Programming in C

This is a set of notes on Programming in C. They were originally prepared as plain ASCII files using a private set of nroff macros and have now been converted to HTML.

Most of the small programming examples are available on-line. Just select the link about the program and then use your WWW browser's ability to save pages in a local file.

There are various other sources of information available on-line. If you are really interested in the C programming language you should

1. Read the comp.lang.c newsgroup
2. Obtain and read the comp.lang.c FAQ
3. Study the submissions to the International Obfuscated C Code Contest

If you are interested in systems programming especially in the Unix context there are further notes on a wide range of topics.

Peter Burden jphb@scit.wlv.ac.uk

Introduction to C Programming - Introduction

Chapter chap1 section 1

By time-honoured convention the first C program anybody writes is known as the "hello world" program. It is shown below.

```
main()
{
        printf("hello, world\n");
}
```

When this program is compiled and run the effect is to display

```
hello, world
```

on the screen.

See also

- Basics of Programming
- The "hello world" Program
- Writing Strings
- Program Layout
- Programming Errors
- Standards and History
- C and C++
- Character Codes
- Exercises
- Review questions

Programming with Integers

Introduction to C Programming - Basics of Programming

Chapter chap1 section 2

In order to run the program you need to go through several stages. These are

- 1. Type the text of the program into the computer
- 2. Translate the program text into a form the computer can use
- 3. Run the translated program

If you make a mistake during step 1 then it is likely that the computer will not be able to translate the program so you will never reach step 3.

The text of a program is known as the source of the program. The translation of the program is known as **compilation** and is performed by a special program known as a **compiler.** The result of the translation or compilation is the **executable** or **binary** version of the program.

The program source is kept in a **source file.** You may use whatever method you prefer to enter and modify the text in the source file. Usually this is done using some sort of text editor program. The use of the word-processing software often found on PCs can cause problems as such software often incorporates extra codes for formatting and laying out text in the text file. Such extra codes will not be understood by the compiler and will cause the compilation to fail.

If you are going to do a lot of programming it is well worth the effort to learn an editor designed for use by programmers. vi and emacs are two such editors frequently found on Unix based computer systems and not uncommon on PCs.

If you are going to store the source in a file for future alteration and development then you will need to name the file in accordance with conventions understood by your compiler. Practically all C compilers expect program source to be provided in files with names ending in **.c .** C compilers also make use of files with names ending in **.h .**

The compiler is a large complex program that is responsible for translating the source program into a form that can be executed directly by the computer. Compilers are either supplied with the rest of the standard programs provided on a computer system or they may be purchased separately. There are even some public domain (i.e. free) compilers.

The process of compilation involves the compiler breaking down the source program in accordance with the published rules for the programming language being

used and generating machine code that will cause the computer to execute the programmer's intentions. Machine code is a sequence of 1s and 0s that control the flow of data between the various internal parts of the computer. This is often called the binary or executable version of the program.

Once a program has been compiled or translated into machine code the code may be executed directly or may be saved in a file for future use. If it is saved in a file then the compiled program may be run at any time by telling the host operating system (Unix, MSDOS etc.,) the name of the file and the fact that it should copy the program from the file into the computer memory and then start the program running.

Once you have prepared the source file you are ready to to try to compile the program. Supposing that you were working on the *hello world* program then you might well put the source in a file called *hw.c* . What you actually type to compile the program depends on the particular compiler and operating system you are using.

Under Unix systems, it is almost always

```
cc hw.c
```

You may sometimes use *gcc* or *acc* . Assuming you don't get any error messages you will now find a file called *a.out* has appeared in the current directory. This contains the executable version of your program and it can now be executed by typing the command

```
a.out
```

If you are familiar with Unix you can change the name of the file to *hw* by typing

```
mv a.out hw
```

or you can type

```
cc -o hw hw.c
```

to compile the program. Unix C compilers often have many options, consult the manual for details.

If you are operating under MSDOS you will need to check your compiler manual and, possibly, consult a local expert to find out what command to type. It may be something like

```
CL HW.C
```

or

```
BCC HW.C
```

Most MSDOS C compilers will then generate an executable file called *HW.EXE*

which can be executed by typing

```
HW.EXE
```

Unix C compilers do not write any messages unless there are errors in your program. If the compilation is successful the next thing you will see is the next operating system prompt. MSDOS compilers often generate intermediate messages concerning libraries and code phases, these are only of interest to more advanced programmers.

It is usual to talk about a C program being executed, this means that the compiled version of the C program is executed. Similarly when talking about how a particular piece of C code is executed we are actually referring to the execution of the machine code generated by compilation of the piece of C code in question.

The hello world program.

Introduction to C Programming - Programming Errors

Chapter chap1 section 6

It is quite likely that your attempt to write a *hello world* program will be completely successful and will work first time. It will probably be the only C program you ever write that works first time. There are many different types of mistake you might make when writing C programs. These can result in messages from the compiler, or sometimes the linker, programs that bring the computer grinding to a halt or programs that simply produce the wrong answer to whatever problem they were trying to solve or programs that get stuck in a never-ending loop. As a beginner you are more likely to encounter compiler or linker error messages, particularly if you have already done some programming in another programming language. The rest of this section illustrates some possible errors and their consequences.

In the first example the programmer has forgotten that the C programming language is case sensitive.

```
MAIN()
{
        printf("hello, world\n");
}
```

This error gave rise to some rather mysterious error messages from the linker. Typical examples are shown below. The compiler on the IBM 6150 used to write these notes produced the following error output.

```
ld: Undefined -
        .main
        _main
        _end
```

The compiler on a SUN Sparc Station produced the following messages.

```
ld: Undefined symbol
   _main
Compilation failed
```

The Turbo C integrated environment produced the following.

```
Linker Error: Undefined symbol _main in function main
```

And finally Microsoft C version 5.1 produced the following error messages.

```
hw.c

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: HW.OBJ
Run File [HW.EXE]: HW.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]: SLIBCEC.LIB

LINK : error L2029: Unresolved externals:
```

```
_main in file(s):
 C:\MSC\LIB\SLIBCEC.LIB(dos\crt0.asm)

There was 1 error detected
```

All these errors reflect the fact that the linker cannot put the program together properly. On Unix based systems the linker is usually called "ld". Along with the user supplied main() function all C programs include something often called the **run-time support package** which is actually the code that the operating system kicks into life when starting up your program. The run-time support package then expects to call the user supplied function main(), if there is no user supplied main() then the linker cannot finish the installation of the run-time support package. In this case the user had supplied MAIN() rather than main(). "MAIN" is a perfectly valid C function name but it isn't "main".

The rather extensive set of messages from Microsoft C 5.1 were partly generated by the compiler, which translated the program without difficulty, and partly generated by the linker. The reference to "crt0" is, in fact, a reference to the C Run Time package which tries to call main() to start the program running.

In the second example the programmer, probably confusing C with another programming language, had used single quotes rather than double quotes to enclose the string passed to printf().

```
main()
{
        printf('hello, world\n');
}
```

On the IBM 6150 the compiler produced the following error messages. The reference to "hw.c" is to the name of the source file that was being compiled.

```
"hw.c", line 3: too many characters in character constant
"hw.c", line 3: warning: Character constant contains more than one byte
```

The SUN Sparc Station compiler gave the following error messages.

```
"hw.c", line 3: too many characters in character constant
Compilation failed
```

The Turbo Integrated environment gave the following messages. C:\ISPTESTS\HW.C was the name of the source file.

```
Error C:\ISPTESTS\HW.C 3:
Character constant too long in function main
```

Microsoft C version 5.1 gave the following messages.

```
hw.c
hw.c(3) : error C2015: too many chars in constant
```

In each case the error message referred clearly to the line number in error. The reference to character constants appears because the C language uses single quotes for a different purpose (character constants).

In the third example the programmer, again possibly confused by another programming language, had missed out the semi-colon on line 3.

```
main()
{
```

```
        printf("hello, world\n")
}
```

The IBM 6150 compiler produced the following message.

```
"hw.c", line 4: syntax error
```

The SUN Sparc Station produced the following messages.

```
"hw.c", line 4: syntax error at or near symbol }
Compilation failed
```

Turbo C produced the following messages.

```
Error C:\ISPTESTS\HW.C 4:
Statement missing ; in function main
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c
hw.c(4) : error C2143: syntax error : missing ';' before '}'
```

In all cases the error message referred to line 4 although the error was actually on line 3. This often happens when compilers detect errors in free-format languages such as C. Simply the compiler didn't realise anything was wrong until it encountered the } on line 4. The first error message is particularly unhelpful.

In the fourth example the programmer wrote *print()* rather than *printf().*

```
main()
{
        print("hello, world\n");
}
```

This, not surprisingly, produced the linker error messages shown in below. These are rather similar to the error messages shown earlier when the effects of writing MAIN() rather than main() were shown. The IBM 6150 compiler generated the following messages.

```
ld: Undefined -
        .print
        _print
```

The SUN Sparc Station compiler generated the following messages.

```
ld: Undefined symbol
   _print
Compilation failed
```

Turbo C generated the following messages.

```
Linking C:\ISPTESTS\HW.C 4:
Linker Error: Undefined symbol _print in module HW.C
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c

Microsoft (R) Overlay Linker  Version 3.65
```

```
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: HW.OBJ
Run File [HW.EXE]: HW.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]: SLIBCEC.LIB

LINK : error L2029: Unresolved externals:



_print in file(s):
 HW.OBJ(hw.c)

There was 1 error detected
```

In the final example the programmer left out the parentheses immediately after main.

```
main
{
        printf("hello, world\n");
}
```

The IBM 6150 compiler produced the following messages.

```
"hw.c", line 2: syntax error
"hw.c", line 3: illegal character: 134 (octal)
"hw.c", line 3: cannot recover from earlier errors: goodbye!
```

The SUN Sparc Station compiler produced the following messages.

```
"hw.c", line 2: syntax error at or near symbol {
"hw.c", line 2: unknown size
Compilation failed
```

Turbo C produced the following messages.

```
Error C:\ISPTESTS\HW.C 2: Declaration syntax error
```

The Microsoft C version 5.1 compiler produced the following messages.

```
hw.c
hw.c(2) : error C2054: expected '(' to follow 'main'
```

All of these messages are remarkably unhelpful and confusing except that from Microsoft C, particularly that from the IBM 6150 compiler.

You may find it interesting to try the various erroneous versions of the "hello world" program on your particular system. Do you think your compiler generates more helpful error messages?

If you are using Turbo C you will see the following message, even when compiling a correct version of the *hello world* program

```
Warning C:\ISPTESTS\HW.C 4:
   Function should return a value in function main
```

A warning message means that there is something not quite right with the program but the compiler has made assumptions that the compiler writer thought reasonable. You should never ignore warnings. The

ideal is to modify the program so that there are no warnings, however that would introduce extra complications into the *hello world* program and this particular message can be ignored for the moment.

The message means that the user supplied function main() should return a value to the run-time support package. There is no requirement to do so and most compilers recognise main() as a special case in this respect. Returning a value to the run-time support package should not be confused with returning a value, sometimes known as an exit code, to the host operating system.

Standards and History

Introduction to C Programming - C Standards and History

Chapter chap1 section 7

The C programming language was developed in the years 1969 to 1973, although the first published description did not appear until the book *The C Programming Language"* written by Brian Kernighan and Dennis Ritchie was published in 1978. The early versions of the C language were strongly influenced by a language called BCPL which itself was a derviative of Algol.

The early development of C was closely linked to the development of the Unix operating system. Large portions of the code of the Unix operating system were eventually written in C and problems encountered in transferring Unix to various computers were reflected into the design of the C language. The modest hardware available to the Unix developers was also reflected in the language design, most notably the use of separate library functions for operations such as input and output. Until the early 1980s the language was almost exclusively associated with Unix.

The widespread introduction of microprocessor based computer systems in the early 1980s also saw a rapid growth in the use of C on such systems. C compilers were known to be small and many of the start-up operations that produced the early microprocessor based computer systems were staffed by ex-students who had encountered the language whilst at university.

As early as 1982 it became clear that the informal description of the language in Kernighan & Ritchie's book was not good enough. ANSI established a committee known as X3J11 in 1983. This committee produced a report defining the language at the end of 1989. The report was known as X3.159 but the standard was soon taken over by ISO with the designation ISO/IEC 9899-1990. This version of the language is known as ANSI-C to distinguish it from the earlier version of the language described in Kernighan and Ritchie's book. The earlier version of the language is known as K&R C. C++ and Objective-C are different languages developed from C. The GNU C compiler, often known by the command that invokes it, **gcc** , is public domain software available for both Unix and MSDOS based systems. It supports a version of the language close to the ANSI standard.

All the code presented in these notes, unless specifically indicated otherwise, confirms to the ANSI standard. Many compiler writers and vendors produce C compilers that will compile code conforming to the ANSI standard but they also provide a variety of extensions to suit the particular target environment. Such extensions are ususally extra library routines for functions such as PC screen handling and interfacing direct to MSDOS system functions. Sometimes the

extensions include extra features introduced into the language, usually to cope with some of the problems of memory management on MSDOS based systems. In the Unix environment such extensions are less common, Unix systems are normally supplied with a compiler and the extensions appear as extra libraries for functions such as driving X-Windows displays or network communications.

All C programmers ought to be aware of what is and what isn't standard in their particular environment. The better compiler writers usually make this fairly clear in their manuals.

It is also often possible to obtain further special purpose application specific libraries for use with C compilers to provide facilities such as database handling, graphics, numerical analysis etc.

C and C++

Introduction to C Programming - The "hello world" Program

Chapter chap1 section 3

You may be wondering about the need for all the peculiar symbols in the "hello world" program. They are all essential as can be demonstrated by leaving some of them out, the consequences of such errors are discussed in more detail later. First, however, let's take another look at the *hello world* program.

```
main()
{
        printf("hello, world\n");
}
```

This program, in fact, consists of a single piece or chunk of executable code known as a **function .** Later on we will see programs consisting of many functions. All C programs MUST include a function with the name *main* , execution of C programs always starts with the execution of the function *main* , if it is missing the program cannot run and most compilers will not be able to finish the translation process properly without a function called *main* .

It is important to note that the required function is called *main* not *MAIN* . In the C programming language the case of letters is always significant unlike many older programming languages.

In the simple *hello world* program *main* appears on line 1. This is not essential, program execution starts at *main* not at the first line of the source. However, it is conventional in C programs, to put *main* at, or near, the start of the program.

The round brackets, or **parentheses** as they are known in computer jargon, on line 1 are essential to tell the compiler that when we wrote *main* we were introducing or defining a function rather than something else. You might expect the compiler to work out this sort of thing for itself but it does make the compiler writer's task much easier if this extra clue is available to help in the translation of the program.

On lines 2 and 4 you will see curly brackets or braces. These serve to enclose the body of the function *main* . They must be present in matched pairs. It is nice, but not essential, to line them up in the fashion shown here.

Line 3 is the meat of this simple program. The word *printf* clearly has something to do with printing. Here, of course, it is actually causing something to be displayed to the screen, but the word print has been used in this context since the time before

screens were generally used for computer output. The word *printf* was probably derived from *print formatted* but that doesn't matter very much, all you need to know is that this is the way to get something onto the screen.

In computer jargon *printf* is a library function. This means that the actual instructions for displaying on the screen are copied into your program from a library of already compiled useful functions. This process of putting together a program from a collection of already compiled bits and pieces is known as **linking** and is often done by a **linker** or **loader** program as distinct from a compiler. Most compilation systems, sensibly, hide this complication from the user but you will see references to linkage or linker errors from time to time. More advanced programmers can build their own private libraries of useful things.

Line 3 is a **statement.** A statement tells the computer to do something. In computer jargon we say that a statement is executed. This simple example consists of the use of a library function to do something. In the C programming language a simple use of a function (library or otherwise) is, technically, an expression, i.e. something that has a value. In computer jargon we say that an expression is evaluated. To convert an expression into a statement a semi-colon must follow the expression, this will be seen at the end of line 3. The actual value of the expression is of no interest in this case and no use is made of it.

A C function normally consists of a sequence of statements that are executed in the order in which they are written. Each statement must, of course, be correctly written before the sequence is acceptable to the compiler. The following version of the "hello world" program shows the use of multiple statements.

```
main()
{
        printf("hello, ");
        printf("world");
        printf("\n");
}
```

Note that each one of the three statements is a printf expression converted to a statement by a terminating semi-colon.

Officially the purpose of *printf* is to write things to the standard output or screen. The list of things to be printed out are enclosed within parentheses immediately after the word printf. They are known as the function parameters or arguments. We sometimes say that the parameters are passed to the function. Incidentally when talking about any function, a C programmer would refer to it, in writing, by its name with the parentheses. We shall adopt this convention from now on.

In the *hello world* program *printf()* has only got a single parameter, this is the sequence of characters

```
hello, world\n
```

This sequence is identified as a single parameter by enclosing it in double quotes. Such a sequence of characters enclosed in double quotes is known as a **string.** Strings are discussed in more detail in the next section

---

Writing strings

Introduction to C Programming - Writing strings

Chapter chap1 section 4

In the previous section we have used the library function *printf()* with a single string as parameter. This appeared as

```
printf("hello world\n");
```

The double quote character used to enclose strings should not be confused with a pair of single quote characters.

If you have followed what has been said so far you may well be wondering why the character \n was included in the string and why it didn't appear in the output. The answer is that this pair of characters is converted by the compiler into a single new-line character that is stored as part of the string. Actual new-line characters in the source of the program are treated as if they were space characters by the compiler, so the escape convention is necessary to get a new-line character into the string.

The program example used earlier to illustrate multiple statements also illustrates the significance of the \n character in the strings. If the \n were left off the end of the last line of program output then the effect would be that the next operating system prompt would appear on the same line as the output of the program.

You can include as many \n's as you like within a string enabling multi-line output to be produced with a single use of the *printf()* function. Here's an example.

```
main()
{
        printf("This is an\nexample of\nmulti-line output\n");
}
```

When the program was compiled and run it produced the following output.

```
This is an
example of
multi-line output
```

It would not be possible to include an actual new-line character in the string in the program source because this would "mess-up" the source and the compiler would not be able to translate the program properly.

However if the string is too long to fit comfortably on a single line of the source listing of program then it is possible to spread a string over several lines by **escaping** the actual new-line character at the end of a line by preceding it with a backslash. The string may then be continued on the next line as the following example shows.

```
main()
{
        printf("hello, \
world\n");
}
```

Note that there are no characters after the backslash on the third line and also note that the continuation string must start at the very start of the next line. The following nicely laid out version of the above program.

```
main()
{
        printf("hello, \
        world\n");
}
```

produced the output

```
hello,         world
```

the indenting spaces at the start of the string continuation being taken as part of the string.

An alternative, and probably nicer, approach is to make use of what is called **string concatenation.** This simply means that two strings which are only separated by spaces are regarded by the compiler as a single string. Actually the two strings can be separated by any combination of spaces, newlines, tab characters and comments. In the jargon of C programming all these things are collectively known as **white space** although perhaps we should now say "content challenged space". The use of string concatenation is shown by the following example.

```
main()
{
        printf("hello," /* space only */
                " world\n");
}
```

Programmers using MSDOS should note that versions of *printf()* supplied with MSDOS compilers normally convert \n to a carriage return and line feed (new-line) character pair, there is no need to try and include a carriage return character in a string. On Unix systems this is the responsibility of the display driving software.

There are several other characters that cannot conveniently be included in a string but which can be included using the \ notation. The complete list is

```
\a      "BELL" - i.e. a beep
\b      Backspace
\f      Form Feed (new page)
\n      New Line (line feed)
\r      Real carriage return
```

```
\t      Tab
\v      Vertical Tab
\\      Backslash
\'      Single Quote
\"      Double Quote
\?      Question Mark
```

It is not necessary to use the \ convention for all the above characters in strings but they will always be understood.

It is also possible to include arbitrary characters in strings by including the three digit octal representation of the character preceded by a \ symbol. For example the "hello world" program could be written in this form.

```
main()
{
        printf("\150ello, world\n");
}
```

In case you didn't know, 150 is the octal code for "h". It is not, generally, possible to use hexadecimal or decimal constants in this fashion. The use of \ in this way is known as "escaping" and the \ is known as an "escape" character, it turns off the normal meaning of the immediately following character.

The effect is actually to send the string of binary 1s and 0s equivalent to octal 150 to the output device which should respond in the manner its designers intended it to. You can use this convention to send all sorts of unusual special characters to all sorts of unusual output devices.

Program Layout

Introduction to C Programming - Program Layout

Chapter chap1 section 5

The *hello world* program listed at the start of this chapter is laid out in accordance with normal C conventions and practice. The C programming language is "free format" which means you can lay out the text of programs in whatever form takes your fancy provided, of course, you don't run words together. The "hello world" program could have been written in this form

```
main(){printf("hello, world\n");}
```

or even this

```
main
(
)
{
printf
(
"hello, world\n"
)
;
}
```

Neither of the above is recommended.

The indentation of the word *printf* in the original version of the *hello world* program is achieved by the use of a single **TAB** character not a sequence of spaces. TAB characters may be generated by hitting the TAB key on normal keyboards, occasionally the TAB key carries a mysterious symbol consisting of 2 arrows. Multiple TAB characters should be used to indent C programs in preference to sequences of spaces. The use of TABs makes for smaller source files, less typing and tidier layout. All C compilers understand the use of TABs. Unfortunately some word processors and editors have strange ideas about standard TAB settings, for C source TABs should be set every 8 characters.

All but the briefest programs will benefit from descriptive comments included in the source. In the C programming language anything enclosed between the character pair **/*** and the character pair ***/** is treated as a comment. The compiler treats any comment, however long, as if it were a single space. Comments may appear in any reasonable place in the source. A commented version of the "hello world" program

appears below.

```
/*      This is the first C program

        Author  : J.P.H.Burden
        Date    : 22 September 1992
        System  : IBM 6150 + AIX Version 2.1
*/
main()
{
        printf("hello, world\n");
}
```

Comments may stretch over several lines but may not be nested, i.e. you cannot include comments within comments. This can cause difficulties if you wish to "comment out" parts of a program during development. The following use of comments is perfectly correct.

```
main()
{
        printf("hello, world\n");
/*
        some very clever code I haven't
        quite sorted out yet.
*/
        printf("the solution to the problem of \
life, the universe and everything is");
/*
        more very clever code still to be
        developed
*/
}
```

But the following attempt to "comment out" further parts of the program.

```
main()
{
        printf("hello, world\n");
/*
        OK let's stop pretending we can write
        really clever programs
        /*
                some very clever code I haven't
                quite sorted out yet.
```

```
        */
                printf("the solution to the problem of \
        life, the universe and everything is");
        /*
                more very clever code still to be
                developed
        */
*/
}
```

Gave the following compiler error messages when using the SUN Sparc Station compiler.

```
"comm2.c", line 17: syntax error before or at: /
Compilation failed
```

The error message refers to the final "*/" in the program.

Program Errors

Introduction to C Programming - C and C++

Chapter chap1 section 8

Although it is the intention of these notes to stick rigidly to the ANSI standard for C, it is quite possible you will be using a compiler, such as Turbo C, that is really a C++ compiler. This is possible because C++ is a strict superset of C which means that any C program should be acceptable to a C++ compiler. In fact some C++ compilers operate by first converting the C++ code into C. As well as the ideas associated with object-oriented programming, C++ introduces a few extra features that the C programmer should be aware of, these include an alternative syntax for comments, several extra keywords that cannot be used in normal programming and a very simple form of input and output known as stream IO.

The alternative comment syntax uses the symbol // to mark the start of a comment and the end of a line to indicate the end of a comment. This means that you cannot have multi-line comments in C++ and it also means that you cannot have a comment in the middle of a line of code. If you have ever programmed in assembler language you'll be familiar with this style of commenting. The problem of nested comments does not arise.

Of course, C++ compilers will still accept C style comments marked by symbols "/*" and "*/". You shouldn't mix the two styles in the same program.

Here is an example of the *hello world* program written using C++ commenting style.

```
//          A simple program to demonstrate
//          C++ style comments
//
//          The following line is essential
//          in the C++ version of the hello
//          world program
//
#include        <stdio.h>
main()
{
        printf("hello, world\n");  // just like C
}
```

The C++ stream output facility is illustrated by the following program. The basic syntax of C++ stream output is

```
cout << value-to-display
```

Note that the program, like the previous example, requires the use of "#include" and a header file name. The meaning of "#include" will be discussed in a later chapter.

```
#include        <iostream.h>
main()
{
        cout << "hello, world\n";
}
```

The "<<" symbol is read as "puts to" but may also be read as "shift to". This is really C's left shift operator. C++ allows hexadecimal chraracter representations in strings unlike ANSI C.

Character Codes

Introduction to C Programming - Character Codes

Chapter chap1 section 9

Here is a list of ASCII character codes in octal and hexadecimal notation along with the equivalent C escape sequence where relevant. This table is not comprehensive.

Octal code	Hexadecimal code	Escape Sequence	Control Sequence	Standard Designation	Function
000	00	-	Ctrl-@	Null	does nothing
001	01	-	Ctrl-A	SOH	-
002	02	-	Ctrl-B	STX	-
003	03	-	Ctrl-C	ETX	Sometimes Break Program
004	04	-	Ctrl-D	EOT	Sometimes End of File
005	05	-	Ctrl-E	ENQ	-
006	06	-	Ctrl-F	ACK	-
007	07	\a	Ctrl-G	BEL	Audible Signal
010	08	\b	Ctrl-H	BS	Backspace
011	09	\t	Ctrl-I	HT	TAB
012	0a	\n	Ctrl-J	NL	Newline or Linefeed
013	0b	\v	Ctrl-K	VT	Vertical TAB
014	0c	\f	Ctrl-L	FF	New Page or Clear Screen
015	0d	\r	Ctrl-M	CR	Carriage Return
016	0e	-	Ctrl-N	SO	-
017	0f	-	Ctrl-O	SI	-
020	10	-	Ctrl-P	DLE	-
021	11	-	Ctrl-Q	DC1	Flow Control - On
022	12	-	Ctrl-R	DC2	-

023	13	-	Ctrl-S	DC3	Flow Control - Off
024	14	-	Ctrl-T	DC4	-
025	15	-	Ctrl-U	NAK	Sometimes Line Delete
026	16	-	Ctrl-V	SYN	-
027	17	-	Ctrl-W	ETB	-
030	18	-	Ctrl-X	CAN	Sometimes Line Delete
031	19	-	Ctrl-Y	EM	-
032	1a	-	Ctrl-Z	SUB	Sometimes End of File
033	1b	-	Ctrl-[	ESC	Often Escape for Screen Control
034	1c	-	Ctrl-\	FS	-
035	1d	-	Ctrl-]	GS	-
036	1e	-	Ctrl-^	RS	-
037	1f	-	Ctrl-_	US	-
040	20	-	-	SP	Space - Hit the space bar

And the normal printing character codes with hexadecimal and octal equivalent values.

```
21 041 !  22 042 "  23 043 #  24 044 $  25 045 %  26 046 &
27 047 '  28 050 (  29 051 )  2a 052 *  2b 053 +  2c 054 ,
2d 055 -  2e 056 .  2f 057 /  30 060 0  31 061 1  32 062 2
33 063 3  34 064 4  35 065 5  36 066 6  37 067 7  38 070 8
39 071 9  3a 072 :  3b 073 ;  3c 074 <  3d 075 =  3e 076 >
3f 077 ?  40 100 @  41 101 A  42 102 B  43 103 C  44 104 D
45 105 E  46 106 F  47 107 G  48 110 H  49 111 I  4a 112 J
4b 113 K  4c 114 L  4d 115 M  4e 116 N  4f 117 O  50 120 P
51 121 Q  52 122 R  53 123 S  54 124 T  55 125 U  56 126 V
57 127 W  58 130 X  59 131 Y  5a 132 Z  5b 133 [  5c 134 \
5d 135 ]  5e 136 ^  5f 137 _  60 140 `  61 141 a  62 142 b
63 143 c  64 144 d  65 145 e  66 146 f  67 147 g  68 150 h
69 151 i  6a 152 j  6b 153 k  6c 154 l  6d 155 m  6e 156 n
6f 157 o  70 160 p  71 161 q  72 162 r  73 163 s  74 164 t
75 165 u  76 166 v  77 167 w  78 170 x  79 171 y  7a 172 z
7b 173 {  7c 174 |  7d 175 }  7e 176 ~
```

Exercises

Introduction to C Programming - Exercises

Chapter chap1 section 10

1. Your first task, of course, is to enter, compile and run the *hello world* program on your computer system. Note the commands you used to compile the program, any messages produced and the names of any files generated.
2. Now try some of the programs with errors. Note the error messages produced by your compiler. Do you think they are more infomrative than those described in the notes ?
3. What is the effect of typing the *hello world* program using round brackets (parentheses) rather than curly brackets (braces) ?
4. Write a C program that produces the following output

   ```
   ****************
   *  hello world *
   ****************
   ```

   using multiple printf() calls.
5. Write a C program that prints *hello world* vertically on the screen.
6. Write a C program that prints *hello world* on the screen and then "beeps".
7. On many displays the character sequence ESC [2 J causes the display to go blank. Use this information to write a C program that clears the screen. (If you are working on a PC running MSDOS you'll need to ensure that ANSI.SYS is operational)
8. Write a program that displays hello world in the centre of an otherwise blank screen.

Chapter 1 review questions

1. C's output function *printf()* is
    1. part of the 'C' language
    2. a library function
    3. a function users must write as part of every 'C' program
    4. another name for the function *print*
2. An escape character can be included in a 'C' string by including the following characters in the string
    1. \e
    2. ESC
    3. \033
    4. \0x1B
3. All 'C' programs must include
    1. The keyword 'program'
    2. A function called 'main'
    3. The name of the author
    4. A lest one use of the *printf()* function
4. Conventionally 'C' source files have names ending in
    1. .c
    2. .cpp
    3. .bas
    4. .html
5. The effect of writing *print()* rather than *printf()* is that
    1. The program will not compile
    2. The program will work as expected
    3. The program will display "hello world" in inverse video
    4. The program will not link correctly
6. 'C' programs
    1. must be written with one statement per line
    2. must be written entirely in lower case letters
    3. can be laid out in any reasonable manner

4. can only be written using the editor 'edlin' (under MSDOS) or 'vi' (under Unix)

7. The 'C' program

```
main()
{
        printf("hello "
        "world" "\n");
}
```

   1. will print out "hello world"
   2. will print out "hello" on one line followed by "world" on the next line
   3. will not compile correctly
   4. will just print out "hello"

8. Two strings are concatenated if they are separated by
   1. the string "\040"
   2. a TAB character
   3. a comment
   4. a newline character

9. Much of the Unix operating system is written in 'C' because
   1. 'C' was developed along with Unix to ensure the portability of Unix
   2. It was the only language available on the computers used for developing Unix
   3. IBM wouldn't let the developers of Unix use PL/I
   4. 'C' provides the necessary low-level control and primitives for an operating system
   5. MSDOS had already been written in 'C'

Wrong

Right

Wrong

print() is not a standard 'C' function.

Wrong

It would be nice if \e could be used in this way and the ANSI standardisation effort did consider the possibility but decided that the notion of Escape was not truly portable (mainly to IBM mainframes).

Wrong

ESC is the conventional name for this character. It is not understood by any 'C' compiler. Check the character codes information.

Wrong

Hexadecimal 1B is the same as Octal 33, the code for Escape. ANSI 'C' does not allow hexadecimal constants within strings although this is a common (and non-portable) extension.

Wrong

There is no such keyword in 'C'. You may be thinking of the Pascal programming language.

Wrong

Including the author's name, within a comment is certainly desirable but, like everything else within a comment, it is entirely optional.

Wrong

There are many other ways of creating output in 'C'. A valid program need not generate any output. In fact it need not do anything at all. The following is perfectly OK, if rather pointless

```
main()
{
}
```

Right

If your operating system is not sensitive to the case of letters in file names, then it migt be 'C'.

Wrong

This is used for C++ programs by some systems. Since all valid 'C' programs are also, more or less, valid C++ programs .cpp may be acceptable.

Wrong

Ridiculous.

Wrong

Ridiculous

Wrong

The program will compile. Functions are incorporated at linkage time.

Wrong

If, of course, *print()* were the name of a, non-standard, function with a similar functionality to *printf()* then the program would work as expected.

Wrong

Unless, of course, *print()* does this.

Right

This would be the behaviour in a regular ANSI environment. If your libraries included *print()* as a non-standard function then the program would link but the results of running the program would depend on what the *print()* function actually did.

Wrong

See the notes on program layout.

Wrong

Although 'C' function names, variable names and keywords are conventionally lower case, this is only a convention. You **can** write in upper case if you really want to but remember that standard library function names and keywords are in lower case.

Wrong

Any **text** editor can be used but beware of word processors which insert all sorts of special mark-up characters in the text they are creating.

Right

The three strings are concatenated into a single string. See the related notes.

Wrong

The strings are concatenated into a singel string. See the notes

Wrong

String concatenation, i.e. the assembly of adjacent strings into a single string is part of the ANSI standard. See the notes

Wrong

Separating them by a further string simply results in the separating string being included between them in the concatenated result. In this case the string actually consists of a single space character. (See the codes). The result is that the strings are concatenated but an extra space character appears at the concatenation point.

Right

It was also developed to include the necessary primitives and facilities.

Wrong

Some form of assembly language would have been available.

Wrong

They would, at a price ! More importantly PL/I was not available on the hardware in use (early DEC machines) and was known to be a large and complex language.

Right

Portability was another important consideration

Wrong

MSDOS was developed later than Unix and was written in Assembly language.

Programming With Integers - Introduction

Chapter chap2 section 1

All computers work internally with numbers. All the familiar things computers appear to manipulate such as strings, characters, graphics, sound etc., are represented inside the computer as numbers. In this chapter we will be looking at various elementary techniques for reading in, storing and printing out numbers. We will further confine our study to whole numbers or integers as they universally called by computer scientists and mathematicians. We will also see how to make the computer do some very simple calculations with numbers.

See Also

- A simple program to add up two numbers
- Storing numbers
- Input of numbers
- Output of numbers
- Another program to read in two numbers
- Setting initial values
- Output layout control
- The effects of input errors
- The use of input layout control
- Programming errors
- Keywords
- C and C++
- Exercises

Arithmetic

Programming With Integers - A Simple program to add up two numbers

Chapter chap2 section 2

A simple C program that reads in two numbers, stores them and prints out the sum is listed below.

```
/*      A program to read in two numbers
        and print their sum
*/
main()
{
        int     x,y;    /* places to store numbers */
        printf("Enter x ");
        scanf("%d",&x);
        printf("Enter y ");
        scanf("%d",&y);
        printf("The sum of x and y was %d\n",x+y);
}
```

After compiling the program the following dialogue is typical of operation of the program.

```
Enter x 4
Enter y 5
The sum of x and y was 9
```

There are a number of features of this program that require further explanation. If you have had any experience of programming you will probably guess the following points. The code on line 6 reserves and names places to store the numbers. The library function *scanf()* used on lines 8 and 10 reads from the keyboard. The addition of the two stored numbers is done on line 11 by calculating the value of x+y. Even if you have had programming experience you may well be puzzled by the percent (%) and ampersand (&) symbols appearing on lines 8, 10 and 11.

Storing Numbers

Programming With Integers - Storing Numbers

Chapter chap2 section 3

The first task to be performed when writing any program that manipulates numbers is to decide where to store them and tell the computer about it. Numbers are stored in the computer's memory. Individual memory locations are known to the computer by their reference number or **address** but programmers, being human beings, much prefer to give the locations names that can be remembered easily. It is the compiler's task to manage the relationship between the programmer chosen names of memory locations and the internal reference numbers. This is done using a directory or look-up table. A programmer is not normally in the least bit interested in the internal reference numbers although, as we shall see in a later chapter, a C programmer can determine and display these reference numbers or addresses should she or he want to. The important thing about computer memory is simply that numbers, once stored in a memory location, stay there until either the program finishes or some other number is stored in that location or the computer is switched off.

The parts of a program that tell the compiler that memory locations are required and should be referred to by certain names are known as **declarations.** In its simplest form a declaration consists of a word describing what type of memory is required followed by the names to be used to refer to the locations.

For example line 6 of the sum of two numbers program reads

```
int x,y;
```

This requests that the compiler reserves two memory locations of type *int* and that they can be referred to as *x* and *y* .

At this stage in our study of the C programming language, we will only consider one type of memory location known as **int .** Memory locations of type *int* are used to store integers. There are many other types and variations which will be considered in a later chapter.

The word **int** is special for the compiler. Whenever the compiler sees the word *int* in a source program, the compiler assumes that it is processing the start of a declaration which requests the reservation of one or more memory locations of type *int* . In computer jargon we say that *int* is a **keyword** or **reserved word.** Most programming languages have keywords and misuse of keywords can cause obscure and puzzling error messages. A complete list of C programming language keywords will be found at the end of this chapter. The meaning of them will be made clear in due course, but, in the mean time, you must avoid using any of these words unless

you know what they mean.

The word *int* is followed by a list of names to be used. The names in the list are separated by commas. As in all other programming languages there are rules for acceptable names of memory locations. The rules for the C programming language are

1. Names must start with a letter or underscore symbol. Names starting with underscore symbols are often used for special internal purposes by compilers and linkers and should be avoided in normal programming.
2. Names must only consist of letters, numbers and underscore symbols. No other characters are allowed within names.
3. Names may be as long as the programmer wishes but any characters after the first 31 may be ignored by the compiler, i.e. they are not significant. The actual number of significant characters may vary between compilers, you may rely on the first 31 characters being significant in all but the oldest non-standard compilers.

The fact that the C programming language is case sensitive means that names of memory locations can be in upper case, lower case or a mixture of both. Most C programmers use names constructed entirely from lower case letters as these are easier to type and read.

You cannot declare the same name more than once. The name of a memory location must not clash with the name of a function, i.e. you can't have memory locations called *main* and *printf* in a program that uses these functions. All declarations are terminated by a semi-colon. Declarations may be spread over several lines if you wish. This is shown below.

```
int     x,      /* first number */
        y;      /* second number */
```

Multi-line declarations are probably preferable to declaring several names on a single line because they allow descriptive comments to be placed after each declaration. The following common alternative form is simply two declarations.

```
int     x;      /* first number */
int     y;      /* second number */
```

Memory locations used for holding numbers in this way are often referred to as **variables** because it is always possible to change the number stored in the location. The names associated with memory locations are known as **identifiers.** It is common practice to talk about *the variable x* , this is, occasionally, misleading because what is really meant is *the number stored in the location whose address is x* . The circumlocution is, not surprisingly, not commonly used but you should remember that this is what is really going on or you will get very confused when

addresses, as distinct from contents of addressed locations, are the subject of arithmetic in more advanced programming.

All variables used in a program must be declared. All variable declarations must appear before any executable statements.

The question of the initial values stored in memory locations will be discussed later.

Input of Numbers

Arithmetic and Data Types - Introduction

Chapter chap4 section 1

In this chapter we will see how the C programming language handles a wider range of numerical data types than the simple integers we have studied so far. For the various data types we will see how to read them in, how to print them out, how to write constant values and how expressions involving numbers of more than one type are evaluated.

There are three basic data types. The notion of a data type reflects the possibility of the 1s and 0s stored in a computer memory location being interpreted in different ways. For full details an appropriate text on computer architecture should be consulted.

- The first basic data type we shall consider is the floating point number. Floating point numbers provide a way of storing very large and very small numbers including fractions. There are often several different ways of storing floating point numbers offering a choice of precision or number of significant digits. High precision storage of floating point numbers uses up more computer memory and results in slower arithmetic. Floating point numbers are sometimes called real numbers although mathematicians would object to this usage, there being many numbers that cannot be stored exactly as floating point numbers.
- We have already discussed integers but here we shall see that, like floating point numbers they can be stored in several different ways providing more or fewer significant digits and also offering the option of storing them without a sign.
- Finally the character data type provides a way of storing and manipulating the internal codes that represent the symbols that appear on output devices and the symbols that are engraved on keyboard keytops.

The C programming language does not support the fixed point and binary coded decimal data types widely used in commercial programming to store monetary information.

See also

- Floating Point Numbers
- Floating point data type mismatch in *printf()*
- Output Layout and Constant Formats for floating point numbers

- Accuracy of floating point arithmetic
- Integer Data Types
- The effect of precision specification on integer output
- Usigned Integers, Octal and Hexadecimal Conversions
- Bit-wise Operators
- Character Data Types
- Arithmetic with numbers of different data types
- Conversions and promotions between numbers of different types
- Use of casts for numeric conversions
- Operator precedence
- C and C++
- Exercises

Programming With Integers - Keyword List

Chapter chap2 section 12

The final part of this chapter is the promised list of keywords which is given below. Do not use any of these words unless you know what they mean. This restriction does not, of course, apply to text within strings.

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

If you have any previous experience of programming, you will recognise quite a number of these. Some compilers may support various extra keywords. You will need to consult your compiler manual carefully for details but the following list includes some of the more likely ones. Remember that the use of these keywords and the associated facilities constitute extensions but it is wise to avoid them as they could cause problems when programs are moved to systems whose compilers support such extensions.

```
asm*        fortran     near+       public*
catch*      friend*     new*        readonly
cdecl       generic     operator*   template*
class*      globaldef   pascal      this*
defined     globalref   pragma      throw*
delete*     globalvalue private*    try*
entry       huge+       protected*  virtual*
far+        inline*
```

The words indicated with an asterisk are keywords for the C++ programming language. The words indicated with a plus sign are keywords used by many MSDOS C compilers for extended memory operations.

C and C++

Programming With Integers - Initial Values of Variables

Chapter chap2 section 7

You may have wondered what values are stored in the variables *x* and *y* in the sum of two numbers programs before any values have been read in. The answer is that the values are indeterminate, this means that you cannot make any assumptions about what values are initially in any location. On many systems you will find that the initial value is zero but you **must not** rely on this.

The following program demonstrates what happens if you do not set initial values.

```
main()
{
        int     x;
        int     y;
        printf("Initial value of x is %d\n",x);
        printf("Initial value of y is %d\n",y);
}
```

On the IBM 6150 it produced the following output.

```
Initial value of x is 0
Initial value of y is 0
```

On the SUN Sparc Station the result was

```
Initial value of x is 0
Initial value of y is 32
```

And finally Turbo C gave the following results

```
Initial value of x is 0
Initial value of y is 248
```

Should you want a variable to have some defined initial value then this can be included within the declaration by following the variable name by an equals symbol (=) and a value in the declaration. This is known as **initialisation.** The following program shows the initialisation of variables.

```
main()
{
```

```
        int     x=3,y=4;
        printf("The sum of %d and %d is %d\n",x,y,x+y);
}
```

When compiled and run it produced the following output.

```
The sum of 3 and 4 is 7
```

The initialisation is part of the declaration. Initialised and uninitialiased variables can be mixed in the same declaration. For example

```
int x,y=7,z;
```

In fact a variable can be initialised to any expression whose value can be determined by the compiler. This means that initialisations such as

```
int     x=4+7;
int     z=3,y=z+6;
```

are acceptable although rather pointless and not conventional programming.

Output Layout Control

Programming With Integers - Input of Numbers

Chapter chap2 section 4

The library function **scanf()** performs several tasks. It determines which keys the user has pressed, calculates, in the computer's internal form, the number that the user has typed and finally stores the number in a particular memory location.

In many elementary programs *scanf()* will be used to read from the keyboard. In fact *scanf()* reads from something called the **standard input ,** this is usually the keyboard but the host operating system has complete freedom to connect something else to the standard input. This may, typically, be a stream of data read from a file.

The simple examples of the use of *scanf()* in the sum of two numbers program each have two parameters. If a function is called with more than one parameter then the parameters are supplied as a comma separated list.

The first parameter is a string of the sort we have already seen used with *printf()* as will be seen from the enclosing double quotes. The purpose of the string is to tell *scanf()* what rules to apply when converting from the set of keys pressed by the user to the internal form of number storage used by the computer. This is necessary because when you type a number such as 27 on the keyboard this causes the two characters "2" and "7" to be sent to the computer. Internally computers store numbers using the binary system, which is convenient for electronic manipulation but totally unsuitable for human use. The library function *scanf()* will determine that the keys pressed were "2" and "7" and use this information to generate the internal binary number 0000000000011011. (The number of zeroes at the front varies from computer to computer.)

Within the string *%d* means convert from external decimal form to an internal binary integer. External decimal form is simply the way you normally write or type integers. There are many other forms of conversion that will be described later. The initial percent symbol (**%**) in the string means that the following "d" is a conversion type specification rather than specifying that *scanf()* is expecting to find a d in the input.

The second parameter is the address of the place to store the number read in and converted. The ampersand (&) means "address of" and **must not** be left out. Leaving out the ampersand almost always results in chaos. If you want to see what sort of chaos results skip on towards the end of this chapter.

Output of Numbers

Programming With Integers - Programming Errors

Chapter chap2 section 11

`integer` rather than `int`

As with the earlier *hello world* program it is quite possible to make a variety of errors with the sum of 2 numbers programs. In the first example the programmer, possibly confused by another programming language has written `integer` rather than `int` .

```
main()
{
        integer x,y;
        printf("Enter x ");
        scanf("%d",&x);
        printf("Enter y ");
        scanf("%d",&y);
        printf("The sum of x and y is %d\n",x+y);
}
```

This resulted in various error messages from the compilers. First the error messages produced by the IBM 6150 compiler.

```
"sum2.c", line 3: integer undefined
"sum2.c", line 3: syntax error
"sum2.c", line 5: x undefined
"sum2.c", line 7: y undefined
```

The following error messages were produced by the SUN Sparc Station compiler.

```
"sum2.c", line 3: integer undefined
"sum2.c", line 3: syntax error at or near variable name "x"
"sum2.c", line 5: x undefined
"sum2.c", line 7: y undefined
Compilation failed
```

And finally the following error messages were produced by the Turbo C compiler. These have been edited slightly by removing the file name that the Turbo C compiler puts at the start of each line.

```
3: Undefined symbol "integer" in function main
3: Statement missing ; in function main
```

```
5: Undefined symbol 'x' in function main
7: Undefined symbol 'y' in function main
```

A variable called `int`

In the next example the programmer tried to use two variables called `int` and `inp` . `int` is, of course, a keyword and, not surprisingly, the compilers gave various error messages.

```
main()
{
        integer int,inp;
        printf("Enter int ");
        scanf("%d",&int);
        printf("Enter inp ");
        scanf("%d",&inp);
        printf("The sum of int and inp is %d\n",int+inp);
}
```

First the messages from the IBM 6150.

```
"sum2.c", line 3: illegal type combination
"sum2.c", line 3: syntax error
"sum2.c", line 5: syntax error
"sum2.c", line 8: syntax error
```

And now the SUN Sparc Station error messages.

```
"sum2.c", line 3: integer undefined
"sum2.c", line 3: syntax error at or near type word "int"
"sum2.c", line 5: syntax error at or near type word "int"
"sum2.c", line 7: inp undefined
"sum2.c", line 8: syntax error at or near type word "int"
Compilation failed
```

And finally the edited Turbo C error messages.

```
3: Too many types in declaration in function main
3: Need an identifier to delcare in function main
5: Expression syntax in function main
7: Undefined symbol 'inp' in function main
8: Expression syntax in function main
```

Confused variable names

In the next example the programmer has declared variables called n1 and n2 but has attempted to read into variables called x and y.

```
main()
{
        int     n1,n2;
        printf("Enter x ");
        scanf("%d",&x);
        printf("Enter y ");
        scanf("%d",&y);
        printf("The sum of x and y is %d\n",n1+n2);
}
```

This, not surprisingly, resulted in messages about undefined symbols. First from the IBM 6150.

```
"sum2.c", line 5: x undefined
"sum2.c", line 7: y undefined
```

Second, almost identically, from the SUN Sparc Station

```
"sum2.c", line 5: x undefined
"sum2.c", line 7: y undefined
Compilation failed
```

and finally from Turbo C

```
5: Undefined symbol 'x' in function main
7: Undefined symbol 'y' in function main
```

Misplaced declarations

In the next example the programmer has interleaved declarations and executable statements.

```
main()
{
        int     x;
        printf("Enter x ");
        scanf("%d",&x);
        int     y;
        printf("Enter y ");
        scanf("%d",&y);
```

```
        printf("The sum of x and y was %d\n",x+y);
}
```

This resulted in the following error messages from the IBM 6150 compiler

```
"sum2.c", line 6: syntax error
"sum2.c", line 8: y undefined
```

and the following from the SUN Sparc Station compiler

```
"sum2.c", line 6: syntax error at or near type word "int"
"sum2.c", line 8: y undefined
Compilation failed
```

Surprisingly the Turbo C compiler accepted the program without complaint and it worked quite happily and correctly. (See later notes on C++). The ANSI C standard does not sanction this sort of coding. Cases where compiler writers allow things that are not allowed by the published standard rules are called "extensions". If all your programs are always going to be compiled by the same compiler then there is no harm in using extensions and they are sometimes rather useful, so much so that they sometimes make their way into new published standards. However if you intend to publish your program in any way or anticipate it being moved from one computer system to another you must avoid such extensions at all costs.

Missing ampersands in *scanf()*

In the next example of a faulty program the programmer has left out the ampersands in the *scanf()* function calls.

```
/*      A program to read in two numbers
        and print their sum
*/
main()
{
        int     x,y;    /* places to store the numbers */
        printf("Enter x ");
        scanf("%d",x);
        printf("Enter y ");
        scanf("%d",y);
        printf("The sum of x and y was %d\n",x+y);
}
```

As was promised various varieties of chaos resulted. On all three systems the program compiled without any difficulty but attempts to run it had strange consequences. Both the IBM 6150 and the SUN Sparc Station produced the following dialogue.

```
$ sum2
Enter x 3
Memory fault - core dumped
$
```

ls -l command) revealed the following state of affairs on the SUN Sparc Station.

```
total 105
-rw-r--r--  1 jphb      8421808 Sep 23 14:47 core
-rwx------  1 jphb        24576 Sep 23 14:47 sum2
-rw-------  1 jphb          138 Sep 23 14:47 sum2.c
```

The effects on the IBM 6150 were similar but the file called *core* . was much smaller. The behaviour of Turbo C was quite different producing the following dialogue

```
Enter x 7
Enter y 6
The sum of x and y is 502
Null pointer assignment
```

To understand what has happened here it is necessary to remember that the values of the second and third parameters of *scanf()* are required to be the addresses of the locations that the converted numbers are stored in. In the C programming language the address of a memory location can always be obtained by proceeding its name with an ampersand. If the ampersand is omitted in the *scanf()* parameter list then the value passed to *scanf()* is not the address of "x" but the value that happened to be stored in "x", however the code of the *scanf()* library function knows nothing of this and assumes that the value it has received is an address and attempts to store the converted number in that memory location.

As was seen in an earlier example the initial value of "x" happened to be 0 on both the IBM 6150 and the SUN Sparc Station so, when *scanf()* had converted 3 to internal form it then attempted to store it in memory location 0. Unfortunately this memory location was clearly used for something else important and the over-writing of this important data has wrecked or, to use the jargon, **crashed** the program. The effects of such a crash depend very much on the particular compiler and host operating system.

On Unix systems the effect is that the "event" is detected or caught by the operating system which writes a program memory image to a file called, for historical reasons, *core* . There are various debugging tools which can be used to examine such *core* files in an attempt to find out what went wrong. Under MSDOS the effects are harder to predict depending on the particular compiler you are using, you may well have to re-boot the system.

The behaviour of the program compiled by the Turbo C compiler is particularly puzzling, after producing an incorrect answer it then produced a message that nobody other than a professional C programmer could reasonably be expected to understand. At least the Unix systems produced a message that a reasonable user could take to indicate

that something has gone wrong.

Missing `printf()` parameters

The final example shows the interesting consequences of a piece of gross carelessness on the part of the programmer who has left out the "x+y" and the preceding comma in the final call to *printf()* in the sum of two numbers program. (I've done this myself more than once !)

```
main()
{
        int     x,y;
        printf("Enter x ");
        scanf("%d",&x);
        printf("Enter y ");
        scanf("%d",&y);
        printf("The sum of x and y was %d\n");
}
```

When this program was compiled and run on a SUN Sparc Station the result was

```
Enter x 3
Enter y 4
The sum of x and y was -134218272
```

The effect on the IBM 6150 was similar only the result was quoted as 536872196 and Turbo C gave the result 4.

The compiler did not generate any error messages on any of the systems, this is because, as far as the compiler is concerned, there is nothing wrong with the final *printf()*. The compiler does not check that the parameters after the format string match up with the conversion specifications included within the format string, the difficulty only shows up when the function *printf()* is executed.

This failure of C compilers to check that the parameters supplied to library function calls are consistent with the layout specification is unfortunate but almost universal. In the next chapter we shall see further examples of the consequences of such errors, it is important to recognise them.

What is happening is that, once *printf()* has determined, from the format string, that it needs an *int* , it expects one to have been provided in a standard place, unfortunately the correct number has not been stored in that standard place.

Keyword List

Programming With Integers - Output of Numbers

Chapter chap2 section 5

We have already encountered the library function *printf()*. Notice how the \n was omitted from the strings on lines 7 and 9 of the sum of two numbers program so that the prompt and the user response appeared on the same line when the program was executed.

The parameters associated with *printf()* on line 11

```
printf("The sum of x and y was %d\n",x+y);
```

are more interesting. The string always associated with *printf()* is, in fact, a **format string** or **layout specification** that consists of a mixture of things to be copied directly to the output and output conversion specifications. The second parameter and any subsequent parameters of *printf()* will be expressions whose values will be converted from internal binary form to external form in accordance with the conversion specifications in the format string.

As with *scanf()* there are a wide variety of possible conversion specifications. A conversion specification within a format string is always introduced by a percent symbol (%) and terminated by one of several key letters identifying a conversion type. Within format strings you must write **%%** if you want an actual **%** symbol to appear in the output. In this chapter we will only use the "d" conversion type. This means convert from internal binary to external decimal.

The value of **x+y** is, as you might have expected, the sum of the values stored in locations x and y. "x+y" is, technically, an **expression.** Whenever an expression occurs in a program the effect is that the computer calculates the value of the expression and leaves the value in a standard place. Expressions of arbitrary complexity may be included as parameters of *printf()* as well as numbers and variables which are special simple cases of expressions.

Another program to read in two numbers.

Programming With Integers - Reading in Two Numbers

Chapter chap2 section 6

It is possible to use *scanf()* to read in two numbers from a single line of input. This requires that *scanf()'* s conversion specification be modified to specify two conversions and also that the addresses to store both numbers be supplied as parameters to the *scanf()* function. A sum of two numbers program that works this way is shown below.

```
/*      Another sum of two numbers
        program
*/
main()
{
        int     n1;
        int     n2;
        printf("Enter two numbers ");
        scanf("%d%d",&n1,&n2);
        printf("The sum is %d\n",n1+n2);
}
```

Several runs of the program are shown below. The dollar symbol (**$**) is the operating system (Unix) prompt. *sum2* is the name of the file holding the executable version of the program.

```
$ sum2
Enter two numbers 11 4
The sum is 15
$ sum2
Enter two numbers 21                    73
The sum is 94
$ sum2
Enter two numbers 44  -10
The sum is 34
$ sum2
Enter two numbers 23
11
The sum is 34
```

Although the conversion specification is **%d%d** suggesting that the numbers should be concatenated, *scanf()* will accept numbers with arbitrary intermediate space and

will, in fact, not accept concatenated numbers. This works because *scanf()*, when using d conversion rules, accepts an arbitrary amount of leading space. As far as *scanf()* is concerned a space is generated by the user hitting the SPACE bar, the TAB key or the RETURN key.

The final example is particularly interesting showing a slightly peculiar feature of the operation of *scanf()*. The user had hit the keys 2 and 3 and then hit the RETURN key. *scanf()*, quite properly, assumed that the first number the user had entered was 23 but the input specification required two decimal numbers so *scanf()* carried on reading input until it had obtained another number. The user could have hit RETURN many times before typing 11 and the program would have produced the same result.

You will also notice that *scanf()*, not surprisingly, is quite happy with negative numbers.

Initial values of variables

Programming With Integers - Control of Output Layout

Chapter chap2 section 8

By including extra codes between the % symbol introducing an output conversion specification and the conversion type letter it is possible to exercise considerable control over the appearance of the output produced by *printf()*.

When an internal value is converted to external form by *printf()* the set of printing positions occupied by the external form is known as an output **field .** The number of characters in such a field is the **width** of the field. The simple %d conversion specification specifies an output field whose width is always just sufficient to accommodate the required number of digits and, if required, a negative sign.

By incorporating a number between the "%" symbol and the "d" you can specify the output field width. The output field width will never be less than the width you have specified, leading spaces will be generated by *printf()* as necessary. If the number to be converted is too big to fit in the output field then *printf()* will increase the size of the output field so leading digits are not lost.

The specification of output field widths is useful and important if you are attempting to print tidy columns of figures or print in particular positions on pre-printed stationery. It is important to consider the maximum values you are going to have to print out and define the output field width appropriately.

The following version of the sum of two numbers program shows the use of output field width specification. >> and << are included in the layout specification so the output field limits are clearly visible in the output.

```
main()
{
        int     i,j;
        printf("Enter numbers ");
        scanf("%d%d",&i,&j);
        printf("The sum was >>%3d<<\n",i+j);
}
```

Here's the output produced by the program known as *sum3* . In the first run note the two leading spaces before the 7 and in the final run note that the output field width has expanded to 4.

```
$ sum3
```

```
Enter numbers 3 4
The sum was >>  7<<
$ sum3
Enter numbers 120 240
The sum was >>360<<
$ sum3
Enter numbers 999 999
The sum was >>1998<<
```

If the field width specification includes a leading zero then the leading spaces will become leading zeroes in the output. This may be useful for displaying times using the 24 hour clock or compass bearings. For example.

```
main()
{
        int     i,j;
        printf("Enter numbers ");
        scanf("%d%d",&i,&j);
        printf("The sum was >>%06d<<\n",i+j);
}
```

A run of this program is shown.

```
$ sum4
Enter numbers 123          512
The sum was >>000635<<
$ sum4
Enter numbers 500000 1
The sum was >>500001<<
```

There are various other options associated with *d* output conversion. For further information you should read the manual pages for the *printf()* function.

Input Errors

Programming With Integers - Input Errors

Chapter chap2 section 9

When you write a program that takes input from human beings, you ought to consider the possibility of the human being pressing the wrong key. At this stage in our study of the C programming language we do not know enough to include comprehensive checks and precautions in our programs and, anyway, such checks would completely obscure the simple purpose of the programs.

However it is worth knowing what happens when the user does make a typing error. The behaviour of *scanf()* is very simple and fairly well-defined, in the presence of invalid input it simply gives up its attempt to convert from external to internal form. The following sample dialogue, using the basic sum of two numbers program demonstrates what happens. The program was called *sum* and was run on a SUN Sparc Station. $ is the Unix operating system prompt.

```
$ sum
Enter x 3
Enter y two
The sum of x and y was 35
$ sum
Enter x one
Enter y The sum of x and y was 32
$ sum
Enter x 1.25
Enter y The sum of x and y was 33
```

In the first example, the first number (x) was read in satisfactorially but *scanf()* was unable to process the second input value so gave up without altering the value of the variable "y", which seems to have been 32. See earlier example.

The second example is rather more interesting. When *scanf()* failed to convert the input *one* from external decimal to internal binary it left the characters "o", "n" and "e" waiting to be processed. The second call to *scanf()*, after the second input prompt, found these characters waiting to be processed and also failed to convert them, it never got round to getting any more input from the user. This failure to get more user input is the reason that the program result appears on the same line as the second input prompt, the user never got a chance to type anything, in particular the RETURN at the end of his line of input.

In computer jargon we might say that *scanf()* doesn't **flush the input buffer.**

The third example shows the same sort of behaviour as the second example. When *scanf()* attempts a *%d* conversion it does not expect to find a decimal point in the input and gives up as soon as it finds the decimal point.

Input Layout

Programming With Integers - Input Layout

Chapter chap2 section 10

The format specification strings used by a *printf()* are very similar to those used by *scanf()*, however there are some differences.

A field width specfication in an input format defines the maximum number of characters to be examined after leading spaces have been skipped. This may be useful for reading fixed format numerical items, such as the output from other programs but suffers from the disadvantage that any unread characters are left in the input buffer. The following example shows the effects of input field width specification.

```
main()
{
        int     x,y;
        printf("Enter two 3 digit numbers ");
        scanf("%3d%3d",&x,&y);
        printf("The numbers were %d and %d\n",x,y);
}
```

And some typical runs of the program called inf.

```
bash$ inf
Enter two 3 digit numbers 123456
The numbers were 123 and 456
bash$ inf
Enter two 3 digit numbers 123       456
The numbers were 123 and 456
bash$ inf
Enter two 3 digit numbers 1234 567
The numbers were 123 and 4
```

In the final run note the effect of the character "4" left in the input buffer.

You can include further characters in the *scanf()* format string if they are not part of a conversion specification. *scanf()* will then expect to find the relevant characters in the input. The following program expects two numbers separated by a comma.

```
/*      A program to read in two numbers
        and print their sum
```

```
*/
main()
{
        int     x,y;    /* places to store the numbers */
        printf("Type two numbers in the form n1, n2 ");
        scanf("%d,%d",&x,&y);
        printf("The sum of x and y was %d\n",x+y);
}
```

Test runs of the program, known as sum9, are shown below.

```
$ sum9
Type two numbers in the form n1, n2 1, 3
The sum of x and y was 4
$ sum9
Type two numbers in the form n1, n2 100,200
The sum of x and y was 300
$ sum9
Type two numbers in the form n1, n2 1 , 2
The sum of x and y was 33
$ sum9
Type two numbers in the form n1, n2 200 400
The sum of x and y was 232
```

In the final two runs a spurious space was typed before the expected comma and, in the final run, the comma was left out altogether. In both cases *scanf()* gave up as soon as it found something other than the specified comma after the first number. There are more complex ways of specifying optional input characters in *scanf()* format specifications, they will be discussed later.

Input Errors

Addresses, Pointers, Arrays and Strings - Strings

Chapter chap6 section 7

It is possible to have aggregates of all sorts of things including aggregates, pointers to anything including functions and user defined data types. In this section we will look particularly at aggregates of characters which are so very widely used that they have some special properties and there are special library functions for handling such aggregates. Under certain circumstances an aggregate of characters may be known as a **string .**

A string is a collection of characters terminated by a character all of whose bits are zero, i.e. a **NUL** (not a NULL). It is entirely possible to initialise an aggregate to a string by writing code such as

```
char msg[]={'H','e','l','l','o','\0'};
```

but this would never be done in practice as the C language allows the alternative notation

```
char    msg[] = "Hello";
```

which is always used in practice. The rules for writing string constants are exactly the same as those that were discussed much earlier in the course when the use of *printf()* was introduced. It should be noted that the size of the aggregate "msg" is 6 bytes, 5 for the letters and 1 for the terminating NUL.

There is an interesting and important difference between the following declarations which may seem equivalent.

```
char    msg[] = "Hello";
```

and

```
char    *msg = "Hello";
```

The first declaration reserves an aggregate of 6 characters in the memory space currently being allocated, the data space is initialised to the relevant set of character values. This is only initialisation, the values and hence the text of the string can be altered.

The second declaration reserves enough space to hold a pointer to a character, this pointer is initialised to point to a "secret" system place within your program where

the actual character string "Hello" is stored. This is likely to be the same general place that is used for storing layout specification strings used by the *printf()* and *scanf()* functions. The value stored in "msg", which is only a pointer to a character, can be altered so it points to a different character. However, and this is encouraged by the ANSI standard, the actual string constant should not be alterable, although this restriction is seldom enforced.

The conversion type "s" may be used for the input and output of strings using *scanf()* and *printf()*. Width and precision specifications may be used with the %s conversion, the width specifies the minimum output field width, if the string is shorter then **space padding** is generated, the precision specifies the maximum number of characters to display. If the string is too long, it is truncated. A negative width implies **left justification** of short strings rather than the default **right justification.**

The following program illustrates the use of the "%s" conversion.

```
main()
{
        char    *msg="Hello, World";
        printf(">>%s<<\n",msg);
        printf(">>%20s<<\n",msg);
        printf(">>%-20s<<\n",msg);
        printf(">>%.4s<<\n",msg);
        printf(">>%-20.4s<<\n",msg);
        printf(">>%20.4s<<\n",msg);
}
```

producing the output

```
>>Hello, World<<
>>         Hello, World<<
>>Hello, World         <<
>>Hell<<
>>Hell                 <<
>>                 Hell<<
```

The ">>" and "<<" symbols were included in this program so that the limits of the output fields were clearly visible in the output.

- String input using s conversion
- String input using scanset conversion
- String input using c conversion
- String input using the gets() function

Programming With Integers - C and C++

Chapter chap2 section 13 In the C++ programming language the restriction that all declarations should appear before all executable statements is relaxed, this is why the Turbo C compiler (which is really a C++ compiler) happily accepted the version of the sum of two numbers program with the declaration of y after the input of x.

As well as the use of the library functions *scanf()* and *printf()* for input and output the C++ language supports an alternative form of input and output known as an **IO stream.** This uses the syntax

```
cout << value
```

for the output of a single value and the syntax

```
cin >> name-of-variable
```

for the input of a value to a variable. Note that, unlike *scanf()*, the IO stream input requires just the name of the variable without any preceding "&". The symbol ">>" is read as "gets from" and "<<" is read as "puts to". Before any IO stream operations are performed you must put the header line

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

in the program. The sum of two numbers program can be re-written in C++ in this form

```
//      Sum of two numbers program in C++

#include        <iostream.h>
main()
{
        int     x,y;
        cout << "Enter first number ";
        cin >> x;
        cout << "Enter second number ";
        cin >> y;
        cout << "The sum of the two numbers was ";
        cout << x+y;
        cout << "\n";
}
```

Note that there is no need to provide conversion information, the C++ compiler

inspects the values and variables associated with IO stream input and output and arranges for the correct conversions. `cout` itself is a rather special sort of variable, it is actually of type IO-stream. More interestingly the expression

```
cout << x+y
```

is also of type IO-stream so multiple outputs can be written in this fashion

```
cout << "The sum of x and y was " << x+y << "\n"
```

To provide output width control there are several special functions associated with cout. These include *cout.width()* and cout.fill() . The function *cout.width()* sets the output field width for the next output operation only. It takes a single integer valued parameter. The function *cout.fill()* specifies the character to be used to fill the leading part of the next output field. It takes a single character constant as parameter. This consists of an actual constant enclosed in single quotes. You may be surprised that the names of these functions include a single dot, this does not mean that C++ allows dots within variable names, this means that the width and fill functions are associated with *cout* . The following two programs produce identical output. The first program uses C style output layout control, the second uses C++ style control.

```
main()
{
        int     x=25,y=50;
        printf("x = %5d, y = %05d\n",x,y);
}
```

The C++ version follows.

```
#include        <iostream.h>
main()
{
        int     x=25,y=50;
        cout << "x = ";
        cout.width(5);
        cout.fill(' ');
        cout << x << ", y = ";
        cout.width(5);
        cout.fill('0');
        cout << y << "\n";
}
```

Both programs produced the following single line of output

```
x =    25, y = 00050
```

You will, almost certainly, think that the C version is simpler, it certainly involves a lot less coding. The advantages of the C++ approach will only become clear in more advanced and complex programs.

Exercises

Programming With Integers - Exercises

Chapter chap2 section 14

1. Write a C program to print out the sum of two numbers in the form

   ```
   The sum of 3 and 4 was 7
   ```

   I.e. display the numbers as well as their sum.
2. Modify the program of exercise 1 to obtain the two numbers interactively from the user.
3. By specifying a plus-sign in the *scanf()* format write a program that will read in simple sums such as

   ```
   12+34
   ```

   and print the result.
4. Write a program that declares at least three variables and initialises them to fairly large values. By suitable use of the *printf()* layout print them out in a neat column.
5. If the symbol "-" appears immediately after the "%" in a d conversion specification for use by printf() then the displayed number is left-justified within the output field rather than the usual right justification. Write a program to demonstrate this, can you think of any use for this facility ?
6. If the symbol "+" appears immediately after the "%" in a d conversion specification for use by printf() then the number displayed is always preceded by a sign. Write a program to demonstrate this, can you think of any use for this facility ?

Arithmetic - Introduction

Chapter chap3 section 1

In this chapter we will explore the integer arithmetic capabilities of the C programming language. We will also see another way of putting numbers in memory locations. We will still only be concerned with integer arithmetic.

See also

- Expressions
- Expression Evaluation
- Operator Precedence
- Operator Types
- Assignment Operators
- Increment and Decrement Operators
- Summary
- Program Layout
- Function parameter evaluation order
- Exercises

Data Types

Arithmetic - Expressions

Chapter chap3 section 2

In the previous chapter we saw how we could the computer to add up two numbers by writing the expression

```
x+y
```

The appearance of such an expression causes the computer to calculate the value of the expression when executing the statement that includes the expression. It also seems obvious that one can add up three numbers using an expression such as

```
x+y+z
```

In both these expressions are both variables such as "x", "y" and "z" and the symbol "+". The symbol "+" means add up the two surrounding numbers. It is one of several elementary **operators .**

```
+       meaning add
-       meaning subtract
*       meaning multiply
/       meaning divide
```

The reason that "*" is used for multiplication rather than an "X" or plain juxtaposition of variables as is done in ordinary algebraic notation, is that an "X" could easily be confused with the name of a variable and juxtapostion of two variable names looks like a third variable name.

Strictly the operators shown above are **binary operators** which means that they operate on two numbers. The values associated with an operator may be

1. actual numbers called constants
2. the names of variables, in which case the value of the variable is intended
3. expressions, in which case the value of the expression is intended

The following are all valid expressions assuming x,y and z are variables

1. x+23
2. 4500
3. x+y*11
4. z

The following program shows some simple examples of the use of the operators shown above.

```
main()
{
```

```
        int     x = 3;
        int     y = 2;
        int     z = 6;
        printf("This is a constant -- %d\n",213);
        printf("The value of %d-%d is %d\n",x,y,x-y);
        printf("The value of %d*%d is %d\n",x,y,x*y);
        printf("The value of %d/%d is %d\n",z,y,z/y);
}
```

It produced the following output

```
This is a constant -- 213
The value of 3-2 is 1
The value of 3*2 is 6
The value of 6/2 is 3
```

The value of any expression involving integers will itself be an integer. This is straightforward for addition, subtraction and multiplication but division requires further consideration. Division of two integers results in an integer valued quotient and a remainder that plays no further part in the proceedings.

Alternatively you can regard division as producing an answer including a fractional part and the resultant number being truncated by discarding the fractional part. The ANSI standard talks about **truncation** although all computers actually do integer division and discard the remainder.

So when 20/7 is calculated, the result can be thought of either as 2 remainder 6 or as 2.85714..... In the first case discarding the remainder gives the result 2 and in the second case truncating the result also gives the result 2.

The ANSI standard says that when two positive integers are divided, the result is **truncated towards zero.** If the division involves negative numbers then the result may be truncated up or down.

The following program shows the values of some actual quotients

```
main()
{
        int     x = 5;
        int     y = 8;
        int     p = -3;
        int     q = -5;
        printf("The value of %2d divided by %2d is %2d\n",
                        y,x,y/x);
        printf("The value of %2d divided by %2d is %2d\n",
                        y,p,y/p);
```

```
        printf("The value of %2d divided by %2d is %2d\n",
                        p,q,p/q);
        printf("The value of %2d divided by %2d is %2d\n",
                        q,x,q/x);
}
```

When compiled and run the program produced the following output.

```
The value of  8 divided by  5 is  1
The value of  8 divided by -3 is -2
The value of -3 divided by -5 is  0
The value of -5 divided by  5 is -1
```

Although the ANSI standard allows computers to give either -2 or -3 as the value of 8/-3, all three systems tested, when writing these notes, gave the result -2.

The effects of attempting to divide by zero are officially **undefined .** The ANSI standard does not require compiler writers to do anything special, so anything might happen. Of course we tried this by changing the value of x to zero in the previous program. Turbo C spotted what was going on and displayed the message

```
Divide error
```

The Unix systems were slightly less informative producing the following messages

```
Arithmetic exception (core dumped)
Breakpoint - core dumped
```

on the SUN Sparc station and IBM 6150 respectively. Both Unix systems produced the *core* file described in the previous chapter.

Whether it is reasonable to expect the computer to check every division operation by examining the divisor before actually executing the division is a debatable point, unless there is special hardware for detecting the condition it can slow programs down.

Evaluation of Expressions

Arithmetic - Evaluation of Expressions

Chapter chap3 section 3

The expression

```
x+y+z
```

appeared earlier. This expression looks innocent but illustrates an important point. Since the operator "+" is a binary operator this expression has to be evaluated in 2 steps. These could be either

```
Evaluate the expression x+y
Add z to the result of the previous evaluation
```

or

```
Evaluate the expression y+z
Add x to the result of the previous evaluation
```

The ANSI standard allows computers to do it either way and it clearly makes no difference to the result. However for the expression

```
x-y+z
```

it clearly does make a difference as the following quick calculation shows.

First assume x has the value 1, y has the value 2 and z has the value 3. Evaluating x-y and then adding z to the result gives the value +2 whereas evaluating y+z and then subtracting the result from x gives the value -4.

The ANSI standard is quite definite about what should happen. It states that the "+" and "-" operators **group** or **associate** left-to-right. This means that an expression such as x-y+z is evaluated from left to right corresponding to the first alternative described above. If we weren't certain what this meant, a quick program confirms that our compiler writer interpreted it the same way as we did.

```
main()
{
        int     x=1,y=2,z=3;
        printf("Value of \"%d-%d+%d\" is %d\n",
                x,y,z,x-y+z);
        printf("value of \"%d+%d-%d\" is %d\n",
                x,y,z,x+y-z);
```

```
}
```

Giving the following output

```
Value of "1-2+3" is 2
value of "1+2-3" is 0
```

Suppose now that we did want to calculate y+z first and then subtract the result from x. This problem can be solved by using an extra piece of notation and writing the expression as

```
x-(y+z)
```

The **parentheses** (jargon for round brackets) enclose an expression and expressions enclosed within parentheses are evaluated before other expressions according to the standard. Further expressions contained within parentheses can be enclosed within parentheses to an arbitrary depth, this is called **expression nesting.** Expressions enclosed within parentheses are sometimes called **sub-expressions** but this isn't really very helpful as they are proper expressions in their own right. Again a simple programming example is sufficient to convince ourselves that parentheses work as advertised.

```
main()
{
        int     x=1,y=2,z=3;
        printf("Value of \"%d-(%d+%d)\" is %d\n",
                x,y,z,x-(y+z));
}
```

Resulting in the output

```
Value of "1-(2+3)" is -4
```

Notice how we used escaped double quotes within the *printf()* layout specification to ensure that actual double quotes appeared in the output.

The ANSI standard rather grandly calls parentheses **primary grouping operators.**

There is an important difference between the "+" and the "-" operators. The jargon says that "+" operator is a **commutative** operator whereas - is not. The adjective commutative applied to a binary operator means that it doesn't matter which order it takes its operators in. In other words

```
value1 operator value2
```

has exactly the same value as

```
value2 operator value1
```

This is clearly true for "+" and equally clearly false for "-".

The ANSI standard says that expressions involving only one of the commutative and associative operators can be evaluated in any order. An **associative** operator is an operator which has the property

```
value1 operator ( value2 operator value3 )
```

has the same value as

```
( value1 operator value2 ) operator value3
```

allowing one to write, unambiguously,

```
value1 operator value2 operator value3
```

The only associative and commutative operators we have met so far are "+" and "*".

You may wonder if it could possibly matter how x+y+z is calculated. The answer is that it might matter if an arithmetic overflow occurred during the calculation, the problems of arithmetic overflow will be discussed, briefly, later in this chapter.

Operator Precedence

Arithmetic - Operator Precedence

Chapter chap3 section 4

The expressions

```
x+y*z and x*y+z
```

suffer from the same potential ambiguity as

```
x+y-z and x-y+z
```

however the problem is handled in a different way. The normal mathematical expectation is that multiplication is performed before addition. There are various ways of saying this, we could say that the "*" operator **binds more tightly** or we could say, and will say, that the "*" operator has a higher **precedence** than the "+" operator. Again a simple programming example confirms this point.

```
main()
{
        int     x=2,y=7,z=5;
        printf("The value of \"%d*%d+%d\" is %d\n",
                        x,y,z,x*y+z);
        printf("The value of \"%d+%d*%d\" is %d\n",
                        z,x,y,z+x*y);
}
```

resulting in the output

```
The value of "2*7+5" is 19
The value of "5+2*7" is 19
```

Of course, we could use parentheses if we actually wanted addition performed before multiplication as the following example shows.

```
main()
{
        int     x=2,y=7,z=5;
        printf("The value of \"%d*(%d+%d)\" is %d\n",
                        x,y,z,x*(y+z));
        printf("The value of \"(%d+%d)*%d\" is %d\n",
                        z,x,y,(z+x)*y);
}
```

resulting in the output

```
The value of "2*(7+5)" is 24
The value of "(5+2)*7" is 49
```

Operator Types

Arithmetic - Types of Operators

Chapter chap3 section 5

The operators "+" and "-" are called **additive** operators. The operators "*" and "/" are called **multiplicative** operators. There is one further multiplicative operator, this is "%" which is a **remaindering** or **modulo** operator. The value of

```
value1 % value2
```

is the remainder when value1 is divided by value2. It is guaranteed by the ANSI standard that the value of an expression such as

```
(a/b)*b + a%b
```

is equal to a. A programming example is in order.

```
main()
{
        int     x=20,y=6;
        printf("The quotient of %d divided by %d is %d\n",
                        x,y,x/y);
        printf("The remainder of %d divided by %d is %d\n",
                        x,y,x%y);
}
```

resulting in the output

```
The quotient of 20 divided by 6 is 3
The remainder of 20 divided by 6 is 2
```

The behaviour of both the "/" and "%" operators is, officially, **undefined,** if the second operator is zero.

All the multiplicative operators associate or group **left-to-right** and have the same precedence. This is demonstrated by considering the output of the following program.

```
main()
{
        int     x=20;
        int     y=3;
        int     z=5;
        printf("Value of \"%d/%d*%d\" is %d\n",
                x,y,z,x/y*z);
        printf("Value of \"%d/%d/%d\" is %d\n",
                x,y,z,x/y/z);
        printf("Value of \"%d%%%d*%d\" is %d\n",
```

```
                x,y,z,x%y*z);
}
```

which produced the output

```
Value of "20/3*5" is 30
Value of "20/3/5" is 1
Value of "20%3*5" is 10
```

In the first line of output "20/3" is first calculated giving 6 by the rules of integer division, the result is multiplied by 5. In the second line of output the value of "20/3" is calculated and the result is divided by 5 giving the value 1. In the final line of output the value of "20%3" is calculated, its value is 2 which is then multiplied by 5.

The C programming language has a particularly large set of operators compared with many other programming languages, it is important to keep track of the precedence and grouping of operators when writing complicated expressions.

Assignment Operators

Arithmetic - Assignment Operators

Chapter chap3 section 6

Another useful set of elementary operators are the **assignment** operators. The simplest of these is "=". Unlike the operators we have seen so far the choice of first operand is restricted to the name of a variable. A typical example would be

```
x=7
```

The value of this expression is 7 but as a **side effect** the value 7 is stored in the variable x. The side effect is often more important than the value of the expression. Again a programming example is in order.

```
main()
{
        int     x=4;
        printf("The value of x is %d\n",x);
        printf("The value of \"x=8\" is %d\n",x=8);
        printf("The value of x is %d\n",x);
}
```

resulting in the output

```
The value of x is 4
The value of "x=8" is 8
The value of x is 8
```

Another example underlines the point

```
main()
{
        int     x=2,y=3;
        printf("x = %d, y = %d\n",x,y);
        printf("Value of \"(y+(x=8))\" is %d\n",
                        y+(x=8));
        printf("x = %d, y = %d\n",x,y);
}
```

producing the output

```
x = 2, y = 3
```

```
Value of "(y+(x=8))" is 11
x = 8, y = 3
```

These examples might look a little odd to those familiar with other programming languages who might be happier with the following program.

```
main()
{
        int     x=4;
        printf("The value of x is %d\n",x);
        x=8;
        printf("The value of x is %d\n",x);
}
```

which produced exactly the same first and last lines of output. The line between the two *printf()* function calls is a single statement consisting of the expression

```
x=8
```

The final semi-colon is necessary to ensure that this line is a statement. There are several good reasons, mainly concerned with keeping compiler design simple, for treating "=" as an operator rather than treating assignment as a special sort of statement, the technique adopted by many other programming languages.

Technically the thing that appears at the left hand side of an assignment expression must be an **lvalue ,** this means the address of a memory location in which a value can be stored. In the simple assignment expression

```
x=x+1
```

the "x" on the left hand side represents the address of a memory location whereas the "x" on the right hand side of the "=" symbol represents the value stored in the location with the address "x". This confusion is common to nearly all programming languages.

The expression

```
x=x+1
```

repays further examination. This expression includes the two operators "=" and "+" and it is proper to ask whether it means

```
evaluate the expression x=x
add 1 to the result of the first evaluation
```

or

```
evaluate the expression x+1
```

```
assign the result to x
```

We have already seen that problems of this nature can be resolved by considering the relative precedence of the operators. The ANSI standard states that all the assignment operators have a lower priority or precedence than any other operator except the rather bizarre comma operator which we won't meet for some time. This means that the second interpretation of "x=x+1" is the correct interpretation.

The value of an assignment expression may, of course, be assigned to a variable. This leads to constructs such as

```
x = y = z = p+3
```

Unlike the arithmetic operators discussed earlier assignment operators associate or group **right to left.** This means that, in the above example, the first step is the evaluation of the expression p+3 followed by assignment of its value to z. The value of the expression

```
z = p+3
```

is then assigned to y and so on.

There are several more assignment operators that provide a short-hand way of writing

```
variable = variable operator expression
```

The following are based on the arithmetic operators we have discussed so far

```
+= -= *= /= %=
```

There are others which will be discussed later in the course. The economy of writing

```
count_so_far += input_value
```

compared with

```
count_so_far = count_so_far + input_value
```

is obvious and these assignment operators are widely used by C programmers. In all cases the value of the assignment expression is the value assigned to variable whose name is the first operand of the expression. The following program exercises the assignment operators. Note the "%%" in the penultimate output *printf()* format.

```
main()
{
        int     x=4;
        int     n1=2,n2=1,n3=4,n4=5,n5=3;
        printf("Initial value of x is %d\n",x);
```

```
        printf("Value of x += %d is %d\n",n1,x+=n1);
        printf("Value of x -= %d is %d\n",n2,x-=n2);
        printf("Value of x *= %d is %d\n",n3,x*=n3);
        printf("Value of x /= %d is %d\n",n4,x/=n4);
        printf("Value of x %%= %d is %d\n",n5,x%=n5);
        printf("Final value of x is %d\n",x);
}
```

The output is shown below. It is an interesting exercise to try and follow through the execution of the program and predict the values that are printed out.

```
Initial value of x is 4
Value of x += 2 is 6
Value of x -= 1 is 5
Value of x *= 4 is 20
Value of x /= 5 is 4
Value of x %= 3 is 1
Final value of x is 1
```

If you peeked, try changing the initial values, predicting the results and then try it on a computer.

If you couldn't figure out what was going on in the previous example let's go through it step by step.

When the second *printf()* is executed it is necessary to calculate the value of "x+=n1", since n1 has the value 2 and x has the value 4 the value of "x+n1" is 6 and remembering that

```
x+=n1
```

is equivalent to

```
x = x + n1
```

it is now clear that x takes the value 6 and that this is also the value of the assignment expression.

On the next line the value of "x-=n2" is required. As n2 has the value 1 and x was changed to 6 on the previous line the value of this is 5 which value is assigned to x.

On the fourth *printf()* line the value of x*=n3 is required, since n3 is 4 and x is now 5 this is 20. On the fifth line is value of "x/=n4" is the same as "x=x/n4" which is 4.

Finally "x%=n5" is calculated, remembering that x is now 4 and n5 is 3 and that the expression is equivalent to "x=x%n5", it is easy to see that the value is 1, the remainder when 4 is divided by 3.

The ++ and -- operators

Arithmetic - The ++ and -- Operators

Chapter chap3 section 7

The final operators considered in this chapter are the "++" and "--" operators. These provide a handy way of incrementing or decrementing the value of a variable, a very common programming requirement. Each operator comes in two flavours known as **prefix** and **postfix.** The difference concerns whether the value of the expression is the new value of the variable or the old value of the variable. If the operator precedes the name of the variable then the value of the expression is the new value, otherwise it is the original value. Note that these operators can only be associated with variables, they cannot be associated with constants or expressions. The jingles **use and increment** or **increment and use** may be helpful here and, of course, another programming example is called for.

```
main()
{
        int     x=3;
        int     y=4;
        printf("Value of x++ is %d\n",x++);
        printf("Value of ++y is %d\n",++y);
        printf("Value of y++ is %d\n",y++);
        printf("Value of ++x is %d\n",++x);
        printf("Final value of x is %d, y is %d\n",x,y);
}
```

Producing the output

```
Value of x++ is 3
Value of ++y is 5
Value of y++ is 5
Value of ++x is 5
Final value of x is 5, y is 6
```

Again careful consideration of the output will be rewarded by greater understanding of what is actually going on.

The first statement displays the value of x++. This is the postfix flavour meaning use-and-increment so the value of the expression x++ is the original value (3) but as a side effect of evaluating x++ the value of x is changed to 4, however this will not be apparent until the next use of x.

The next statement displays the value of ++y. This is the prefix or increment-and-use flavour so the value of the expression is the value after incrementation of the original value. The net effect is that the value of y is changed to 5.

The next statement works in exactly the same way as the first statement showing the value of y before performing the incrementation implied by y++. During the execution of the previous statement the value of y had been incremented to 5, it, of course, retains this value when the execution of the next statement starts. After the execution the value of y will be 6.

The final statement takes the current value of x, which had been changed to 4 during the execution of the first statement, and increments it further. The postfix "++" means increment-and-use so the value displayed is 5. The "--" operator is totally analagous to the "++" operator only it decrements rather than increments.

Fairly obviously, the "++" and "--" operators can only be applied to variables (or more properly lvalues).

The "++" and "--" operators are very widely used in programming loops and other iterative constructs which we will be using shortly.

The so-called **side-effects** associated with both the increment and decrement operators and the assignment operators are often confusing to programmers who are familiar with other languages or who have attended courses on software engineering where such things are deprecated as confusing and mysterious. The widespread use of and reliance on side-effects is, however, very typical of normal C programming. It can, in the wrong hands, give rise to very obscure code, in the right hands in can result in remarkably terse programs that will be small and efficient when compiled.

"++" and "--" are examples of **unary** operators which means that they are associated with a single operand rather than two operands. Other examples are "+" and "-" used in contexts such as

```
x = -y
```

and

```
y = +z
```

The unary "+" operator doesn't actually do anything useful, it is there for completeness.

Summary of Arithmetic Operators

Arithmetic - Summary of Arithmetic Operators

Chapter chap3 section 8

We can now summarise the operators we have seen so far. These are, in order of decreasing precedence

Symbol	Type
( )	Primary Grouping
- +	Unary
-- ++	Unary
* / %	Binary Multiplicative
+ -	Binary Additive
= += -= *= /= %=	Assignment Operators

Program Layout

Arithmetic - Program Layout

Chapter chap3 section 9

You may have noticed that, in some of the programs listed earlier, the statements involving *printf()* spread over two lines. For example.

```
printf("The value of \"%d+%d*%d\" is %d\n",
                z,x,y,z+x*y);
```

This is possible because the C language is a **free format** language, which means that statements and expressions may be written in any way to improve readability. This freedom also applies to the list of parameters for a function such as *printf()* and was used earlier to avoid long lines spoiling the appearance of the printed text.

In general, C compilers regard any sequence of TAB characters, space characters and newline characters as equivalent to a single space. These are collectively known as **white-space** characters. Sequences of characters enclosed within the character pairs /* and */ are also equivalent to a single space. None of this applies to characters enclosed within double quotes.

It is also not permissible to write things such as

```
p r i n t f
```

rather than

```
printf
```

The reason is both interesting and important. As part of the compilation process the source program you have typed in is converted into a sequence of items known as **tokens** which might be names of variables, names of functions, keywords, operators or constants. The rules for identifying tokens are rather complicated to state but the effect is fairly obvious. The following example shows the "hello world" program listed one token per line

```
main
(
)
{
printf
(
"hello, world\n"
)
;
```

```
}
```

This was given as an example of how not to lay out programs in chapter 1. Breaking the name of a variable into separate pieces causes the compiler to see each piece as a separate token. Some equally bizarre looking examples of code can be understood by considering tokenisation. For example

```
x-=-3
```

would be split into the token stream "x", "-=", "-" and "3", there being no other valid way of splitting it into tokens. It would have been better if the programmer had written

```
x -= -3
```

Oddities such as

```
x+++++y
```

can probably only be validly tokenised as

```
x ++ + ++ y
```

but it is much better to write the second form rather than the first.

Order of Evaluation of Functional Parameters

Arithmetic - Order of Evaluation of Functional Parameters

Chapter chap3 section 10

A final important point in this chapter is to consider the output produced by the following program

```
main()
{
        int     x=5;
        printf("Values are %d and %d\n",x++,++x);
}
```

Before revealing the results let's see if we can work out what the output of the program will be. You might typically argue along the following lines.

We need to consider the values passed to the *printf()* function. The first of these is the value of the expression "x++". This is the use-and-increment (prefix) flavour of "++" so the value of the expression is 5 and as a side effect of evaluating the expression the value of x is increased to 6. The value of the expression "++x" is now calculated, this is the "increment-and-use" flavour of "++" so the value of the expression is clearly 7. Thus the expected output is

```
Values are 5 and 7
```

and compiling and running the program on the SUN Sparc Station produced exactly the expected output. What problem could there possibly be with this simple program ? Trying the same program using the Turbo C compiler resulted in the output

```
Values are 6 and 6
```

This is rather surprising but what has happened is actually quite easy to understand, if rather inconvenient. The C programming language standard rules quite specifically allow the parameters to be passed to a function to be evaluated in any convenient order. The SUN Sparc station compiler worked left to right, which seems more natural, whereas the Turbo C Compiler worked right to left which may be more efficient in some circumstances.

This must be remembered when writing programs that are to be compiled on many different machines. Some compilers provide flags or options to allow the user to control the order of functional parameter evaluation.

A similar difficulty arises when considering the output of a program such as

```
main()
{
        int     x = 4;
        printf("Result = %d\n",x++ + x);
}
```

Since the standard allows expressions involving commutative associative operators such as "+" to be evaluated in any order a moment's thought shows that the value printed out would be 8 for right-to-left evaluation and 9 for left-to-right evaluation. On the SPARC system the output was

```
Result = 8
```

whereas the Turbo C compiler gave the result

```
Result = 9
```

Strictly the behaviour of the program is **undefined,** which means that anything might happen.

Exercises

Arithmetic - Exercises

Chapter chap3 section 11

1. Write a program to read in 3 integers and print their sum.
2. Write a program to read in two integers and display both the quotient and the remainder when they are divided. How does your computer treat the results of division by a negative number? What happens when you try and divide by zero?
3. What error message does your compiler give when you write

   ```
   x+1=x
   ```

   in a program?
4. Write a program to test the behaviour of your computer system when it processes

   ```
   printf("%d %d\n",x++,++x);
   ```
5. How do you think the computer will evaluate

   ```
   x+=x++
   ```

   Try and calculate the value for a particular value of x, then write a program to see what really happens. Do you think any other computer might come up with a different result?
6. Does

   ```
   x-=x
   ```

   make any sense? What do you think it means?
7. How would

   ```
   x%y%z
   ```

   be evaluated? Write a program to find out what happens.
8. Write a program to read in two integers and display one as a percentage of the other. Typically your output should look like

   ```
   20 is 50% of 40
   ```

   assuming that the numbers read in where 20 and 40

Arithmetic and Data Types - Floating Point Numbers

Chapter chap4 section 2

The **floating point** data type provides the means to store and manipulate numbers with fractional parts and a very large range of sizes. The ANSI standard describes three types of floating point storage known as **float , double** and **long double .** Different C compilers running on different computer systems are allowed by the ANSI standard to implement the various types of floating point numbers in different ways but certain minimum standards must be met. The basic characteristics are summarised in the following table.

Type	Maximum Value	Significant Digits	Context
float	1.0×10^{37}	6	ANSI specified minimum acceptable
double	1.0×10^{37}	10	
long double	1.0×10^{37}	10	
float	3.403×10^{38}	6	Actual characteristics on SUN Sparc station
double	1.798×10^{308}	15	
long double	1.798×10^{308}	15	
float	3.4×10^{38}	7	Actual characteristics on a PC using the Turbo compiler
double	1.7×10^{308}	15	
long double	1.1×10^{4932}	19	

For example a *double* variable on the SUN Sparc Station ANSI compiler will store numbers up to 1.798×10^{308} (that's 308 zeroes) to an accuracy of about 15 decimal places.

It will be noted that *double* and *long double* are the same on the SUN Sparc station, this is clearly allowed by the standard which equally clearly allows *long double* to support a larger maximum value and more significant digits if the compiler writer so wishes and the underlying hardware can manipulate such numbers. It should be noted that most C programmers tend to use the *double* floating point data type rather than *float* or *long double* for largely historical reasons.

Memory locations of any of the floating point data types can be declared by giving the type name and a list of identifiers. Floating point locations can be initialised as part of the declaration. There are no special rules for naming floating point data locations. Declarations for memory locations of different data types must be separate declarations but as many memory locations of a single type as required can

be declared in a single declaration. The names of memory locations of all the various types must be distinct.

The values of floating point numbers can be written using the conventional notation involving a decimal point. A notation such as 3.7 implies a constant of type double. In the unlikely circumstances that a constant of a particular type is needed then one of the letters "f" or "F" for a float constant or "l" or "L" for a long double constant can be written as the last character of the constant. There are other notations which will be described later.

Floating point numbers can be converted to external form using the *printf()* function with the following conversion specifications

f	float
lf	double
Lf	long double

Between the % that introduces the conversion specification in the *printf()* format string and the f , lf or Lf that terminates the conversion there will usually be a **field specification** of the form

```
w.d
```

where w specifies the overall **field width** and d is the **precision** specification which tells *printf()* how many digits to print after the decimal point. If the precision is not specified then a default of 6 is used. For the use of precision specifications with integers see later. The technique for adjusting the field width and precision while the program is running is also discussed later.

The following program illustrates the declaration, initialisation and output of floating point numbers.

```
main()
{
        double  x=213.5671435;
        double  y=0.000007234;
        printf("x = %10.5lf\n",x);
        printf("y = %10.5lf\n",y);
        printf("x = %5.2lf\n",x);
        printf("y = %10lf\n",y);
        printf("x = %3.1lf\n",x);
}
```

It produced the following output.

```
x =  213.56714
y =    0.00001
x = 213.57
y =   0.000007
x = 213.6
```

There are several interesting points to notice here. On the second, third and fifth lines notice that the output has been rounded. On the fourth line note the default precision and on the fifth line note the output field width has expanded to accommodate the actual data.

It is, of course, quite admissible to omit the field width entirely and just quote a precision with preceding period, then the required number of digits will be displayed and the field width will expand suitably.

All the arithmetic operations described in the previous chapter with the exception of those involving the modulo operator ("%") may be applied to floating point numbers. The division operator applied to floating point numbers yields a floating point quotient, there are no complications with remainders or truncation. The effect of applying any of the operators to a mixture of floating point data types or a mixture of floating point and integer data types will be discussed later. There are no extra arithmetic operators for floating point data types, in particular there is no operator for raising a floating point number to a power or taking its square root. The ANSI standard defines library functions for these and many other common mathematical functions such as sines, cosines etc.

Floating point numbers may be read in using the *scanf()* library function in exactly the same way as integers were read in, only you need to use the appropriate **floating point conversion** specification. Any normal way of writing a floating point value may be used externally including integers which are properly converted to the equivalent floating point number.

Floating point arithmetic is illustrated by the following program.

```
main()
{
        double  data;
        double  x=490;
        data = (2.0*x)/3.5;
        printf("data = %20.10lf\n",data);
        x = 1.0/data;
        printf("   x = %20.10lf\n",x);
}
```

which produced the output

```
data =       280.0000000000
   x =         0.0035714286
```

And finally the following program called *fp3* illustrates floating point input.

```
main()
{
        double  x,y;
        printf("Enter values for x and y ");
        scanf("%lf%lf",&x,&y);
        printf("The sum of x and y is %10.5lf\n",x+y);
}
```

which proceeded as follows

```
$ fp3
Enter values for x and y 234.567 987.654321
The sum of x and y is 1222.22132
$ fp3
Enter values for x and y 1 2
The sum of x and y is    3.00000
```

- floating point data type mismatch in *printf()*
- display of floating point numbers
- Accuracy of floating point arithmetic
- integer data types

Arithmetic and Data Types - Floating point data type mismatch in *printf()*

Chapter chap4 section 3

It is essential that the data types in *printf()* and *scanf()* parameter lists match up, in type and number of items, with the conversion specifications in the function format string. The effects of errors are illustrated in the following examples. First an attempt to display an integer using floating-point conversion.

```
main()
{
        printf("The number was %10.5lf\n",2445);
        printf("The number was %10.5lf\n",-2445);
}
```

Which produced the output

```
The number was    0.00000
The number was       -NaN
```

The behaviour of this program may differ on different computers. On a PC using Turbo C, the computer just hung and had to be rebooted. What has gone wrong here is fairly easy to understand. The *printf()* function, when it saw a floating point conversion specified in the format string, expected a floating point number as the first parameter and attempted to interpret the set of bits it found as if they were a floating point number. In the first case it simply produced a wrong result, in the second case the symbols **NaN** mean *not a number* which means that the string of bits representing -2445 cannot be interpreted as a floating point number.

There are also problems with trying to mix the floating point types as the following example illustrates.

```
main()
{
        float   x = 4.5;
        double  xx = 4.5;
        long    double  xxx = 4.5;
        printf(" x  as a double %10.5lf\n",x);
        printf("xxx as a double %10.5lf\n",xxx);
```

```
        printf(" xx as a float  %10.5f\n",xx);
        printf("xxx as a float  %10.5f\n",xxx);
        printf(" x  as a long double %10.5Lf\n",x);
        printf(" xx as a long double %10.5Lf\n",xx);
}
```

Producing the following output which has been slightly rearranged to fit it all on one page. Each of the long sequences of digits was actually a single line of output.

```
 x  as a double    4.50000
xxx as a double -10560771802106750553398
20964236991361162176664932300208003249 21
49127351068588667107002898112176229354 90
40909540518381565135739181898662686472 91
14315184967339498937635661066552266224 10
63463152069503040346825467754523441614 47
00535546970535461596864773910796684532 99
14986496.00000
 xx as a float     4.50000
xxx as a float  -10560771802106750553398
20964236991361162176664932300208003249 21
49127351068588667107002898112176229354 90
40909540518381565135739181898662686472 91
14315184967339498937635661066552266224 10
63463152069503040346825467754523441614 47
00535546970535461596864773910796684532 99
14986496.00000
Segmentation fault (core dumped)
```

which should be enough to encourage anyone to be careful with type matching when using *printf().* The above spectacular results were obtained using the standard compiler on a SUN Sparc station. Other compilers may give up or produce weird results in different ways.

Before the program crashed it produced 4 lines of output. On the first line there was no particular problem and *printf()* was quite happy to interpret the value of of the *float* variable x as if it were a *double* . Actually the value of *printf()'* s second parameter has been automatically converted from *float* to *double* , this is known as **function parameter promotion** and is part of the ANSI standard. It is discussed later.

When generating the second line of output *printf()* expected a *double* value and mis-interpreted the supplied *long double* bit pattern.

On the third line *printf()* was expecting a *float* but knew that all *float* parameters are

promoted to *double* so correctly interpreted the supplied bit pattern, unfortunately this didn't work when a *long double* was supplied.

The effects of the fifth executable line were rather different, *printf()* expected a *long double* and in trying to get it something went dramatically wrong causing the program to crash. The actual crash shown here was recorded on a Unix system, the message **Segmentation fault** means that the program attempted to access a part of the computer memory that had been allocated to a different program. It does suggest that code such as

```
printf("x=%10.5Lf\n",2.5)
```

is likely to cause a disaster. (It did !). The problem is that "2.5" is seen as constant of type *double* rather than as a constant of type *long double* . A constant can, of course, be forced to be of a particular type as described earlier. The program

```
main()
{
        printf("x = %10.5Lf\n",2.5L);
}
```

was perfectly well behaved producing the output

```
x =    2.50000
```

- Display of floating point numbers

Arithmetic and Data Types - Display of floating point numbers

Chapter chap4 section 4

All the floating point values we have seen so far have been written using the conventional integral part/decimal point/fractional part notation. The C programming language also supports the well-known "e" notation which is handy for very large or very small floating point numbers.

A floating point number can be written in the form

```
"number" e "exponent"
```

without any internal spaces meaning simply `number X 10`$^{\texttt{exponent}}$ E may be used instead of e if preferred. Constants written using the "e" notation are normally of type double, they may be forced to other floating point types by appending a suitable letter as described earlier. The output e conversion displays a floating point number in the form shown above. The precision specification specifies the number of digits to appear after the decimal point in the output. Here are some examples in a program

```
main()
{
        double  x = 1.23456789e+15;
        double  y = 100;
        double  z = 0.5e-15;
        /* shows e format - width=10, precision=5 */
        printf(" x = %10.5le\n y = %10.5le\n z = %10.5le\n",x,y,z);
        /* shows e format - width=20, precision=10 */
        printf(" x = %20.10le\n y = %20.10le\n z = %20.10le\n",x,y,z);
        /* f format with default options */
        printf(" x = %lf\n y = %lf\n z = %lf\n",x,y,z);
}
```

producing the output

```
 x = 1.23457e+15
 y = 1.00000e+02
 z = 5.00000e-16
 x =     1.2345678900e+15
 y =     1.0000000000e+02
 z =     5.0000000000e-16
 x = 1234567890000000.000000
 y = 100.000000
 z = 0.000000
```

printf() also supports the g and G conversions for floating point numbers. These provide "f" conversion style output if the results fit in the field and "e" or "E" style output otherwise. With "g" conversions the precision specification in the field specifies the **total number** of significant digits to display rather than the number after the decimal point. The following program illustrates the use of the "g" conversion.

```
main()
{
        double  x = 12.3456789;
```

```
        printf("%10.4lg\n",x);
        printf("%10.4lg\n",x*x);
        printf("%10.4lg\n",x*x*x);
        printf("%10.4lg\n",x*x*x*x);
}
```

producing the output

```
     12.35
     152.4
      1882
 2.323e+04
```

For input via *scanf()* any of the floating point conversion styles can be specified in a *scanf()* input specification string and any style of floating point number will be accepted and converted.

- Accuracy of floating point arithmetic
- Integer data types

Arithmetic and Data Types - Accuracy of floating point arithmetic

Chapter chap4 section 5

As a final point concerning floating point numbers, it is worth remembering that not all numbers can be stored exactly using floating point representation, as the following program shows.

```
main()
{
        double  z,y;
        z = 1.0/3.0;    /* one third */
        y = 1.0 - z - z - z;    /* should be zero */
        printf("%20.18le\n",y);
}
```

producing the output

```
1.11022302462515654e-16
```

This, clearly, isn't the expected zero. You will probably get different results on different computers. Of course, it is no great surprise that 1/3 cannot be represented exactly, in conventional decimal notation 0.3333.... is only ever an approximation. However it is less widely realised that fractions that can be expressed exactly in decimal notation (such as 1/10) cannot be stored exactly using computer floating point formats. The reason is that computers use an underlying binary notation rather than a decimal notation and only fractions with powers of 2 in the denominator (such 376/1024) can be represented exactly.

The following example illustrates the effects of precision. The program repeatedly adds smaller and smaller numbers to a variable originally holding 50000.0.

First the example using *double* variables

```
main()
{
        double  x = 50000.0;
        double  y = 0.01;
        x += y; /* adds 0.01 */
        printf("x = %30.15lf\n",x);
```

```
        y /= 100;
        x += y; /* adds 0.0001 */
        printf("x = %30.15lf\n",x);
        y /= 100;
        x += y; /* adds 0.000001 */
        printf("x = %30.15lf\n",x);
        y /= 100;
        x += y; /* adds 0.00000001 */
        printf("x = %30.15lf\n",x);
        y /= 100;
        x += y; /* adds 0.0000000001 */
        printf("x = %30.15lf\n",x);
        y /= 100;
        x += y; /* adds 0.000000000001 */
        printf("x = %30.15lf\n",x);
}
```

producing the output

```
x =          50000.010000000002037
x =          50000.010099999999511
x =          50000.010100999999850
x =          50000.010101009997015
x =          50000.010101010098879
x =          50000.010101010098879
```

The results aren't quite what a mathematician would expect, we've got 15 signifcant digits of accuracy guaranteed by our compiler but we've displayed the results to 20 significant figures, the last 5 should, of course, be regarded with suspicion. Notice that the final addition has made no difference to the stored number.

Modifying the program by changing the variables to *float* and the output conversion from "lf" to "f" and recompiling gave the following results.

```
x =          50000.011718750000000
x =          50000.011718750000000
x =          50000.011718750000000
x =          50000.011718750000000
x =          50000.011718750000000
x =          50000.011718750000000
```

Detailed comment hardly seems necessary. However this simple example does underline the fact that care needs to be taken with any program that performs floating point calculations, for fuller details a textbook on numerical analysis should be consulted.

- Integer Data Types

Arithmetic and Data Types - Integer Data Types

Chapter chap4 section 6

The C programming language supports a variety of integer data types. These are

```
short   int
int
long    int
```

The data type *int* may correspond to either *short int* or *long int* . All the above data types may also be either *signed* (the default) or *unsigned* .

Declarations may be preceded by the keywords *signed* or *unsigned* . *short* and *long* may be written instead of *short int* and *long int* . If an integer data type is described as *unsigned* it means that the contents of a memory location of that type will always be interpreted as a positive number. If you are familiar with binary number representations this is equivalent to saying that the most significant bit is taken as part of the number rather than being taken as the sign bit.

The ANSI standard requires, indirectly, that a *short int* occupy at least 16 bits of computer memory and that a *long int* occupy at least 32 bits of computer memory. These limits are usually those actually in operation on most computers. On PC's an unqualified *int* is usually equivalent to a *short int* and on Unix based systems an unqualified *int* is usually equivalent to a *long int* .

If you are uncertain whether your compiler's *int* defaults to short or long, try the following simple program.

```
main()
{
        int     x = 30000;
        int     y;
        y = x+x;
        printf("twice x is %d\n",y);
}
```

On a system with long default *int*, such as the SUN Sparc Station the output would be

```
twice x is 60000
```

On a system with short default *int*, such as the Turbo C compiler running on a PC the output is likely to be

```
twice x is -5536
```

The problem here is that the number 60000 is simply too big to be stored in 16 bits of

computer memory. When the arithmetic circuits of the computer generated 60000 the result was simply too big to fit into the memory locations set aside for the purpose, this is known as an **arithmetic overflow. The ANSI standard says that behaviour is undefined under such circumstances, which means that anything might happen. It would, perhaps, be better if the arithmetic overflow were detected in the same way as division by zero but practically all computer systems simply truncate the generated sequence of bits giving wildly inaccurate results. If you are interested in computer architecture you might care to note that -5536 is 60000 - 2 X 32768.**

It is equally possible to make the SUN Sparc Station compiler give erroneous results by replacing 30000 by 2000000000 (that's 2 followed by 9 zeroes) in the previous program. Some PC compilers have options to force *int* to default to *long* rather than *short* , check the relevant manuals for details.

Input and output conversions for signed integers are

conversion	data type
hd	short int
d	int
ld	long int

Field width specifications work exactly the same way for all three conversions. As with floating point numbers, it is important to ensure that the conversion specification types match up with the variable types. The following program

```
main()
{
        short   int     si = 400;
        long    int     li = 400;
        printf(" si = %ld\n",si); /* short converted as long */
        printf(" li = %hd\n",li); /* long converted as short */
}
```

producing the output

```
 si = 26214800
 li = 400
```

shows what happens when you get it wrong. The above output was produced using Turbo C on a PC, the SUN Sparc station compiler gave completely correct results. It also shows that functional parameters of type *short int* are promoted to the system default *int* type, such promotion has no effect on a PC but on the SUN Sparc station accounts for the correct operation of the program.

It is clearly important, when designing and coding programs, to understand the limitations of the *int* data types in the environment being used. Problems are, not surprisingly, most commonly encountered when moving functional programs from a 32-bit *int* environment to a 16-bit *int* environment. It might be thought that portability problems could be avoided by declaring all integer variables as explicitly *long* . This is wrong because many library functions simply take *int* parameters using the local default

and unwise because it means that programs running in 16-bit environments will occupy more memory and take longer to do arithmetic than necessary.

- **The effect of precision specification on integer output**
- **Unsigned integers**
- **Bit-wise operators**

Arithmetic and Data Types - The effect of precision specification on integer output

Chapter chap4 section 7

The "%d" conversion which has been described earlier as well as including a field-width specification may also include a **precision** specification similar to that used with the floating point specifications. With a "%d" specification the precision specifies the **minimum** number of output digits to appear. This may include leading zero padding. A precision specification of zero will result in a totally blank output field when displaying the value 0, this is sometimes useful when printing large tables. The following program illustrates the effect of precision on integer output.

```
main()
{
        int     x=5,y=0,z=25;
        /* output always has at least 2 non-blank digits */
        printf(" x = %4.2d y = %4.2d z = %4.2d\n",x,y,z);
        /* complete suppression of zero value */
        printf(" x = %4.0d y = %4.0d z = %4.0d\n",x,y,z);
}
```

producing the output

```
 x =   05 y =   00 z =   25
 x =    5 y =      z =   25
```

Unsigned Integer Data Types

Arithmetic and Data Types - Unsigned Integer Data Types

Chapter chap4 section 8

The unsigned integer data types are

```
unsigned        short   int
unsigned        int
unsigned        long    int
```

unsigned may be written instead of **unsigned int .** When an integer is declared as *unsigned* the meaning is that the most significant bit in the internal representation is taken as significant data rather than a sign. All previous comments on signed integers apply. The rules of arithmetic are slightly different for unsigned integers in an obvious way. The main differences concern different input and output conversion codes and the existence of a number of extra operators available to manipulate unsigned values.

The main use of unsigned integers is to provide a convenient way to access the underlying sequence of bits in computer memory. This is important when the bit values are used to control output devices or to determine the status of input devices. Under these circumstances it is common for a program to need to determine the value of a particular bit or to need to set a particular bit or group of bits to particular values. Such operations are sometimes called bit-twiddling.

For input and output the unsigned *int* conversions u , o , x and X are used. They may be preceded by "l" or "h" for *long* or *short* unsigned integers. The "u" conversion is for conversion to or from an unsigned decimal form. The "o" conversion is for conversion to or from an octal representation of a value. The "x" and "X" conversions are for conversion to or from a hexadecimal representation of a value. The "X" conversion uses capital letters to represent the hexadecimal digits greater than 9, the "x" conversion uses lower case letters. Octal representation is widely used in the Unix environment for historical reasons. In the Unix environment the "x" conversion is used in preference to the "X" conversion. These conversions may be used with field specifications in exactly the same way as the "d" conversion seen earlier.

There is no output conversion for binary notation. The library functions *strtol()* and *strtoul()* which are discussed later provide facilities for input of binary values via strings. There is no way of writing a binary constant. You have to accept the grouping of bits in threes or fours implicit in octal or hexadecimal notation.

Where appropriate numerical values or constants can be written using octal or hexadecimal notation. Such constants are always unsigned but may be included in normal arithmetic expressions as a consequence of the rules for evaluating expressions involving objects of different data types.

An octal constant is a sequence of the digits 0-7 with an initial 0. A hexadecimal constant is a sequence of the characters 0-9,A-F,a-f with an initial 0x or 0X. Note that this means that any numerical constant with an initial or leading zero is **not** interpreted as a decimal constant. An unsigned decimal constant may be written by putting a "U" or "u" symbol at the end of the constant.

The following program shows the use of octal and hexadecimal constants and conversions.

```
main()
```

```
{
        unsigned        int x = 0300;   /* octal constant */
        unsigned        int y = 300U;   /* decimal constant */
        unsigned        int z = 0x300;  /* hexadecimal constant */
        /* output using unsigned decimal conversion */
        printf("unsigned decimal x = %4u y = %4u z = %4u\n",x,y,z);
        /* output using octal conversion */
        printf("          octal x = %4o y = %4o z = %4o\n",x,y,z);
        /* output using hexadecimal conversion */
        printf("    hexadecimal x = %4x y = %4x z = %4x\n",x,y,z);
}
```

producing the output

```
unsigned decimal x =  192 y =  300 z =  768
          octal x =  300 y =  454 z = 1400
    hexadecimal x =   c0 y =  12c z =  300
```

- Bitwise Operations
- Conversions and promotions between data types

Arithmetic and Data Types - Bitwise Operations

Chapter chap4 section 9

To manipulate individual bits there are a number of "bit-wise" operators.

Operator	Meaning
|	Or
&	And
^	Exclusive Or
>>	Right Shift
<<	left Shift
~	Complement

They may only be sensibly applied to unsigned integers. "~" is a unary operator, the others are binary operators (i.e have two operands). The following corresponding assignment operators are also available

```
|=
&=
^=
>>=
<<=
```

The "|", "&" and "^" operators perform the OR, AND and EXCLUSIVE OR functions between the bits of the two operands. The ">>" and "<<" operators shift the bit patterns to the left or right by the indicated number of places. 1s or 0s that are shifted to the end of a word simply disappear whilst 0s are generated and inserted at the other end. There is no way of rotating a bit pattern in C.

```
x<<1
```

is a lazy and obscure way of saying two times "x". The unary operator "~" flips or complements all the bits of its operand. Technically this is a 1's complement not a 2's complement.

For details of bitwise operator precedence see the table at the end of this chapter.

The following program, called *int3*, enables the user to enter a value and determine the value (1 or 0) of a particular bit.

```
main()
{
        /*  Notation used here is that Bit 0 is the
                least significant bit
        */
        unsigned        x;
        int     n;
```

```
        printf("Enter a number ");
        scanf("%u",&x);
        printf("Which bit do you want to see ");
        scanf("%d",&n);
        printf("It was %u\n",(x>>n)&01);
}
```

A typical dialogue is shown below

```
$ int3
Enter a number 4
Which bit do you want to see 2
It was 1
$ int3
Enter a number 27
Which bit do you want to see 5
It was 0
```

The expression

```
(x>>n)&01
```

is evaluated by first shifting the number x n positions to the right so that the required bit is in the least significant position. The parentheses are necessary because the "&" operator has a higher precedence than the ">>" operator. There is a list of operator precedences at the end of this chapter. After shifting, all the bits other than the least significant are set to zero by ANDing the shifted pattern with the binary bit pattern 0.....0001, here represented by the octal constant 01.

The next example, int4, sets a particular bit to the value 1.

```
main()
{
        unsigned        x;
        int     n;
        printf("Enter a number ");
        scanf("%u",&x);
        printf("Which bit do you want to set ");
        scanf("%d",&n);
        x |= 01<<n;
        printf("After setting bit %d value is %o (Octal)\n",n,x);
        printf("   and %d (decimal)\n",x);
}
```

Again, a typical dialogue

```
$ int4
Enter a number 27
Which bit do you want to set 4
After setting bit 4 value is 33 (Octal)
   and 27 (decimal)
$ int4
Enter a number 27
```

```
Which bit do you want to set 5
After setting bit 5 value is 73 (Octal)
   and 59 (decimal)
```

The expression "01 << n" constructs a bit pattern with the relevant bit set. The assignment operator "|=" performs the setting of the particular bit by ORing 1 with whatever was already in that position. In all other positions in the variable x, 0 is ORed with whatever is already there.

To set a specific bit to 0 rather than 1, the program could be modified to include the following expression

```
x &= ~(01 << n)
```

- The character data type
- Operator precedence

Arithmetic and Data Types - The character Data Type

Chapter chap4 section 10

The **character** data type invariably corresponds to a single byte or 8 bits of computer memory. A variable of character data type is declared using the keyword **char .** Input and output conversions are performed using the %c conversion code. This doesn't actually cause any conversion to take place, the bits in the variable are sent to the output or read from the input device unaltered. Character constants can be written by enclosing the character symbol in single quotes. Normally only a single character can appear between the character constant single quotes but the following conventions are understood.

Notation	Meaning
'\''	Single Quote
'\"'	Double Quote
'\?'	Question Mark
'\\'	Backslash
'\a'	Audible Signal (BEL)
'\b'	Back space
'\f'	Form feed (Page throw)
'\n'	New line (line feed)
'\r'	Carriage return
'\t'	Tab
'\v'	Vertical Tab

The above are called **escape sequences.** There are also escape sequences that allow the bit representations of characters in octal or hexadecimal notation. Typically these look like

```
'\16' or '\xe'
```

the first being an octal escape sequence and the second being a hexadecimal escape sequence. Octal escape sequences may consist of up to three digits. Unlike octal constants there is no need for an initial zero. The use of character constants is illustrated in the following program.

```
main()
```

```
{
        char    x1='H',x2,x3='l';
        x2='e';
        printf("%c%c%c%c%c\n",x1,x2,x3,x3,'o');
}
```

producing the output

```
Hello
```

There are better ways of doing this.

Surprisingly *char* variables may be *signed* or *unsigned* . This only has an effect when *char* variables are being mixed with other variable types in arithmetic expressions. The implications will be discussed in the next section .

- Mixed data type arithmetic
- Conversions and promotions between data types

Arithmetic and Data Types - Mixed Data Type Arithmetic

Chapter chap4 section 11

This section discusses the problems of evaluating expressions involving values of different data types. Before any expression involving a binary operator can be evaluated the two operands must be of the same type, this may often require that one (or sometimes both) of the values of the operands be converted to a different type. The rules for such conversions depend on both the types of the operands and the particular operator. Since the C language has 45 operators and 12 different data types this seems a daunting task, suggesting that there are something like 24000 combinations to consider. Fortunately, it is not that complicated, even so the rules are far from simple. Some programming languages take the simple way out and prohibit any expressions involving values of different data types and then provide special type conversion functions. This is called **strong typing** and such languages are called strongly typed. The C language is not strongly typed and instead infers the required type conversions from the context.

There are six basic methods of converting values from one type to another. In discussing the methods it is common to talk about the width of a data object, this is simply the number of bits of computer memory it occupies. The methods are.

1. Sign Extension

   This technique is adopted when converting a signed object to a wider signed object. E.g converting a *short int* to a *long int* . It preserves the numerical value by filling the extra leading space with 1's or 0's.
2. Zero Extension

   This is used when converting an unsigned object to a wider unsigned object. It works by simply prefixing the value with the relevant number of zeroes.
3. Preserve low order data - truncate

   This is used when converting an object to a narrower form. Significant information may be lost.
4. Preserve bit pattern

   This is used when converting between signed and unsigned objects of the same width.
5. Internal conversion

   This uses special hardware to convert between floating point types and from integral to floating point types.
6. Truncate at decimal point

   This is used to convert from floating point types to integral types, it may involve loss of significant information.

The basic conversions listed above are those that take place on assignment. Some examples are shown in the following program.

```
main()
{
```

```
        signed          short   int     ssi;
        signed          long    int     sli;
        unsigned        short   int     usi;
        unsigned        long    int     uli;
        ssi = -10;
        sli = ssi;      /* sign extension - sli should be -10 */
        printf("ssi = %8hd sli = %8ld\n",ssi,sli);
        usi = 40000U;  /* usigned decimal constant */
        uli = usi;      /* zero extension - uli should be 40000 */
        printf("usi = %8hu uli = %8lu\n",usi,uli);
        uli = 0xabcdef12;       /* sets most bits ! */
        usi = uli;      /* will truncate - discard more sig bits */
        printf("usi = %8hx uli = %8lx\n",usi,uli);
        ssi = usi;      /* preserves bit pattern */
        printf("ssi = %8hd usi = %8hu\n",ssi,usi);
        ssi = -10;
        usi = ssi;      /* preserves bit pattern */
        printf("ssi = %8hd usi = %8hu\n",ssi,usi);
}
```

This produced the following output.

```
ssi =      -10 sli =      -10
usi =    40000 uli =    40000
usi =     ef12 uli = abcdef12
ssi =    -4334 usi =    61202
ssi =      -10 usi =    65526
```

It may be interesting to note that the difference between the pairs of values on the last two lines is 65536. Conversions between *signed long* and *unsigned short* are typically undefined. The next program shows conversions to and from floating point types.

```
main()
{
        double  x;
        int     i;
        i = 1400;
        x = i;  /* conversion from int to double */
        printf("x = %10.6le i = %d\n",x,i);
        x = 14.999;
        i = x;  /* conversion from double to int */
        printf("x = %10.6le i = %d\n",x,i);
        x = 1.0e+60;    /* a LARGE number */
        i = x;  /* won't fit - what happens ?? */
        printf("x = %10.6le i = %d\n",x,i);
}
```

producing the output

```
x = 1.445000e+03 y = 1445
x = 1.499700e+01 y = 14
x = 1.000000e+60 y = 2147483647
```

This program was compiled and run on a SUN Sparc station. The loss of significant data, a polite way of saying the answer is wrong, in the final conversion should be noted.

There is an extra complication concerning variables of type *char* . The conversion rules to be applied depend on whether the compiler regards *char* values as signed or unsigned. Basically the ANSI C standard says that variables of type *char* are promoted to type *unsigned int* or type *signed int* depending on whether the type *char* is signed or unsigned. An *unsigned int* may then be further converted to a *signed int* by bit pattern preservation. This is implementation dependent. The following program shows what might happen.

```
main()
{
        char    c;
        signed          int     si;
        unsigned        int     usi;
        c = 'a';        /* MS bit will be zero */
        si = c;         /* will give small +ve integer */
        usi = c;
        printf("   c = %c\n  si = %d\n usi = %u\n",c,si,usi);
        c = '\377';     /* set all bits to 1 */
        si = c;         /* sign extension makes negative */
        usi = c;
        printf("  si = %d\n usi = %u\n",si,usi);
}
```

producing the output

```
   c = a
  si = 97
 usi = 97
  si = -1
 usi = 65535
```

The output shown above was produced using the SUN Sparc Station compiler, identical output was produced by Turbo C but the IBM 6150 gave the result

```
si = 255 usi = 255
```

for the final line.

Clearly both the SUN Sparc Station and the Turbo C compiler regarded *char* as a signed data type applying sign extension when assigning the *signed char* c to the *signed int* si . The conversion from *signed char* c to *unsigned int* usi is more interesting. This took place in two stages the first being sign extension and the second being bit pattern preservation. On the IBM 6150 *char* is treated as an unsigned data type, both assignments using bit pattern preservation. The following program with forced signing of *char* further clarifies the point.

```
main()
{
        unsigned        char    uc;
        signed  char    sc;
        unsigned        int     ui;
        signed  int     si;
```

```
        uc = '\377';
        ui = uc;
        si = uc;
        printf("Conversion of unsigned char ui = %u si = %d\n",
                                ui,si);
        sc = '\377';
        ui = sc;
        si = sc;
        printf("Conversion of signed char ui = %u si = %d\n",
                                ui,si);
}
```

producing the output

```
Conversion of unsigned char ui = 255 si = 255
Conversion of signed char ui = 4294967295 si = -1
```

For the first line of output the variable "uc" is an *unsigned char* and the conversion of its value to either the *signed int* si or the *unsigned int* ui is by bit pattern preservation. For the second line of output the variable "sc" is a *signed char* and the conversion of its value to the *signed int* si is a simple case of sign extension whereas the conversion to the *unsigned int* ui is by sign extension followed by bit pattern preservation.

The distinction between signed and unsigned *char* data types only becomes significant when a char data type is used to hold a value with the most significant bit set. For the normal ASCII character set this will not happen, but if you are using extended ASCII character codes (e.g. the box drawing characters found on PCs) then the most significant bit will be set.

It is also quite common practice to use variables of type *char* to store small integer values.

And a final example.

```
main()
{
        unsigned        char    uc = '\377';
                        char    c = '\377';
        signed          char    sc = '\377';
        int     v1,v2,v3;
        v1 = 20 + uc;   /* unsigned arithmetic */
        v2 = 20 + c;    /* default */
        v3 = 20 + sc;   /* signed arithmetic */
        printf("v1 = %d v2 = %d v3 = %d\n",v1,v2,v3);
}
```

producing the output

```
v1 = 275 v2 = 19 v3 = 19
```

The most significant point here is the value of "v1". The expression

```
20 + uc
```

involves a *signed integer* (20) and an *unsigned char* (uc). The *unsigned char* has been converted to an *unsigned int* by bit pattern preservation and the *signed integer* 20 converted to an *unsigned integer* prior to the execution of the + operation.

- The usual arithmetic conversions and promotions

Arithmetic and Data Types - The usual arithmetic conversions and promotions

Chapter chap4 section 12

The binary arithmetic operations are only defined when applied to two values of the same type. If the evaluation of an expression involves values of different types various conversions are applied to one (or both) of the values to give two values of the same type. These are known as the **usual arithmetic conversions .** The following description is taken straight from the ANSI C standard.

First, if either operand has type *long double* , the other operand is converted to *long double* .

Otherwise, if either operand has type *double* , the other operand is converted to *double* .

Otherwise, if either operand has type *float* , the other operand is converted to type *float* .

Otherwise, the integral promotions are first applied to both operands and then the following rules are applied.

If either operand has type *unsigned long int*, the other operand is converted to *unsigned long int.*

Otherwise, if one operand has type *long int* and the other has type *unsigned int,* if a *long int* can represent all values of an *unsigned int,* the operand of type *unsigned int* is converted to *long int;* if a *long int* cannot represent all the values of an *unsigned int,* both operands are converted to *unssigned long int*

Otherwise, if either operand has type *long int*, the other operand is converted to *long int.*

Otherwise, if either operand has type *unsigned int*, the other operand is converted to *unsigned int.*

Otherwise, both operands have type *int.*

The **integral promotions** specify the conversion of *char* and *short int* data types to *int* data types in arithmetic contexts. Functional promotions specify type conversion of functional parameters. The following table shows the integral promotions

Original type	Type after promotion

unsigned char	unsigned int
signed char	int
unsigned short int	unsigned int
short int	int

The type of a promoted *char* depends on whether the *char* data type is signed or unsigned. Integral promotions are applied in the context of evaluation or arithmetic expressions. The **functional promotion** rules are the same as the integral promotion rules with the extra rule that *float* data types are converted to *double*

A good rule is to avoid mixing data types, especially the signed and unsigned varieties when the rules get particularly complex. However there are some occasions when normal practice expects the use of mixed data types. Consider the following program which demonstrates the mixture of integral and floating-point types in evaluating an expression, here constants of the various types are mixed but similar effects would be seen using variables.

```
main()
{
        double  x,y;
        x = 1 + 2 / 3;
        y = 1 + 2 / 3.0;
        printf("x = %5lf, y = %5lf\n",x,y);
}
```

producing the output

```
x = 1.000000, y = 1.666667
```

Consider the expressions on the right hand side of the two assignments.

In the assignment to x the first step in evaluation of the expression is the evaluation of

```
2/3
```

Both operands are integers. Since "/" has higher precedence than "+" and there are no type conversions, the rules of integer division apply yielding the result 0 (as an integer). The addition operator then has two integral operands so yields an integral result (1) which is converted to floating point on assignment.

The second assignment proceeds rather differently. Again the first step is the evaluation of the expression

```
2/3.0
```

only in this case one of the operands is floating point so the other is converted to floating point and the rules of floating point arithmetic apply yielding the result 0.666666... The addition now has one floating point operand so the other is converted to floating point and the result assigned without conversion.

A final example of arithmetic involving objects of different types, in this case integers and characters is shown below. The program reads in a two digit number reverses the digits and displays both the number and its reversed form. The program would fail if it were not supplied with a two digit number.

```
main()
{
        int     n1=0,n2;
        char    c1,c2;
        printf("Enter a two digit number ");
        scanf("%c%c",&c1,&c2);
        n1 = (c1-'0')*10 + c2-'0';
        n2 = (c2-'0')*10 + c1-'0';
        printf("The number was %d\n",n1);
        printf("The number reversed was %d\n",n2);
        printf("The sum was %d\n",n1+n2);
}
```

A typical dialogue is

```
Enter a two digit number 17
The number was 17
The number reversed was 71
The sum was 88
```

The interesting point here concerns the expressions

```
c1-'0' and c2-'0'
```

The effect of evaluating these expressions is to subtract the internal representation of the character 0 (zero) from whatever was actually read in. If the internal representation of the digits 0-9 is continuous, as it would be if your computer used the ASCII character set, then this technique provides a handy way of converting from external character form to internal binary form, however it is often better to use a library function such as *atoi()*.

- Casts

Arithmetic and Data Types - Casts

Chapter chap4 section 13

Sticking to the rule of thumb that says don't mix data types in expressions and relying on assignment conversions can lead to very clumsy code and it is sometimes useful if we can force conversions when we actually need to. This is easily done in the C programming language using a construct known as a **cast.** A cast is of the form

```
(data type name) expression
```

The expression

```
1 + 2 / 3
```

used in the program in the previous section could have been written

```
1 + 2 / (double) 3
```

The expression after the "/" operator is now

```
(double) 3
```

which is of type double. When working with constants this seems clumsy, it is certainly easier to write 3.0 but if the expression had involved integer variables then it does avoid unnecessary intermediate variables. Supposing k,l and m were all integer variables then the expression

```
k + l / m
```

would be evaluated using all integer arithmetic, to evaluate it using floating point arithmetic, a cast could be used thus

```
k + l / (double) m
```

- Operator Prcedences

Arithmetic and Data Types - Operator Precedences

Chapter chap4 section 14

To finish off this rather long chapter here is a list, in precedence order, of all the operators we have seen so far.

Operators	Associativity
`() ++(postfix) --(postfix)`	Left to Right
`++(prefix) --(prefix) ~ (type) +(unary) -(unary)`	Right to Left
`* / %`	Left to Right
`+ -`	Left to Right
`<< >>`	Left to Right
`&`	Left to Right
`^`	Left to Right
`|`	Left to Right
`= += *= <<= >>= /= %= -= &= |= ^=`	Right to Left

- Exercises

Arithmetic and Data Types - C and C++

Chapter chap4 section 15

As well as the cast mechanism for explicit type conversion, C++ also allows data type names to be used as if they were functions. This type of explicit type conversion is more typical of older program languages. For example, in C++, you could write

```
int(5.3)
```

instead of

```
(int)5.3
```

The C style cast is available in C++ because C++ is a strict superset of C, the function-like usage is more appropriate in more advanced applications.

The fact that the operators "<<" and ">>" can be used to mean two completely different things in C++ is particularly interesting. The C++ compiler decides which interpretation to apply by examining the data types of the variables associated with the operators. If the operator to the left of a "<<" or ">>" symbol is of IO stream type then input or output code is generated, otherwise shift instructions are generated.

This use of the same operator to mean different things is called **operator overloading.** In fact, ANSI C also has operator overloading, the operator & means something quite different (take the address) when it used a unary operator in front of variable compared with its use as a binary operator between two integers (bitwise AND).

Arithmetic and Data Types - Exercises

Chapter chap4 section 16

1. Write a program similar to exercise 8 of the previous chapter only displaying the percentage correct to 2 decimal places.
2. Run the program that adds 30000 to 30000 on your computer. What size are your ints?
3. Write a program that reads in a temperature expressed in Celsius (Centigrade) and displays the equivalent temperature in degrees Fahrenheit.
4. If the variable x is of type *double* or type *float* then adding a suitably small number to it will not change the actual value stored. Write a program to read in an integer, calculate its reciprocal and store it in a variable called "r". Calculate and display the value of

   ```
   (1000+r)-1000
   ```

   Display the value using a "g" or "e" conversion. Run your program for various input integers. How big does the integer have to be before adding the reciprocal has no effect ?
5. Write and run a program to find out whether your computer's char's are signed or unsigned.
6. Write a program to read in a four letter word and print it out backwards.
7. Write a program to read in a three digit number and produce output like

   ```
   3 hundreds
   4 tens
   7 units
   ```

   for an input of 347. There are two ways of doing this. Can you think of both of them? Which do you think is the better?
8. The digital root of an integer is calculated by adding up the individual digits of a number. Write a program to calculate this (for numbers of up to 4 digits). Is it true that the digital root of a number divisible by 3 is also divisible by 3?

   Does this still hold if the number is expressed in hexadecimal or octal notation? I.e. if you add up the hexadecimal or octal digits is the sum divisible by 3. Write a program to find out.
9. The notes include a program that got into trouble printing out floating point numbers producing very long lines of digits. Try this on your system.

Addresses, Pointers, Arrays and Strings - String input using "s" conversion

Chapter chap6 section 8

Strings may be read in using the %s conversion with the function *scanf()* but there are some irksome restrictions. The first is that *scanf()* will only recognise, as an external string, a sequence of characters delimited by whitespace characters and the second is that it is the programmer's responsibility to ensure that there is enough space to receive and store the incoming string along with the terminating null which is automatically generated and stored by *scanf()* as part of the %s conversion. The associated parameter in the value list must be the address of the first location in an area of memory set aside to store the incoming string.

Of course, a field width may be specified and this is the maximum number of characters that are read in, but remember that any extra characters are left unconsumed in the input buffer.

Simple use of *scanf()* with %s conversions is illustrated in the program shown below.

```
main()
{
        char    strspace[50];   /* enough ?? */
        printf("Enter a string ");
        scanf("%s",strspace);
        printf("The string was >>%s<<\n",strspace);
}
```

The program was called *str2* and a typical dialogue is illustrated below.

```
$ str2
Enter a string fred
The string was >>fred<<
$ str2
Enter a string fred and joe
The string was >>fred<<
$ str2
Enter a string    fred
The string was >>fred<<
```

```
$ str2
Enter a string "fred and joe"
The string was >>"fred<<
$ str2
Enter a string fred\ and\ joe
The string was >>fred\<<
$
```

It will be noted that attempts to quote a string with internal spaces or to escape the internal spaces (both of which normally work in the Unix command environment) did not work.

- String input using scanset conversion
- String input using c conversion
- String input using the gets() function

Addresses, Pointers, Arrays and Strings - String input using scanset conversion

Chapter chap6 section 9

The **scanset** conversion facility provided by *scanf()* is a useful string input method although it can appear dauntingly complex. This conversion facility allows the programmer to specify the set of characters that are (or are not) acceptable as part of the string. A scanset conversion consists of a list of acceptable characters enclosed within square brackets. A range of characters may be specified using notations such as "a-z" meaning all characters within this range. The actual interpretation of a range in this context is implementation specific, i.e. it depends on the particular character set being used on the host computer. If you want an actual "-" in the scanset it must be the first or last character in the set. If the first character after the "[" is a "^" character then the rest of the scanset specifies unacceptable characters rather than acceptable characters.

The use of scansets is shown by this program, called *str3*.

```
main()
{
        char    strspace[50];
        printf("Enter a string in lower case ");
        scanf("%[ a-z]",strspace);
        printf("The string was >>%s<<\n",strspace);
}
```

And a typical dialogue is shown here.

```
$ str3
Enter a string in lower case hello world
The string was >>hello world<<
$ str3
Enter a string in lower case hello, world
The string was >>hello<<
$ str3
Enter a string in lower case abcd1234
The string was >>abcd<<
$
```

Note that, in all cases, conversion is terminated by input of something other than a

space or lower-case letter.

- String input using s conversion
- String input using c conversion
- String input using the gets() function

Addresses, Pointers, Arrays and Strings - String input using c conversion

Chapter chap6 section 10

An alternative method for the input of strings is to use *scanf()* with the %c conversion which may have a count associated with it. This conversion does not recognise the new-line character as special. The count specifies the number of characters to be read in. Unlike the %s and %[] (scanset) conversions the %c conversion does not automatically generate the string terminating NUL and strange effects will be noted if the wrong number of characters are supplied. Its use is demonstrated by the following program.

```
main()
{
        char    inbuf[10];
        int     i;
        while(1)
        {
                printf("Enter a string of 9 characters ");
                scanf("%10c",inbuf);
                inbuf[9]='\0';  /* Make it a string */
                printf("String was >>%s<<\n");
                if(inbuf[0] == 'Z') break;
        }
}
```

typical dialogue is shown below.

```
$ str4
Enter a string of 9 characters 123456789
String was >>123456789<<
Enter a string of 9 characters abcdefghi
String was >>abcdefghi<<
Enter a string of 9 characters abcdefghijklmnopqr
String was >>abcdefghi<<
Enter a string of 9 characters 123456789
String was >>klmnopqr
<<
Enter a string of 9 characters ttttttttt
String was >>23456789
```

There are some rather odd things going on here. The first point to note is that, contrary to the prompt, 10 characters are being converted. This is done so that the newline character at the end of the input line is also read in, otherwise it would be left in the input buffer to be read in as one of the input characters the next time round. The effect of providing too many input characters is that "unconsumed" input characters (including new-line characters) are left in the input buffer, these will be "consumed" by the next call to *scanf()*, if too few input characters are provided then *scanf()* hangs (or **blocks**) until it has got enough input characters. Both types of behaviour can be seen in the above example.

The complexities of *scanf()'* s behaviour suggest that it is not really suitable for reliable general purpose string input.

- String input using s conversion
- String input using scanset conversion
- String input using the gets() function

Addresses, Pointers, Arrays and Strings - String input using the gets() library functions

Chapter chap6 section 11 The best approach to string input is to use a library function called **gets().** This takes, as a single parameter, the start address of an area of memory suitable to hold the input. The complete input line is read in and stored in the memory area as a null-terminated string. Its use is shown the program below.

```
main()
{
/*      this program reads in strings until it has read in
        a string starting with an upper case 'z'
*/
        char    inbuf[256];     /* hope it's big enough ! */
        while(1)
        {
                printf("Enter a string ");
                gets(inbuf);
                printf("The string was >>%s<<\n",inbuf);
                if(inbuf[0] == 'Z') break;
        }
}
```

and a typical dialogue is shown below

```
$ str5
Enter a string hello world
The string was >>hello world<<
Enter a string 1
The string was >>1<<
Enter a string
The string was >><<
Enter a string ZZ
The string was >>ZZ<<
$
```

We will see shortly how to use the functional value associated with *gets()*, this will provide a better way of terminating the input in loops such as that shown above. You should, by now, have a pretty good idea of the likely consequences of the input string being too long for the buffer area. *gets()* simply does not handle this problem, you can either, as is done here, declare a fairly large buffer and hope or use the more advanced function *fgets()* that will be described in the section on file handling.

Library functions for handling Input Strings

Addresses, Pointers, Arrays and Strings - The library function sprintf() and puts()

Chapter chap6 section 13

The library function **sprintf()** is similar to *printf()* only the formatted output is written to a memory area rather than directly to standard output. It is particularly useful when it is necessary to construct formatted strings in memory for subsequent transmission over a communications channel or to a special device. Its relationship with *printf()* is similar to the relationship between *sscanf()* and *scanf()*. The library function **puts()** may be used to copy a string to standard output, its single parameter is the start address of the string. *puts()* writes a new-line character to standard output after it has written the string.

A simple example of the use of *sprintf()* and *puts()* is shown below.

```
main()
{
        char    buf[128];
        double  x = 1.23456;
        char    *spos = buf;
        int     i = 0;
        sprintf(buf,"x = %7.5lf",x);
        while(i<10) puts(spos+i++);
}
```

producing the output

```
x = 1.23456
 = 1.23456
= 1.23456
 1.23456
1.23456
.23456
23456
3456
456
56
```

If "\n" had been incorporated in the format string of the *sprintf()* then the output would have been double spaced because the *sprintf()* function would have put a newline character in the generated string and *puts()* would then generate a further newline.

Note that this program is slightly naughty, strictly *sprintf()'* s layout specification string should be a string constant.

The correct way to adjust field width and precision at run time is to replace the width and/or precision with a star ("*") and include an appropriate integer in the parameter list. This value will be used **before** the actual value to be converted is taken from the parameter list. Here is a

program showing the facility in use.

```
main()
{
        double  x=1234.567890;
        int     i=8,j=2;
        while(i<12)
        {
                j=2;
                while(j<5)
                {
                        printf("width = %2d precision = %d "
                               "display >>%*.*lf<<\n",i,j,i,j,x);
                        j++;
                }
                i++;
        }
}
```

The program displays the effects of various widths and precisions for output of a *double* variable. Here is the output.

```
width =  8 precision = 2 display >> 1234.57<<
width =  8 precision = 3 display >>1234.568<<
width =  8 precision = 4 display >>1234.5679<<
width =  9 precision = 2 display >>  1234.57<<
width =  9 precision = 3 display >> 1234.568<<
width =  9 precision = 4 display >>1234.5679<<
width = 10 precision = 2 display >>   1234.57<<
width = 10 precision = 3 display >>  1234.568<<
width = 10 precision = 4 display >> 1234.5679<<
width = 11 precision = 2 display >>    1234.57<<
width = 11 precision = 3 display >>   1234.568<<
width = 11 precision = 4 display >>  1234.5679<<
```

The ">>" and "<<" are used to indicate the limits of the output field. Note that the variables "i" and "j" appear twice in parameter list, the first time to give the values in the annotation and the second time to actually control the output.

Arrays of Strings

The Pre-Processor and standard libraries - Mathematical functions

Chapter chap8 section 7

The **math.h** include file contains prototypes for a useful set of mathematical functions.

```
asin    atan    atan2   cos     sin     tan
cosh    sinh    tanh    exp     frexp   ldexp
log     log10   modf    pow     sqrt    ceil
fabs    floor   fmod
```

For details of what these do you should consult the appropriate manual. There is no standard set of mathematical constants (such as "pi") defined in any header file, you've got to set these up for yourself although many implementations do provide such information, frequently in the *math.h* include file.

If a mathematical function fails due to an inappropriate input value or the output value lying outside the range of representable numbers then the global variable **errno** is set to a value indicating the type of error. The value returned by the mathematical function is implementation dependent. The include file **errno.h** includes the declaration of *errno* and #define's for the two likely values **EDOM** and **ERANGE** used for indicating errors in mathematical function evaluation. *EDOM* means the input value is inappropriate, a domain error and *ERANGE* means that the output value is out of range.

On some systems you may need to include extra command line arguments for the C compiler to make it look in the mathematical functions library. On a Unix system

```
cc summ.c -lm
```

would be typical, the "-lm" flag specifying the maths library. This shouldn't be necessary on an ANSI system. The following program which tabulates squares, square roots and cube roots shows the use of the maths library functions *sqrt()* and *pow()*.

```
#include        <math.h>
main()
{
        int     i=0;
```

```
        while(i++<16)
                printf("%2d %4d %8.6lf %8.6lf\n",
                        i,i*i,sqrt(i),pow(i,1.0/3));
}
```

producing the output

```
 1    1 1.000000 1.000000
 2    4 1.414214 1.259921
 3    9 1.732051 1.442250
 4   16 2.000000 1.587401
 5   25 2.236068 1.709976
 6   36 2.449490 1.817121
 7   49 2.645751 1.912931
 8   64 2.828427 2.000000
 9   81 3.000000 2.080084
10  100 3.162278 2.154435
11  121 3.316625 2.223980
12  144 3.464102 2.289428
13  169 3.605551 2.351335
14  196 3.741657 2.410142
15  225 3.872983 2.466212
16  256 4.000000 2.519842
```

Notice that the operand of the *sqrt()* library function was of type *int*, this did not cause any problems since the prototype for *sqrt()* specifies a parameter of type double and the compiler has simply included a type conversion in the code that generates the value of the parameter.

The string handling macros and functions

The Pre-Processor and standard libraries - String handling

Chapter chap8 section 8

The **string.h** include file includes prototypes for the following library functions

1. **strcpy**

   Copy a string from one area to another. The use of this function was illustrated earlier.

2. **strncpy**

   Copy a string up to a maximum number of characters or end of source string. This avoids the problems associated with *strcpy()* The function has three parameters which specify the destination address, the source address and the maximum number of bytes to copy. A typical use is illustrated by the following code fragment

   ```
   char      inbuf[512];
   char      wkbuf[20];
     .
     .
   strncpy(wkbuf,inbuf,19);
     .
   ```

   This copies at most 19 characters from *inbuf* to *wkbuf* The number of bytes to copy is set to 19, one less than the size of *wkbuf* because copying of long strings is terminated by the exhaustion of number of characters to copy not the need to store a NUL in the destination space.

   The use of *strncpy()* is generally preferable to the use of *strcpy()*

3. **strcat**

   Concatenate strings The use of this function was illustrated earlier.

4. **strncat**

   Concatenate strings with limit on size. This, like *strncpy()*, imposes a restriction on the maximum number of characters concatenated. It has three parameters specifying the destination and source addresses and the maximum number of characters to append. A typical use is shown below

   ```
   int l;
   char dirpath[80];
   char file[256];
         .
         .
    l = strlen(pname); /* how much space already used ?? */
    strncat(pname,"/",80-l-1); /* append a slash */
    strncat(pname,fname,80-l-2);  /* and the file name to give
                                     full path name (if poss) */
         .
         .
   ```

   This code first determines the length of the string already in the destination area then appends as much as possible of the string in input.

5. **strcmp**

   Compare strings. The use of this function was illustrated earlier. The return value of this function is positive or negative depending on the difference of the first pair of characters that differ that are found in the strings being compared.

6. **strcoll**

   Compares strings in a manner dependent on the current locale. The value returned by *strcmp()* is dependent on the host system's character set. This function allows the user control the return value to give a non-standard ordering of strings.

7. **strncmp**

Compares strings with a limit on size

8. **strxfrm**

   Transforms a string in a manner dependent on the current locale. This can be used instead of *strcoll()* to generate a transformed version of a string that can be used with *strcmp()* to give the effect of a non-standard ordering sequence. A separate transformed string is stored explicitly.

9. **strchr**

   Search a string for a particular character. This function takes two parameters the first is the string to be searched and the second is the character to be searched for. The return value is a pointer to the first occurence of the character within the string or NULL if the character was not found.

   The following example program called *findsl* shows the function in use.

```
#include <string.h>

main()
{
        int     input[256];
        char    *cptr;
        printf("Enter a string ");
        gets(input);
        cptr = strchr(input,'/');
        if(cptr)
        {
                printf("String from first / is %s\n",cptr);
        }
        else
                printf("No / in input\n");
}
```

   A typical dialogue is

```
$ findsl
Enter a string http://www.wlv.ac.uk/
String from first / is //www.wlv.ac.uk/
$ findsl
Enter a string the cat sat on the mat
No / in input
```

10. **strcspn**

    Computes length of initial substring of a string comprising characters NOT from a specified set of characters. It is the opposite of *strspn()*

11. **strpbrk**

    Similar to strchr() only a set of characters may be specified. Here's an example program called showbrk.

```
#include  <string.h>
main()
{
        char    inbuf[256];
        char    *cptr;
        printf("Enter string ");
        gets(inbuf);
        cptr = strpbrk(inbuf,"abcdefghijklmnopqrstuvwxyz");
        if(cptr)
                printf("First lower case letter at position %d\n",cptr-inbuf);
        else
                printf("No lower case letters\n");
}
```

and a typical dialogue

```
$ showbrk
Enter string ABCDEfGHIHJKlm
First lower case letter at position 5
$ showbrk
Enter string ABCDEFG
No lower case letters
```

12. **strrchr**

    Similar to strchr() only works from end of string. Here's an example program called *findlast* that demonstrates its use.

```
#include        <string.h>
main()
{
        char    *cptr;
        char    input[256];
        printf("Enter full path name ");
        gets(input);
        cptr = strrchr(input,'/');
        if(cptr)
                printf("File name %s\n",cptr+1);
        else
                printf("Not a full path name\n");
}
```

    Note the use of "cptr+1" in the *printf()* parameter list, since *strrchr()* returned the address of the '/'.

13. **strspn**

    Computes length of initial substring of a string comprising characters from a specified set of characters. It is closely realted to *strpbrk()* as will be seen from the example of that function.

14. **strstr**

    Looks for a string as a sub-string of another string.

15. **strtok**

    Breaks string into sequence of tokens. This function has two parameters the first is a string to be examined and the second is a string consisting of the token separators. The function will be used in a loop, on the first call, with the first parameter set to point to the string, the function makes a private note of the parameters and where it has got to in breaking the string into tokens. On subsequent calls the first parameter should be NULL. The return value is a pointer to the token start address or NULL if there are no more tokens.

    It is important to realise that *strtok()* operates by converting token separators into NULLs in the input string.

    Here's an example of it in use.

```
#include        <string.h>
main()
{
        char    inbuf[256];
        char    *cptr;
        int     count = 0;
        printf("Enter path name ");
        gets(inbuf);
        cptr = strtok(inbuf,"/");
        while(cptr)
        {
                printf("Component %d = %s\n",++count,cptr);
                cptr = strtok(NULL,"/");
        }
```

```
}
```

and the dialogue

```
Enter path name home/staff/acad/jphb/cbook/new/chap8/chap8.txt
Component 1 = home
Component 2 = staff
Component 3 = acad
Component 4 = jphb
Component 5 = cbook
Component 6 = new
Component 7 = chap8
Component 8 = chap8.txt
```

16. **strerror**

    Generates an error message. This is used in conjunction with the standard global variable *errno* to give standard error messages describing the reason for failure of library routines or system calls. The actual messages are, of course, host system dependant. It takes a single integer parameter and returns a pointer to the relevant error message.

17. **strlen**

    Determines the length of a string The use of this function was illustrated earlier

18. **memset**

    Fills a memory area with a particular character.

    ```
    memset(buff,'\0',256);
    ```

    could be used to fill 256 bytes starting at *buff* with zeroes.

19. **memcpy**

    This copies characters from one memory area to another. Unlike *strcpy()* it does not stop when a NUL is encountered in the data being copied.

20. **memmove**

    This is similar to memcpy() except that it works via an intermediate area giving defined behaviour if the areas overlap.

21. **memcmp**

    Compare areas of memory

22. **memchr**

    Search memory for a particular character.

The #define directive

Addresses, Pointers, Arrays and Strings - Arrays of Strings

Chapter chap6 section 14

It is important to realise that a string constant such as

```
"hello, world"
```

is really a constant of type *pointer to character* that points to the system place where the actual string is stored. It is possible to have aggregates of strings which will, of course, really be aggregates of pointers to char. The following program, called *stall1*, demonstrates the point.

```
main()
{
        char    *days[] = {
                "Sunday",
                "Monday",
                "Tuesday",
                "Wednesday",
                "Thursday",
                "Friday",
                "Saturday"
                        };
        char    *dname;
        int     nday;
        printf("Enter day of week [1-7] ");
        scanf("%d",&nday);
        dname = days[nday-1];
        printf("That is %s\n",dname);
}
```

A typical dialogue is shown below

```
$ stall1
Enter day of week [1-7] 4
That is Wednesday
bash$ stall1
Enter day of week [1-7] 2
That is Monday
```

```
$
```

The final 2 lines of code could have been replaced by the single line

```
printf("That is %s\n",ndays[n-1]);
```

The relationship between string constants, pointers and aggregates is well illustrated by the following program, which prints out a string constant backwards.

```
main()
{
        int     i=4;
        char    c;
        do
        {
                c = "hello"[i];
                printf("%c",c);
                i--;
        } while(i >= 0);
        printf("\n");
}
```

The output was

```
olleh
```

The key to understanding this is to remember that a string constant, such as "hello" in this case, is really the address of the place where the actual string is stored, the index operator can then, quite naturally, be associated with such an address.

The next program devlops the idea rather more obscurely. The purpose of the program, called stall2, is to display an integer in hexadecimal notation.

```
main()
{
        int     i;
        unsigned        int     m;
        printf("Enter an integer ");
        scanf("%d",&i);
        m = i;
        i = 0;
        do
        {
                printf("%c","0123456789abcdef"[m>>28]);
                m <<= 4;
```

```
        } while(++i<8);
        printf("\n");
}
```

A typical dialogue is given below.

```
$ stall2
Enter an integer 10
0000000a
$ stall2
Enter an integer 33
00000021
$ stall2
Enter an integer -9
fffffff7
$
```

The code will repay careful study. The string constant

```
"0123456789abcdef"
```

is, of course, the address of the start of an aggregate so the index operator may properly be used in this context. The value of the expression

```
m>>28
```

is simply the most significant half-byte of the (shifted) value of m which is used as index to select the relevant character from the string. Finally

```
m <<= 4
```

shifts the bit pattern 4 bits to the left so that progressively less significant half-bytes become the most significant half-byte after evaluation of "m>>28". It is essential that "m" be *unsigned* otherwise the shift operations would not work correctly.

The use of *printf()* or some other library function is a much more obvious and straightforward way of coding this conversion but it may well be useful if only a limited amount of memory is available for storing the code.

Library string handling function

Addresses, Pointers, Arrays and Strings - Library string handling functions

Chapter chap6 section 15

The C programming language does not, in fact, support a string data type, however strings are so useful that there is an extensive set of library functions for manipulating strings. Three of the simplest functions are

Name	Function
strlen()	determine length of string
strcmp()	compare strings
strcpy()	copy a string

The first of these, *strlen(),* is particularly straightforward. Its single parameter is the address of the start of the string and its value is the number of characters in the string excluding the terminating NUL.

The second function, *strcmp(),* takes the start addresses of the two strings as parameters and returns the value zero if the strings are equal. If the strings are unequal it returns a negative or positive value. The returned value is positive if the first string is greater than the second string and negative if it is less than. In this context the relative value of strings refers to their relative values as determined by the host computer character set (or **collating sequence**).

It is important to realise that you cannot compare two strings by simply comparing their start addresses although this would be syntactically valid.

The third function, *strcpy(),* copies the string pointed to by the second parameter into the space pointed to by the first parameter. The entire string, including the terminating NUL, is copied and there is no check that the space indicated by the first parameter is big enough.

A simple example is in order. This program, *stall3*, has the opposite effect to the example given earlier.

```
main()
{
        char    *days[] = {
                "Sunday",
                "Monday",
                "Tuesday",
                "Wednesday",
                "Thursday",
                "Friday",
                "Saturday"
                        };
        int     i;
        char    inbuf[128];
        printf("Enter the name of a day of the week ");
        gets(inbuf);
        do
        {
                if(strcmp(days[i++],inbuf)==0)
                {
                        printf("day number %d\n",i);
                        exit(0);
```

```
                }
        } while(i<7);
        printf("Unrecognised day name\n");
}
```

A typical dialogue

```
$ stall3
Enter the name of a day of the week Tuesday
day number 3
$ stall3
Enter the name of a day of the week Bloomsday
Unrecognised day name
$ stall3
Enter the name of a day of the week Friday
day number 6
$
```

The program is totally unforgiving of any errors in the input layout such as leading and trailing spaces or entry all in lower case or entry of abbreviations.

To demonstrate the use of *strlen()*, here is a simple program, called *stall4*, that reads in a string and prints it out reversed, a tremendously useful thing to do. The repeated operation of this program is terminated by the user entering a string of length zero, i.e. hitting the RETURN key immediately after the program prompt.

```
main()
{
        char    inbuf[128];     /* Hope it's big enough */
        int     slen;   /* holds length of string */
        while(1)
        {
                printf("Enter a string ");
                gets(inbuf);
                slen = strlen(inbuf);   /* find length */
                if(slen == 0) break;    /* termination condition */
                while(slen > 0)
                {
                        slen--;
                        printf("%c",*(inbuf+slen));
                }
                printf("\n");
        }
}
```

The program operates by printing the characters one by one, starting with the last non-NUL character of the string. Notice that "slen" will have been decremented before the output of the character, this is correct since the length returned by *strlen()* is the length excluding the NUL but the actual characters are aggregate members 0 .... length-1.

A typical dialogue is illustrated below.

```
$ stall4
Enter a string 1234
```

```
4321
Enter a string     x
x
Enter a string abc def ghi
ihg fed cba
Enter a string
$
```

Here is another version of the same program re-written using a more typical C programming style.

```
main()
{
        char    inbuf[128];     /* Hope it's big enough */
        int     slen;   /* holds length of string */
        while(1)
        {
                printf("Enter a string ");
                gets(inbuf);
                if((slen = strlen(inbuf)) == 0) break;
                while(slen--)printf("%c",*(inbuf+slen));
                printf("\n");
        }
}
```

It illustrates the use of side-effects and address arithmetic and should be compared with the first version.

The next prorgam is designed to drive home the point about comparing strings as distinct from comparing their start addresses.

```
main()
{
        char    x[22],*y;
        strcpy(x,"A Programming Example");
        y = x;

/*      First test - compare y with constant */

        if( y == "A Programming Example")
                printf("Equal 1\n");
        else
                printf("Unequal 1\n");

/*      Second test - compare using strcmp() */

        if(strcmp(x,"A Programming Example") == 0)
                printf("Equal 2\n");
        else
                printf("Unequal 2\n");

/*      Assign constant address and compare */

        y = "A Programming Example";
        if( y == "A Programming Example")
```

```
                printf("Equal 3\n");
        else
                printf("Unequal 3\n");
}
```

It produced the following output

```
Unequal 1
Equal 2
Unequal 3
```

The first comparison compares the address held in the variable "y" with the address of the system place where the string constant "A Programming Example" is stored. Clearly the start address of the aggregate "x" is different from the address of the system place where the string constant "A Programming Example" is stored, since *strcpy()* has only copied the string.

The second test used *strcmp()* to compare the strings rather than their start addresses, the result is, not surprisingly, that the strings were, in fact, equal.

The final test looks rather surprising. A value has been assigned to "y" and "y" has then been immediately compared with that value and found to be different. The explanation is that the compiler has not been clever enough to spot the repeated use of the same string constant and has made multiple copies of this constant in memory. This underlines the fact that the actual value of a string constant is the address of the first character. Some compilers may be clever enough to avoid this problem. The ANSI standard does not specify any particular behaviour.

Finally an example using *strcpy()*. This program, called *stall5* twiddles the case of every character in the input string.

```
main()
{
        char    istr[128];      /* input buffer */
        char    tstr[128];      /* translated string here */
        int     i;
        int     slen;           /* string length */
        while(1)
        {
                printf("Enter a string ");
                gets(istr);
                if((slen=strlen(istr))==0) break;       /* terminate */
                strcpy(tstr,istr);      /* make a copy */
                i = 0;
                while(i < slen) /* translate loop */
                {
                        if(     tstr[i] >= 'A' &&
                                tstr[i] <= 'Z') /* upper case */
                                        tstr[i] += 'a'-'A';
                        else if(tstr[i] >= 'a' &&
                                tstr[i] <= 'z') /* lower case */
                                        tstr[i] += 'A'-'a';
                        i++;    /* to next character */
                }
                printf("   Original string = %s\n",istr);
                printf("Transformed string = %s\n",tstr);
        }
```

```
}
```

The following dialogue is typical

```
$ stall5
Enter string aBDefgXYZ
   Original string = aBDefgXYZ
Transformed string = AbdEFGxyz
Enter string ab   CD   123
   Original string = ab   CD   123
Transformed string = AB   cd   123
Enter string :::x:::y:::Z:::
   Original string = :::x:::y:::Z:::
Transformed string = :::X:::Y:::z:::
Enter string
```

The program has preserved the original string by copying it to a different memory area before manipulating it.

It is important that there is somewhere to copy the string to. A common programming error is illustrated below. This variation on the previous program is called *stall6.*

```
main()
{
        char    istr[128];
        char    *tstr;
        int     i;
        int     llen;
        while(1)
        {
                printf("Enter string ");
                gets(istr);
                if((llen=strlen(istr))==0) break;
                strcpy(tstr,istr);
                i = 0;
                do
                {
                        if(tstr[i]>='A' && tstr[i]<='Z')
                                tstr[i] += 'a'-'A';
                        else if(tstr[i]>='a' && tstr[i]<='z')
                                tstr[i] += 'A'-'a';
                } while(i++<=llen);
                printf("   Original string = %s\n",istr);
                printf("Transformed string = %s\n",tstr);
        }
}
```

This is what happened

```
$ stall6
Enter string abcdefghjikl
Segmentation fault (core dumped)
```

The programmer has probably assumed that there really is such a data type as a string and that *strcpy()*

provides the facility to assign strings. The failure of the program is not surprising once you think about the initial value of "tstr". The initial value of non-initialiased variables was discussed earlier. Clearly copying the input character string to whatever location tstr happened to point to, has overwritten something important or has attempted to access a memory location not available to the program. Occassionally this error will not cause program failure because "tstr" happens to point to somewhere relatively safe and the program has only been tested with strings that were not long enough to cause damage when copied to whatever place "tstr" pointed to.

Exercises

The Pre-Processor and standard libraries - The #define directive

Chapter chap8 section 9

The **#define** preprocessor directive is used to associate a particular string of characters with a replacement string. The operation of the preprocessor is then very similar to an editor performing a global substitution. The basic syntax is

```
#define <identifier> <replacement>
```

The replacement can be any sequence of characters and is terminated by the end of line. The normal C language rules apply to the formation of identifiers. This is particularly useful for referring to frequently used items in a program, for example the size of a buffer. The following variant of the line reversing program illustrates the point.

```
#include        <stdio.h>
#define LINMAX  80

main()
{
        char    inbuf[LINMAX];
        int     c;
        while(1)
        {
                int     i=0;
                while((c=getchar())!='\n' && c!=EOF)
                        if(i!=LINMAX)inbuf[i++]=c;
                if(c==EOF) break;
                if(i--)while(i>=0)putchar(inbuf[i--]);
                putchar('\n');
        }
}
```

The virtue of this approach is that it is possible to change the size of the buffer by simply altering the single line that defines the symbol **LINMAX ,** once this change has been made the program can be re-compiled and the preprocessor will replace every occurrence of the symbol *LINMAX* by the new actual value. In a larger program with several references to *LINMAX* the advantages of this approach are considerable.

If the actual buffer size were quoted then it would be necessary to track down every

reference and alter it when the buffer size was changed. It would be all to easy to overlook one or, possibly worse, change an instance of a number that happened to look the same but was completely unrelated to the buffer size.

It is good programming practice to use upper case letters for identifiers defined by *#define* directives, this makes them stand out in the actual C code and makes it clearer to the programmer that they are not normal variables. Remember that the preprocessor operates purely by textual substitution so the error in

```
#define MAXCNT  100
.
.
.
MAXCNT = 50;
```

is not immediately obvious unless you realise what **MAXCNT** actually stands for.

Identifiers defined using *#define* in this fashion are sometimes called **manifest constants** although this notation does not really reflect the full capabilities of *#define* as the example below will make clear. Occasionally they are called **object-like macros .**

It is quite possible to use *#define* directives for all sorts of purposes as the following program shows.

```
#include        <stdio.h>
#define LINMAX  80

#define BEGIN   {
#define END     }
#define AND     &&
#define FOREVER while(1)
#define NEWLINE '\n'

main()
BEGIN
        char    inbuf[LINMAX];
        int     c;
        FOREVER
        BEGIN
                int     i=0;
                while((c=getchar())!=NEWLINE AND c!=EOF)
                        if(i!=LINMAX)inbuf[i++]=c;
                if(c==EOF) break;
                if(i--)while(i>=0)putchar(inbuf[i--]);
```

```
                putchar('\n');
        END
END
```

The program now looks quite unlike a C program and bears a marginal resemblance to certain other programming languages. This practice is to be strongly discouraged even though it does demonstrate the text substitution nature of the preprocessor.

Function Like Macros

Loops and Conditions - Introduction

Chapter chap5 section 1

So far all the programs we have written have been straight-line programs, this means that they have consisted of a sequence of statements to be executed one after the other. In this chapter we will see how to write programs that do different things depending on conditions detected by the program and how to write programs that repeatedly execute a set of statements. Before we do this, however, we will need to learn how to detect conditions.

See Also

- Relational Operators
- Logical Operators
- The If and Else Statements
- The Dangling Else Problem
- Local Variables in Compound Statements
- The Equality and Assignment Operators
- The While, Break and Continue Statements
- The Do Statment
- The Trinary (:?) Operator
- Exercises

Loops and Conditions - Relational and Logical Operators

Chapter chap5 section 2

In the C programming language there a number of operators that allow us to construct expressions whose value depends on some condition between the values of the operands. The following set of four operators, known as the **relational operators,** are basic examples.

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
<	Less than

These may be used in expressions of the form

```
<value1> <relational operator> <value2>
```

Such integer valued expressions, known as **relational expressions,** have either the value 0 or the value 1. The expression has the value 1 if the relationship between the values is true and 0 if the relationship between the operators is false. They are illustrated by the following program.

```
main()
{
        int     x = 3;
        int     y = 4;
        printf("Value of %d > %d is %d\n",x,y,x>y);
        printf("Value of %d >= %d is %d\n",x,y,x>=y);
        printf("Value of %d <= %d is %d\n",x,y,x<=y);
        printf("Value of %d < %d is %d\n",x,y,x<y);
}
```

which produced the following output

```
Value of 3 > 4 is 0
Value of 3 >= 4 is 0
Value of 3 <= 4 is 1
Value of 3 < 4 is 1
```

If you are familiar with other programming languages you might find this rather surprising. The C programming language does not have such things as logical or Boolean data types.

As well as the relational operators, the C programming language also provides two **equality operators.**

Operator	Meaning
==	Is equal to
!=	Is not equal to

Expressions involving these operators evaluate to 1 or 0 in exactly the same way as the relational operators described earlier.

If a relational or equality operator is associated with operators of different types then the usual arithmetic conversions take place. This can, occasionally, produce surprising results as is shown in the program below.

```
main()
{
        int     x = -4;
        unsigned        int     y = 3;
        printf("Value of %u<%d is %d\n",y,x,y<x);
}
```

producing the output

```
Value of 3<-4 is 1
```

This output says that 3 is less than -4, which is apparently wrong. What has happened is that the "<" operator has two operands, one of type *int* and the other of type *unsigned int* , in this case the arithmetic conversion rules require the signed operand to be converted to unsigned by bit pattern preservation before the comparison. The normal representation of negative numbers involves setting high order bits to indicate this fact so unsigned comparison will see the value with the high order bits set as the larger. This can be even more confusing with the *char* data type, since the signed or unsigned nature of *char* is not always obvious.

The logical operators

Loops and Conditions - The logical operators

Chapter chap5 section 3 There are three operators that are commonly used to combine expressions involving relational operators. These are

Operator	Meaning
&&	and
||	or
!	not

The ! operator is unary, the others are binary. The operators "&&" and "||" should not be confused with the bitwise operators "&" and "|". The rules for evaluating expressions involving the logical operators are.

- **&&** If either of the values are zero the value of the expression is zero, otherwise the value of the expression is 1. If the left hand value is zero, then the right hand value is not considered.
- || If both of the values are zero then the value of the expression is 0 otherwise the value of the expression is 1. If the left hand value is non-zero, then the right hand value is not considered.
- **!** If this operator is applied to a non-zero value then the value is zero, if it is applied to a zero value, the value is 1.

The effects of these operators is summarised in the following table.

Operand 1	Operand 2	op1 || op2	op1 && op2	! op1
0	0	0	0	1
0	non-zero	1	0	1
non-zero	0	1	0	0
non-zero	non-zero	1	1	0

Some further examples of expressions involving these operators are now in order.

```
main()
{
        int     x=1,y=2,z=3;
        int     p,q;
        p = (x>y) && (z<y);     /* False i.e. 0 */
        q = (y>x) || (y>z);     /* True i.e. 1 */
```

```
        printf(" %d && %d = %d\n",p,q,p&&q);
        printf(" %d || %d = %d\n",p,q,p||q);
        /* Can mix "logical" values and arithmetic */
        printf(" %d && %d = %d\n",x,q,x&&q);
        printf(" %d || %d = %d\n",p,y,p||y);
        /* Exercise the NOT operator */
        printf("    ! %d = %d\n",p,!p);
        printf("    ! %d = %d\n",q,!q);
        /* NOT operator applied to arithmetic */
        printf("    ! %d = %d\n",z,!z);
}
```

producing the output

```
 0 && 1 = 0
 0 || 1 = 1
 1 && 1 = 1
 0 || 2 = 1
    ! 0 = 1
    ! 1 = 0
    ! 3 = 0
```

These are, of course, very simple examples. Before considering more elaborate cases we must note the precedence and associativity of these new operators. Here is yet another table of operator precedences.

Operator	Associativity
() ++(postfix) --(postfix)	Left to Right
++(prefix) --(prefix) ~ (type) +(unary) -(unary) !	Left to Right
* / %	Left to Right
+ -	Left to Right
<< >>	Left to Right
< <= => >	Left to Right
== !=	Left to Right
&	Left to Right
^	Left to Right
|	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to Left

= += *= <<= >>= /= %= -= &= |= ^=	Right to Left

The ?: operator will be described later in this chapter. The next example illustrates the sometimes surprising effects of associativity.

```
main()
{
        int     a = 9;
        int     b = 8;
        int     c = 7;
        printf("Value of %d>%d>%d is %d\n",a,b,c,a>b>c);
}
```

producing the output

```
Value of 9>8>7 is 0
```

This is, initially, rather surprising because 0 means that the relation is false and yet the relation appears to be true. However if it is remembered that an expression like this is evaluated left to right then the result makes sense. The first step is the evaluation of

```
9>8
```

this is clearly true and the value of the expression is 1. The next, and final step, is the evaluation of

```
1>7
```

this is, equally clearly false, and has the value 0.

If you want to determine whether x lies between the values a and b you must write

```
b > x && x > a
```

and not

```
b > x > a
```

There is no need for parentheses because ">" has a higher precedence than "&&". Note that there is nothing wrong with writing "b>x>a", it will be accepted by the compiler, it simply doesn't do what you might expect it to do.

You may have noted the odd rules about ignoring the right hand value when evaluating "&&" and "||" expressions. This is sometimes called **fast track evaluation.** It is clearly sensible not to evaluate a, possibly complex, part of an expression when the value of the complete expression can already be determined. It

is also useful in avoiding possible problems. Consider the following program, called relop4, that might make use of the fact that a floating point number has either the value zero or a reciprocal less than 0.75.

```
main()
{
        double  x;
        int     flag;
        printf("Enter a number ");
        scanf("%lf",&x);
        flag = ( x==0 || 1.0/x < 0.75 );
        printf("Flag value = %d\n",flag);
}
```

A typical dialogue is shown below

```
$ relop4
Enter a floating point number 1.5
Flag value is 1
$ relop4
Enter a floating point number 0.75
Flag value is 0
$ relop4
Enter a floating point number 0
Flag value is 1
```

The important point here is the behaviour of the program for the input value zero. If both operands of the "&&" operator were evaluated before determining the value of the expression then the program would fail when it attempted to compute 1.0/x.

You might think, quite correctly, that the operator "&&" is commutative and it doesn't make any difference which way round you write the operands. If you consider the effects of reversing the order of operands in the program, it is fairly obvious that the behaviour with the input value 0 would be quite different. If the operands were reversed then the first step in determining the value to be assigned to "flag" is to determine the value of

```
1.0/x < 0.75
```

If the input value were 0, then the program would crash when it attempted to determine the value of "1.0/x".

If and Else statements

Loops and Conditions - The trinary (?:) operator

Chapter chap5 section 10

Consider the following code which sets the variable x to the maximum of a and b.

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

This looks innocent and straightforward but if the main requirement is to deliver the maximum of a and b as a value within an expression it is remarkably clumsy requiring, amongst other things, the use of an extra intermediate variable "x" to convey the result of the statement execution to the expression evaluation part of the code.

The C programming language offers an alternative via the "?:" operator. This is a trinary operator which means that three operands are associated with the operator. The syntax is

```
<expression1> ? <expression2> : <expression3>
```

The value of the expression is the value of expression2 if the value of expression1 is non-zero and the value of expression3 if the value of expression1 is zero. The setting of x to the maximum of a and b can now be achieved by the following code

```
x=a>b?a:b
```

Here is a very simple example.

```
main()
{
        int     a,b;
        printf("Enter 2 numbers ");
        scanf("%d%d",&a,&b);
        printf("Maximum is %d\n",a>b?a:b);
        printf("Minimum is %d\n",a>b?b:a);
}
```

A typical dialogue with this program, called *trin*, is shown below.

```
$ trin
```

```
Enter 2 numbers 3 5
Maximum is 5
Minimum is 3
$ trin
Enter 2 numbers 4 4
Maximum is 4
Minimum is 4
$
```

Another simple and useful example of the use of the trinary operator is shown by the following program.

```
main()
{
        int     i=0;
        while(i<3)
        {
                printf("%d error%c\n",i,i==1?' ':'s');
                i++;
        }
}
```

producing the output.

```
0 errors
1 error
2 errors
```

Notice the neat way in which the final "s" is suppressed on the output.

Exercises

Loops and Conditions - If and Else statements

Chapter chap5 section 4

It is now time to consider some new types of statement. The first of these are the **if-statement** and the **else-statement .** The syntax of an *if-statement* is

```
if (<expression>) <statement>
```

and the meaning is that if the expression has a non-zero value then the statement included within the *if-statement* , sometimes called the **controlled statement,** is executed. If the value of the expression is zero then the controlled statement is not executed.

An *if-statement* may be followed immediately by an *else-statement* . The syntax of an *else-statement* is

```
else <statement>
```

and the meaning is that the controlled statement is executed if the controlling expression of the immediately preceding *if-statement* had the value zero.

A simple example program called if1 is in order.

```
main()
{
        int     x;
        printf("Enter a number ");
        scanf("%d",&x);
        if(x%2 == 1)    printf("%d is odd\n",x);
        else            printf("%d is even\n",x);
}
```

and a typical dialogue is

```
$ if1
Enter a number 27
27 is odd
$ if1
Enter a number 0
0 is even
$ if1
Enter a number 1000
1000 is even
```

To understand the behaviour of this program it is only necessary to remember what the "%" operator does. The value of the expression "x%2" is the remainder when x is divided by 2, i.e. 0 when x is even and 1 when x is odd, so the value of "x%2 == 1" is 1 if x is odd and 0 if x is even.

It is quite common practice to omit the "== 1" in the condition in cases such as this since the value of "x%2" is 0 or 1 anyway.

The next example uses the relational and logical operators. It reads in three numbers and determines whether the middle number lies between the other two. Note how the controlled statements are indented for greater readability, this is common practice. This program is called if2.

```
main()
{
        int     p,q,r;
        printf("Enter three numbers ");
        scanf("%d%d%d",&p,&q,&r);
        if(( p<q && q<r ) || ( p>q && q>r))
                printf("%d lies between %d and %d\n",q,p,r);
        else
                printf("%d does not lie between %d and %d\n",
                                q,p,r);
}
```

and a typical dialogue is shown below

```
$ if2
Enter three numbers 1 2 3
 2 lies between 1 and 3
$ if2
Enter three numbers 3 2 1
 2 lies between 3 and 1
$ if2
Enter three numbers 10 10 11
 10 does not lie between 10 and 11
```

The expression associated with the "if" could have been written

```
p<q && q<r || p>q && q>r
```

The parentheses were not essential because "&&" has a higher precedence than "||", however the human reader may well have forgotten this and the extra typing is a small price to pay for greater readability and clarity.

The controlled statement associated with an *if-statement* may be any valid statement including another *if-statement* , however it can only be a single statement. In the following program called if3 the programmer has attempted to associate two statements with the *if* .

```
main()
{
        int     x;
        printf("Enter a number ");
        scanf("%d",&x);
        if(x%3 == 0)
                printf("The number %d",x);
                printf("is divisible by 3\n");
}
```

The program is intended to report whether the input number is divisible by 3. Remember that the value of the expression x%3 is non-zero if x is not divisible by 3. What actually happened is shown below.

```
$ if3
Enter a number 27
The number 27 is divisible by three
$ if3
Enter a number 10
```

```
is divisible by three
```

The problem here is, of course, that the final *printf()* statement is not controlled by the *if* and is, in fact, the statement after the *if-statement* so it is always executed. What is required is some way of grouping statements together in a single "super" statement. C provides just such a mechanism known as a **compound statement.** A compound statement is simply a list of ordinary statements enclosed in braces, i.e. {..}. A compound statement can be used anywhere where the ANSI standard requires a statement. The correct version of the if3 program, called if4, is shown below.

```
main()
{
        int     x;
        printf("Enter a number ");
        scanf("%d",&x);
        if(x%3 == 0)
        {
                printf("The number %d",x);
                printf("is divisible by 3\n");
        }
}
```

and a typical dialogue

```
$ if4
Enter a number 27
The number 27 is divisible by three
$ if4
Enter a number 10
$
```

The expression associated with the "if" could have been written

```
!(x%3)
```

the parentheses being necessary since the ! operator has a higher precedence than the % operator. A further, slightly more complex example is shown below. This program is called if5.

```
main()
{
        double  x;
        printf("Enter a number ");
        scanf("%lf",&x);
        if(x > 0.0)
        {
                printf("%10.5lf is positive\n",x);
                printf("%10.5lf is the reciprocal\n",1.0/x);
        }
        else
        {
                if (x == 0.0)
                {
                        printf("Zero - no reciprocal\n");
                }
                else
```

```
                {
                        printf("%10.5lf is negative\n",x);
                        printf("%10.5lf is the reciprocal\n",1.0/x);
                }
        }
}
```

and a typical dialogue

```
$ if5
Enter a number 3.5
A positive number It's reciprocal is    0.28571
$ if5
Enter a number 0.0
Zero - no reciprocal
$ if5
Enter a number -2.89
A negative number It's reciprocal is   -0.34602
```

Notice how the *if* and *else* keywords and the { and } symbols marking out the controlled compound statements are all lined up. This is highly desirable and makes understanding complex programs much easier for the human reader. As in all examples in these notes the indentation has been achieved using TABs in the source file.

Some programmers prefer to lay this program out in a slightly different way. The initial "{" appears on the same line as the condition, the statements within the controlled compound statement are indented and the final "}" is lined up with the *if* keyword.

```
main()
{
        double  x;
        printf("Enter a number ");
        scanf("%lf",&x);
        if(x > 0.0) {
                printf("%10.5lf is positive\n",x);
                printf("%10.5lf is the reciprocal\n",1.0/x);
        } else {
                if (x == 0.0) {
                        printf("Zero - no reciprocal\n");
                } else {
                        printf("%10.5lf is negative\n",x);
                        printf("%10.5lf is the reciprocal\n",1.0/x);
                }
        }
}
```

There are some other conventional layouts, consult any reputable textbook to see examples. The choice of layout is left to personal taste and local standards. However it is important to be consistent in layout once you have decided what convention to adopt.

In the both the previous examples there was a compound statement associated with the condition "x==0.0". The list of statements within the {..} consisted of a single statement. Under these circumstances there is no need for the braces and the following is perfectly correct. The program could be further shortened and simplified by combining successive *printf()* s into a single *printf()*.

```
main()
{
        double  x;
        printf("Enter a number ");
        scanf("%lf",&x);
        if(x > 0.0)
        {
                printf("%10.5lf is positive\n",x);
                printf("%10.5lf is the reciprocal\n",1.0/x);
        }
        else
        {
                if (x == 0.0)
                        printf("Zero - no reciprocal\n");
                else
                {
                        printf("%10.5lf is negative\n",x);
                        printf("%10.5lf is the reciprocal\n",1.0/x);
                }
        }
}
```

A more interesting example of the use of multiple *if* statements is illustrated by the following example. This program reads in a simple expression with a very restricted format and prints out its value.

```
main()
{
        int     n1,n2;
        int     val;
        char    op;
        printf("Enter a simple expression ");
        scanf("%d%c%d",&n1,&op,&n2);
        if(op == '+')
                val = n1 + n2;
        else if(op == '-')
                val = n1 - n2;
        else if(op == '/')
                val = n1 / n2;
        else if(op == '*')
                val = n1 * n2;
        else
        {
                printf("?? operator %c\n",op);
                exit(1);
        }
        printf("%d%c%d = %d\n",n1,op,n2);
}
```

A typical dialogue is shown below.

```
$ if6
Enter a simple expression 2+2
```

```
2+2 = 4
$ if6
Enter a simple expression 29-11
29-11 = 18
$ if6
Enter a simple expression 23*5
23*5 = 115
$ if6
Enter a simple expression 11%5
Unknown operator %
```

A particularly interesting point concerns the sequence of **else if** lines. To understand this, you should note that an *else* statement is associated with the immediately preceding *if* statement in spite of any impressions to the contrary given by program layout. If you're uncertain about this it is illustrative to rearrange the text of the program to conform to the layout described earlier.

Program execution proceeds by evaluating the relational expressions "c == '+'", "c == '-'", etc., one after the other until one has the value 1. The associated controlled statement is then executed and then the statement after the final *else* statement is executed.

```
main()
{
        int     n1,n2;
        int     val;
        char    op;
        printf("Enter a simple expression ");
        scanf("%d%c%d",&n1,&op,&n2);
        if(op == '+')
                val = n1 + n2;
        else    if(op == '-')
                        val = n1 - n2;
                else    if(op == '/')
                                val = n1 / n2;
                        else    if(op == '*')
                                        val = n1 * n2;
                                else
                                {
                                        printf("?? operator %c\n",op);
                                        exit(1);
                                }
        printf("%d%c%d = %d\n",n1,op,n2);
}
```

This program was obtained from the previous one simply by changing the indentation to show the structure more clearly. Either layout is correct and acceptable, if there are a large number of conditions the advantages of the first form are obvious.

A new library function, **exit(),** appears in this program. The effect of this library function is to terminate the program immediately and return to the host operating system. The integer parameter of *exit()* may be accessible to the host operating system as a program return or exit code. The details of this mechanism are, of course, host operating system dependant. Different values can be used to indicate successful or unsuccessful operation of the program. Returning 0 to indicate succesful operation and using non-zero integer values to indicate various errors is a common practice. If you don't include a call to *exit()* in your program, it simply "drops off the end", this is perfectly safe but does mean that the value returned to the

host operating system environment is indeterminate.

- The dangling else problem.
- equality and assignment operator confusion
- Local Variables in compound statements

Loops and Conditions - Exercises

Chapter chap5 section 11

1. Write a program that will read in 4 numbers and print out their average.
2. Write a program to read in a set of numbers and print out their average. The program will start by prompting the user for the number of numbers to be read in and will then prompt for the individual numbers with a prompt such as

   ```
   Enter Number 23
   ```

   to indicate to the user which data item is currently being entered. Do something special when prompting for the last number.

   Note that there is no need to store all the individual numbers, it is sufficient to maintain a running total.
3. Modify the previous program to print out the largest and smallest number read in as well as the average. Also change the prompt to show the number of numbers still to be entered.
4. Write a program to prompt the user for an integer and calculate the sum of all the integers up to and including the input value. Print out the result.
5. Modify the previous program to use floating point arithmetic to add up the reciprocals of all the integers up to and including the input value.
6. Further modify the previous program to print out a list of reciprocal sums for every integer up to and including the input value.

   I.e. print out the sum of the reciprocals of the integers up to and including 1, up to and including 2, up to and including 3 etc., etc.
7. Write a program to print out the integers from 40 to 127 in decimal, octal, hexadecimal and also print out the equivalent character.
8. Modify the previous program to list the values 4 to a line.
9. As a seasonal amusement some programmers like to print out a picture of a Chirstmas Tree looking like this.

```
  *
 ***
  *
 ***
*****
  *
```

```
   ***
  *****
 *******
    |
  ---+---
```

The tree consists of a series of tiers (three in this case) of increasing size. Write a program to produce such a display having prompted the user for the number of tiers.

You could try putting a few baubles on the tree (using o or O).

10. A year is a leap year if it is divisble by 4 unless it is a century year (one that ends in 00) in which case it has to be divisible by 400. Write a program to read in a year and report whether it is a leap year or not.

Loops and Conditions - The dangling else problem

Chapter chap5 section 5

The following program underlines the importance of understanding the relationship between *if* statements, *else* statements and the associated controlled statements.

```
main()
{
        int     a = 2;
        int     b = 2;
        if (a == 1)
                if (b == 2)
                        printf("a was 1 and b was 2\n");
        else
                printf("a wasn't 1\n");
}
```

When compiled and run this program did not produce any output. A properly laid out version of this program makes it clear what is actually going on.

```
main()
{
        int     a = 2;
        int     b = 2;
        if (a == 1)
                if (b == 2)
                        printf("a was 1 and b was 2\n");
                else
                        printf("a wasn't 1\n");
}
```

This shows something called the **dangling else** problem. With the program in its original form it is quite likely that the programmer thought the *else* statement

```
else
        printf("a wasn't 1\n");
```

would be associated with the first *if;* it wasn't. An *else* always associates with the immediately preceding *if* as the alternatively laid out version of the program makes quite clear. The reason for the complete absence of output is the fact that there is no *else* statement associated with the first *if* .

In order to achieve the effect that the programmer probably originally intended, it is

necessary to re-arrange the program in the following form.

```
main()
{
        int     a = 2;
        int     b = 2;
        if (a == 1)
        {
                if (b == 2) printf("a was 1 and b was 2\n");
        }
        else printf("a wasn't 1\n");
}
```

Local Variables in compound statements

Loops and Conditions - The equality and assignment operators

Chapter chap5 section 7

A final point in this section concerns a common error made by newcomers to the C programming language who are familiar with other programming languages. It is remarkably easy to write

```
if ( x = 5 ) ...
```

when you should have written

```
if ( x == 5 ) ...
```

both are perfectly valid code. The first has the effect of assigning the value 5 to x, the value of the expression "x = 5" is, of course, 5 and the statement associated with the *if* will always be executed. The second compares the value of x with 5 and the action taken depends on the value of x.

Many authors have criticised this aspect of the C language. It could have been avoided had more symbols been available to the designers. However C already used every available symbol except $ and ` (back quote) in the normal printing ASCII character set.

The while, break and continue statements.

Loops and Conditions - Local variables in compound statements

Chapter chap5 section 6 You might have noticed that the "body" of all the C programs we have seen so far consisted of a single compound statement that included declarations within the compound statement as well as executable statements. Declarations can always be included within compound statements as is shown by this program.

```
main()
{
        double  x;
        int     flag=0;
        printf("Enter a number ");
        scanf("%lf",&x);
        if(x > 1.5)
        {
                int     flag = 1;
                printf("flag = %d, x = %10.5lf\n",flag,x);
        }
        else
        {
                int     flag = -1;
                printf("flag = %d, x = %10.5lf\n",flag,x);
        }
        printf("flag = %d, x = %10.5lf\n",flag,x);
}
```

A typical dialogue with the program called if7 is shown below

```
$ if7
Enter a number 1.0
flag = -1, x =    1.00000
flag = 0, x =    1.00000
$ if7
Enter a number 2.0
flag = 1, x =    2.00000
flag = 0, x =    2.00000
```

There are 3 distinct and separate variables called **flag** in this program, the first is declared at the start of the program and the other two are declared within the compound statements associated with the *if* and *else* . Changing one of the flag variables within a compound statement has no effect on the "flag" declared at the start of the program. Once the path of execution enters the compound statement the "flag" declared at the start of the program is effectively **hidden** from view. The flag declared within the compound statement ceases to exist once the path of execution leaves the compound statement. The set of statements

where a particular variable definition is effective is called the **scope** of the variable.

The use of the same variable name for multiple different variables in this way is **not** good programming style. It is easy to forget which instance you are actually referring to, however the possibility of creating extra variables of purely local scope is sometimes useful as the following program illustrates.

```
main()
{
        double  x;
        printf("Enter a number ");
        scanf("%lf",&x);
        {  /* this compund statement can be commented out */
                double  a; /* scope is this block ONLY */
                if(x < 0)       a = -x;
                else            a = x;
                printf("Debug - absolute value %10.5lf\n",a);
        }
        printf("Reciprocal of %10.5lf is %10.5lf\n",x,1/x);
}
```

The compound statement that includes the variable "a" is used purely for debugging. Once the programmer is happy this block can be commented out (having first removes the comments inside the block !!). The variable "a" is used solely for debugging and the removal of the block containing also removes the declaration so that the main code is not cluttered up with declarations of variables that are used purely for debugging. This is particularly useful when the debugging code block uses a loop, the "loop control variable" only needs to be declared within the block this avoiding any interference with similarly named variables in the main code.

Equality and Assignment operators

Loops and Conditions - The while, break and continue statements

Chapter chap5 section 8

The next statement to be described in this chapter is the **while** statement. This provides a means for **repeated execution** of controlled statements. The basic form is

```
while ( <expression> ) statement
```

and the meaning is that the expression is evaluated and if the value of the expression is non-zero then the controlled statement is executed otherwise the statement after the *while* statement is executed. An example program is in order.

```
main()
{
        int     i = 0;
        while(i < 10)   /* OK to continue ? */
        {
                printf("%d %2d %3d\n",i,i*i,i*i*i);
                i++;
        }
}
```

The output it produced is

```
0  0   0
1  1   1
2  4   8
3  9  27
4 16  64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
```

The behaviour should be fairly obvious. This program is a **looping** or **iterative** program. When the program starts "i" has the value 0. The first value of the expression

```
i<10
```

is 1 since the value of "i" (0) is clearly less than 10. This continues to be the case until the expression

```
i<10
```

is evaluated after the execution of the *printf()* with i having the value 9. At this stage the value of "i" is 10 and the value of the controlling expression is now zero so the controlled *printf()* is not executed and the path of execution drops through to whatever follows, in this case the end of the program.

Of course the statement controlled by the *while* can be a compound statement and may include further while statements.

The following example shows **nested while loops** being used to print out multiplication tables.

```
main()
{
        int i=1,j;
        while(i <= 12)  /* goes "down" page */
        {
                j = 1;
                while(j <= 12) /* goes "across" page */
                {
                        printf(" %3d",i*j);
                        j++;
                }
                printf("\n");   /* end of line */
                i++;
        }
}
```

as follows

```
  1   2   3   4   5   6   7   8   9  10  11  12
  2   4   6   8  10  12  14  16  18  20  22  24
  3   6   9  12  15  18  21  24  27  30  33  36
  4   8  12  16  20  24  28  32  36  40  44  48
  5  10  15  20  25  30  35  40  45  50  55  60
  6  12  18  24  30  36  42  48  54  60  66  72
  7  14  21  28  35  42  49  56  63  70  77  84
  8  16  24  32  40  48  56  64  72  80  88  96
  9  18  27  36  45  54  63  72  81  90  99 108
 10  20  30  40  50  60  70  80  90 100 110 120
 11  22  33  44  55  66  77  88  99 110 121 132
 12  24  36  48  60  72  84  96 108 120 132 144
```

Input operations can also be included within a loop. An interactive program using a *while* statement and illustrating this is shown below.

```
main()
{
        double  x = 1.0;        /* non-zero to avoid immediate exit */
        printf("Program to display reciprocals, "
               "squares and cubes\nEnter 0 to"
               " terminate\n");
        while(!(x == 0))        /* i.e. while valid data */
        {
                printf("Enter a number ");
                scanf("%lf",&x);
                if(!(x==0))     /* valid data ?? */
                {
                        printf("Reciprocal = %10.5lf\n",1.0/x);
                        printf("Square     = %10.5lf\n",x*x);
                        printf("Cube       = %10.5lf\n",x*x*x);
                }
        }
}
```

A typical dialogue is shown below

```
Program to display reciprocals, squares and cubes
Enter 0 to terminate
Enter a number 4
Reciprocal =    0.25000
Square     =   16.00000
Cube       =   64.00000
Enter a number 0.25
Reciprocal =    4.00000
Square     =    0.06250
Cube       =    0.01562
Enter a number -3
Reciprocal =   -0.33333
Square     =    9.00000
Cube       =  -27.00000
Enter a number 0
```

Occasionally there is, apparently, no need for the controlled statement associated with a *while* , everything you want to do can be done via **side-effects** of the evaluation of the expression. Under these circumstances there must still be a statement after the expression but it will be a **null** statement that is represented simply by a semi-colon. An example is shown below.

```
main()
{
        int     i=2;
        while(printf("%d\n",i++)!=3);
}
```

This produced the output

```
2
3
4
5
6
7
8
9
10
```

To understand what happened you need to know the little known fact that the value of the *printf()* function is the number of characters actually printed which may include a new-line. It is also worth noting that the production of output by the function *printf()* is a **side-effect .** Another simple example of a *while* statement whose controlled statement is a null statement is a simple **spin delay.**

```
int     delay = 10000;
while(delay--);
```

Another common use of the *while* statement is the **forever loop** which might look like

```
while(1)
{
```

```
        .
        .
        .
}
```

The controlled statement is executed until the loop is **broken** by some means. The simplest and commonest way of doing this is by using a **break** statement. A *break* statement consists simply of the word break followed by a semi-colon. Its effect is to cause immediate exit from the enclosing *while* statement. (It has the same effect when used with **do** and **for** statements, these will be described in due course.) It provides a means for writing a simpler and tidier version of the interactive program that appeared earlier in these notes. The revised version is.

```
main()
{
        double  x;
        printf("Program to display reciprocals, "
               "squares and cubes\nEnter 0 to"
               " terminate\n");
        while(1)
        {
                printf("Enter a number ");
                scanf("%lf",&x);
                if(x == 0) break;    /* end of valid data ? */
                printf("Reciprocal = %10.5lf\n",1.0/x);
                printf("Square     = %10.5lf\n",x*x);
                printf("Cube       = %10.5lf\n",x*x*x);
        }
}
```

The condition "x==0.0" might be called the **termination condition** as far the *while* controlled statement is concerned. The use of *break* avoids the need for testing for the condition twice with one of the tests being logically inverted and used in conjunction with an *if* statement to prevent the rest of controlled statement being executed when the termination condition is detected.

Closely associated with the *break* statement is the **continue** statement which like the *break* causes the rest of the loop to be skipped but unlike the *break* does not exit the loop. Its use is illustrated in the following example which sums all the numbers up to and excluding a user entered value excluding the numbers that are divisible by 4.

```
main()
{
        int     max;
        int     sum=0,count=0;
        int     i=0;
        printf("Enter largest number ");
        scanf("%d",&max);
        while(1)
        {
                i++;
                if(i == max) break;
                if(i%4 == 0) continue;
                count++;
                sum += i;
```

```
        }
        printf("There are %d numbers not divisible by 4"
               " and less than %d\nTheir total is %d\n",
                       count,max,sum);
}
```

This produced the output

```
bash$ while5
Enter largest number 15
There are 11 numbers not divisible by 4 and less than 15
Their total is 81
bash$ while5
Enter largest number 16
There are 12 numbers not divisible by 4 and less than 16
Their total is 96
```

The do statement

Loops and Conditions - The do statement

Chapter chap5 section 9

The *while* statement may be described as **test-before-execute .** If the controlling expression is initially zero then the controlled statement is never executed. There is an alternative version that is sometimes useful. This is the **do** statement which may be described as **execute-before-test .** This is sometimes called a **one-trip-loop** referring to the fact that the loop "body" is always executed at least once. The syntax is

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

The controlled statement is always executed at least once. It is sometimes useful in interactive programs where the user must provide some input and there is no sensible initial setting.

The trinary (?:) operator

Switch and For Statements, Command Line and File Handling - The for statement

Chapter chap9 section 2

The **for** statement provides an alternative way of programming loops that, up to now, we have programmed using *while* or *do...while* statements. The basic syntax is simply

```
for(exp1;exp2;exp3) statement
```

The meaning is that first the expression *exp1* is evaluated, then provided expression *exp2* has a non-zero value the statement is executed and expression *exp3* is evaluated. Unlike similar constructs in other programming languages there is no restriction on the nature of the three expressions that are associated with the *for* statement. If the expression *exp2* has a zero value then the flow of control passes through to the statement after the *for* statement. The basic *for* statement is equivalent to writing

```
        exp1;
        while(exp2)
        {
                statement
                exp3;
        }
```

and is defined in this fashion by the ANSI standard. The *for* statement provides a convenient way of writing this kind of loop. A simple example is in order.

```
main()
{
        int     i;
        for(i=0;i<10;i++)
                printf("%d %2d %4d\n",i,i*i,i*i*i);
}
```

producing the output

```
0  0    0
1  1    1
2  4    8
```

```
3  9   27
4 16   64
5 25  125
6 36  216
7 49  343
8 64  512
9 81  729
```

The *break* and *continue* statements may be used within *for* statements in exactly the same way as they are used within *while* and *do...while* statements. Any one, or all three, of the expressions associated with the *for* statement may be omitted but the associated semi-colons must **not** be omitted.

```
for(;;) statement
```

is another way of writing a **repeat..forever** loop. The controlled statement may, of course, be compound.

It is occassionally useful to incorporate several expressions in the expression positions of a *for* statement. To do this the **comma** operator is used. This has the lowest precedence of any C operator. The syntax is simple

```
expression-1, expression-2
```

is an expression and its value is simply the value of expression-2, the value of expression-1 is discarded. It should not be confused with the listing of expressions as functional parameters. The comma operator is illustrated by the following program

```
#include        <math.h>
main()
{
        int     i;
        double  x;
        for(i=0,x=0;i<10;i++,x+=0.5)
                printf("%d %10.7lf\n",i,sqrt(x));
}
```

which produced the output

```
0  0.0000000
1  0.7071068
2  1.0000000
3  1.2247449
4  1.4142136
5  1.5811388
```

```
6  1.7320508
7  1.8708287
8  2.0000000
9  2.1213203
```

The switch statement

Switch and For Statements, Command Line and File Handling - The switch statement

Chapter chap9 section 3

The **switch** statement provides a very useful alternative to multiple *if* statements. It is used in conjunction with the **case** and **default** statements. The syntax is

```
switch(integral expression) statement
```

the controlled statement, known as the **switch body** will consist of a sequence of *case* statements. The syntax of a *case* statement is

```
case constant-integral-expression : statement
```

this is really nothing more than a labelled statement. The meaning of all this is that flow of control passes to the statement whose case label matches the value of the switch expression. The flow of control then continues from that point until a *break* is encountered or the end of the switch body is encountered. A *break* statement takes control out of the switch body.

If none of the case labels match the value of the switch expression then no part of the code in the switch body is executed unless a *default* statement appears within the switch body, this acts as a "catch-all" label when no other case label matches.

An example is in order. This is a variant of a program that has already been seen. It reads in simple expressions and evaluates them.

```
#include        <stdio.h>
main()
{
        int     n1,n2;
        char    c;
        char    inbuf[30];
        while(1)
        {
                printf("Enter Expression ");
                if(gets(inbuf) == NULL) break;
                sscanf(inbuf,"%d%c%d",&n1,&c,&n2);
                switch(c)
                {
                        case '+' :
                                printf("%d\n",n1+n2);
                                break;
                        case '-' :
                                printf("%d\n",n1-n2);
                                break;
                        case '*' :
                                printf("%d\n",n1*n2);
```

```
                        break;
                case '/' :
                        printf("%d\n",n1/n2);
                        break;
                default :
                        printf("Unknown operator %c\n",c);
                }
        }
}
```

A typical dialogue is shown below

```
Enter Expression 345+45
390
Enter Expression 212/6
35
Enter Expression 234-5
229
Enter Expression 234%4
Unknown operator %
```

Notice the frequent *break* statements in the switch body, these are necessary to avoid the **drop-through** between cases. Many people think this *drop-through* is annoying and the language would be better if a break from the switch body was implicit at the end of each *case* , however it can sometimes be useful as this example shows.

```
#include        <stdio.h>
main()
{
        int     c;
        int     dcnt = 0;       /* digits */
        int     wcnt = 0;       /* white space count */
        int     ocnt = 0;       /* others count */
        while((c=getchar())!=EOF)
                switch(c)
                {
                        case '0' :
                        case '1' :
                        case '2' :
                        case '3' :
                        case '4' :
                        case '5' :
                        case '6' :
                        case '7' :
                        case '8' :
                        case '9' :
                                dcnt++;
                                break;
                        case ' ' :
                        case '\t' :
```

```
                        case '\n' :
                                wcnt++;
                                break;
                        default :
                                ocnt++;
                }
        printf("%d digits\n%d white spaces"
                "\n%d others\n",dcnt,wcnt,ocnt);
}
```

When it consumed its own source the program produced the following output

```
13 digits
164 white spaces
336 others
```

A more elaborate use of switch statements is illustrated by the following program that extracts comments from C programs. It is not fooled by comment-like things within string constants but escaped double quotes within strings would cause the program to fail.

```
/*      A program to extract comments */
#include        <stdio.h>
#define LEAD    0       /* In normal text */
#define PSCOM   1       /* Possible start of comment */
#define INCOM   2       /* Processing Comment */
#define PECOM   3       /* Possible end of comment */
#define INSTR   4       /* In string constant */
main()
{
        int     c;      /* input character */
        int     state = LEAD;   /* current status */
        char    *dummy = "/* comment in string */";
        while((c=getchar())!=EOF)
        {
                switch(state)
                {
                case LEAD :
                        switch(c)
                        {
                                case '/' :
                                        state = PSCOM;
                                        break;
                                case '"' :
                                        state = INSTR;
                                        break;
                        }
                        break;
                case PSCOM :
                        switch(c)
                        {
                                case '*' :
```

```
                                state = INCOM;
                                break;
                        case '"' :
                                state = INSTR;
                                break;
                        default :
                                state = LEAD;
                                break;
                }
                break;
        case INCOM :
                switch(c)
                {
                        case '*' :
                                state = PECOM;
                                break;
                        default :
                                putchar(c);
                                break;
                }
                break;
        case PECOM :
                switch(c)
                {
                        case '/' :
                                state = LEAD;
                                putchar('\n');
                                break;
                        default :
                                state = INCOM;
                                putchar('*');
                                putchar(c);
                                break;
                }
                break;
        case INSTR :
                switch(c)
                {
                        case '"' :
                                state = LEAD;
                                break;
                        default :
                                break;
                }
                break;
        }
    }
}
```

When presented with its own source as input the program produced the following output.

```
       A program to extract comments
In normal text
Possible start of comment
Processing Comment
Possible end of comment
In string constant
input character
current status
```

This use of nested switches, whilst it makes for rather clumsy coding, stems from a very simple design technique known as a **state switch.** The whole operation of the program is described by the following transition table.

	LEAD	PSCOM	INCOM	PECOM	INSTR
/	PSCOM	LEAD	INCOM print /	LEAD print \n	INSTR
*	LEAD	INCOM	PECOM	INCOM print * *	INSTR
"	INSTR	INSTR	INCOM print "	INCOM print * "	LEAD
other	LEAD	LEAD	INCOM print ch	INCOM print * ch	INSTR

The table entries represent the new value of "state" and the action to be taken depending on the current input character (shown in the left hand column) and the current value of state (shown in the top row). In some circumstances as well as changing the value of state it is necessary to take some sort of action, in this case printing out a particular character or the current input character ("ch").

Command Line Arguments

Addresses, Pointers, Arrays and Strings - Introduction

Chapter chap6 section 1

In this chapter we shall see how to determine the actual addresses of variables and how to manipulate them. We will also see how the C programming language handles collections of variables known as aggregates or arrays and how to reference individual members of such a collection. Finally we will look at the special facilities for handling and manipulating a particular sort of collection of characters known as a string.

See Also

- Addresses
- Arrays and Aggregates
- MSDOS Memory Management
- Initialisation of Arrays and Aggregates
- More on Operator Precedence
- Strings
- String input using "s" conversion
- String input using scanset conversion
- String input using "c" conversion
- String input using the *gets()* function
- Processing strings
- Arrays of Strings
- *sprintf()* and *puts()*
- Library String handling functions
- Exercises

Multi-dimensional arrays are discussed later.

Addresses, Pointers, Arrays and Strings - Addresses

Chapter chap6 section 2

We have actually already seen how to obtain the **address** of a variable. This is done very simply by preceding the name of a variable with an ampersand **(&).** We have used this syntax with the parameters of the *scanf()* library function which required the addresses of the locations to receive the converted input values. It is rare that the actual addresses of variables are of any interest to the outside world, however it is possible to display them as this simple program demonstrates.

```
main()
{
        int     a,b;
        double  x;
        int     z;
        printf("Address a = %08x\n",&a);
        printf("Address b = %08x\n",&b);
        printf("Address x = %08x\n",&x);
        printf("Address z = %08x\n",&z);
}
```

The output produced by this program on the SUN Sparc Station was

```
Address a = f7fffc04
Address b = f7fffc00
Address x = f7fffbf8
Address z = f7fffbf4
```

and on the IBM 6150

```
Address a = 3fffe478
Address b = 3fffe47c
Address x = 3fffe480
Address z = 3fffe488
```

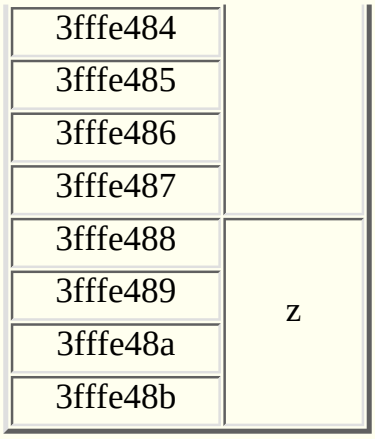
The differences in the results reflect the different ways the two machines allocate actual memory locations. It is interesting to note that the SUN Sparc Station allocates successive locations in reverse order whereas the IBM 6150 worked forwards. You may also note the different amount of memory associated with a floating point number. This program would give different results on different machines. The actual addresses are byte addresses, here expressed in hexadecimal notation.

The following diagrams show the memory layout on the SUN Sparc Station and the IBM 6150 respectively.

Byte Address	Memory Location
f7fffbf4	z
f7fffbf5	
f7fffbf6	
f7fffbf7	
f7fffbf8	x
f7fffbf9	
f7fffbfa	
f7fffbfb	
f7fffbfc	
f7fffbfd	
f7fffbfe	
f7fffbff	
f7fffc00	b
f7fffc01	
f7fffc02	
f7fffc03	
f7fffc04	a
f7fffc05	
f7fffc06	
f7fffc07	

The IBM 6150

Byte Address	Variable
3fffe478	a
3fffe479	
3fffe47a	
3fffe47b	
3fffe47c	b
3fffe47d	
3fffe47e	
3fffe47f	
3fffe480	x
3fffe481	
3fffe482	
3fffe483	



Addresses cannot be stored in any of the types of variable we have seen so far. It is tempting to think that an address occupies 32 bits, and this is very commonly the case on Unix based systems, and thus assume that addresses could be stored in *long int* s. However the internal storage of addresses on PC's, especially when running MSDOS, is a very different matter, requiring a combination of **segment numbers** and **offsets.** The use of hexadecimal conversion for the display of addresses is not strictly ANSI standard, particular implementations may display addresses in various other forms.

There are special variable types for storing addresses or **pointers** as they are often called. There are separate variable types for storing the addresses of each separate type of data object, the reasons for this will become clear in due course. The declaration of, for example, a variable suitable for storing the address of an *int* may be mixed in with declarations of variables of type *int* because the syntax is simply to precede the name with an asterisk (*****).

The declaration

```
int x,*y,z=0
```

declares three variables x,y and z. "x" and "z" are variables of *int* type whereas y is a variable of type **pointer to int .** You can only, meaningfully, assign the address of an *int* to a variable of type *pointer to int*, however you can perform certain arithmetic operations on a *pointer to int.* You can also perform the unary operation * on a variable (or expression) of type *pointer to int.* The effect of this operation is to obtain the **value** of the thing **pointed to.** A demonstration program is in order.

```
main()
{
        int     *x;     /* pointer */
        int     y = 2;
        int     z = 3;
        x = &y;
        printf("x points to a location holding %d\n",*x);
        x = &z;
        printf("x points to a location holding %d\n",*x);
        x++;
```

```
        printf("x points to a location holding %d\n",*x);
}
```

This produced the following output on a SUN Sparc Station

```
x points to a location holding 2
x points to a location holding 3
x points to a location holding 2
```

The most interesting point about this program is the effect of the operation "x++". One might be excused for imagining that this added 1 to the address stored in x and that this would cause problems since an earlier program suggested that successive *int* variables had addresses 4 units apart. This did not cause problems because the type of the variable "x" is *pointer to int* and incrementing a *pointer to int* is taken to mean alter the value to point to the next integer (i.e. add 4). Note also that the operation of the program depends on the reverse order of allocation of memory locations to integers used by the SUN Sparc Station compiler, the program would not work correctly on the IBM 6150. The following example further illustrates the **typed** nature of **pointer arithmetic.**

```
main()
{
        double  x = 3.5;
        double  y = 2.5;
        double  z = 1.5;
        double  *fptr = &z;
        printf("Address = %08x Value = %5.2lf\n",fptr,*fptr);
        fptr++;
        printf("Address = %08x Value = %5.2lf\n",fptr,*fptr);
        fptr++;
        printf("Address = %08x Value = %5.2lf\n",fptr,*fptr);
}
```

producing the output

```
Address = f7fffbf0 Value =  1.50
Address = f7fffbf8 Value =  2.50
Address = f7fffc00 Value =  3.50
```

Again the operation of this program depends on the SUN Sparc Station order of variable allocation. Note that the incrementation of "fptr" has increased its value by 8, the number of bytes occupied by a variable of type *double* .

As well as using the "*" unary operator to mean the value of the thing pointed to when it appears within an expression, the unary "*" operator may be used on the left hand side of an assignment expression to mean that the value is to be stored in the addressed location. The following program demonstrates the point.

```
main()
{
        int     x=4;
        int     y=2;
        int     z=5;
        int     *iptr;
        iptr = &z;
        *(iptr+2) = y+z;
        printf("x = %d, y = %d, z = %d\n",x,y,z);
}
```

producing the output

```
x = 7, y = 2, z = 5
```

here the expression

```
*(iptr+2)
```

appearing on the left hand side of the assignment operator means that the **assignment target** is two *int* locations past the *int* location whose address is stored in "iptr". With the SUN Sparc Station backwards way of doing things this means that the assignment is to the location "x".

A pointer variable can be initialised to point to something as part of its declaration in exactly the same way as an ordinary variable can be initialised as part of its declaration. It is important to realise that a pointer variable that has not been initialised will contain an **unpredictable value,** just like an uninitialised ordinary variable. This can cause some unpredictable results.

It cannot be too **strongly emphasised** that you cannot rely on variables being stored in any particular order in memory or there being any particular relationship between the addresses of successively declared variables.

Aggregates and Arrays

Addresses, Pointers, Arrays and Strings - Aggregates and Arrays

Chapter chap6 section 3

It is tedious to think of a whole series of names for a set of variables. The C language, like most other programming languages, provides a means of associating a single name with a set of objects of the same data type. Such a collection or set is called an **aggregate** or **array.** An aggregate is declared in a similar manner to an ordinary variable only the number of elements in the aggregate appears in square brackets after the name. The name of an aggregate is not the name of a variable but may be thought of as a pointer to or address of the first object in the aggregate. The following program shows the declaration of an aggregate and simple use of the aggregate to store data.

```
main()
{
        int     agg[5]; /* declares 5 ints */
        int     i = 0;
        int     sum = 0;
        do      /* loop to "fill" aggregate */
        {
                *(agg+i) = i*i;
                i++;
        } while(i<5);
        i = 0;
        do      /* loop to add up values */
        {
                sum += *(agg+i);
                i++;
        } while(i<5);
        printf("Sum = %d\n",sum);
}
```

It produced the following output, which is the sum of the squares of the integers from 0 to 4.

```
Sum = 30
```

The following program demonstrates the input of numbers into the members of an aggregate, and then lists the numbers in the reverse order of entry (i.e. last-in, first-out). The program is called "agg2"

```
main()
```

```
{
        int     agg[5];
        int     i = 0;
        while(i < 5)
        {
                printf("Enter number %d ",i+1);
                scanf("%ld",agg+i);
                i++;
        }
        i = 4;  /* i would be 5 after leaving input loop */
        printf("The numbers were ");
        while(i >= 0)
        {
                printf("%d ",*(agg+i));
                i--;
        }
        printf("\n");
}
```

A typical dialogue is shown below.

```
$ agg2
Enter number 1 45
Enter number 2 789
Enter number 3 2154
Enter number 4 32
Enter number 5 176
The numbers were 176 32 2154 789 45
$
```

There are one or two points of particular interest in this program. Note the expression "i+1" in the input prompt, this is done because users are generally happier to think of the first thing they enter as number 1 rather than number 0. Note also the expression

```
agg+i
```

in the parameter list for the *scanf()* library function. The value of this expression is, of course, the address of the location to receive the input value. Finally note the exclusion of "\n" from the format strings in all but the last of the 3 *printf()* function calls in the program.

In standard implementations of the C language there is no mechanism whatsoever to prevent your program running off the end (or the beginning) of an aggregate. The effects are almost always unpredictable and usually disastrous. The designers of the C language were of the opinion that such checks would be computationally expensive and, anyway, systems programmers ought to be sufficiently alert and intelligent to design

their programs to avoid such problems. These problems are sometimes known as **array bound violations.**

However an example of running off the end is in order. This has been deliberately concocted not to do anything disastrous, just mysterious.

```
main()
{
        int     x = 45;
        int     elems[20];
        int     y = 66;
        int     i = 0;
        printf("x = %d y = %d\n",x,y);
        while(i<=20)    /* fill aggregate */
        {
                *(elems+i) = i;
                i++;
        }
        /*      The following statement does
                not reference any part of the
                aggregate                */
        printf("x = %d y = %d\n",x,y);
}
```

producing the output

```
x = 45 y = 66
x = 20 y = 66
```

The output is puzzling since there is no reference to "x" or "y" between the two *printf()* statements in the program. The problem here is where the value of "i" is stored when "i" has the value 20. Since the aggregate "elems" has only got 20 elements their addresses will elems+0, elems+1, ... elems+19. The address elems+20 will **not** refer to a member of the aggregate but some other memory location, probably one lying either immediately before or immediately after the area of computer memory allocated to the aggregate. With the "backwards" memory allocation policy adopted by the SUN Sparc station compiler, it seems fairly reasonable to expect the address elems+20 to refer to the integer location immediately prior to the aggregate and this does seem to be what happened. The following listing shows what the program produced on the IBM 6150 which uses "forward" allocation.

```
x = 45 y = 66
x = 45 y = 20
```

Clearly the location after the aggregate has been changed in this case.

On both systems the program has changed the contents of a memory location without

referring to it in any conventional way. This unanticipated corruption arose, perhaps, because the programmer thought that a 20 member aggregate would have members bearing reference numbers 1,2,...20.

There is an alternative notation available and commonly used to refer to members of an aggregate. The notation

```
*(agg+i)
```

may be replaced by

```
agg[i]
```

where "agg" is the name of the aggregate and "i" is an integer. Most C compilers actually convert the second notation into the first notation internally. "[..]" is actually an operator, sometimes called the **indexing** operator or the **subscript** operator. The use of the subscript operator is demonstrated by the following version of the first program using aggregates. It should be compared with the earlier listing.

```
main()
{
        int     agg[5];
        int     i = 0;
        int     sum = 0;
        do
        {
                agg[i] = i*i;
                i++;
        } while(i < 5);
        i = 0;
        do
        {
                sum += agg[i];
                i++;
        } while(i < 5);
        printf("Sum = %d\n",sum);
}
```

It operated in exactly the same way. If you are familiar with other programming languages, you'll certainly be more comfortable with this notation, until you read the next paragraph anyway.

The "[..]" operator is commutative which means that either of the following are acceptable and both have exactly the same effect.

```
agg[i] i[agg]
```

This is hardly surprising since the compiler, as just explained, will convert them into

```
*(agg+i) *(i+agg)
```

respectively. The following program, which is yet another variation on the first aggregate example, further illustrates the relation between the "[..]" operator and address variables.

```
main()
{
        int     agg[5];
        int     *collection = agg;
        int     i=0;
        int     sum = 0;
        while(i<5)
        {
                collection[i] = i*i;
                i++;
        }
        i = 0;
        while(i<5) sum += collection[i++];
        printf("sum = %d\n",sum);
}
```

This looks slightly odd to programmers familiar with the concept of arrays as implemented in certain other programming languages, since *collection* is being used as an array although it was not declared as such. Reflecting on the fact that *collection[i]* is synonymous with **(collection+i)* should make it clear what is happening. It is better, for a variety of reasons, not to think about arrays when programming in C but to think about aggregates which have some rather different properties. Multi-dimensional arrays and aggregates are discussed later.

- MSDOS memory models
- Aggregate initialisation

Functions and storage organisation - Multi-Dimensional Aggregates and Arrays

Chapter chap7 section 11

It is not very surprising that since the C programming language allows aggregates of any data type or object it allows aggregates of aggregates. The syntax is also not very surprising unless you are thinking in terms of two dimensional arrays. A typical declaration is

```
char names[2][4];
```

This declares an aggregate of two elements, each element is an aggregate of four characters. Elements of such an object can be referred to in a fairly obvious fashion by expressions such as

```
names[1][2]
```

Aggregates of aggregates can be initialised using, not very surprisingly, an initialiser which consists of a set of initialisers. For example

```
char    names[2][4] = {
        {'f','r','e','d'},
        {'a','n','n','e'},
                };
```

An aggregate to hold the image of a character screen could be declared in the following fashion

```
char screen[25][80];
```

This clearly reflects a structure of 25 rows of 80 characters. Interestingly the complete image could be sent to the screen by code like

```
while(i<2000) sendchar(*(*screen+i++));
```

To understand this it is useful to think of "screen" as a variable of type *pointer-to-pointer-to-char* so the expression *screen is of type *pointer-to-char*, the addition of i++ to this value generates the address of a character and the "outer" asterisk operator yields the actual character. The following simple program illustrates what happens

```
main()
{
        char    oxo[4][2] = {
```

```
                {'a','b'},{'c','d'},{'e','f'},{'g','h'}
                        };
        int     i = 0;
        while(i<8) printf("%c",*(*oxo+i++));
        printf("\n");
}
```

producing the output

```
abcdefgh
```

The following example further illustrates some aspects of aggregates of aggregates.

```
char    x[3][5];        /* extern for zero initialisation */
main()
{
        char    (*y)[5]; /* what data type is y ? */
        int     i;
        strcpy(x[0],"abcd");
        strcpy(x[1],"efgh");
        strcpy(x[2],"ijkl");
        /* output first two strings */
        printf("%s %s\n",x[0],x[1]);
        y = &(x[1]);
        /* what does y+1 point to ? */
        printf("%s %s\n",y,y+1);
}
```

Producing the following output

```
abcd efgh
efgh ijkl
```

Here the variable x is the name of an aggregate of 3 aggregates of 5 characters each. The three calls to *strcpy()* put strings in each of the aggreagtes of 5 characters. Remember that every character within "x" is zero initially since it is of storage class extern. The variable "y" is of type pointer to an aggregate of 5 characters. Notice that the expression y+1 correctly points to the **next** aggregate of 5 characters after that containing the string "efgh". The need for the parentheses in the declaration of "y" arises because without it the declaration

```
char *y[5]
```

would give an aggregate of 5 *pointers to characters* which is not what was required. The analogy with the syntax for the declaration of pointers to functions should be noted. This type of operation is sometimes referred to as **array slicing**

There are obvious extensions to higher dimensioned arrays.

If you are used to the syntax for multi-dimensional arrays in older languages such as Pascal and Fortran you will certainly find the C notation confusing. Even more confusing is the fact that a C compiler will accept

```
x[2,5]
```

when you meant to write

```
x[2][5]
```

The previous expression is an example of the rarely used *comma* operator which is discussed later.

Some programmers may wonder whether 'C' two-dimensional arrays are stored row-first or column-first. The idea of of rows and columns relates to a graphical presentation of a two-dimensional array (or matrix) and a mathematical convention that associates the first index with the row and the second index with the column. Examination of the examples above shows that when "walking" through storage the first index changes most rapidly so 'C' uses column first storage.

Exercises

Addresses, Pointers, Arrays and Strings - MSDOS Memory Models

Chapter chap6 section 4

On computers using the Intel 80n86 series of processors there are two different ways of storing an address. These may be called **near** and **far .** A near address requires 16 bits of storage whereas a far address requires 32 bits of storage. An actual far address is constructed from the sum of a 16 bit segment number and a 16 bit offset with a 4 bit overlap so it is effectively 28 bits long. Near addresses use less memory and can be manipulated more quickly and more simply. The use of near addresses implies a severe limitation on the amount of data a program can handle and on the amount of code that can make up the program.

C compilers intended for use in such environments often have options to generate either type of address. The selection of such options is usually controlled from the compiler command line. The choices are usually called **memory models.**

The Microsoft C version 5.1 compiler typically offers a choice of 5 memory models known as small, medium, compact, large and huge. Their characterisitics are summarised below.

- **small**. This means that near addresses are used for both data objects and code objects (functions). The data and code spaces are thus restricted to 64 KByte each.
- **medium**. Near addresses are used for data but far addresses are used for code. This restricts the data space to 64 KByte and the code space to 256 MByte. This suggests a large complex program manipulating a small amount of data.
- **compact**. Far addresses are used for data but near addresses are used for code. This restricts the data space to 256 MByte and the code space to 64 KByte. This suggests a small simple program manipulating a massive amount of data.
- **large**. Far addresses are used for both data and code. The data and code spaces are thus restricted to 256 MByte each. This suggests large programs manipulating large amounts of data.
- **huge**. The ability to handle large amounts of data in the large and compact memory models suggests that the data may be stored in large aggregates or arrays. Unfortunately the large and compact models calculate addresses within data objects by adding a near offset to the base address of the object. This implies a restriction of 64 KByte on the size of a data object. This restriction is relaxed in the huge memory model which is otherwise similar to

the large model.

Similar memory models are supported by Turbo C with the addition of a "tiny" model in which all the code and data occupy a single 64 KByte segment of memory.

An address, either of a data object or a function (code object) may be declared to be of other than the default type for the current memory model by using the non ANSI standard keywords *near , far* and *huge* as appropriate. A further complication concerns the standard libraries which come in different forms for each memory model. If you wish to compile a program using a particular memory model you need to have and link the set of standard libraries appropriate to that memory model. Vendors of hard disc systems love this.

The use of the "x" conversion with *printf()* to display pointer values is of doubtful portability. It only works when addresses consist of a single component which can be converted to an unsigned integer in a straightforward way. The ANSI standard provides the "p" conversion for use with pointers. This is specified as converting a pointer to a sequence of printable characters in an implementation defined manner.

The first program in this chapter was modified to use the "p" conversion rather than the "08x" conversion and compiled by the Turbo C compiler using the large memory model. The output produced was.

```
Address a = 47CA:0FFE
Address b = 47CA:0FFC
Address x = 47CA:0FF4
Address z = 47CA:0FF2
```

The figure before the colon (47CA) is the hexadecimal high part of the far address and the the figure after the colon is the low part of the far address.

For fuller details of memory models and associated topics you must consult your compiler manual. This section is only intended to give a flavour of what happens.

Array and Aggregate Initialisation

Addresses, Pointers, Arrays and Strings - Array and Aggregate Initialisation

Chapter chap6 section 5

It is possible to initialise an aggregate as part of the declaration using a syntax similar to that used to initialise normal variables. Since an aggregate is being initialised it is necessary to provide a sequence of values. Such a sequence, known as an **initialiser,** consists of a comma separated list of values enclosed within braces. The following example, yet another variant of the first aggregate program demonstrates the point.

```
main()
{
        int     agg[5] = {0,1,4,9,16};
        int     i = 0;
        int     sum = 0;
        do
        {
                sum += *(agg+i);
                i++;
        } while(i<5);
        printf("Sum = %d\n",sum);
}
```

It produced exactly the same output. It should be noted that older non-standard implementations of the C programming language may not allow the initialisation of aggregates declared inside compound statements. The number of data items provided within the initialiser must match the aggregate size, if too few are provided then the final members of the aggregate will not be initialised, if too many are supplied then the behaviour is unpredictable. Attempting to compile the following program

```
main()
{
        int     agg[5] = {0,1,4,9,16,25};
        int     i = 0;
        int     sum = 0;
        do
```

```
        {
                sum += *(agg+i);
                i++;
        } while(i<5);
        printf("Sum = %d\n",sum);
}
```

produced the following output

```
"agg7.c", line 3: too many array initializers
Compilation failed
```

If the size of an initialised aggregate is not specified then the compiler will count the number of initialisers and create an aggregate of the correct size. This program demonstrates the facility.

```
main()
{
        int     agg[] = {0,1,4,9,16,25};
        int     i=0;
        int     sum = 0;

        do sum += *(agg+i);
                while(++i<5);
        printf("Sum = %d\n",sum);
}
```

It produced the output

```
Sum = 30
```

Although it is advantageous to be able to put extra values in a collection of values and have the size expand automatically, this program has failed to deliver the correct result (55) because the number of times round the loop has not been altered. It would be nice if there were some way of telling the program to go through every member of the aggregate and then stop. If this could be done then any program that processed an initialised aggregate could easily be modified to process a larger or smaller aggregate simply by changing the initialiser list.

This effect can be achieved using the **sizeof** operator. This comes in two very similar flavours.

The first is

```
sizeof expression
```

and the second is

```
sizeof ( type-name )
```

The first expression gives the number of basic units of storage (invariably bytes) occupied by the expression which must be the name of a variable or the name of an aggregate. The second expression gives the number of basic units of storage occupied by a single object of named type. So to determine the number of elements in an aggregate you could use the following code

```
int     agg[] = {......};
nelements = sizeof agg / sizeof ( int );
```

The following program shows the use of the *sizeof* operators in determining the range of values to be used when working through an array. This program reads in an input value and checks whether it is one of the values in the array.

```
main()
{
        int     okvals[] = {23,48,61,44};
        int     i = 0;
        int     input;
        printf("Enter a code value ");
        scanf("%ld",&input);
        do
        {
                if(okvals[i] == input)
                {
                        printf("Accepted\n");
                        exit(0);
                }
                i++;
        } while(i < sizeof okvals/sizeof (int));
        printf("Not accepted\n");
}
```

Here are some examples of its operation. The program was called *agg9*

```
abash$ agg9
Enter a code value 23
Accepted
bash$ agg9
Enter a code value 44
Accepted
```

```
bash$ agg9
Enter a code value 98
Not accepted
bash$ agg9
Enter a code value 4
Not accepted
```

The array is to be regarded as a list of acceptable values. If further values become acceptable or if fewer values become acceptable then it is only necessary to change the list of values and recompile the program. There is no need to track down every reference to the array size in code that scans the array, the limits are set automatically during compilation.

Expressions involving the *sizeof* operators are evaluated at compile time rather than run time, this means that the expression in the first flavour of *sizeof* must be a **constant expression,** i.e. one whose value does not change in the course of program execution and hence can be evaluated once when the program is being compiled.

In the example shown above you could have written

```
sizeof(agg)
```

since the enclosing an expression in parentheses has no effect on the value of the expression.

Operator Precedence

Functions and storage organisation - Exercises

Chapter chap7 section 12

1. Devise and code a function with two parameters, the first of type *char* and the second of type *int.* The purpose of the function will be to print out the first parameter the specified number of times. Write a simple interactive program to test out your function.
2. Write a function that takes a string as parameter and prints out the string surrounded by a box of asterisks. Incorporate your function in a program that prompts the user for a string, prints it out within an enclosing box and then constructs a further string from the supplied string (you might, for example, add "This is your string" on the front) and prints out the modified string in an enclosing box.
3. The square root of a number can be calculated by making repeated approximations using the formula

   ```
   new = (old + original/old)/2
   ```

   where "old" is the previous approximation and "original" is the number you are trying to find the square root of. Stop making repeated approximations when the difference between the new and old approximations is small enough. Write a function to calculate square roots and incorporate it in a simple interactive program.

   This method is known as Newton's method, square roots are normally calculated using a library function.

   Write a set of functions called *isdecimal(), ishex()* and *isoctal()* that can be applied to a string to determine what type of numerical constant the string represents. Incorporate your functions in a program that reads in a string, converts it to internal numerical form (use *sscanf()*) and prints out the internal value in decimal, octal and hexadecimal form.
4. In a mathematical text a function f(x,y) is defined thus

   ```
   f(x,y) = x - y  if x<0 or y<0
   f(x,y) = f(x-1,y)+f(x,y-1)      otherwise
   ```

   Write a C program to evaluate this function for user supplied values of x and y.
5. Modify the program of the previous exercise to print out a table of values of the function defined for all values of x and y in the range 0-10.

Warning this might take quite a long time to run.

Addresses, Pointers, Arrays and Strings - Operator Precedence

Chapter chap6 section 6

We have seen several new operators in this chapter (unary *, unary &, [] and sizeof). It is time to update our table of operator precedences and associativities. Here it is.

Operators	Associativity
`() [] ++(postfix) --(postfix)`	Left to Right
`++(prefix) --(prefix) ~ (type) +(unary) -(unary) ! sizeof *(unary) &(unary)`	Right to Left
`* / %`	Left to Right
`+ -`	Left to Right
`<< >:>`	Left to Right
`< <= => >`	Left to Right
`== !=`	Left to Right
`&`	Left to Right
`^`	Left to Right
`|`	Left to Right
`&&`	Left to Right
`||`	Left to Right
`?:`	Right to Left
`= += *= <<= >>= /= %= -= &= |= ^=`	Right to Left

Strings

Switch and For Statements, Command Line and File Handling - File Handling

Chapter chap9 section 5

The standard library used in conjunction with *stdio.h* provides some quite useful facilities for handling files. There are, inevitably, some host system dependencies in file handling. These will be mentioned when relevant.

All file handling is done via objects of type *pointer to **FILE.*** This compound data type is defined in *stdio.h* . The following file handling functions are provided in the standard library.

Function	Action
fopen()	Open a file
fclose()	Close a file
fprintf()	Formatted write to a file
fscanf()	Formatted read from a file
fputc()	Write a character to a file
fgetc()	Read a character from a file
fputs()	Write a string to a file
fgets()	Read a string from a file
putc()	Write a character to a file (macro)
getc()	Read a character from a file (macro)
ungetc()	Un-read a character from a file
fread()	Unformatted read from a file
fwrite()	Unformatted write to a file
fgetpos()	Determine current position in a file
fseek()	Adjust current position in a file
fsetpos()	Adjust current position in a file
ftell()	Determine current position in a file
rewind()	Set current position to start of file
feof()	Tests whether end-of-file has been seen
ferror()	Tests whether a file error has occurred
clearerr()	Clears file error indicator
remove()	Delete a file
rename()	Rename a file
tmpfile()	Create a temporary file

tmpnam()	Create a unique file name
fflush()	Force writing of data from buffer to file
freopen()	Opens a file using a specific FILE object

There are also a number of pre-defined constants in stdio.h The following are most likely to be useful.

constantmeaning

FOPEN_MAX	Maximum number of open files
FILENAME_MAX	Maximum length of file name
SEEK_CUR	Used with fseek() function
SEEK_END	Used with fseek() function
SEEK_SET	Used with fseek() function
stderr	Pre-opened file
stdout	Pre-opened file
stdin	Pre-opened file

The standard file handling functions view a file as an array of bytes. Some of the functions are line oriented regarding a line as a sequence of bytes terminated by a new-line character (in the Unix tradition). If the host operating system is not Unix then the file handling functions will attempt to make the files look like Unix files, this can cause some portability problems.

A simple example program designed to copy a file called *data1* to a file called *data2* is listed below.

```
#include        <stdio.h>
main()
{
        FILE    *ifp,*ofp;
        int     c;
        if((ifp=fopen("data1","r")) == NULL)
        {
                printf("Couldn't open \"data1\" for input\n");
                exit(1);
        }
        if((ofp=fopen("data2","w")) == NULL)
        {
                printf("Couldn't open \"data2\" for output\n");
                exit(1);
        }
        while((c=getc(ifp)) != EOF) putc(c,ofp);
}
```

There are several points of interest in this program.

The objects *ifp* and *ofp* are used to hold pointers to objects of type FILE (note the case), they are no different from any other pointer data type. The library function **fopen()** actually opens the file. It takes two parameters that are both pointers to characters. The first parameter is the name of the file to open, in this simple example the file names are "hard-wired" into the program but they could have been obtained interactively or from the command line. The second parameter is the file opening mode. This is also a string of characters, even though, in this case there is only a single character. ANSI C recognises the following opening modes

mode	meaning
r	Open text file for reading
w	Truncate to zero size or create text file for writing
a	Append - open or create text file for writing at end of file

These basic modes may be modified by two extra characters. A "+" may be used to mean that the file is to be opened for both **reading and writing (updating).** A b may be used to mean opening in **binary mode,** this means that the library functions will present the underlying data to the program without any attempt to make the file look Unix like, under Unix it has no effect.

The return value from *fopen()* is either a pointer to an object of type FILE or the value *NULL* if it was not possible to open the named file. The macros **getc()** and **putc()** should be noted, the entire action of copying the files is handled by the single line of code at the end of the program. There is no need to specifically close the files, when the program returns to the host environment all open files are automatically closed as part of the return to host environment mechanism.

The use of the library function **fgets()** is illustrated in the following example. *fgets()* is the file handling equivalent of *gets()*, it may be thought of as a function that reads in the next record from the file by equating a record with a sequence of characters terminated by a newline, which is the normal Unix convention. Unlike *gets()* which requires the programmer to be careful about the input buffer size, *fgets()* has a parameter that specifies the maximum number of characters to transfer to the input buffer. The prototype of *fgets()* is

```
char *fgets(char *, int, FILE *)
```

The first parameter is, of course, the start address of the input buffer. The second parameter is the buffer size, the actual number of characters read is, at most, one less than the buffer size allowing a string terminating *NUL* to be placed at the end of the buffer. The final parameter identifies the input file.

If the input record is not too big for the input buffer then it is copied to the input buffer complete with the input line terminating newline character. If the record is too big then there will be no newline character in the input buffer and the next call to *fgets()* will carry on where the last one left off, getting the next portion of the record.

The use of *fgets()* is illustrated by the following example which analyses the number of lines in a file and their maximum and minimum lengths.

```
#include        <stdio.h>
```

```
/*      Program to report the number of records, their
        average size & the smallest and largest record sizes.
*/

int     getrec(FILE *);

main(int argc, char *argv[])
{
        int     n;
        int     recno = 0;      /* number of records */
        int     minrec;         /* smallest */
        int     maxrec;         /* largest */
        long    cumrec = 0;     /* cumulative size */
        FILE    *dfp;           /* file to analyse */
        char    started = 0;
        if(argc != 2)
        {
                printf("Usage : fil2 f\n");
                exit(1);
        }
        if( (dfp = fopen(argv[1],"r")) == NULL)
        {
                printf("error opening %s\n",
                                        argv[1]);
                exit(1);
        }
        while (1)
        {
                if((n=getrec(dfp))==EOF) break;
                recno++;
                cumrec += n;
                if(!started)
                {
                        minrec = maxrec = n;
                        started = 1;
                }
                if(n<minrec) minrec = n;
                if(n>maxrec) maxrec = n;
        }
        printf("%4d records\n",recno);
        printf("average size %5.1f\n",
                                (double)cumrec/recno);
        printf("smallest %2d\nlargest %4d\n",
                                minrec,maxrec);
}
int     getrec(FILE *f)
/*      function to read a record and return (as
        functional value) the record size. If the
```

```
        end of file is encountered then EOF is
        returned.
*/
{
        int     rs=0;
        int     length;
        char    buff[25];
        while(1)
        {
                if(fgets(buff,25,f)==NULL)
                        return EOF;
                length = strlen(buff);
                rs += length;
                if(buff[length - 1] == '\n')
                        return rs;
        }
}
```

With a suitably large file named on the command line, the program produced the following output.

```
8788 records
average size  99.2
smallest 25
largest  282
```

There are several interesting points about this program. The function **getrec()** reads the file in chunks of 24 bytes, this is ridiculously small. In order to determine how much data has actually been read in the function *strlen()* is applied to the string in the input buffer, remembering that the count returned by *strlen()* excludes the string terminating *NUL*. It is particularly important to look at the character before the string terminating *NUL* in the input buffer, if this is a newline then the repeated calls to *fgets()* have encountered the end of the current record and the accumulated record size can be returned.

---

stdin, stdout and stderr

Addresses, Pointers, Arrays and Strings - Library Functions for processing strings

Chapter chap6 section 12

There are a variety of library functions for handling input data that has been read in using *gets()*. The most useful include **sscanf()** and the function **atoi().** The function *sscanf()* applies *scanf()* type conversions to data held in a program buffer as a single string rather than to data read from standard input. The *atoi()* function converts a character string from external decimal form to internal binary form.

The use of *sscanf()* in conjunction with *gets()* is illustrated by the following program, known as *getin1*. The purpose of the program is to read in an integer. Unlike simple uses of *scanf()*, input errors are detected and the prompt repeated until a valid integer is entered.

```
main()
{
        char    error;
        char    inbuf[256];     /* hope it's big enough */
        int     i;
        char    c;
        while(1)
        {
                error = i = 0;
                printf("Enter an integer ");
                gets(inbuf);    /* get complete input line */
                while(*(inbuf+i) == ' ') i++;   /* skip spaces */
                if(*(inbuf+i) == '-' || *(inbuf+i) == '+') i++;
                while(c = *(inbuf+i++)) /* while string end NUL */
                {
                        if(c>'9' || c<'0')      /* non-digit ? */
                        {
                                printf("Non-Numeric Character %c\n",c);
                                error = 1;
                                break;
                        }
                }
                if(!error)      /* was everything OK ? */
                {
                        int     num;    /* local variable */
                        sscanf(inbuf,"%d",&num);        /* conversion */
                        printf("Number was %d\n",num);
                        break;
                }
        }
}
```

A typical dialogue is shown below

```
$ getin1
Enter an integer a123
Non-Numeric Character a
```

```
Enter an integer 123a
Non-Numeric Character a
Enter an integer     1234.56
Non-Numeric Character .
Enter an integer     1234
Number was 1234
$ getin1
Enter an integer    +43
Number was 43
$
```

There are some interesting points about this program. The main processing loop first skips any leading spaces leaving the address "inbuf+i" pointing to the first non-blank character in the input text. An initial sign is also skipped. After the optional initial sign all input characters must be digits until the input string terminating NUL is encountered. If anything other than a digit, including trailing blanks, is encountered the loop is broken and an error indicator set.

The condition

```
c = *(inbuf+i++)
```

associated with the loop that checks for digits is a typical piece of C code that does several things in one go. The value of the expression "*(inbuf+i++)" is the next character from the input buffer *inbuf*. The value of the expression inbuf+i++ being the **address** of the character and the * operator yielding the value of the addressed item. In the course of calculating the address of the character, the variable "i" is incremented as a side-effect. The character value is assigned to the variable "c" to be used in the test for being a digit on the following line, the value of the assignment expression being, of course, the value assigned. The value of this expression only becomes zero, so terminating the loop, when the character in question is the string terminating NUL.

In practice the code of this program would be incorporated into a user defined function that might well return the value of the entered integer.

The function *sscanf()* is similar to *scanf()* except that it has an extra parameter that is the address of the start of the memory area that holds the character string to be processed. The library function *atoi()* could have been used instead of *sscanf()* in this example by changing the appropriate line to read

```
num = atoi(inbuf);
```

The function i *atoi()* takes the address of an area of memory as parameter and converts the string stored at that location to an integer using the external decimal to internal binary conversion rules. This may be preferable to *sscanf()* since *atoi()* is a much smaller, simpler and faster function. *sscanf()* can do all possible conversions whereas *atoi()* can only do single decimal integer conversions.

The library functions sprintf() and puts()

Switch and For Statements, Command Line and File Handling - stdin, stdout and stderr

Chapter chap9 section 6

The predefined files known as **stdin , stdout** and **stderr** normally correspond to the keyboard and, for both *stdout* and *stderr* , the display. Most host operating systems provide facilities to reconnect these to files but such reconnection or redirection is totally invisible to the program.

All three are, in fact, objects of type *pointer to FILE*, and they may be used in any file handling function in just the same way as a pointer returned by *fopen().* In fact the macro *putchar(c)* is really nothing more than

```
putc(c,stdout)
```

It is sometimes useful to initialise a pointer to FILE to point to one of the standard items, to provide a "standard input as default" type of operation.

```
FILE *ifp = stdin;
```

being a typical definition.

stderr is subject to separate host operating system redirection to *stdout* and is commonly used to write error messages. Typically a programmer would write

```
fprintf(stderr,"Couldn't open \"%s\"\n",filename);
```

Direct Access File Handling

Addresses, Pointers, Arrays and Strings - Exrecises

Chapter chap6 section 16

1. Write a program to read text lines into a buffer using *gets()*. Calculate the length of each line. By using input redirection arrange for your program to read a text file (such as its own source) and print out

   ```
   The number of lines read
   The length of the longest line
   The length of the shortest line
   The average line length
   ```

   You should arrange for a line of length zero to terminate your file.
2. By using a text editor (or otherwise) create a file containing a significant number (80-100) integers in the range 0-100. Place the integer 999 at the end of the file.

   Write a program to read the file using input redirection and calculate the number of input integers in each of the ranges 0-9, 10-19, 20-29 etc., Print out the results of your calculations.
3. Modify the program of the previous exercise to print out a histogram for each of the ranges consisting of a sequence of asterisks running across the screen.
4. Modify the program of the previous exercise to produce a vertical histogram rather than a horizontal histogram.
5. Write a program that will read text from standard input and produce a list of the characters encountered in the input and the number of times each character occurred. You will find it most convenient to use *gets()* to read each line of input into a buffer and terminate your input with an empty line. Run the program with input re-direction to read from a file. Do not print out the character and the count if it did not occur in the input.
6. Modify the program of the previous exercise to print out the character frequency table in lines of not more than 8 entries going across the screen rather than one long list.
7. Write a program that will prompt the user for two strings of characters, assemble them into a single string and then print the string out reversed.

Functions and storage organisation - Introduction

Chapter chap7 section 1

In this chapter we will see how to construct packaged code chunks known as functions or routines.

See Also

- Functions
- Prototypes
- Pre ANSI function declarations
- Aggregates and Arrays as function parameters
- Pointers to functions
- Recursion
- Storage classes and function local variables.
- The stack
- Type qualifiers
- Multi-dimensional arrays and aggregates
- Exercises

Functions and storage organisation - Functions

Chapter chap7 section 2

We have been using functions since the very start of the course. *printf()* is a library function as are *scanf(), gets(), strcmp()* etc., We have also been writing functions since our first example program. *main()* is nothing more or less than a function.

Functions provide a convenient way of packaging up pieces of code so that they can be used over and over a again. Consider a simple interactive program that reads in three integers and calculates and displays the sum of the three integers. It might look like

```
main()
{
        int     n1,n2,n3;
        printf("Enter a number ");
        scanf("%d",&n1);
        printf("Enter a number ");
        scanf("%d",&n2);
        printf("Enter a number ");
        scanf("%d",&n3);
        printf("Sum is %d\n",n1+n2+n3);
}
```

But this direct use of *scanf()* is not a very good way to get data from a fallible human user. In the previous chapter we saw a program that vetted the user's input to insure that it really was numeric. We could, of course, re-write the above program to incorporate all the checking and prompting for each input number but this would involve a clumsy, wasteful and error-prone replication of the code. It would be much better if we could re-use the same code three times. Functions provide precisely this facility. Using a get-an-integer function the program, called *fex1*, might look like.

```
main()
{
        int     n1,n2,n3;
        n1 = getint();
        n2 = getint();
        n3 = getint();
        printf("Sum = %d\n",n1+n2+n3);
}
getint()
{
        char    error;
        char    inbuf[256];
        int     i;
        char    c;
        while(1)
        {
                i = 0;
                error = i = 0;
                printf("Enter an integer ");
                gets(inbuf);
```

```
                while(*(inbuf+i) == ' ') i++;
                if(*(inbuf+i) == '-' || *(inbuf+i) == '+') i++;
                while(c = *(inbuf+i++))
                {
                        if(c>'9' || c<'0')
                        {
                                printf("Non-Numeric Character %c\n",c);
                                error = 1;
                                break;
                        }
                }
                if(!error)
                {
                        int     num;
                        sscanf(inbuf,"%d",&num);
                        return  num;
                }
        }
}
```

And a typical dialogue is illustrated below.

```
$ fex1
Enter an integer one
Non-Numeric Character o
Enter an integer 1
Enter an integer hello
Non-Numeric Character h
Enter an integer 23098
Enter an integer 0xfed
Non-Numeric Character x
Enter an integer 23
Sum = 23122
$
```

If you compare this program with the one in the previous chapter you will find that it is almost identical with the second part (or function) in the current program. There are two important differences.

1. The first line has been changed from *main()* to *getint()* . *"getint"* is the name of the function. The rules for naming C functions are the same as the rules for naming variables. A variable and a function may not have the same name.
2. Rather than printing out the number read in, it is returned by the statement

   ```
   return num;
   ```

   When the assignment

   ```
   n1 = getint()
   ```

   is evaluated, the value of the function *getint()* is the value associated with the last executed **return statement** within the function.

The program shown above consists of two function **definitions.** The first for the function *main()* and the second for the function *getint()* . We will see shortly that the declaration of a function is not quite the same thing as the definition of a function.

A C program normally consists a series of function definitions. The line of code that starts *getint()* is identified as a **function definition** by two facts,

1. it is not included within the compound statement that forms the executable body of the function *main()*
2. the parentheses that appear immediately after the name *getint* .

Notice also that when *getint()* was called although we did not supply the function with any parameters we still had to write the parentheses normally used to enclose the parameter list. These function definitions may appear in any order although it is conventional to start with *main* and follow it with functions called by *main* followed by functions called by functions that are called by *main* and so on. This order makes programs much easier to read. Function definitions may **not** be included within other function definitions, i.e. they cannot be **nested.** The *return statement* can appear anywhere in a function, if the path of execution of a function reaches the end of the function definition then the function returns to the calling environment exactly as if an explicit return had been included in the code.

Of course the simple *getint()* function does exactly the same thing every time it is called. It would be nice if it could produce a variable prompt. This is easily achieved by making the prompt a parameter. When using parameters with functions the C programming language talks about **formal** parameters and **actual** parameters. The formal parameter is a sort of dummy variable associated with the function definition, it provides storage space for the value of the actual parameter which appears as part of the function call.

To declare a formal parameter as part of a function definition, include the type and name to be associated with the formal parameter within the parentheses that appear at the start of the definition. A variant of the previous program using a parameter is shown below.

```
main()
{
        int     sum=0,n[3],i=0;
        while(i<3)
        {
                n[i] = getint(i+1);
                sum += n[i++];
        }
        printf("Sum = %d\n",sum);
}
getint(int n)
{
        char    error;
        char    inbuf[256];
        int     i;
        char    c;
        while(1)
        {
                i = 0;
                error = i = 0;
                printf("Enter integer %d ",n);
                gets(inbuf);
                while(*(inbuf+i) == ' ') i++;
                if(*(inbuf+i) == '-' || *(inbuf+i) == '+') i++;
                while(c = *(inbuf+i++))
                {
                        if(c>'9' || c<'0')
                        {
                                printf("Non-Numeric Character %c\n",c);
```

```
                                error = 1;
                                break;
                        }
                }
                if(!error)
                {
                        int     num;
                        sscanf(inbuf,"%d",&num);
                        return  num;
                }
        }
}
```

A typical dialogue

```
$ fex2
Enter integer 1 23
Enter integer 2 11
Enter integer 3 one
Non-Numeric Character o
Enter integer 3 1
Sum = 35
$
```

Here "n" is the formal parameter.

It is important to realise that the formal parameter holds a **copy** of the value of the actual parameter at the time the function was called. It is possible to change the value of the formal parameter but this has no effect on the actual parameter as the following example shows.

```
main()
{
        int     n[3]={1,2,3};
        int     sum = 0;
        addup3(n[0],n[1],n[2],sum);
        printf("sum = %d\n",sum);
}
addup3(int a, int b, int c, int total)
{
        total = a+b+c;
        printf("total = %d\n",total);
}
```

This produced the output

```
total = 6
sum = 0
```

It is clearly desirable that the value of the actual parameter cannot be changed from within the function because the code of the function has no way of telling whether the actual parameter was the name of a variable or an expression or even a constant.

It is possible to return values via the parameters by providing the function with the address of a memory location. The formal parameter then has to be of one of the pointer types. Consider the following

example.

```
main()
{
        int     n[3]={1,2,3};
        int     sum = 0;
        addup3(n[0],n[1],n[2],&sum);
        printf("sum = %d\n",sum);
}
addup3(int a, int b, int c, int *total)
{
        *total = a+b+c;
        printf("total = %d\n",*total);
}
```

This produced the following output.

```
total = 6
sum = 6
```

Throughout the function code it is quite clear that the final formal parameter is the address of an integer.

Prototypes

Functions and storage organisation - Prototypes

Chapter chap7 section 3

So far we have dodged any consideration of the type of value returned by a function. In the examples shown above it was quietly assumed that the function *getint()* returned an integer. It is quite possible to have functions returning values of other types, however the compiler will assume that all functions return integers unless it has information to the contrary as the following example shows.

```
main()
{
        int     n[3]={23,14,7};
        int     i=0;
        while(i<3)
        {
                double  x;
                x = reciprocal(n[i]);
                printf("1/%d = %10.7lf\n",n[i],x);
                i++;
        }
}
double  reciprocal(int j)
{
        return 1.0/j;
}
```

The data type name associated with the definition of the function *reciprocal()* should be noted. This says that the function *reciprocal()* returns a value of type double. Unfortunately the compiler had already seen the reference to *reciprocal()* earlier in the program and had assumed, in the absence of information to the contrary, that *reciprocal()* was of type *int*. The following error message was generated by the compiler.

```
"fex5.c", line 14: identifier redeclared: reciprocal
Compilation failed
```

The solution to this problem is to provide something called a **function prototype** that declares the function name, its type and the types of all its formal parameters. Such prototypes can appear anywhere in the program and need not be part of a function definition. They usually appear at the start of a program before the function *main()* and obviously should appear before any call to the function in question. The following modified version of the previous program shows what it looks like.

```
double  reciprocal(int);
```

```
main()
{
        int     n[3]={23,14,7};
        int     i=0;
        while(i<3)
        {
                double  x;
                x = reciprocal(n[i]);
                printf("1/%d = %10.7lf\n",n[i],x);
                i++;
        }
}
double  reciprocal(int j)
{
        return 1.0/j;
}
```

producing the output

```
1/23 =  0.0434783
1/14 =  0.0714286
1/7 =  0.1428571
```

The very first line of the program is the function prototype. It should be noted that there is no need to define a name to be associated with the formal parameter in the prototype, just the typing information.

With prototypes the compiler can perform various extra checks on actual calls as the following example shows.

```
int     addup3(int,int,int,int*);
main()
{
        int     n[3] = {4,12,9};
        int     sum;
        addup3(n[0],n[1],&sum);
        printf("sum = %d\n",sum);
        addup3(1,2,3,4,sum);
        printf("sum = %d\n",sum);
        addup3(1.2,2.9,2.7,&sum);
        printf("sum = %d\n",sum);
}
int     addup3(int x,int y,int z, int *res)
{
        *res = x + y + z;
}
```

Not surprisingly, the compiler complained producing the following error messages.

```
line 6: warning: improper pointer/integer combination: arg #3
line 6: prototype mismatch: 3 args passed, 4 expected
line 8: warning: improper pointer/integer combination: arg #4
line 8: prototype mismatch: 5 args passed, 4 expected
Compilation failed
```

Line 6 was

```
addup3(n[0],n[1],&sum)
```

and line 8 was

```
addup3(1,2,3,4,sum)
```

The messages about the number of parameters are self-explanatory. The other messages require further explanation as does the fact that no error messages seem to have been generated for the final call of *addup3()* in which all the parameters were of the wrong type. Let's first fix up the errors on the first two calls of *addup3()*.

```
int     addup3(int,int,int,int*);
main()
{
        int     n[3] = {4,12,9};
        int     sum;
        addup3(n[0],n[1],n[2],&sum);
        printf("sum = %d\n",sum);
        addup3(1,2,3,&sum);
        printf("sum = %d\n",sum);
        addup3(1.2,2.9,2.7,&sum);
        printf("sum = %d\n",sum);
}
int     addup3(int x,int y,int z, int *res)
{
        *res = x + y + z;
}
```

This compiled without errors and produced the following output

```
sum = 25
sum = 6
sum = 5
```

The sum of the 3 values for the final call of *addup3()* is 6.8, 5 is wrong but not disastrously. What has happened is that the compiler noted that the function *addup3()* required three parameters of type *int* but was called with three parameters of type *double*, it, obligingly, generated extra code that converted the values of the parameters to the correct type when the function *addup3()* was called. The extra error messages produced above can now be explained,

1. the prototype told the compiler to expect an *int* for the third argument but a *pointer to int* was supplied,
2. a *pointer to int* was expected for the fourth argument but an actual *int* was supplied. There is no standard conversion between integer data types and pointer data types so the compiler could not generate any conversion code.

You may be puzzled as to why *printf()* and *scanf()* aren't handled in this helpful fashion. The answer is that *printf()* and *scanf()*, for partly historical reasons, take a variable number of parameters of variable types and no checking is possible. In fact the formal prototype of *printf()* is

```
int printf(char *,...)
```

where the symbol "...", known as an **ellipsis** means an arbitrary number (including zero) of parameters of arbitrary type.

If a function does not actually take any parameters then the keyword **void** may be used in place of the parameter list in the prototype declaration and function definition. Just leaving out the parameters simply tells the compiler that any number of parameters of any type are acceptable, i.e. no checking will be possible. *void* indicates quote explicitly that there are **no** parameters. Similarly if the function is not going to return a value then its type should be declared as "void". Examples will be seen later in these notes.

The values of parameters of certain types will be converted under all circumstances irrespective of prototypes. This mechanism is known as functional parameter promotion and was described earlier in the discussion on arithmetic type conversions.

Pre ANSI Function Declarations
Aggregates and Arrays as Function Parameters

Functions and storage organisation - Pre ANSI Function Declarations

Chapter chap7 section 4

The provision of information on function parameters in prototypes was only introduced in the ANSI version of the C language. Pre-ANSI prototypes only refer to the type of the functional value and also have a different way of defining the formal parameters of a function. The pre-ANSI style prototypes and function definitions are accepted by ANSI compilers but the checking of the number and type of function parameters will not take place leading to potential obscure errors.

A typical pre-ANSI function definition would look like

```
double  harmonic_mean()
double  x,y,z;
{
        .
        .
```

The ANSI equivalent is

```
double  harmonic_mean(double x, double y, double z)
{
        .
        .
```

Aggregates and Arrays as function parameters

Functions and storage organisation - Aggregates and Arrays as Function Parameters

Chapter chap7 section 5

There is no restriction on the possible number or type of parameters that may be associated with a function. The following program demonstrates the syntax associated with the use of an aggregate as a formal parameter.

```
int     sumagg(int[],int);
main()
{
        int     x[]={1,5,3,2,7,4};
        printf("Sum of numbers = %d\n",
                sumagg(x,sizeof x/sizeof (int)));
}
int     sumagg(int q[],int nels)
{
        int     i=0;
        int     sum = 0;
        while(i<nels) sum+=q[i++];
        return  sum;
}
```

This produced the output

```
Sum of numbers = 22
```

The function definition includes the formal parameter

```
int q[]
```

There is no need to specify the size of the aggregate at this stage because the aggregate itself is not being passed to the function. Remember that the name of an aggregate is, effectively, a synonym for the address of element number zero and also remember that aggregates are closely related to pointers. The following version of the program listed above is also perfectly valid.

```
int     sumagg(int[],int);
main()
```

```
{
        int     x[]={1,5,3,2,7,4};
        printf("Sum of numbers = %d\n",
                sumagg(x,sizeof x/sizeof (int)));
}
int     sumagg(int *q,int nels)
{
        int     i=0;
        int     sum = 0;
        while(i<nels) sum+=q[i++];
        return  sum;
}
```

This differs from the previous example only in the definition of the first formal parameter in the function definition. The aggregate notation

```
q[]
```

and the pointer notation

```
*q
```

are clearly totally equivalent and interchangeable. Notice that in this example program it was necessary to pass the aggregate size as a separate parameter.

Another example of the use of aggregates as functional parameters is shown in this example of a function to calculate the length of a string. There is actually no need to write such a function, the library function *strlen()* is perfectly satisfactory and entirely standard.

```
int     lenstr(char*);
main()
{
        char    inbuf[256];
        int     len;
        int     tlen = 0,cnt = 0;
        while(1)
        {
                printf("Enter String ");
                gets(inbuf);
                if((len = lenstr(inbuf)) == 0) break;
                printf("Length = %d\n",len);
                cnt++;tlen+=len;
        }
        printf("Strings Processed = %d\n"
        "Total Characters = %d\n",cnt,tlen);
```

```
}
int     lenstr(char *cp)
{
        char    *start = cp;
        while(*cp++);
        return cp - start - 1;
}
```

A typical dialogue is shown below.

```
$ fpx3
Enter String hello
Length = 5
Enter String goodbye
Length = 7
Enter String this is some test data
Length = 22
Enter String and some more
Length = 13
Enter String
Strings Processed = 4
Total Characters = 47
$
```

Pointers to functions

Functions and storage organisation - Pointers to Functions

Chapter chap7 section 6

Although it is not possible to pass functions (as distinct from their values) to other functions as parameters it is possible to pass a pointer to a function as a parameter. In order to do this it is first necessary to look at the syntax of the declaration of a variable of type **pointer to function.** Such variables carry the type of value returned by the function. A typical declaration of variable of type **pointer to integer valued function** is

```
int (*f)();
```

The odd looking syntax arises because the grouping operator "()" has a higher precedence than the address operator "*". The parentheses around "*f" are necessary to ensure that f is seen as pointer **before** it is seen as a function. It is also, of course, possible to have an aggregate of pointers to functions. For example

```
double (*valuator[6])();
```

The unadorned (i.e. no parentheses) name of a function is an object of type pointer to function and may be used to initialise a pointer to function as part of the declaration or may be assigned to a pointer to function. An example is in order.

```
int     sum(int,int);
int     difference(int,int);
int     product(int,int);
int     quotient(int,int);
main()
{
        char    inbuf[256];
        int     (*math)();
        int     n1,n2;
        char    *iptr = inbuf;
        printf("Enter an expression ");
        gets(inbuf);
        while(*iptr >= '0' && *iptr <= '9') iptr++;
        n1 = atoi(inbuf);
        if(*iptr == '+') math = sum;
        if(*iptr == '-') math = difference;
```

```
        if(*iptr == '*') math = product;
        if(*iptr == '/') math = quotient;
        iptr++;
        n2 = atoi(iptr);
        printf("Value = %d\n",math(n1,n2));
}
int     sum(int p,int q)
{
        return p+q;
}
int     difference(int x,int y)
{
        return x-y;
}
int     quotient(int a,int b)
{
        return a/b;
}
int     product(int c,int d)
{
        return c*d;
}
```

This program reads in a simple expression such as

```
5+12
```

and displays its value as the following typical dialogue demonstrates. No attempt is made to check the input and the required input layout is totally unexplained to the user. The program is called *fpx4* .

```
$ fpx4
Enter an expression 15*8
Value = 120
$ fpx4
Enter an expression 22/5
Value = 4
$
```

Recursion

Functions and storage organisation - Recursion

Chapter chap7 section 7

C functions may call themselves in an entirely natural manner. This is known as **recursion** and is illustrated in the following classical example which calculates factorial n, i.e. the product of all the integers from 1 to n.

```
int     factorial(int);
main()
{
        int     i=1;
        do
                printf("Factorial %d = %d\n",i,factorial(i));
        while(i++<12);
}
int     factorial(int x)
{
        if(x==1) return 1;
        else    return x*factorial(x-1);
}
```

producing the output

```
Factorial 1 = 1
Factorial 2 = 2
Factorial 3 = 6
Factorial 4 = 24
Factorial 5 = 120
Factorial 6 = 720
Factorial 7 = 5040
Factorial 8 = 40320
Factorial 9 = 362880
Factorial 10 = 3628800
Factorial 11 = 39916800
Factorial 12 = 479001600
```

Storage Classes

Functions and storage organisation - Storage Classes

Chapter chap7 section 8

In all the examples of functions we have seen so far all variables have been declared and defined within the compound statements that form the body of the functions. Such variables are **local** to the functions and cannot be referred to outside the function, furthermore they cease to exist once the program control path leaves the function. Such local variables are created and initialised freshly each time the program control path enters the function. This point was mentioned earlier. An example illustrates the point.

```
void    function(void);
main()
{
        int     i=0;
        while(i<4)
        {
                printf("i = %d\n",i);
                function();
                i++;
        }
}
void    function(void)
{
        int     i = 4;
        printf("Initial Value of i = %d\n",i);
        i = 7;
        printf("  Final Value of i = %d\n",i);
}
```

producing the output

```
i = 0
Initial Value of i = 4
  Final Value of i = 7
i = 1
Initial Value of i = 4
  Final Value of i = 7
i = 2
Initial Value of i = 4
```

```
  Final Value of i = 7
i = 3
Initial Value of i = 4
  Final Value of i = 7
```

It will be noted that there are two variables called *i* whose values change independently.

If an aggregate is declared within a function then it too springs into existence when the program control path enters the function and goes out of existence on exit. If the aggregate is initialised then it is re-initialised every time the function is entered, the initialisation is actually performed by assignment and the inefficiencies involved meant that some pre-ANSI compiler writers did not allow such aggregates to be initialised.

It is possible to declare memory locations in such a way that they retain their values between function calls. This can be done in two ways, the memory locations declared can be publicly accessible from within all functions, this is sometimes called a **global declaration,** alternatively the memory locations can be made local to functions just like all the declarations we have seen so far but unlike them values are retained.

In the C programming language the various types of declaration are known as **storage classes.** There are four storage classes known as **auto, static, extern** and **register.** All the variables we have seen up to now have occupied memory locations of storage class auto.

Memory locations accessible from within all the functions of a program are of storage class extern. The syntax of such declarations is similar to that of auto storage class declarations with two important differences.

1. Such declarations must not lie within functions, they usually appear at the start of a program before the function *main().*
2. Memory locations in this storage class are guaranteed by the ANSI standard to be initialised to zero unless they are specifically initialised to some other value.

Memory locations that are local to a function and retain their values from one function call to the next are of storage class *static* . Such declarations are similar to *auto* storage class declarations in that they appear within functions but they are preceded by the keyword *static* . They are similar to *extern* storage class declarations in that zero initialisation is guaranteed.

An example of the use of extern storage is given below. This uses a "global" variable to return a value from a function.

```
int     sum;    /* storage class extern */
void    addup3(int,int,int);
main()
{
        int     n[]={1,2,3};
        addup3(n[0],n[1],n[2]);
        printf("Sum = %d\n",sum);
}
void    addup3(int a,int b,int c)
{
        sum = a+b+c;
}
```

producing the output

```
Sum = 6
```

The use of global variables in this way is generally not very good practice but it is sometimes useful to maintain a global set of flags that control the detailed behaviour of a program and possibly also global data buffers where the whole purpose of the program is to manipulate the contents of the buffers via the various functions.

The use of static storage is illustrated in the following program.

```
char    getnext(void);
main()
{
        int     c;
        int     i = 0;
        while(c=getnext())
        {
                if(c=='\n') printf(" ");
                else    printf("%c",c);
                if(++i == 20)
                {
                        printf("\n");
                        i = 0;
                }
        }
        printf("\n");
}
char    getnext(void)
{
        static  char    inbuf[256];
        static  char    *iptr = inbuf;
```

```
        int     slen;
        if(!*iptr)
        {
                gets(inbuf);
                iptr = inbuf;
                slen = strlen(inbuf);
                *(inbuf+slen) = '\n';
                *(inbuf+slen+1) = '\0';
                if(slen == 1 && *iptr == 'Z')
                        return 0;
        }
        return *iptr++;
}
```

This program reads in text and prints it out 20 characters to a line irrespective of the input line lengths. An input newline character is replaced by a space. Input is terminated when an input line consisting of the single character Z is encountered; there are much better ways of terminating input, they will be described later. The program was run using the following input data file, using the Unix host systems input redirection facility.

```
The cat sat on the mat 1234567890 times and got cold
today is thursday
this is a very short line
this is a long line ************************************
and the end of file follows
Z
```

and produced the output

```
The cat sat on the m
at 1234567890 times
and got cold today i
s thursday this is a
 very short line thi
s is a long line ***
********************
************* and th
e end of file follow
s
```

The particularly nice feature of this program is that the input buffer is entirely **private** to the function *getnext()*.

The keyword *auto* may be applied to *auto* storage class declarations but this is never

done in practice. The keyword *extern* must not be applied to *extern* storage class declarations, its significance will be discussed later.

The keyword *register* applied to a declaration implies that the object declared should occupy memory of storage class *register*. This means that, if possible, the variables should be allocated to processor registers rather than main memory, this makes for faster reference in time-critical applications. On many modern compilers, particularly those operating on multiple general purpose register architecture processors, the use of *register* makes no difference to the speed of the generated program, the compiler is clever enough to spot frequently referenced variables and keep them in registers anyway. If you are coding for a time critical application you may wish to experiment with *register* variables but be warned that the declaration of more *register* variables than the number of registers actually available may result in slower programs.

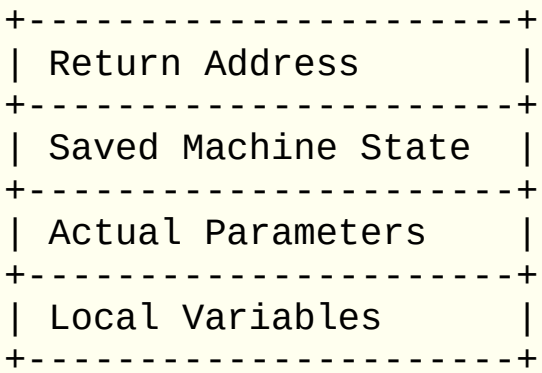
The Stack

Functions and storage organisation - The Stack

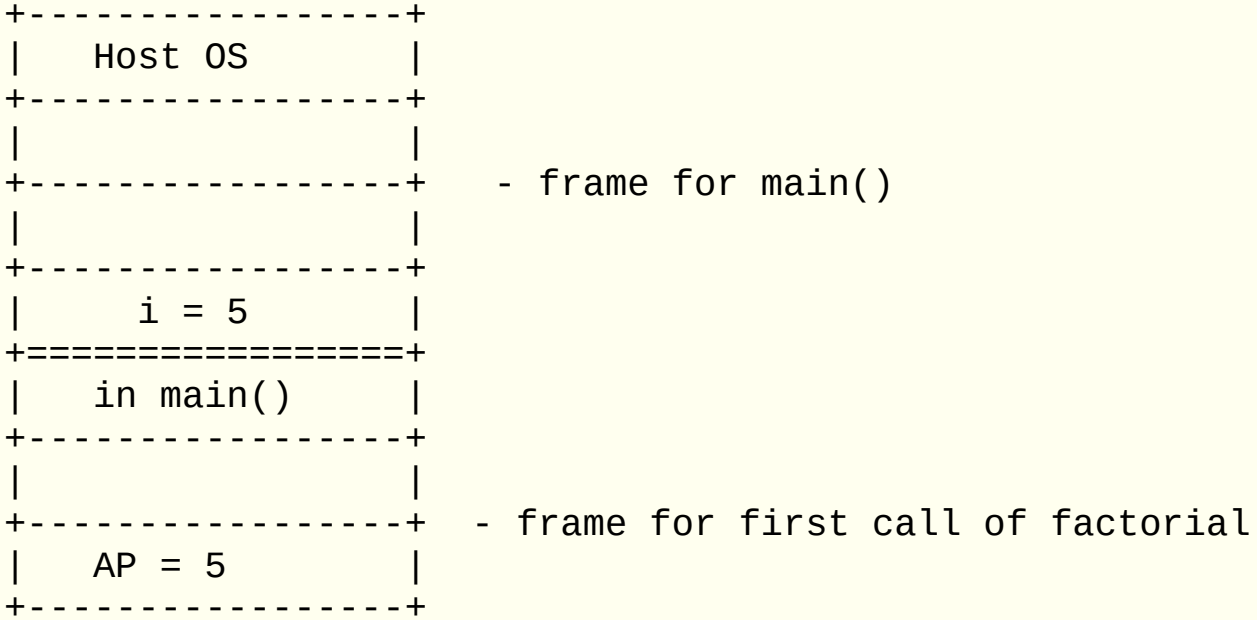
Chapter chap7 section 9

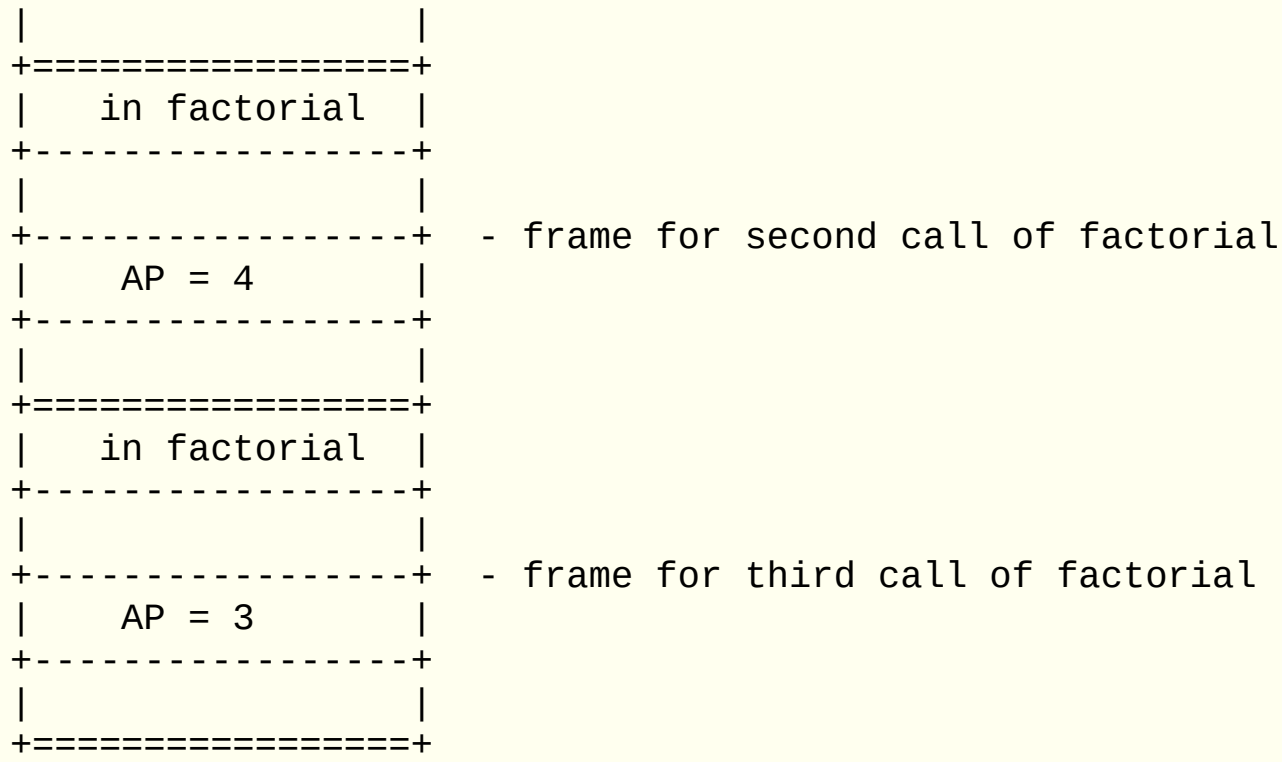
Normally when functions call other functions the processor maintains a data structure called a **stack .** This consists of a series of blocks of memory called **stack frames** that are stored consecutively. Each time a function is called a stack frame is built and when the function exits the stack frame is released.

The stack frame will include the **return address** from the function, the **machine state** on entry to the function, **copies of the actual parameters** and space for all the function's **local variables.** The following is fairly typical.



Static and extern storage class variables are not stack based. The following diagram shows the state of the stack during the execution of the factorial program shown earlier.





If a complex program fails producing a core dump in the Unix environment then there are tools to examine the stack that is part of the memory image written to the *core* file. Such a tool is **dbx** which may be invoked by typing

```
dbx <object file name> <core file name>
```

The core file name defaults to *core* and the object file name defaults to *a.out* . *"dbx"* will produce a **stack trace back** by typing i> where in response to dbx's prompt. This shows a history of all the functions that have called other functions. It may well include unfamiliar library functions. There are many other facilities available to the user of "dbx", see the manual for details.

- Storage qualifiers

Functions and storage organisation - Storage qualifiers

Chapter chap7 section 10

There are two storage qualifiers **const** and **volatile** that may be associated with a variable declaration.

The qualifier *const* implies that the variable may not be changed be program action. It may be changed by other means, such as the operation of input devices. It is important to realise that the *const* qualifier is **not** in any way direct way associated with the use of constants. The compiler will normally detect and report attempts to change variables with the *const* qualifier.

The qualifier *volatile* implies that the variable may change in ways outside the control of the program in an unpredictable fashion.

The *const* and *volatile* qualifiers are rarely used in normal programming but are useful when programming special purpose and dedicated hardware systems that control devices directly rather than via a host operating system. A common implication of the *const* qualifier is that memory locations in question may be associated with read only memory. Variables can readily have both *const* and *volatile* qualifiers. For example

```
const volatile int real_time_clock;
```

the implication being that the program is **not** going to change the clock setting and that the clock setting will be changed by other means.

Multi-Dimensional Aggregates and Arrays

The Pre-Processor and standard libraries - Introduction

Chapter chap8 section 1 In this chapter we will see how the C language pre-processor is used and we will also begin to look in detail at some of the standard libraries provided with the C language.

See also

- Compiling and Linking
- The #include pre-processor directive
- The standard headers
- The ctype macros and functions
- The stdio standard input and output macros and functions
- The mathematical macros and functions
- The string handling macros and functions
- The #define pre-processor directive
- Function like pre-processor macros
- The #ifdef pre-processor diretive
- Exercises

The Pre-Processor and standard libraries - Compiling and Linking

Chapter chap8 section 2

compiling a C language is usually a four stage process. These processes are

1. Preprocessing
2. Compilation to Assembly Language
3. Conversion of Assembly Language to Machine Code
4. Linking in Libraries

Under the Unix system it is possible to stop the compilation after each of these phases by special command line flags. The pre-processing phase is, essentially, a text substitution phase. The output of the pre-processing phase is still recognisable C language code, although parts can look very obscure. Preprocessing is controlled by various directives incorporated into the C language source code. A preprocessor directive is recognised by the fact that the first non-whitespace character on the source line is a "#" (hash).

The following pre-processor directives are understood by ANSI C compilers.

```
define  elif    else    endif
error   if      ifdef   ifndef
include line    pragma  undef
```

The most widely used directives are "define", "include" and "if -- endif". They are commonly referred to with the preceding "#" as, for example, "#define". Whitespace characters may appear between the "#" and the actual directive. A preprocessor directive is terminated by the end of the source line unless the last character on the source line is a backslash. A line consisting of just a "#" symbol has no effect, Unix programmers should **not** confuse this with the Bourne shell conventions for writing comments.

The commonest uses of the pre-processor are to handle **include** or **header** files and provide a facility for constants and macros, the latter are sometimes called **object-like** and **function-like** macros. The pre-processor behaves in a manner reminiscent of an editor relacing one text item by text from another source, possibly building the replacement text in accordance with fairly complex rules.

The #include pre-processor directive

The Pre-Processor and standard libraries - The #include directive

Chapter chap8 section 3

The **include** directive is used to switch compiler input to a named file. This is most commonly used to incorporate files containing prototypes for library functions, definitions of standard constants and definitions of data structures used by library functions.

The form of an *include* directive is either

```
        #include        <file-name>
or
        #include        "file-name"
```

An *include* directive is simply replaced by the entire contents of the named file. If the file-name is enclosed within angle brackets then the compiler looks in various standard places, if the file-name is enclosed in quotes then the compiler looks in the current directory before looking in the standard places. The actual set of places searched for files to include is implementation specific and the rules just quoted are also implementation specific. It is usually possible to change the compiler's idea of the standard places by setting an environment variable or a compiler command line flag ("-I" under Unix).

Files that are included in this fashion are commonly called **header** files or **include** files. They commonly have a ".h" suffix. It is possible to include compilable C code in an include file but this is regarded as bad practice and it can cause tricky problems when working with large programs, it is also very inefficient. If you are familiar with the Pascal programming language you may want to call these files *libraries* . This is completely wrong, all C compilation systems use the word library to refer to a collection of already compiled functions etc. You'll see how to construct your own libraries in a future chapter.

You can, of course, write your own header files and, in such cases, you can call them whatever you wish to call them. Include files may include other include files and so on to an unspecified maximum depth of nesting.

Standard headers

The Pre-Processor and standard libraries - Standard library header files

Chapter chap8 section 4

There are a number of header files specified in the C language standard. These are included into a user programmer via the pre-processor mechanism. They include useful constants defined using the #define pre-processor directive prototypes and some standard structures.

The standard headers are

1. **assert.h**

   Correctness assertions. A simple debugging aid. Produces run-time error messages that refer to the source line number. See the manual pages

2. **ctype.h**

   Character typing

3. **errno.h**

   Error reporting and definition of various error types and the global variable errno.

4. **float.h**

   Characteristics of floating point numbers on the particular implementation. Examine the file for details. This specifies precision, range etc., for all the floating point data types available. A typical such file is listed below.

```
#define FLT_ROUNDS      1

#define FLT_RADIX       2
#define FLT_MANT_DIG    24
#define FLT_EPSILON     1.19209290E-07F
#define FLT_DIG         6
#define FLT_MIN_EXP     (-125)
#define FLT_MIN         1.17549435E-38F
#define FLT_MIN_10_EXP  (-37)
#define FLT_MAX_EXP     (+128)
#define FLT_MAX         3.40282347E+38F
#define FLT_MAX_10_EXP  (+38)
```

```
#define DBL_MANT_DIG    53
#define DBL_EPSILON     2.2204460492503131E-16
#define DBL_DIG         15
#define DBL_MIN_EXP     (-1021)
#define DBL_MIN         2.2250738585072014E-308
#define DBL_MIN_10_EXP  (-307)
#define DBL_MAX_EXP     (+1024)
#define DBL_MAX         1.7976931348623157E+308
#define DBL_MAX_10_EXP  (+308)

#define LDBL_MANT_DIG    53
#define LDBL_EPSILON     2.2204460492503131E-16
#define LDBL_DIG         15
#define LDBL_MIN_EXP     (-1021)
#define LDBL_MIN         2.2250738585072014E-308
#define LDBL_MIN_10_EXP  (-307)
#define LDBL_MAX_EXP     (+1024)
#define LDBL_MAX         1.7976931348623157E+308
#define LDBL_MAX_10_EXP  (+308)
```

The meanings of most of these is fairly obvious. The initial **FLT , DBL** or **LDBL** refers to one the three floating point numeric data types. The items with **MAX** or **MIN** after the type give the largest and smallest positive number, those with **MANT_DIG** give the number of significant decimal digits, those with **EPSILON** give the least number that can be added to 1.0 and cause a change. To understand the other items remember floating point numbers are stored internally in the form

mantissa X $\text{radix}^{\text{exponent}}$

where mantissa is always numerically less than 1. The maximum and minimum exponent is defined both as a power of 10 (**MAX_10_EXP** and **MIN_10_EXP**) and as a power of the radix (**MAX_EXP** and **MIN_EXP**), the radix itself being **FLT_RADIX .** The number of radix digits (bits if the radix is 2) is defined by the **MANT_DIG** items. **FLT_ROUNDS** defines how floating point numbers are rounded, 1 means to the nearest number.

5. **limits.h**

   Characteristics of integers and characters on the particular implementation. This specifies maximum and minimum values. A typical usch file is listed below.

```
#define CHAR_BIT                8
#define SCHAR_MIN               (-128)
```

```
#define SCHAR_MAX                127
#define UCHAR_MAX                255
#define CHAR_MIN                 SCHAR_MIN
#define CHAR_MAX                 SCHAR_MAX
#define SHRT_MIN                 (-32768)
#define SHRT_MAX                 32767
#define USHRT_MAX                65535
#define INT_MIN                  (-2147483647-1)
#define INT_MAX                  2147483647
#define UINT_MAX                 4294967295
#define LONG_MIN                 (-2147483647-1)
#define LONG_MAX                 2147483647
#define ULONG_MAX                4294967295
#define MB_LEN_MAX               4
```

Again these are mostly obvious. Note that several of these are defined in terms of earlier definitions in the include file. The initial part of the define'd items refers to the various integer and character data types. **CHAR_BIT** is the number of bits in a character and **MB_LEN_MAX** is the maximum size of a multi-byte character that would be used with certain national character sets (such as Chinese).

6. **locale.h**

   This is concerned with internationalization and provides definitions for a number of variables that collectively define a **locale .** The values within the locale determine how certain functions display numbers, especially currency, dates and times. There are library functions for setting the locale.

   To support non-English character sets which typically include many more characters than the familiar ASCII codes, C implementations can use either **wide characters** as a distinct data type or can use **multi-byte characters.** There are special library functions for handling such things. They are referred to occasionally in these notes but not discussed further.

7. **math.h**

   This includes prototypes for common mathematical functions.

8. **setjmp.h**

   A facility for making non-local jumps. It is possible to associate labels with statements in C, however the scope of such labels is purely within the range of the function within which they are defined. The *setjmp()* and *longjmp()* functions and associated data structures declared here relax this restriction by providing support for the necessary stack unwinding.

9. **signal.h**

This provides some support via the *signal()* library function to catch and handle exceptional conditions that might be detected by the host operating system or hardware. The *raise()* library function may be used to signal such events. ANSI specifies the following special events as being detectable.

1. **SIGABRT** Generated by the *abort()* function
2. **SIGFPE** Generated by erroneous arithmetic operations
3. **SIGILL** Generated by an attempt to execute an illegal operation
4. **SIGINT** Generated by the receipt of an interactive attention signal
5. **SIGSEGV** Generated by an invalid attempt to access memory
6. **SIGTERM** A termination request sent to the program

The mechanisms that detect these conditions and the precise circumstances that cause them are, of course, host system dependent. The handling of signals is more fully developed in the Unix environment. For fuller information see the manual

10. **stdarg.h**

    Various things to allow the programmer to define and use functions with a variable number of parameters. Used to be called *varargs.h*

11. **stddef.h**

    Definitions of alternative names for some common data types that are widely used but may correspond to different basic types on different systems. ANSI specifies the following

    1. **ptrdiff_t** Type of variable used for storing difference between two pointers.
    2. **size_t** Type of result of sizeof operator
    3. **wchar_t** Type of object used for storing extended characters

12. **stdio.h**

    This contains prototypes and definitions for a large number of functions and constants that are primarily associated with file handling which is discussed more fully in the next chapter. There are also a number of function prototypes, macros and constants that are useful when working with standard input and standard output.

13. **stdlib.h**

    This header file contains prototypes for a number of generally useful library functions.

function name	function action
atoi	Ascii to integer

atof	Ascii to double
atol	Ascii to long
strtod	Ascii to double
strtol	Ascii to long
strtoul	Ascii to unsigned long
rand	Random number generator
srand	Seed random number generator
calloc	Allocate memory block
free	Release allocated memory block
malloc	Allocate memory block
realloc	Re-allocate memory block
abort	Cause abnormal termination
atexit	Defines function to be called when exit() is called
exit	Causes normal termination
getenv	Get environment information
system	Issue command to host operating system
bsearch	Binary search of sorted array
qsort	Sort array
abs	Integer absolute value
div	Performs integer division
labs	Long integer absolute value
ldiv	Performs long integer division

The functions *mblen(), mbtowc(), wctomb(), mbstowcs()* and *wcstombs()* are provided for handling multi-byte characters.

14. **string.h**

    Prototypes for string handling functions. These are discussed more fully later in this chapter.

15. **time.h**

    This includes prototypes and data type definitions for handling the date and time. Time is commonly stored either in system form which is the number of seconds since some arbitrary origin or epoch or it is stored in a date/time structure. The functions are

Function name	**Function action**

clock	Processor time used by program
difftime	Difference between two dates/times
mktime	Convert from date/time to system form
time	Determines time in system form
asctime	Converts date/time to string
ctime	Converts system time to string
gmtime	Converts system time to GMT date/time
localtime	Converts system time to local date/time
strftime	Converts date/time to formatted form. There are many formatting options. The behaviour is affected by the locale.

Some of these such as *assert.h* and *setjmp.h* are not often used. Others such as *signal.h* and *stdarg.h* are best left to a more advanced course. On Unix systems the include files are commonly found in the directory */usr/include*.

The ctype macros and functions

The standard input and output macros and functions

The mathematical macros and functions

The Pre-Processor and standard libraries - The ctype macros

Chapter chap8 section 5

One of the simplest and most useful standard headers is *ctype.h* . This defines a set of function-like macros that look like functions that can be used to determine the type of a character. All the macros look like

```
isathing(c)
```

and deliver a zero or non-zero value depending on whether the single character parameter has or does not have a certain property. The functions are

Macro name	Macro action
isalnum(c)	Non-zero for a letter or number
isalpha(c)	Non-zero for a letter
iscntrl(c)	Non-zero for a control character
isdigit(c)	Non-zero for a digit
isgraph(c)	Non-zero for a printing character (excl. space)
islower(c)	Non-zero for a lower case letter
isprint(c)	Non-zero for a printing character (incl. space)
ispunct(c)	Non-zero for any printing character excluding space, letters and numbers
isspace(c)	Non-zero for whitespace characters including space, form feed, new line, carriage return, horizontal tab and vertical tab.
isupper(c)	Non-zero for an upper case letter

A simple example is in order.

```
#include        <ctype.h>
main()
{
        char    inbuf[256];     /* input buffer */
        int     lcnt = 0;       /* letter count */
        int     ncnt = 0;       /* number count */
        int     scnt = 0;       /* space count */
        int     nchar = 0;      /* character count */
        char    c;
```

```
        printf("Enter String ");
        gets(inbuf);
        while(c = *(inbuf+nchar++))
        {
                if(isdigit(c)) ncnt++;
                if(isalpha(c)) lcnt++;
                if(isspace(c)) scnt++;
        }
        printf("%d characters\n%d digits\n"
                "%d letters\n%d spaces\n",
                nchar,ncnt,lcnt,scnt);
}
```

A typical dialogue is shown below.

```
Enter String the cat sat on the mat 1234567890 times
40 characters
10 digits
22 letters
7 spaces
```

The use of the "isathing" macros is preferable to writing code such as

```
if(c>='a' && c<='<z' || c>='A' && c<='Z')
```

because each "isathing" macro is converted into a table lookup and because it involves much less coding.

The Standard input and output macros and functions

Switch and For Statements, Command Line and File Handling - Introduction

Chapter chap9 section 1

In this chapter we will see how to process the command line arguments that will enable us to write C programs that can be used in the same way as normal operating system commands. We will also see how to do some elementary handling of files from within programs rather than using host system input and output redirection.

Before we do that, however, we are going to look at two C language statements that we have not discussed previously. These are the **for** and **switch** statements.

The Pre-Processor and standard libraries - Standard input and output functions and macros

Chapter chap8 section 6

The include file *stdio.h* contains a number of useful function declarations, macros and defined constants. Many of these are concerned with file handling but the following are more generally useful.

function name	function action
getchar()	Get character from standard input. This is usually a macro.
gets()	Gets string from standard input
printf()	Formatted write to standard output
putchar()	Puts character to standard output. This is usually a macro.
puts()	Puts string to standard output
scanf()	Formatted read from standard input
sprintf()	Formatted write to store
sscanf()	Formatted read from store

The prototypes for some of these functions, such as *printf()*, are only partial since they accept a variable argument list. As well as these functions *stdio.h* also includes definitions for a number of useful constants

constant name	meaning
BUFSIZ	Standard Buffer Size for I/O
EOF	End of File Coded Value
NULL	Pointer that doesn't point to anything

Just in case you're interested here's a typical macro-expansion of getchar().

```
(--(stdin)->_cnt>=0? ((int)*(stdin)->_ptr++):_filbuf(stdin))
```

stdin is a composite data type (see next chapter but 1) that includes both a count of available characters and a pointer to the next available character. *_filbuf()* is a library function to fill the buffer associated with the composite data object *stdin* .

A simple example is in order. This program, called *revlin* , reverses a line of input text.

```
#include        <stdio.h>
main()
{
        char    inbuf[BUFSIZ];
        int     c;
```

```
        int     nchar = 0;
        while((c=getchar()) != '\n') *(inbuf+nchar++) = c;
        while(nchar--) putchar(*(inbuf+nchar));
        putchar('\n');
}
```

A typical example of its operation

```
The cat sat on the mat 1234567890 times
semit 0987654321 tam eht no tas tac ehT
```

The use of *getchar()* is generally preferable to the complicated behaviour of *scanf()*. A more interesting example is the following program which does the same thing repeatedly until it encounters the end of the input file.

```
#include        <stdio.h>

main()
{
        char    inbuf[BUFSIZ];
        int     c;
        while(1)
        {
                int     nchar = 0;
                while((c=getchar()) != '\n')
                {
                        if(c==EOF) exit(0);
                         *(inbuf+nchar++) = c;
                }
                while(nchar--) putchar(*(inbuf+nchar));
                putchar('\n');
        }
}
```

This is what it did when presented with itself as input

```
>h.oidts<       edulcni#
)(niam
{
;]ZISFUB[fubni  rahc
;c      tni
)1(elihw
{
;0 = rahcn      tni
)'n\' =! ))(rahcteg=c((elihw
{
;)0(tixe )FOE==c(fi
;c = )++rahcn+fubni(*
}
;))rahcn+fubni(*(rahctup )--rahcn(elihw
```

```
;)'n\'(rahctup
}
}
```

The functions *getchar(), putchar()* and the constant *EOF* repay further study.

The function *getchar(),* actually a macro, reads a character from standard input and returns it as a functional value. The value of the function *getchar()* is of type *int* for a very important reason. This is that if the system calls that underlie *getchar()* detect an **end of file** condition then they need to return something that cannot be confused with any possible character. The only safe way of doing this is to return an integer with some of the bits in the high byte set, something that would never happen with valid character input (always assuming, of course, that you didn't have a Chinese keyboard). The constant value written as *EOF* is commonly -1 but you musn't assume this. It is a common beginner's mistake to declare the variable "c" of type *char* rather than type *int* in the preceding program, this results in high-byte truncation on assignment and the end of file condition never being seen.

If the input for a program is being taken from a file, using input re-direction under MSDOS or Unix, then it is clearly quite straightforward for the host operating system to detect the end of the file and raise the end-of-file condition. If the the program is taking input from un-redirected standard input (i.e. the keyboard) this is more difficult. The usual approach is for the keyboard driving software (that also does such things as echoing the input and making the ERASE key work properly) to detect a special key (commonly Ctrl-Z or Ctrl-D) at the start of an input line and use this circumstance to raise the end-of-file condition. Details, obviously, vary between host systems.

The function, again really a macro, *putchar(c)* takes a character as parameter and transfers it to the standard output.

It is useful to note that the function *gets(),* that we have already used, returns a character pointer. If the function completed successfully then the pointer points to the start of the buffer that the string was read in to, if the function fails or, more likely, encounters an *end of file* condition then *gets()* will return a pointer with the value *NULL.* A simple program, called *filean* , that reports on the contents of a file read from standard input illustrates its use.

```
#include        <stdio.h>
#include        <ctype.h>
main()
{
        char    inbuf[BUFSIZ];
        int     lcnt = 0;       /* letters */
        int     dcnt = 0;       /* digits */
        int     nchar = 0;      /* total number of characters */
        int     lines = 0;      /* number of lines */
        while(gets(inbuf) != NULL)
        {
                int     i=0;
                char    c;
                nchar += strlen(inbuf);
                lines++;
```

```
                while(c = *(inbuf+i++))
                {
                        if(isalpha(c))lcnt++;
                        if(isdigit(c))dcnt++;
                }
        }
        printf("%d lines\n%d characters\n"
                "%d letters\n%d digits\n",
                lines,nchar,lcnt,dcnt);
}
```

When asked to process the file *text* using the Unix command

```
filean < text
```

it produced the following output

```
195 lines
7309 characters
5917 letters
12 digits
```

This differs from the output produced by the Unix command

```
wc text
```

which produced the output

```
     195    1204    7504 text
```

This indicates 195 lines, 1204 "words" and 7504 characters. Because the *filean* program did not count the newline characters at the end of each line since these were converted to *NUL* characters by *gets()*.

The mathematical macros and functions

The Pre-Processor and standard libraries - Function Like Macros

Chapter chap8 section 10

An extension of the *#define* directive allows parameters to be associated with preprocessor directives in a notation very reminiscent of a function. These are normally called **macros** or **function-like macros .** *getchar(), putchar()* and the *isathing()* macros are examples that are defined in the header files stdio.h and ctype.h respectively.

The basic syntax is

```
#define identifier(identifier list) replacement-string
```

A simple example might be

```
#define cbroot(x) pow(x,1.0/3)
```

this means that code such as

```
cbroot(y+17)
```

is converted into

```
pow(y+17,1.0/3)
```

by the preprocessor. This is very handy, however there are some problems. Consider the example

```
#define square(a) a*a
```

This is quite satisfactory for usages such as

```
square(17)
```

which is converted to

```
17*17
```

but the conversion of

```
square(x+2)
```

to

```
x+2*x+2
```

is unlikely to have the effect that the programmer intended. To avoid this problem it

is essential to write

```
#define square(a) (a)*(a)
```

Once this has been done the preprocessor will convert

```
square(x+2)
```

to

```
(x+2)*(x+2)
```

Of course there are still some problems with usages such as

```
1/square(x+2)
```

being converted to

```
1/(x+2)*(x+2)
```

which always has the value 1. To solve this problem a macro such as square is always defined as

```
#define square(a) ((a)*(a))
```

the immediate previous example converting to

```
1/((x+2)*(x+2))
```

Unfortunately there are some problems that cannot be resolved. Consider the following program.

```
#define square(x)       ((x)*(x))
main()
{
        int     i=0;
        while(i<16)
                printf("%2d %4d\n",i,square(i++));
}
```

The output produced is

```
 2    0
 4    6
 6   20
 8   42
10   72
12  110
14  156
```

```
16  210
```

There is clearly something seriously wrong here. The loop has gone up in steps of two rather than one and the numbers in the right hand column aren't even squares. The macro expansion of "square(i++)" is

```
((i++)*(i++))
```

which, of course, increments "i" twice not once. Note also the right to left order of function parameter evaluation. This sort of problem causes most C programmers to use function-like macros with considerable care.

Of course macro definitions within comments are not seen by either the compiler or the pre-processor; comments are stripped from the source program before pre-processing. Macros and *#define'* d identifiers are not seen when they are referenced inside strings however the value associated with a pre-processor definition may be a string.

Also if the formal parameter associated with a macro is preceded by a "#" symbol in the replacement string then a string complete with enclosing quotes and all relevant escapes is formed. This operation is known as **stringizing.** It is illustrated by the following program.

```
#define string(x)       #x
main()
{
        printf(">>%s<<\n",string(a string));
        printf(">>%s<<\n",string(" quoted "));
}
```

producing the output

```
>>a string<<
>>" quoted "<<
```

The following example is more interesting. It also shows that a macro can be defined in terms of another macro which is in turn defined in terms of yet another macro.

```
#define showx(x)        printf( #x " = %d  ",(x))
#define show1(x)        showx(x);printf("\n")
#define show2(x,y)      showx(x);show1(y)
#define show3(x,y,z)    showx(x);show2(y,z)
#define show4(x,y,z,p)  showx(x);show3(y,z,p)

main()
```

```
{
        int     a = 4;
        int     b = 9;
        int     c = 7;
        int     d;
        show1(c+d);
        show3(a,b+c,d=a+b);
        show4(a*b,b*c,c*d,d*a);
}
```

producing the output

```
c+d = 7
a = 4  b+c = 16  d=a+b = 13
a*b = 36  b*c = 63  c*d = 91  d*a = 52
```

It might be useful to consider the steps in the expansion of *show1(c+d)* in the above program. The first step of macro expansion conversion changes this to

```
        "showx(c+d);printf("\n");
```

The expansion of showx() yields

```
        printf("c+d" " = %d  ",(c+d))
```

after the stringizing indicated by "#x". The rule concerning the concatenation of adjacent strings separated only by white space then operates to give a single layout specification. Warning - many supposed ANSI compilers don't seem to get this quite right, it should be used with care.

If it is not possible or convenient to write the complete text of a function-like macro on one line then it can be written over several lines if there is an escaping backslash immediately before the end of each line.

The pre-processor also provides facilities to paste together tokens to form a single token after pre-processing. This is called **token pasting** and is achieved using the ## symbol. This simply means join up the two items on either side of the ## as if they were a single text token. Token pasting is illustrated by the following program

```
#include        <stdio.h>
#define shown(x)        printf("%d",n##x)
main()
{
        int     n1 = 3;
        int     n2 = 4;
        int     n3 = 5;
```

```
        shown(1);
        shown(2);
        shown(3);
        putchar('\n');
}
```

producing the output

```
345
```

When the pre-processor processed *shown(1)* it generated

```
        printf("%d",n1)
```

rather than the

```
        printf("%d",n 1)
```

that would have been generated without pasting.

There are further complex rules defining the behaviour of the preprocessor when handling replacement strings which include identifiers that were *#define'* d in an earlier pre-processor directive.

The #ifdef and #endif directives

Switch and For Statements, Command Line and File Handling - Command Line Arguments

Chapter chap9 section 4

Under an operating system such as Unix or MSDOS a command may typically be issued by typing something such as

```
cut -d: -f2,3 data
```

Here *cut* is the name of a the command and the assumption is that there is an executable file called *cut* in a standard system place. The remaining items on the user supplied input are options or flags controlling the precise behaviour of the command. They are known as **command line arguments.**

A C program can always access the command line arguments associated with its invocation via formal parameters associated with the definition of the function *main()*. The definition of *main()* may typically start

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

the identifiers *argc* and *argv* are conventional. The value of *argc* is the number of command line arguments, including the name of the program and *argv* is an aggregate of character pointers that point to the actual arguments, each of which is presented to the program as a *NUL* terminated string. It is a simple matter to write a program such as that given below that simply echoes its command line arguments. The program is called *arg1* .

```
main(int argc, char *argv[])
{
        int     i = 0;
        printf("%d command line arguments\n",argc);
        do
                printf("Argument %d = >>%s<<\n",i,argv[i]);
        while(++i<argc);
}
```

and typical examples of use

```
$ arg1 a b c
4 command line arguments
Argument 0 = >>arg1<<
Argument 1 = >>a<<
Argument 2 = >>b<<
Argument 3 = >>c<<
$ arg1 test_data -f -deg
4 command line arguments
Argument 0 = >>arg1<<
Argument 1 = >>test_data<<
Argument 2 = >>-f<<
Argument 3 = >>-deg<<
$ arg1
1 command line arguments
```

```
Argument 0 = >>arg1<<
$
```

The symbols ">>" and "<<" are used to make the limits of the strings clear in the output.

It should be understood that the host operating system is likely to manipulate the command line arguments typed by the user into something quite different. Before designing applications that depend on the command line arguments some experimentation is wise. For example typing

```
arg1 *
```

on the Unix system used to write these notes gave the result

```
5 command line arguments
Argument 0 = >>arg1<<
Argument 1 = >>arg1<<
Argument 2 = >>arg1.c<<
Argument 3 = >>arg1.log<<
Argument 4 = >>text<<
```

whereas under MSDOS (and the Turbo C compiler) the following results were obtained.

```
2 Command line arguments
Argument 0 = >>.\arg1.exe<<
Argument 1 = >>*<<
```

The reason for the difference is nothing to do with the C programming language but stems from the fact that the Unix command interpreting shell expands the file naming wild card "*" into a list of file names whereas MSDOS's COMMAND.COM does not and requires the user program to make various subtle system calls to perform this expansion. As stated above the possibility of such differences must be remembered when designing programs that take information from the command line.

A common alternative declaration for the function *main()* is

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

and it would be possible to re-write the previous program to read.

```
main(int argc, char **argv)
{
        printf("%d Arguments\n",argc);
        while(*argv) printf("%s\n",*argv++);
}
```

The operation is similar to the previous example. Its operation depends, amongst other things, on the fact that *argv[argc]* is guaranteed by the ANSI standard to be *NULL* so the loop will terminate properly. The ANSI standard also guarantees that the strings *argv[]* retain their values throughout the execution of the program. They may also be changed by the program though this is seldom useful or good practice.

A common requirement is to recognise command line arguments, possibly with associated values and set flags that will control the behaviour of parts of the program. Such command line arguments may typically look like

Typical argument	Type
-f	A "binary" flag. Either present or not present.
-t:	A single character value. The argument introduces a value ":" in this case.
-x1234	A flag introducing a numeric value

-ddata	A flag introducing a string value

Typical code to recognise flags of these various types is shown in the example below.

```
int     fflg = 0;       /* 1 implies -f seen */
char    tchar = 'x';    /* character from -t option */
int     xval = 0;       /* value from -x option */
char    filename[50];   /* string from -d option */
main(int argc, char *argv[])
{
        int     i;
        for(i=1;i<argc;i++)
        {
                if(argv[i][0] == '-')
                {
                        switch(argv[i][1])
                        {
                        case 'f' :
                                fflg = 1;
                                break;
                        case 't' :
                                tchar = argv[i][2];
                                break;
                        case 'x' :
                                xval = atoi(argv[i]+2);
                                break;
                        case 'd' :
                                strcpy(filename,argv[i]+2);
                                break;
                        default :
                                printf("Unknown option %s\n",argv[i]);
                        }
                }
        }
        if(fflg) printf("-f seen\n");
        printf("\"tchar\" is %c\n",tchar);
        printf("x value is %d\n",xval);
        printf("File name is %s\n",filename);
}
```

and some typical logs

```
$ arg3 -t^ -x44 -dfred
"tchar" is ^
x value is 44
File name is fred
$ arg3 -ddata -f
-f seen
"tchar" is x
x value is 0
File name is data
$ arg3 -v -t5
```

```
Unknown option -v
"tchar" is 5
x value is 0
File name is
$
```

You should note that the options are sensed correctly irrespective of their order on the command line and that the variables associated with the options retain their initial values if the options do not appear on the command line. File Handling

Switch and For Statements, Command Line and File Handling - Direct Access Files

Chapter chap9 section 7

The C language view of a file is to see it as an array of bytes. This means that all files may be treated as direct access files and the library function **fseek()** may be used to provide immediate access to any particular part of the file. This assumes that the start byte address of the record is known. The following program demonstrates the construction and use of a table of record start positions using **ftell().**

```
/*      program to display selected
        records from a file */

#include        <stdio.h>
#define MAXREC  1000
long    index[MAXREC];

int     makeindex(FILE *);
void    showrec(FILE *,int);

main(int argc, char *argv[])
{
        int     nrec;   /* record to show */
        int     nrecs;  /* records in file */
        FILE    *d;     /* the data file */
        if(argc != 2)
        {
                fprintf(stderr,"Usage : showrec f\n");
                exit(1);
        }
        if((d=fopen(argv[1],"r"))==NULL)
        {
                fprintf(stderr,"error opening %s\n",
                                        argv[1]);
                exit(1);
        }
        if((nrecs=makeindex(d))==0)
        {
```

```
                fprintf(stderr,
                        "file too big or empty\n");
                exit(1);
        }
        printf("%d records\n",nrecs);
        while(1)
        {
                printf("Enter record number ");
                scanf("%d",&nrec);
                if(nrec<0) exit(0);
                if(nrec >= nrecs)
                        printf("Out of range\n");
                else showrec(d,nrec);
        }
}
int     makeindex(FILE *f)
/*      builds the (global) index table
        and returns the number of records
        or 0 if the file has too many
        records or is empty
*/
{
        int     c;
        int     i=0;
        while(1)
        {
                if((c=getc(f))=='\n')
                {
                        index[++i]=ftell(f);
                        if(i==MAXREC) return 0;
                }
                else
                        if(c==EOF) return i;
        }
}
void    showrec(FILE *f, int n)
/*      display required record - simply copies
        characters to stdout
*/
{
        char    c;
        fseek(f,index[n],0);
        while(1)
```

```
        {
                putchar((c=getc(f)));
                if(c=='\n') return;
        }
}
```

A typical dialogue using the source file as input is shown below.

```
$ fil3 fil3.c
76 records
Enter record number 44
        or 0 if the file has too many
Enter record number 11
        FILE    *d;     /* the data file */
Enter record number 76
Out of range
Enter record number 21
                exit(1);
Enter record number -1
```

There are a number of interesting points here. The direct access functions always work with long integers and it is traditional to declare the associated variables as being of type *long int* . The record numbering starts at zero and the file examination part of the program is terminated by a negative input. Strictly the final parameter of *fseek()* ought to have been *SEEK_SET* not zero.

The value returned by *ftell()* is the byte position of the byte about to be read from the file so when a newline is encountered this is the start address of the next record.

The functions **fsetpos()** and **fgetpos()** do the same things as *fseek()* and *ftell()* only they use parameters of type **fpos_t** rather than *long int*. This, potentially, allows for larger files to be handled and the use of these functions is to be preferred.

Records and Fields

Switch and For Statements, Command Line and File Handling - Records and Fields

Chapter chap9 section 8

For many applications it is necessary, or useful, to regard the records of a file as consisting of a set of fields. The standard file handling functions provide no support for fields. There are two common approaches to splitting records into fields. The first approach, understood by many Unix utilities, is to designate a special character as the **field separator** character, this allows a variable number of variable width fields; the alternative is to define fields as being of a specific width.

When handling data organised into fields within records it is often useful to represent the record as a sequence of strings. The following program shows how this might be done.

```
#include        <stdio.h>
#define MAXCHAR 120     /* maximum record size */
#define MAXFLDS 20      /* maximum number of fields */
#define SEPCHAR ':'     /* field separator */
main()
{
        char    ibf[MAXCHAR];   /* input buffer */
        char    *fields[MAXFLDS];  /* field pointers */
        int     i;
        int     j;
        int     nfields;        /* number of fields */
        while(gets(ibf)!=NULL)
        {
                fields[0] = ibf;
                i=0;
                for(j=0;ibf[j];)
                {
                        if(ibf[j]==SEPCHAR)
                        {
                              fields[++i] = &ibf[j+1];
                              ibf[j]='\0';
                        }
                        j++;
                }
                nfields = i+1;
                printf("%d fields\n",nfields);
                for(j=0;j<nfields;j++)
```

```
                printf("field %2d >>%s<<\n",j,fields[j]);
        }
}
```

The program, which reads the data from its standard input, produced the following output

```
3 fields
field  0 >>a<<
field  1 >>b<<
field  2 >>c<<
1 fields
field  0 >>some data<<
2 fields
field  0 >><<
field  1 >>starts with a null field<<
2 fields
field  0 >>ends with a null field<<
field  1 >><<
```

when presented with the input

```
a:b:c
some data
:starts with a null field
ends with a null field:
```

Exercises

Switch and For Statements, Command Line and File Handling - Exercises

Chapter chap9 section 9

1. Write a program that will evaluate a simple arithmetic expression read from the command line. I.e. the user will type

   ```
   myprog 3+4
   ```

   and the program will respond with 7.
2. Write a version of the file copying program that prompts the user for the source and destination file names.
3. Write a version of the file copying program that takes the source and destination file names from the command line. Devise suitable tests to detect the incorrect number of command line arguments and invalid file names. Modify your program to determine whether the destination file already exists and generate a suitable warning if it does.
4. Write a program using nested for loops to print out a multiplication table. See earlier notes for an example of this done using the while statement.

Structures, Unions and Typedefs - Introduction

Chapter chap10 section 1

In this chapter we will see how to construct data objects with complex internal structure. We will also see how to access the components of such objects and the operations that can be performed on the objects themselves.

See also

- Structures (basic)
- Structures (examples)
- Structures (use, basic)
- Structures (use, extended)
- Unions
- Alignment Constraints
- Typedefs
- Bit Fields
- enum data types

Structures, Unions and Typedefs - Basic Structures

Chapter chap10 section 2

A C **structure** is a compound data object. A compound data object consists of a collection of data objects of, possibly, different types. It may be thought of as a private or user defined data type as distinct from the standard data types that are provided by the C programming language.

The C programming language provides facilities to declare such objects which means define their internal structure via a **template** and to declare a **tag** to be associated with such objects so that it is not necessary to repeat the definition. Given both the declaration of the structure and the associated tag it is only necessary to use the tag when declaring actual instances of structures. A simple example of a structure declaration and definition is given below. The C keyword " **struct** is used to indicate that structures are being defined and declared.

```
struct  date    /* the tag */
{       /* start of template */
        int     day;    /* a member */
        int     month;  /* a member */
        int     year;   /* a member */
        char    dow;    /* a member */
} dates[MAXDAT],today,*next;    /* instances */
```

dates is an aggregate of instances of " *struct date* , *today* is a simple instance of a *struct date* and next is simply a pointer to a *struct date* . Once the code given above has appeared in the program, further instances of *struct date* can be declared in the following manner.

```
struct  date    my_birthday;
struct  date    end_of_term;
```

In this simple example the structure tag is " *Tag names conform to the same rules as variable names but belong to a separate "universe" so a variable and a tag can have the same name. It is, thus, quite legal and perfectly acceptable to write*

```
struct date date;
```

*The template tells the compiler how the structure is laid out in memory and gives details of the **member** names. A (tagged) template does not reserve any instances of the structure, it only tells the compiler what it means.*

Structure member declarations conform to the same syntax as ordinary variable declarations. Structure member names should conform to the same syntax as ordinary variable names and structure tags but again belong to a different "universe". I.e. the same name could be used for a structure tag, an instance of the structure and a member of the structure. Each structure defines a separate universe as far as naming structure members is concerned.

The following rather bizarre and confusing code is perfectly legal

```
struct  u
{
        int     u;
        int     v;
} v;
struct  v
{
        char    v;
        char    u;
} u;
```

Pre-ANSI C compilers often had rather more restrictive rules about the separate naming universes.

Structure members can be any valid data type, including other structures, aggregates and pointers including pointers to structures and pointers to functions. A structure may not, for obvious reasons, contain instances of itself but may contain pointers to instances of itself.

Structures may be initialised in the same fashion as aggregates using initialisers. for example

```
struct date Christmas = {25,12,1988,3};
```

Individual members of a structure may be referred as shown in the following examples

```
dates[k].year
today.month
(*next).day
```

The . (dot) operator selects a particular member from a structure. It has the same precedence as () and [] which is higher than that of any unary or binary operator. Like () and [] it associates left to right. The basic syntax is

```
<structure name>.<member name>
```

The syntax requires that the first component be a structure so

```
dates.year[k] /* WRONG */
```

would be wrong because dates is an aggregate of structures and year is not an aggregate, similarly

```
*next.day /* WRONG */
```

*would be wrong because the . (dot) operator has a higher priority than the * (star) operator. This incorrect usage attempts to use next as a structure and access the object whose address is in the member day of this structure.*

The correct way of referring to a member of a structure whose address is given is typically

```
(*next).day
```

This is so common that the alternative syntax

```
next->day
```

is part of the C Programming language. The -> operator has the same precedence as the . (dot) operator.

Structures may be assigned, used as formal function parameters and returned as functional values. Such operations cause the compiler to generate sequences of load and store instructions that might pose efficiency problems. C programmers particularly concerned about program speed will avoid such things and work exclusively with pointers to functions.

There are few actual operations that can be performed on structures as distinct from their members. The only operators that can be validly associated with structures are "=" (simple assignment) and "&" (take the address). It is not possible to compare structures for equality using "==", nor is it possible to perform arithmetic on structures. Such operations need to be explicitly coded in terms of operations on the members of the structure.

Simple examples of structures

Structures, Unions and Typedefs - Simple examples of structures

Chapter chap10 section 3

The following simple structure declarations might be found in a graphics environment.

```
struct point
{
        double  x;
        double  y;
};

struct circle
{
        double rad;
        struct point cen;
};
```

With the declarations given above the following simple C function may be written to give the area of a circle

```
double area(struct circle circle)
{
        return PI*circle.rad*circle.rad;
}
```

The example assumes that PI was #define'd suitably. To determine whether a given point lay inside a circle the following C function could be used.

```
incircle(struct point point,struct circle circle)
{
        double dx,dy;
        dx = point.x - circle.cen.x;
        dy = point.y - circle.cen.y;
        return dx*dx+dy*dy <= circle.rad*circle.rad;
}
```

Further graphics structure declarations such as

```
struct line
{
        struct point start;
        struct point end;
};
```

and

```
struct triangle
{
        struct point pt[3];
};
```

can be made in a natural and useful fashion.

Use of Structures

Structures, Unions and Typedefs - Use of Structures

Chapter chap10 section 4

This section presents and discusses a program that makes use of structures. The first version of the program is listed below. The purpose of the program is to read text from the standard input and produce a list of the number of times each distinct word occurs in the input. The first version of the program requires that the input be presented one word per line.

```
#include        <string.h>
#include        <stdio.h>

#define MAXWRDS 1000
#define MAXCHRS 4000

struct  word
{
        char    *text;  /* pointer to word text */
        int     count;  /* number of occurences */
}       word[MAXWRDS];

int     nwords;         /* number of different words */

void    findwrd(char *);
void    showwrds(void);

main()
{
        char    inbuf[BUFSIZ];
        while(gets(inbuf))
                findwrd(inbuf);
        printf("%d different words\n",nwords);
        showwrds();
}
void    findwrd(char *s)
{
        static  char    wspace[MAXCHRS];
        static  char    *wptr = wspace;
        int     i=0;
        int     l;
        while(word[i].count)    /* look for word */
        {
                if(strcmp(s,word[i].text)==0)
                {
```

```
                        word[i].count++;
                        return;
                }
                i++;
        }       /* didn't find it */
        if((l=strlen(s))+wptr >= wspace+MAXCHRS)
        {
                fprintf(stderr,"Out of string space\n");
                exit(1);
        }
        strcpy(wptr,s);         /* save it */
        if(i>=MAXWRDS)
        {
                fprintf(stderr,"Out of word space\n");
                exit(1);
        }
        word[i].text = wptr;    /* save info */
        word[i].count = 1;
        nwords++;
        wptr += l+1;
}
void    showwrds(void)
{
        int     i=0;
        while(word[i].count)
        {
                printf("%2d %s\n",word[i].count,word[i].text);
                i++;
        }
}
```

When presented with the following data file

```
this
is
the
data
the
cat
sat
on
the
mat
```

it produced the following output

```
8 different words
 1 this
```

```
1 is
3 the
1 data
1 cat
1 sat
1 on
1 mat
```

Of course this is not a very convenient way of presenting input.

The following features of the program should be noted. The program starts by defining the structure *word* that will hold the frequency count and a pointer to the place where the text of the word is stored. It is much better to store a pointer to the place where the word is stored rather than storing the actual text of the word in the structure because, in the latter case, it would be necessary to reserve enough space within every instance of *struct word* to hold the complete text of the largest possible word. It is difficult to predict the largest size and the operation is likely to be very wasteful of memory.

What is hoped to be a suitable number of instances of *struct word* are declared, in a future chapter we shall see to create new instances of *struct word* dynamically. The aggregate *word* is of storage class extern for two reasons. The first is to take advantage of the default zero initialisation of objects of this storage class and the second is that its manipulation is the main function of the program so it is likely to be referenced by most of the various functions of the program.

Most of the work of the program is performed in the function *findwrd()* This maintains the set of words seen so far in the private buffer *words* , it also maintains a private pointer *wptr* to the next free location in this buffer. The first task of *findwrd()* is to search through the aggregate *word* to see if the word has already been encountered, this is done very simply by evaluating the expression

```
strcmp(s,word[i].text)
```

If the word has been found then the associated count is incremented and the program returns from *findwrd()* If the word is not found then it has to be copied into the text buffer and the relevant data stored in the next member of the aggregate *word* Checks are made that there is enough free space to store the information.

The function *showwrds()* simply lists the stored information.

A natural next step in the development of this program is to list the words in some sort of order, the most natural being in order of decresaing frequency and alphabetic order if they occur the same number of times. This can, of course, be achieved by post-processing the output using the system *sort* utility but it is instructive to see how it is done using the library function *qsort()* The program listed above requires three modifications, the first is to call the function *qsort()* immediately before *showwrds()* , the second is to define and declare the function *compare()* and the third is to include *stdlib.h* to get the prototype for *qsort()* The prototype of *compare()* is

```
int compare(struct word *,struct word *);
```

Rather then repeat the complete program the functions *main()* and *compare()* are listed below.

```
        .
int     compare(struct word *,struct word *);
        .
        .
        .
        .
        .
main()
{
        char    inbuf[BUFSIZ];
        int     (*comp)() = compare;
        while(gets(inbuf))
                findwrd(inbuf);
        printf("%d different words\n",nwords);
        qsort(word,nwords,sizeof (struct word),comp);
        showwrds();
}
        .
        .
        .
int     compare(struct word *w1,struct word *w2)
{
        if(w1->count != w2->count)
                return w2->count - w1->count;
        return strcmp(w1->text,w2->text);
}
```

main() has been modified to call *qsort()* In order to provide *qsort()* with parameters of the correct type, it was necessary to declare a variable *comp* of type pointer to integer valued function.

The library function *qsort()* operates by repeatedly calling the comparison function provided as a parameter to determine whether to re-arrange the elements of the array. It provides the comparison function with pointers to the two elements that are to be compared. Within the comparison function note the use of the "->" operator. Note also that the first use of return, which is used if the counts are different has the returned value set up in such a way that the sort is in reverse numerical order, i.e. larger counts sort low. If the counts are the same the value of the *strcmp()* function provides the necessary ordering information.

The output produced by processing the test data displayed earlier is now

```
8 different words
 3 the
 1 cat
 1 data
```

```
 1 is
 1 mat
 1 on
 1 sat
 1 this
```

A more realistic test is for the modified program to process its own source code after this has been pre-processed by the Unix utility *tr* convert all sequences of non-alphabetic characters into single newline characters. The first few lines of the output were

```
63 different words
19 word
13 i
10 count
 8 int
 8 w
 7 text
 6 char
 6 struct
 6 void
 5 s
 5 wptr
 4 if
 4 n
 4 nwords
 4 of
 3 MAXCHRS
 3 MAXWRDS
```

Structure (use - extended)

Structures, Unions and Typedefs - Extended use of structures

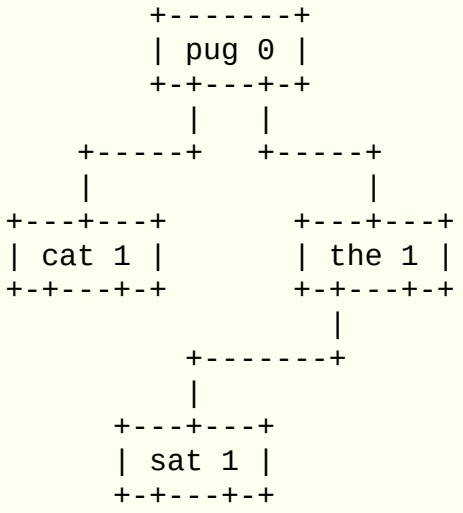
Chapter chap10 section 5

Apart from the problems caused by the required input layout which can always be solved by pre-processing, the technique of searching through the words seen so far is rather clumsy. It would be much easier if we kept the words in some sort of order. The next version of the program does this by organising the storage of information about words in a more elaborate fashion. Rather than having a simple aggregate of *struct word* each instance of a *struct word* is, potentially, associated with two other instances of *struct word* The association or linkage is maintained by two extra elements in a *struct word* , these are both pointers to other instances of a *struct word* The first such pointer points to an instance of *struct word* representing text that is less than that in the main instance and the second points to an instance with text greater than. The declaration of *struct word* now becomes

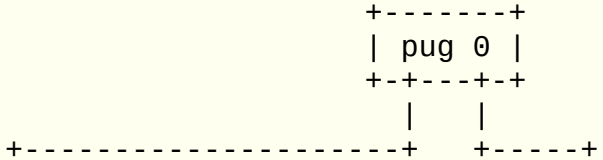
```
        struct  word
        {
                int     count;
                char    *text;
                struct  word    *lptr;
                struct  word    *rptr;
        }
```

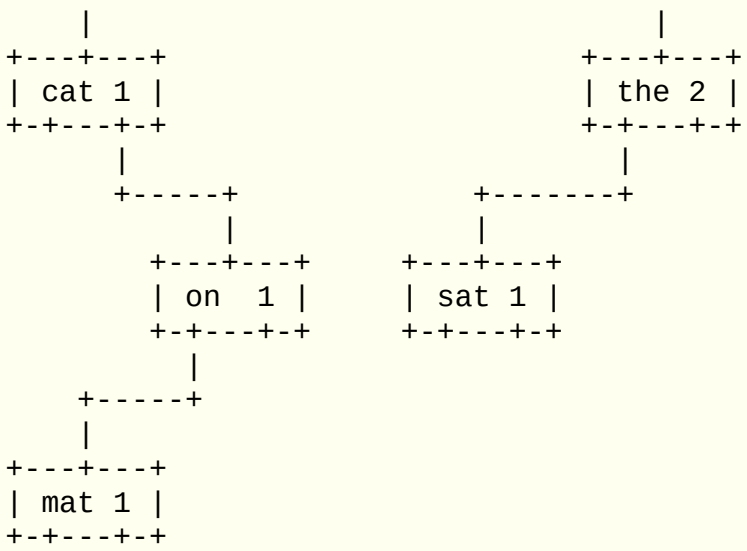
As the program processes input words it builds a hierarchy of such structures, this hierarchy is called a " **tree** , for reasons that are obvious if you draw a diagram showing how the entries point to other instances. When a word is encountered, a very simple search will either locate it or come to a pointer that doesn't point to anything (a NULL pointer).

The following diagram shows how the data would be organised after processing the sequence of words "the", "cat" and "sat".



The entry for "pug" is a dummy entry that represents the start of the tree. After further processing the words "on", "the" and "mat", the data organisation would look like





The program now looks very different. It is listed in separate sections. First the global declarations.

```
#include <stdio.h>

#define MAXCHS  4000    /* maximum characters */
#define MAXWDS  800     /* maximum different words */
int     wdct;           /* number of different words */
int     maxocc;         /* maximum count */

struct word
   {
         int     count;          /* occurences */
         char    *body;          /* actual text */
         struct  word    *lptr;  /* left pointer */
         struct  word    *rptr;  /* right pointer */
   };

void    putword(char *,struct word *);
void    listwords(struct word *,int);
void    error(int);
struct  word *new(char *);
```

This part of the code defines some constants, the compound data type *struct word* and the two global variables *wdct* and *maxocc* It also includes the function prototypes and *#include* 's *stdio.h* The next part is the function *main()*

```
main()
{
        char    inbuf[BUFSIZ];
        int     i;
        struct word *root;
        root=new("ZZZZ");  /* establish root of tree */
        root->count=0;     /* it's only a dummy entry !! */
        while(gets(inbuf))      putword(inbuf,root);
        printf("%d Different Words\n",wdct-1);
        for(i=maxocc;i;i--)listwords(root,i);
}
```

This calls the function *new()* to obtain a pointer to a currently unused instance of a *struct word* and makes this

the root of the linked collection of data. Repeated calls to *putword()* process each input word as it is encountered and finally the function *listwords()* displays all the words that have occurred a specific number of times. The next listing shows the function *putword()* that stores information about the current word.

```
void    putword(char *s,struct  word *w)
/*      Routine to increment word occurence
        count for words already encountered
        and create a properly linked tree
        leaf for words not already encountered.
        The arguments are :-
        (1)     word in input
        (2)     tree leaf to look at
        The routine descends recursively if the
        current leaf is not the required one.
*/
{
        int     flag;
        flag = strcmp(s,w->body);
        if(flag==0)
        {
                if(++w->count > maxocc) maxocc = w->count;
                return;
        }
        if(flag<0)
        {
                if(w->lptr==NULL)
                        w->lptr = new(s);
                else
                        putword(s,w->lptr);
        }
        else
        {
                if(w->rptr==NULL)
                        w->rptr = new(s);
                else
                        putword(s,w->rptr);
        }
}
```

This is fairly straightforward. The input word is compared with the word in the current instance of *struct word* , if they match the count is incremented and the function returns. If they do not match that the left or right pointers are examined as appropriate. If the pointer examined is zero then a new instance of a *struct word* is obtained from *new()* , if the pointer is not zero then the function *putword()* is examined with this instance of *struct word* pointed to as parameter to see if the input word matches the text associated with the *struct word* instance pointed to.

The next listing is the function *new()*

```
struct word *new(char *s)
/*      Routine to set up new tree leaf containing information concerning
        the word supplied as argument. The function returns a pointer to
        the tree leaf which is used by the putword function to link it up
        properly with the rest of the tree.
*/
{
        static  char    text[MAXCHS];
```

```
        static  char    *tptr=text;
        static  struct  word    words[MAXWDS];
        struct  word     *w;
        int     slen;
        if(wdct>MAXWDS) error(2);
        slen = strlen(s) + 1;
        if(tptr+slen > text+MAXCHS) error(3);
        strcpy(tptr,s);
        w = &(words[wdct++]);
        w->body = tptr;
        tptr += slen;
        w->count = 1;
        w->lptr = NULL;
        w->rptr = NULL;
        return(w);
}
```

This function maintains a private aggregate of *struct word* This can be thought of as a pool of objects of this type, the function fishes fresh objects from the pool, initialises them and returns their address. It also maintains the internal buffer holding the bodies of the actual input words.

The final listing is the functions *listwords()* and *error()*

```
void    listwords(struct        word *w,int count)
/*      Simple recursive tree walking   */
{
        if(w->lptr!=NULL)listwords(w->lptr,count);
        if(w->count == count)
                printf("%3d %s \n",w->count,w->body);
        if(w->rptr!=NULL)listwords(w->rptr,count);
}
void    error(int n)
/*      Error reporting routine - all errors are fatal  */
{
        printf("Error %d : ",n);
        exit(0);
}
```

These do not require any detailed explanation.

It should be noted that both the functions *putword()* and *listwords()* are recursive.

The program is still open to criticism of its arbitrary choice of the amount of space set aside for *struct word* objects and the storage of input word text. This problem can be resolved by using the library function *malloc()* to allocate space as required from the host system. Some minor change were necessary to the program, these were the inclusion of *stdlib.h* for *malloc()* 's prototype and the deletion of the MAXWRDS and MAXCHARS *#define* 's. The function *new()* has been completely re-written and the new version looks like

```
struct word *new(char *s)
/*      Routine to set up new tree leaf
        containing information concerning
        the word supplied as argument.
        The function returns a pointer to
        the tree leaf which is used by the
        putword function to link it up
        properly with the rest of the tree.
*/
{
```

```
        int     slen;
        struct  word    *w;
        if((w = (struct word *)
                malloc(sizeof (struct word)))==NULL) error(2);
        slen = strlen(s) + 1;
        if((w->body = (char *)malloc(slen))==NULL) error(3);
        strcpy(w->body,s);
        w->count = 1;
        w->lptr = NULL;
        w->rptr = NULL;
        wdct++;
        return w;
}
```

The most interesting point is the use of the function *malloc()* This takes a single parameter which is the amount of space to be allocated, the function returns a generic pointer (*void* *) to the freshly allocated space, this has to be cast to a pointer to the relevant data type, *struct word* in this case. If it is not possible to allocate the required amount of memory, *malloc()* returns NULL. The initial contents of the freshly allocated memory are indeterminate.

The similar function *calloc()* could have been used, this returns a memory area with all bits set to zero but it requires an extra parameter to indicate how many objects to allocate. Using *calloc()* , the function *new()* could be re-written as

```
struct word *new(char *s)
/*      Routine to set up new tree leaf
        containing information concerning
        the word supplied as argument.
        The function returns a pointer to
        the tree leaf which is used by the
        putword function to link it up
        properly with the rest of the tree.
*/
{
        int     slen;
        struct  word    *w;
        if((w = (struct word *)
                calloc(1,sizeof (struct word)))==NULL) error(2);
        slen = strlen(s) + 1;
        if((w->body = (char *)
                calloc(slen,sizeof (char)))==NULL) error(3);
        strcpy(w->body,s);
        w->count = 1;
        wdct++;
        return w;
}
```

The setting of *w->lptr* and *w->rptr* to NULL is no longer necessary.

Memory blocks allocated by *calloc()* or *malloc()* are administered by these functions as a collection of blocks often called the **heap** . Reference to locations outside an allocated block can break the heap . The function free() may be used to release an allocated block.

Unions

Structures, Unions and Typedefs - Unions

Chapter chap10 section 6

The members of a structure are laid out in memory one after the other, a **union** is syntactically identical (except that the keyword union is used instead of struct). The difference between a struct and a union is that in a union the members overlap each other. The name of a structure member represents the offset of that member from the start of the structure, in a union all members start at the same location in memory. The members of a union may themselves be structs and the members of a struct may themselves be unions.

A typical application is illustrated by the following code fragment. If data, in the form of floating point numbers in internal form is stored in a file then it is difficult to read the file since all the standard C file handling functions operate character by character. The fragment shown below resolves the difficulty by using a union whose two members consist of a character array and a floating point number. It is assumed that a floating point number occupies 8 characters.

```
union ibf
{
        char    c[8];
        double  x;
} ibf;

double values[...];


for(i=0;i<8;i++) ibf.c[i] = getc(ifp);
values[j] = ibf.x;
```

Typdefs

Structures, Unions and Typedefs - Alignment Constraints

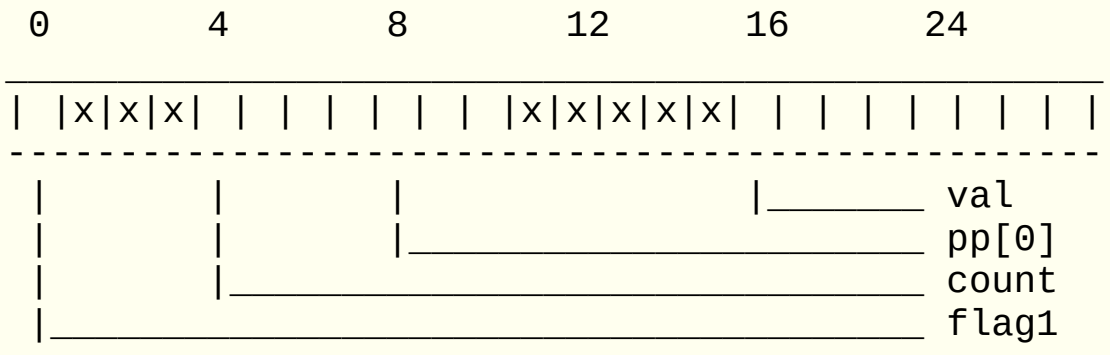
Chapter chap10 section 8

Almost all modern processors support byte addressing, i.e. an address is the address of a byte. However there is often a constraint that larger data items (integers and floating-point numbers) should start at locations whose address is a multiple of the size of the data item. This constraint called, an alignment constraint, much simplifies the handling of such data items.

C compilers can process struct and union declarations in such a way that such alignment constraints do not cause problems. However as a consequence of this there may be unassigned 'holes' left in structures. Similar problems arise in allocating memory when processing normal declarations. Consider for example.

```
char   flag1;  /* 1 byte */
int    count;  /* 4 bytes */
char   pp[3];  /* 3 bytes */
double  val;   /* 8 bytes */
```

This could well be laid out in memory as shown below.



The x's represent unassigned memory locations.

If memory were scarce it would make good sense to change the order of declarations. The problems become more tricky in a non-byte addressing environment (such as the PR1ME 50 series) which allows variables of type char to occupy the next byte in memory but insists that arrays of char be aligned on word boundaries.

Bit fields

Structures, Unions and Typedefs - Typedefs

Chapter chap10 section 7

The use of **typedef** is a simple macro-like facility for declaring new names for data types, including user-defined data types. Typical examples are shown below :-

```
typedef long    BIGINT;
typedef double  REAL;
typedef struct  point
        {
                double  x;
                double  y;
        }       POINT;
```

Given the above declarations, it would be possible to write the following declarations :-

```
POINT   a,b,c;
REAL    a1,a2;
```

It should be emphasised that this does not create new types, it just re-names existing ones. Unlike pre-processor constructs the interpretation of typedefs is performed by the compiler so that checks can be made that typedefs are used correctly.

Strangely typedef is regarded in the standard as a storage class.

Alignment Constraints

Structures, Unions and Typedefs - Bit Fields

Chapter chap10 section 9

The members of a struct may be bit-fields. Strictly these should be called fields rather than bit-fields but the usage is conventional. This means that the definition of the member includes a specification of how many bits the field is to occupy. This can save memory if it is suitably implemented by the compiler. The basic syntax of a bit-field declaration is

```
<type> <name> : <constant expression>
```

The type specification is almost invariably unsigned int, the constant expression is the number of bits that the bit-field is to occupy. The packing of bit-fields into actual words is, of course, implementation dependent. The compiler will generate the appropriate bit-twiddling instructions to allow normal operations to be performed on bit-fields. Most operations on bit-fields are implementation dependent and bit-fields are avoided by most programmers in the interests of portability. At best they allow a saving in memory usage at the price of more code (to extract and store the individual values).

The use of bit-fields is illustrated by the following declaration. Since this is a structure declaration all references to the flags have to be of the form.

```
flags.fracp = 1;
flags.epart = 0;
```

Bit-fields can be initialised in the same way as an ordinary structure. For arithmetic purposes bit-fields are regarded as integers, they may be seen as signed or unsigned depending on the implementation.

The use of bit-fields may save some memory compared with storing items whose values are only ever going to be 1 or 0 in characters, but it should be remembered that extra instructions will be required to perform the necessary packing and unpacking. Bit-fields are very rarely used in practice.

---

enum data types

Structures, Unions and Typedefs - enum data types

Chapter chap10 section 10

enum data types are data items whose values may be any member of a symbolically declared set of values. A typical declaration would be.

```
  enum days {Mon, Tues, Weds, Thurs, Fri, Sat, Sun};
```

This declaration means that the values Mon...Sun may be assigned to a variable of type enum days. The actual values are 0...6 in this example and it is these values that must be associated with any input or output operations. For example the following program

```
enum days  {Mon, Tues, Weds, Thurs, Fri, Sat, Sun };

main()
{
        enum days start, end;
        start = Weds;
        end = Sat;
        printf ("start = %d end = %d\n",start,end);
        start = 42;
        printf ("start now equals %d\n",start);
}
```

produced the following output

```
start = 2 end = 5
start now equals 42
```

It will be noticed that it is possible to assign a normal integer to an enum data type and there is no check made that an integer assigned to an enum data type is within range.

Few programmers use enum data types. The same effects can be achieved by use of #define's although the scoping rules are different.

It is possible to associate numbers other than the sequence starting at zero with the names in the enum data type by including a specific initialisation in the name list. This also effects all following names. For example

```
enum coins { p1=1, p2, p5=5, p10=10, p20=20, p50=50 };
```

All the names except p2 are initialised explicitly. p2 is initialised to value immediately after that used for p1.

Like bit-fields, enum data types are rarely used in practice.

The Pre-Processor and standard libraries - The #ifdef and #endif Directives

Chapter chap8 section 11

The next most useful preprocessor directives are **ifdef** and **endif .** These are always used in pairs with the basic syntax

```
#ifdef  identifier
.
.
.
#endif
```

All the code between the *#ifdef* and the #endif is skipped (i.e. completely ignored) unless the identifier has been the subject of a *#define* directive or has been defined in some other way.

#ifdef may be written as

```
#if defined ...
```

and *#ifndef* may be written as

```
#if !defined ...
```

defined is a special unary operator only understood by the pre-processor. An obvious and useful example is shown below. This program is called *debug* .

```
#include        <stdio.h>
#include        <ctype.h>
#define DEBUG   1
main()
{
        char    inbuf[BUFSIZ];
        int     i = 0;
        int     lcnt = 0;
        gets(inbuf);
        while(*(inbuf+i))
        {
#ifdef  DEBUG
                printf("Character %d Value %c(%o)\n",
```

```
                        i,*(inbuf+i),*(inbuf+i));
#endif
                if(isalpha(*(inbuf+i)))lcnt++;
#ifdef  DEBUG
                printf("Letter Count %d\n",lcnt);
#endif
                i++;
        }
        printf("Total letters %d\n",lcnt);
}
```

A typical dialogue is shown.

```
$ debug
the cat
Character 0 Value t(164)
Letter Count 1
Character 1 Value h(150)
Letter Count 2
Character 2 Value e(145)
Letter Count 3
Character 3 Value  (40)
Letter Count 3
Character 4 Value c(143)
Letter Count 4
Character 5 Value a(141)
Letter Count 5
Character 6 Value t(164)
Letter Count 6
Total letters 6
```

If the line defining *DEBUG* were removed or commented out then the diagnostic *printf()* function calls would not be seen by the compiler and the diagnostic printing would be omitted. This is a common practice.

There are two other ways in which an identifier associated with *#define* may become defined. The first method is from the compiler command line. A typical Unix system call would be

```
cc debug.c -DDEBUG
```

this forces the compiler to see *DEBUG* as defined even though no actual value is associated with *DEBUG* . The usage

```
cc debug.c -DDEBUG=1
```

may be used to associate a particular value with *DEBUG* . The compiler command line definition of an identifier is a particularly useful thing to do because it means that debugging code can be switched on and off without making any alterations whatsoever to the source code.

The second method applies only to certain special symbols. There are certain symbols that are defined internally by the preprocessor. The ANSI standard symbols predefined in this way are

symbol	meaning
__STDC__	defined if the compiler is ANSI
__LINE__	the current source line number
__FILE__	the source file name
__DATE__	the date when the program was compiled
__TIME__	the time when the program was compiled

In case your printer or display didn't make it clear these all consist of four letters preceded by and followed by a pair of underline characters. All except *__STDC__* and *__LINE__* are character strings complete with enclosing double quotes. The following might be found in a program

```
#ifdef  __STDC__
double  reciprocal(double x)
#else
double  reciprocal(x)
double  x;
#endif
{
        return 1.0/x;
}
```

providing a function definition acceptable to both ANSI and pre-ANSI compilers.

The logic associated with *#ifdef* may be extended using **#else** and **#elif** is a fairly obvious way. You may well encounter code such as

```
        #ifdef  UNIX
        #define INTBITS 32
        #else
        #define INTBITS 16
        #endif
```

This sort of thing is common in programs intended to be ported across a wide range of systems although some of the effects can be achieved more cleanly using *limits.h*

if ANSI compilers are available.

To provide greater flexibility any *#define* d identifier can be undefined by the directive **#undef** followed by the identifier. The directive **#line** forces the compiler to have a particular idea of the current line number in the source. This is only useful in conjunction with *__LINE__*.

The directive **#pragma** allows special compiler specific information to be included in the program. The meanings of pragmas are always compiler specific, they may, for example, be used to specify a particular way of storing pointers or generating code.

Exercises

The Pre-Processor and standard libraries - Exercises

Chapter chap8 section 12

1. Using the maths library functions *sin()* and *cos()* write a program that prints out lines of text whose full length represents the range of values -1 to +1 with a suitably positioned asterisk representing the value of *sin()* and a hash symbol representing the value of *cos()*. Produce output for values of input parameter representing angles in the range 0 degrees to 360 degrees in steps of 10 degrees.
2. Define a macro *max(a,b)* to determine the larger of two integers. Define a further macro *max3(a,b,c)* in terms of *max(a,b)* to determine the largest of three numbers. Incorporate your macros in a demonstration program that prompts the user for three numbers and reports the largest.
3. Write a program to read text from the standard input until an end-of-file condition is encountered. When all the text has been read in print out an analysis of the numbers of lines by length.
4. The function *rand()* generates random integers. Write a program to verify this by generating a fairly large number of random integers and analysing them into 20 frequency ranges. Display the number of random numbers generated in each range.
5. Write a program to randomly fill an array of 26 characters with the letters of the alphabet. Print out the randomised alphabet you have generated.
6. Extend the program of the previous exercise to read in text from the standard and replace all letters (preserving case) with the equivalent letter from the shuffled alphabet. Print out the modified text. This could be regarded as a very simple encryption system.

   To ensure that encryption is consistent you can use the library function *srand()* which uses a single integer parameter to initialise the random number generator. The random initialisation can be regarded as an encryption key. Devise a decryption program to recover the text that you encrypted earlier.

Separate Compilation of C Modules - Introduction

Chapter chap11 section 1

This chapter discusses the separate compilation and linkage of modules of C code. So that examples of the compilation and linkage process could be included in these notes it is necessary to focus on a particular host operating system. The choice adopted here is Unix for the very obvious reason that the commands and utilities are pretty much standard compared with the situation under MS-DOS where they are dependent on the particular C compiler in use. The facilities offered in non-Unix environments are often modelled on those available under Unix so it is well worth learning how to do it under Unix.

See also

- Compilation
- The extern keyword
- Simple separate compilation
- Separate compilation for program development
- Static Functions
- Scope rules
- Private Libraries
- The make utility

Separate Compilation of C Modules - Compilation

Chapter chap11 section 2

Under Unix the C compiler is invoked by the command "cc", C source files are expected to have names ending in ".c" and relocatable binary (linkable) modules are expected to have names ending in ".o". The command "cc" will also invoke the linker. Libraries can be maintained using the "ar" command. As an example the Unix command

```
cc -o sums sums.c -lm
```

will compile the program sums.c, link in the maths library and place the executable binary code in a file called sums. With an ANSI compiler there would be no need to include the flag "-lm".

The -o compiler flag followed by a file name specifies that the executable binary code be placed in the named file. If the -o flag is omitted the executable binary code is placed in a file called "a.out".

In the Unix environment C compilation is a multi-stage process. The first stage runs the pre-processor, the second stage generates assembly language code, the third stage converts the assembly language code to relocatable binary code and the final stage links the relocatable binary code modules to generate an executable binary code module.

The compiler -P flag stops the compilation after the pre-processor stage leaving the output in a ".i" file. The compiler -S flag stores the assembly language code in a ".s" file. The compiler -O flag invokes one (or more) optimisation phases. There may well be many other flags and options depending on the particular environment.

In the Unix environment it is sometimes possible to use the various compilation steps as separate free-standing programs. The C pre-processor is called "cpp", the assembler is called "as" and the linker is called "ld". It is very rare to do so.

Modules

In these notes a module is any chunk of code that can be processed by the compiler. Modules should consist of function declarations and declarations of variables of storage class extern. A module may simply contain a single function declaration or may consist entirely of declarations of extern variables. A module is, invariably, a separate source file. Some authors refer to these modules as **compilation units.**

The extern keyword

Separate Compilation of C Modules - The extern keyword

Chapter chap11 section 3

When compiling modules separately it is often the case that the definition of a variable is in one module and references to the variable occur in another module. In this case the variable definition is said to be external to the function referencing the variable. This situation poses a problem for the compiler which needs information about the type of the variable for correct operation.

Declarations which start with the keyword **extern** and do not include any aggregate sizes or initialisation provide sufficient information for the correct operation of the compiler. Actual addresses are filled in when the program is linked.

Simple use of Separate Compilation

Separate Compilation of C Modules - Simple use of Separate Compilation

Chapter chap11 section 4

This example is a program that reads in an integer and prints the name of the associated month. Since an aggregate holding month names is potentially useful in a variety of situations it is compiled separately. Ultimately the compiled month name module could be incorporated into a library.

The month name definition and initialisation is in the file *mnames.c* which is listed below.

```
char    *mn[] =
        {       "january","february","march",
                "april","may","june",
                "july","august","september",
                "october","november","december"
        };
```

This could be compiled with the following command

```
cc -c mnames.c
```

This creates a file called mnames.o. A simple program that uses the information in the aggregate is shown below. It is in the file *mnlp.c*

```
extern  char    *mn[];
main()
{
        int     i;
        while(1)
        {
                printf("Enter a number ");
                scanf("%d",&i);
                if(i<1 || i>12) exit(0);
                printf("month %d is %s\n",i,mn[i-1]);
        }
}
```

This could be compiled using the following command

```
cc -c mnlp.c
```

This creates a file called *mnlp.o* The -c compiler flag specifies that the compiler will compile but will not invoke the linker. The two files could be linked together, along with the relevant library routines, using the following command

```
cc -o mnlp mnlp.o mnames.o
```

This creates an executable file called *mnlp* which prompts for an integer and responds with the month name. An alternative approach would be to use the following command to compile *mnlp.c* and link *mnames.o* in one operation

```
cc -o mnlp mnlp.c mnames.o
```

It should be noted that the declaration of the aggregate *mn[]* in the file *mnames.c* is outside any function declaration and is, thus, of storage class extern. This means that the name of the aggregate (*mn*) is known to the linker. In this example it is interesting to note that the module *mnames.c* does not contain any executable statements.

In line 1 of *mnlp.c* the declaration tells the compiler that the definition of the aggregate *mn[]* is elsewhere so actual references (on line 10) will have to be filled in by the linker.

If the name of the aggregate in *mnames.c* were not *mn* then a linkage error would result when trying to put the program *mnlp* together.

Using Separate Compilation for Program Development

Separate Compilation of C Modules - Using Separate Compilation for Program Development

Chapter chap11 section 5

This section describes a larger scale use of separate compilation facilities. The word counting program described earlier in these notes is subdivided into separate modules. The first module is not really a C module at all but a set of pre-processor directives and function prototypes for use by the various code modules. The file is *sstr.h* and it is listed below.

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

struct word
   {
         int     count;          /* occurences */
         char    *body;          /* actual text */
         struct  word    *lptr;  /* left pointer */
         struct  word    *rptr;  /* right pointer */
   };

void    putword(char *,struct word *);
void    listwords(struct word *,int);
void    error(int);
struct  word *new(char *);

extern  int     wdct;
extern  int     maxocc;
```

This also contains declarations that tell the compiler that the two "global" variables *wdct* and *maxocc* are defined elsewhere and will be fixed up at linkage time.

The second module contains the "global" declarations from the original program. All these declarations are, of course, of storage class extern and appropriate extern declarations need to be put in the other modules by including the header file " *sstr.h* The file is *sstrex.c* and it is listed below.

```
int     wdct;           /* number of different words */
int     maxocc;         /* maximum count */
```

The file can be compiled creating the module sstrex.o using the command

```
cc -c sstrex.c
```

The remaining modules contain one function each. It is not essential to adopt this style. Many functions could be declared in a single file, as is the case with the more conventional style of construction of C source files shown in most of the examples in these notes.

The next module contains the function main and is in the file *sstrmain.c* It is listed below.

```
#include        "sstr.h"
main()
{
        char    inbuf[BUFSIZ];
        int     i;
        struct word *root;
        root=new("ZZZZ");  /* establish root of tree */
        root->count=0;     /* it's only a dummy entry !! */
        while(gets(inbuf))      putword(inbuf,root);
        printf("%d Different Words\n",wdct-1);
        for(i=maxocc;i;i--)listwords(root,i);
}
```

The inclusion of *sstr.h* has achieved four objectives

1. The *#include* directives in *sstr.h* have included all the relevant standard header files. For a large application using many header files this is useful, it avoids tedious error prone repetition.
2. It has provided the declaration of the structure *word* Again the advantages of only actually writing the declaration once are obvious.
3. It has provided the prototypes for *listwords()* and *new()*
4. It has provided declarations for the global variables *wdct* and *maxocc*

The next module contains the function *putword()* and is in the file *sstrputword.c* It is listed below.

```
#include        "sstr.h"
void    putword(char *s,struct  word *w)
{
        int     flag;
        flag = strcmp(s,w->body);
        if(flag==0)
        {
                if(++w->count > maxocc) maxocc = w->count;
                return;
        }
```

```
        if(flag<0)
        {
                if(w->lptr==NULL) w->lptr = new(s);
                else putword(s,w->lptr);
        }
        else
        {
                if(w->rptr==NULL) w->rptr = new(s);
                else putword(s,w->rptr);
        }
}
```

The next module contains the function *new()* and is in the file *sstrnew.c* It is listed below.

```
#include        "sstr.h"
struct word *new(char *s)
{
        int     slen;
        struct  word    *w;
        if((w = (struct word *)malloc(sizeof (struct word)))
                                ==NULL) error(2);
        slen = strlen(s) + 1;
        if((w->body = (char *)malloc(slen))==NULL) error(3);
        strcpy(w->body,s);
        w->count = 1;
        w->lptr = NULL;
        w->rptr = NULL;
        wdct++;
        return w;
}
```

The next module contains the function *listwords()* and is in the file *sstrlistwords.c* It is listed below.

```
#include        "sstr.h"
void    listwords(struct        word *w,int count)
{
        if(w->lptr!=NULL)listwords(w->lptr,count);
        if(w->count == count)
                printf("%3d %s \n",w->count,w->body);
        if(w->rptr!=NULL)listwords(w->rptr,count);
}
```

The final module contains the function *error()* and is in the file *sstrerr.c* It is listed below.

```
#include        "sstr.h"
```

```
void    error(int n)
{
        printf("Error %d : ",n);
        exit(0);
}
```

In this case it was not strictly necessary to include the file *sstr.h* but it was done for consistency.

Once all the module files described above have been created they may be compiled using a series of commands such as

```
cc -c sstrex.c
cc -c sstrmain.c
cc -c sstrputword.c
cc -c sstrnew.c
cc -c sstrerr.c
```

A Unix user would probably use a form of the command such as

```
cc -c sstr*.c
```

this assumes that there are no other *sstr....c* files in the current directory. Whichever form of the cc command is used the result is the same, the creation of a set of .o files in the current directory, assuming, of course, there were no compilation errors.

The compiled modules may now be linked along with the standard libraries using the following command

```
cc -o sstr sstr*.o
```

Alternatively the names of each of the compiled modules could be given separately

```
cc -o sstr sstrex.o sstrmain.o sstrnew.o .....
```

The advantage of this arrangement is that if it is necessary to modify the source code of any of the modules it is only necessary to recompile that particular module (using a "cc -c" command) and then relink the whole program (using a "cc -o" command). This is much faster than recompiling the whole program although it does consume more disc space.

Static Functions

Separate Compilation of C Modules - Static (Private) Functions

Chapter chap11 section 6

It is possible to declare functions of storage class *static* The implication of this is that such functions can only be referenced in the file (or compilation unit) which contains the function definition. This is illustrated by the following example.

This program reads in lines of text and displays them 20 characters to a line on the standard output. This program provides for proper TAB handling (assuming TAB's every 8 character positions). Note that this is not necessarily a good way of de-TABbing input lines. The program is split into 2 modules. The first contains the function *main()* and is in the file *ibad.c* It is listed below.

```
int     getnext(void);
#include        <stdio.h>
main()
{
        int     c;
        int     i=0;
        while( (c=getnext()) != EOF)
        {
                putchar(c);
                if(++i == 20 || c == '\n')
                {
                        i = 0;
                        if(c != '\n') putchar('\n');
                }
        }
}
```

The second module contains the functions *getnext()* and *detab()* and is in the file *iblb.c* It is listed below.

```
#include        <stdio.h>

int     getnext(void);
void    detab(void);
```

```
static  char    ibuf[256];
static  int     iptr;
int     getnext(void)
{
        int     c;
        if(iptr == 0)
        {
                while((c=getchar()) != EOF)
                {
                        ibuf[iptr++] = c;
                        if (c == '\n') break;
                }
                if(c == EOF) return(EOF);
                detab();
                iptr = 0;
        }
        c = ibuf[iptr++];
        if(c == '\n') iptr=0;
        return c;
}
static  void    detab(void)
{
        char    obf[256];
        int     i;
        int     j=0;
        int     imx = 0;
        while(ibuf[imx++] != '\n');
        for(i=0; i<=imx; i++)
        {
                if(ibuf[i]=='\t')
                {
                        if((j%8)==0) obf[j++] = ' ';
                        while(j%8) obf[j++] = ' ';
                }
                else obf[j++] = ibuf[i];
        }
        for(i=0; i<=j; i++) ibuf[i] = obf[i];
}
```

The only external symbol in *iblb.c* is *getnext* The definition of *getenext* looks like a perfectly ordinary function definition. Function names are visible outside files unless they are preceded by the keyword *static* as is the case with the declaration of the function *detab*

The *static* keyword associated with the definitions of *ibuf[]* and *iptr* ensure that they are accessible to all the functions declared in the file *iblb.c* but not by any other functions. This is sometimes called **data** and **function hiding.** If any structures had been defined in *iblb.c* then the declarations would, of course, be private, to *iblb.c* , so providing **structure hiding** It would probably have been better to put the prototype of *getnext* in a header file included in both *ibad.c* and *iblb.c* thus avoiding unreliable duplication.

The names of non-static functions and variables of storage class extern are visible outside the file in which they are defined. In particular they are visible to the linker, this may be a host system utility designed to link archaic languages such as Cobol and Fortran. A consequence of this fact is that the rules for distinguishable external symbols and the ANSI standard says that such symbols may be restricted to as few as 6 characters and that character case may not be significant. In practice this means that any external identifier ought to be unique in its first 6 characters. Most modern linkers are much more accommodating.

Scope Rules

Separate Compilation of C Modules - Scope Rules

Chapter chap11 section 7

Scope rules define where within a program the definition of a function or variable is accessible. I.e. the positions where the defined variable or function can be referenced or used. The scope rules of C are complicated by the use of separate source files. Scopes can be split into 4 categories.

```
function
file
block
function prototype
```

1. Function Scope

   This applies only to labels. It simply means that labels can be used (as "goto" targets) anywhere within the function in which they are defined. This includes use before definition.

2. File Scope

   This means that the identifier can be used anywhere in the current file after the declaration of the identifier. This applies to functions and all variables declared outside functions. File scope variable declarations may be either variable definitions or be preceded by the keyword extern indicating that they are to be linked in from another file. File scope identifiers may be hidden by clashing block scope declarations.

3. Block Scope

   This means that the identifier can only be used in the block in which it is declared. This will apply to auto and register storage class variables declared within a function.

4. Function Prototype Scope

   In order to improve readability function prototypes are usually written with "dummy" variable names. For example

   ```
   double sum(double x, double y);
   ```

   The scope of the identifiers "x" and "y" is the duration of this particular function prototype.

Private Libraries

Separate Compilation of C Modules - Private Libraries

Chapter chap11 section 8

It is possible, with most C compilation systems, to build and maintain private libraries of useful functions to support particular projects. Do not confuse C libraries with the notion of putting the routine source code in a file and #including it into programs. If you create and maintain a library you will also almost certainly wish to create and maintain an associated header file containing function prototypes.

Unix C compilers normally support the command line arguments

```
-L<dir-name> and -I<dir-name>
```

to specify extra directories to be searched for header files and libraries. If a Unix library has a name such as

```
libmylib.a
```

then the linker can be told to scan this library for functions by including the complier command line argument

```
-lmylib
```

Unix library files should always have names beginning with "lib" and with the suffix ".a"

Unix libraries are created and maintained using the utility ar archives. Libraries are built from ".o" or binary files. The operation of the "ar" command is controlled by command line flags that unusually do not require initial dashes. To install binary code units "myf1.o" and "myf2.o" in the library "libmine.a" use the command

```
ar q libmine.a myf1.o myf2.o
```

This will create the library "libmine.a" if it doesn't already exist.

There are various flags associated with "ar". The following are useful.

```
t       List the contents table of the library

r       Replace a named file if it already
        appears in the library

d       Delete the named file from the library
```

```
w       List all the symbols defined in the
        library
```

It is important that the binary code units are placed in the library in the correct order. If one code unit calls a function defined in another then the one making the call should appear first in the library. The a and b flags can be used to put code units in the library after or before another component of the library.

The Unix utilities lorder and tsort can be used in the following fashion to put a set of binary code units into a library in the correct order

```
lorder *.o | tsort | xargs ar q libsubs.a
```

On some systems the linker "ld" can process a library in random order rather than serially processing it from beginning to end. If this is the case it does not matter what order the code units are placed in in the library.

The make utility

Separate Compilation of C Modules - The make utility

Chapter chap11 section 9

When developing and maintaining a large piece of software built out of numerous source files it is important to ensure that if any one file is modified, all files dependent on it are also modified.

The commonest causes of difficulties are when a header file has been modified failing to re-compile all the source files that reference the header file and failing to recompile all altered source files before re-linking. The maintenance of such large pieces of software can be automated using the make utility and an associated makefile. The makefile will consist of a series of entries each consisting a dependency list and a set of commands.

A dependency list consists of the name of a file called the target and after a colon a list of all the other files it depends upon. The following command lines consist of a series of commands that will bring the target up to date. A command line MUST start with a TAB character, a dependency line MUST NOT start with a TAB character.

A typical make file is shown below

```
incdate.o       : incdate.c     date.h
                cc -c incdate.c
vetdate.o       : vetdate.c     date.h
                cc -c incdate.c
diffdate.o      : diffdate.c    date.h
                cc -c diffdate.c
libdates.a      : incdate.o vetdate.o diffdate.o
                ar qr libdates.a incdate.o vetdate.o diffdate.o
mytest.o        : mytest.c      date.h
                cc -c mytest.c
mytest          : mytest.o      libdates.a
                cc -o mytest mytest.o -L. -ldates -lm
```

This should be called 'makefile' in the current directory To bring any of the executables up-to-date after modifying source files it is only necessary to type

```
make <target>
```

If any of the files required to create the target are themselves out of date as determined from their entries in the makefile they are also made.

Efficiency, Economy and Portability - Introduction

Chapter chap12 section 1

Most C programmers are quite happy not to worry about the efficiency, economy and portability of their programs. However there will inevitably be occasions where such things do matter and it is the purpose of this chapter to highlight programming practices that should be avoided or followed in such circumstances.

Efficiency, Economy and Portability - Efficiency, Economy and Portability defined

Chapter chap12 section 2

Efficiency is here taken to refer to the execution speed of a program. In some circumstances, such as an operating system kernel or a real-time program speed of execution may be unusually important. **Economy** is here taken to refer to the amount of memory used by the program. Again there may be special circumstances in which it is particularly important to use as little memory as possible. **Portability** is fairly self-evident, it simply refers to the ease, or otherwise, with which a program may be transferred from one computer to another.

C programs often score highly compared with those written in other languages for economy, efficiency and portability. The rest of this chapter will focus on techniques that enhance economy, efficiency or portability. It should be noted that enhancement of one aspect of programming may have a bad effect on other aspects.

It is not possible to give any totally general advice about efficiency and economy. If efficiency and economy are particularly important you should conduct experiments and tests to identify the strengths and weaknesses of your particular compiler. The material in this chapter will suggest which areas to investigate.

Of course, it goes without saying (almost) that a program should be got working correctly before any attempt is made to enhance its efficiency or economy.

The Hello World program analysed

Efficiency, Economy and Portability - The "Hello World" Program

Chapter chap12 section 3

Examination of the performance of different versions of this program provide some useful insights into factors affecting efficiency and economy. Five different versions of the "Hello World" program were compiled and run using Microsoft C (version 5.1). Extra code was incorporated to display "hello world" 10000 times and report the time elapsed by interrogating the clock. They were actually run on an 80286 based processor (a few years ago!).

The sizes and relative execution times are given below.

	size (bytes)	**execution time**
Program 1	7233	101
Program 2	4525	140
Program 3	2377	112
Program 4	5245	103
Program 5	2361	100

Program 1 was the classic version using *printf()* The efficiency is remarkably good but the *printf()* library function is clearly memory hungry. This is hardly surprising since it includes code for all forms of floating point and integer conversion as well as basic string output.

Program 2. *hw2.c* , listed below used the *stdio.h* macro *putchar()* . This is the least efficient program but it uses the least amount of memory of the portable versions.

```
#include        <stdio.h>
char    *msg="hello, world\n";
main()
{
        while(*msg)putchar(*msg++);
}
```

Program 3, *hw3.c* , listed below used the MS-DOS INT 21 (function 2) system call to write characters in preference to *putchar()* This represents a significant improvement in both speed and more noticeably memory utilisation compared with

program 2. Note that the string needed to include \r for proper operation.

```
#include        <stdio.h>
#include        <dos.h>

char    *msg="hello, world\r\n";
main()
{
        while(*msg)
        {
                bdos(0x02,*msg++,0);
        }
}
```

Program 4, *hw4.c* , listed below, uses the library function *puts()* , efficiency is good but the program is memory hungry.

```
#include        <stdio.h>
main()
{
        puts("hello, world\n");
}
```

Program 5, *hw5.c* , listed below was expected to be the fastest and use the least memory. It met both expectations (just). It is also, arguably, the most obscure. It used the MS-DOS INT 21 (function 9) system call to display a string (quaintly terminated with a $ symbol).

```
#include        <stdio.h>
#include        <dos.h>

char    *msg="hello, world\r\n$";
main()
{
                bdos(0x09,FP_OFF(msg),0);
}
```

The execution times in these examples were probably dominated by screen handling overheads but suggest there is little to be gained (in terms of efficiency) by not using standard library routines but that standard library routines are memory hungry.

Efficient Coding

Efficiency, Economy and Portability - Efficient Coding

Chapter chap12 section 4

Many texts on C state that writing i++ rather than i=i+1 can give more efficient code. The effect is very small and a good compiler may well generate the same code in each case. Similar comments apply to the use of the register storage class for items such as loop control variables.

However there are some practices that can give rise to serious inefficiencies. Things to watch out for are.

1. Structure assignment and the use of structures as functional parameters and values force the generation of code to copy structures byte by byte. It is much better to use structure addresses.
2. Function calls may well involve significant overheads for environment saving, stack frame formation and auto storage class variable initialisation. All these aspects of using functions should be looked at carefully.
3. Think about the data types you use. Long integer arithmetic is usually slower than short integer arithmetic. Floating point arithmetic is much slower than integer arithmetic.
4. Little used aspects of C may well not be well handled by compilers. Particular things to avoid are fields and enum data types.

   If the compiler has an optimisation flag this should be used. The Unix compiler flag is -O. Compiler optimisation should be viewed with caution, it has been known for a compiler optimised version of a program to contain bugs that were absent from the non-optimised version. Compiler optimisation lengthens compilation time so don't use for program development.

   Profiling is a technique supported in some environments that allows the programmer to determine the number of times each function has been called and how much time has been spent in each function. It is worth using if it is available. Profiling is described in more detail later.

   A final, general, point is, of course, to use the most efficient algorithm available consistent with development time (highly efficient algorithms are often rather hard to understand and code).

Economic Coding

Efficiency, Economy and Portability - Economic Coding

Chapter chap12 section 5

If memory is tight there are various things that should be noted.

1. Exercise discretion with the use of library functions. Some are very memory hungry.
2. Rather than declare sufficiently large arrays use *malloc()* to obtain new memory as needed. Be aware that this is inefficient and not as economic of memory as might be hoped due to administrative overheads. Release blocks acquired via *malloc()* (or *calloc()*) using *free()* as soon as they are no longer required.
3. Look out for alignment constraints, particularly amongst elements of structs and unions.
4. Avoid unnecessary use of extern storage (global variables).
5. Try and re-use code as much as possible by careful selection and design of functions.
6. Use pointers to variables as function parameters. Pack up as many variables as possible into a single structure and pass a pointer to that structure rather than use memory variables.
7. Think about the data types of large aggregates. If your program needs an array of 5000 integers consider whether the range of expected values would allow the array to be of type *char* or type *short int* rather than *long int* .

Portability

Efficiency, Economy and Portability - Portability

Chapter chap12 section 6

A portable program is one that can be transferred from one computer system to another. The degree of portability is a measure of the ease with which such a transfer can be made. For C programs there are several dimensions to portablility. The major portability barriers you are likely to encounter are between ANSI-C and "old" C and between the MS-DOS and Unix environments.

Pre-ANSI C compiler writers were free to do whatever they liked, however the language implemented by pre-ANSI compilers is usually a strict subset of the ANSI specification. The biggest problem area is likely to be the pre-processor, many pre-ANSI pre-processors behaved rather differently from the ANSI specification, this is particularly true with Unix based systems. Of course, portability in the other direction, i.e. from ANSI to pre-ANSI is much more difficult, but you are unlikely to meet this as a practical requirement.

Much of the strength of C as a language stems from the careful use of standard libraries. In pre-ANSI environments these were not totally standard, differences may be encountered in both the functionality provided and the particular header files. Some early versions of *printf()* and *scanf()* were rather different from current practice.

Of course, for full portability, you musn't use any library functions that are not part of the ANSI standard. The practice of some manual writers of indicating whether functions are ANSI or not is, unfortunately, far from universal, and you should equip yourself with a list of ANSI functions.

Many compilers and host operating systems provide a lot of useful extra functions that are not part of the ANSI standard. These may well be standardised by different bodies, for example the system calls available Unix systems (and some others) but not to PCs. There are also well known standards for graphics.

The ANSI standard does not totally specify the language. In several areas the compiler writer is left to do whatever seems most appropriate. This is described as implementation-defined behaviour.

The most significant area of implementation-defined behaviour concerns the sizes and natures of the basic data types. The ANSI standard includes a 3 page list of all instances of implementation defined behaviour.

The following points may be worth noting.

1. As far as possible declare all integer variables as either *short int* or *long int* Don't rely on the default.
2. Be aware of constraints on identifiers. Don't use identifiers more than 31 characters long. Be aware of the special constraints that apply to externally visible identifiers, at their most restrictive these state that only the first 6 characters are significant and that alphabetic case may not be significant.
3. Be wary of little used features of the language. enums and (bit) fields are the features most likely to cause problems.
4. Be wary of union data types and the side effects of alignment constraints.
5. Watch out for possible side-effects associated with the order of evaluation of function arguments.
6. Watch out for the effects of promoting non-ASCII valued char variables. Always declare variables as *signed char* or *unsigned char* , never plain *char* .

You should also be aware of the following compiler limits. Many compilers will exceed these limits but no ANSI compiler should enforce tighter limits.

```
   31 Arguments per function call
32767 Bytes in an object
  257 Case labels in a switch
  509 Characters in a source line
  509 Characters in a string constant
   15 Levels of compound statement nesting
  127 Constants in an enum
   15 Levels of do, while and for nesting
   12 Levels of referencing on a declaration
   32 Levels of expressions within expressions
  511 External identifiers in a translation unit
    8 Levels of header file nesting
  127 Local identifiers in a block
   15 Levels of if and switch nesting
    8 Levels of #if, #ifdef and #ifndef nesting
 1024 Macros in a translation unit
  127 Members of a structure or union
   31 Parameters in a function definition
   31 Parameters in a macro
    6 Significant characters in an external identifier
   31 Significant characters in an internal identifier
   15 Levels of struct and union definition nesting
```

Profiling

Efficiency, Economy and Portability - Profiling

Chapter chap12 section 7

This section briefly describes the facilities available to the Unix user to study the performance of programs.

Simply measuring the execution times of a program is straightforward. The Unix command *time* is used in the following manner.

```
time <commands to run program>
```

The following examples show how *time* may be used. It will be noticed that there are two slightly different versions of the *time* command. Normally users will just type *time* rather than the full path names.

```
$ /usr/bin/time isp -4 -5 -6 main > x
239 erroneous records
        2.7 real         1.5 user         0.3 sys
$ /usr/5bin/time isp -4 -5 -6 main > x
239 erroneous records

real        2.7
user        1.5
sys         0.3

bash$ csh
scitsc% time isp -4 -5 -6 main > x
239 erroneous records
1.5u 0.3s 0:02 84% 0+224k 2+3io 0pf+0w
```

The three numbers are elapsed time, CPU time in user mode and CPU time in system mode. They are all expressed in seconds.

If you are using the C shell rather than the Bourne or Bash shells then you can obtain rather more information as shown below.

```
scitsc% time isp -4 -5 -6 main > x
239 erroneous records
1.5u 0.2s 0:02 88% 0+224k 28+2io 25pf+0w
```

The first three times are the user time, system time and elapsed time. The elapsed time is expressed in minutes. The remaining figures are the total CPU time as a

percentage of the elapsed time, the average amount of shared and non-shared memory in Kbytes, the number of block input and output operations and finally the number of page faults and swaps.

To obtain more information about the time a program spends executing particular routines it is necessary to profile the program. Under the Unix system this is done by compiling the program with the "-p" flag on the compiler command line, running it normally and using the utility "prof" to examine the "mon.out" file that the profiled program generates.

Suppose the program "isp.c" were compiled using the command

```
cc -o isp -p isp.c
```

and then run using the command

```
isp -4 -5 -6 main > x
```

This would generate a *mon.out* file in the current directory. The results of the profiling can be displayed using the command

```
prof isp
```

This would, typically, produce the following output.

```
 %time  cumsecs  #call  ms/call  name
  36.2     0.68   8933     0.08  _getfield
  15.4     0.97   8933     0.03  _isbner
  10.1     1.16   9039     0.02  _memccpy
   7.4     1.30    110     1.27  _read
   7.4     1.44                  mcount
   3.7     1.51  73494     0.00  .mul
   3.2     1.57   2630     0.02  __doprnt
   3.2     1.63  65068     0.00  _strlen
   3.2     1.69   8933     0.01  _yearer
   2.7     1.74   8933     0.01  _priceer
   2.1     1.78  26799     0.00  _strcat
   1.6     1.81   7424     0.00  _atoi
   1.6     1.84  22663     0.00  _isblank
   0.5     1.85   8166     0.00  .rem
   0.5     1.86   8934     0.00  _fgets
   0.5     1.87      2     5.00  _fstat
   0.5     1.88      1    10.00  _main
   0.0     1.88      6     0.00  .udiv
   0.0     1.88      3     0.00  .umul
   0.0     1.88      3     0.00  .urem
```

```
  0.0     1.88      1     0.00  ___start_libm
  0.0     1.88    110     0.00  __filbuf
  0.0     1.88      2     0.00  __findbuf
  0.0     1.88      1     0.00  __findiop
  0.0     1.88      2     0.00  __wrtchk
  0.0     1.88      4     0.00  __xflsbuf
  0.0     1.88      1     0.00  _exit
  0.0     1.88      1     0.00  _fflush
  0.0     1.88      1     0.00  _fopen
  0.0     1.88      1     0.00  _fprintf
  0.0     1.88      3     0.00  _free
  0.0     1.88      1     0.00  _freopen
  0.0     1.88      1     0.00  _getpagesize
  0.0     1.88      2     0.00  _ioctl
  0.0     1.88      2     0.00  _isatty
  0.0     1.88   2630     0.00  _localeconv
  0.0     1.88      3     0.00  _malloc
  0.0     1.88      1     0.00  _on_exit
  0.0     1.88      1     0.00  _open
  0.0     1.88   2629     0.00  _printf
  0.0     1.88      1     0.00  _profil
  0.0     1.88      4     0.00  _sbrk
  0.0     1.88      4     0.00  _write
```

Many of the routines mentioned in the above list are library routines and routines called by library routines, however it is clear that the program is spending a lot of time in user routines called *getfields()* and *isbner()* . A quick examination of the source code for the routine *isbner* revealed the following code fragment

```
int     j;
int     cksum = 0;
if(strlen(fld)==0 || isblank(fld))return "";
if(strlen(fld)!=13) return "K";
```

The repeated calls to *strlen()* represent a possible inefficiency. After recoding the program as shown below

```
int     cksum = 0;
int     l;
if((l=strlen(fld))==0 || isblank(fld))return "";
if(l!=13) return "K";
```

profiling gave the following results

```
 %time  cumsecs  #call  ms/call  name
```

28.0	0.53	8933	0.06	_getfield
14.8	0.81			mcount
12.7	1.05	8933	0.03	_isbner
11.6	1.27	9039	0.02	_memccpy
10.6	1.47	110	1.82	_read
5.8	1.58	56889	0.00	_strlen
2.6	1.63	73494	0.00	.mul
2.6	1.68	2630	0.02	__doprnt
2.1	1.72	1	40.00	_main
1.6	1.75	8934	0.00	_fgets
1.6	1.78	22663	0.00	_isblank
1.6	1.81	8933	0.00	_priceer
1.1	1.83	8166	0.00	.rem
1.1	1.85	2629	0.01	_printf
1.1	1.87	26799	0.00	_strcat
1.1	1.89	8933	0.00	_yearer
0.0	1.89	6	0.00	.udiv
0.0	1.89	3	0.00	.umul
0.0	1.89	3	0.00	.urem
0.0	1.89	1	0.00	___start_libm
0.0	1.89	110	0.00	__filbuf
0.0	1.89	2	0.00	__findbuf
0.0	1.89	1	0.00	__findiop
0.0	1.89	2	0.00	__wrtchk
0.0	1.89	4	0.00	__xflsbuf
0.0	1.89	7424	0.00	_atoi
0.0	1.89	1	0.00	_exit
0.0	1.89	1	0.00	_fflush
0.0	1.89	1	0.00	_fopen
0.0	1.89	1	0.00	_fprintf
0.0	1.89	3	0.00	_free
0.0	1.89	1	0.00	_freopen
0.0	1.89	2	0.00	_fstat
0.0	1.89	1	0.00	_getpagesize
0.0	1.89	2	0.00	_ioctl
0.0	1.89	2	0.00	_isatty
0.0	1.89	2630	0.00	_localeconv
0.0	1.89	3	0.00	_malloc
0.0	1.89	1	0.00	_on_exit
0.0	1.89	1	0.00	_open
0.0	1.89	1	0.00	_profil
0.0	1.89	4	0.00	_sbrk
0.0	1.89	4	0.00	_write

Timing

Efficiency, Economy and Portability - Timing and Clock functions

Chapter chap12 section 8

There are ANSI functions that allow the programmer to determine elapsed time and CPU time from within the program. The two library functions *clock()* and *time()* are used. The first of these, *clock()* , returns the amount of CPU time the program has used measured in some arbitrary units. The returned value should be stored in a variable of type *clock_t* The second of these, *time()* , returns the real time, sometimes called the time-of-day or wall-clock-time usually mesured in seconds from some arbitrary origin. It returns -1 if this information is not available. To convert CPU times to seconds they should be divided by the constant CLOCKS_PER_SEC which may conveniently be cast to double. The use of both functions is illustrated by the following program.

```
#include <time.h>
main()
{
        int     i;
        time_t  start,now;
        clock_t start_cpu,now_cpu;
        start=time(NULL);
        start_cpu = clock();
        for(i=0;i<10000000;i++);        /* timing this loop */
        now=time(NULL);
        now_cpu = clock();
        printf("Elapsed time = %d seconds\n",now-start);
        printf("CPU Time = %7.3lf seconds\n",
                (now_cpu-start_cpu)/(double)CLOCKS_PER_SEC);
}
```

which produced the following output

```
Elapsed time = 10 seconds
CPU Time =   4.200 seconds
```

There are some further functions that may be used to convert real times to more useful forms. These are *localtime()* and *gmtime()* that convert a time in seconds from an aribtrary origin to a date/time structure whose elements give the day of the week, the month etc., etc. The functions *asctime()* and *strftime()* convert a date/time structure to a character string. The function *ctime()* converts from time in seconds to a character string.

localtime() gives a date/time structure containing the local time and *gmtime()* gives it in GMT. The format of the strings produced by *asctime()* and *ctime()* are standard, *strftime()* has an elaborate formatting capability and its output is sensitive to the locale. The non-ANSI

function *strptime()* which does the opposite of *strftime()* may sometimes be encountered and is useful in processing logs and such like data files.

Note . The arbitrary origin for *time()* is often Jan 1st, 1970, especially on Unix-based systems. Since the time is held in seconds in a 32-bit number this means that this scheme will fail sometime around Jan 19th, 2038. Are you worried ?

The format of the date/structure known as a *struct tm* defined in the header *time.h* is

```
struct tm
{
        int     tm_sec;         /* seconds after minute */
        int     tm_min;         /* minutes after hour */
        int     tm_hour;        /* hours since midnight */
        int     tm_mday;        /* day of the month */
        int     tm_mon;         /* month of year [0-11] */
        int     tm_year;        /* year since 1900 */
        int     tm_wday;        /* days since Sunday */
        int     tm_yday;        /* days since Jan 1 */
        int     tm_isdet;       /* daylight saving time */
};
```

The following program demonstrates the simple use of *time()* , *localtime()* and *asctime()*

```
#include        <stdio.h>
#include        <time.h>
main()
{
        time_t  now;
        struct  tm      date_time;
        now = time(NULL);
        date_time = *localtime(&now);
        printf("Date is %s",asctime(&date_time));
}
```

producing the output

```
Date is Thu Jan  7 14:46:00 1993
```

There are several rather quirky aspects of these functions. The *time()* function takes a single parameter which may be a pointer to a variable of type *time_t* The value returned by *time()* is also placed in the location addressed by the parameter if the parameter is not NULL. The function *localtime()* takes a pointer to a variable of type *time_t* as parameter and returns a pointer to an object of type *struct tm* The function *asctime()* takes a pointer to an object of type *struct tm* as parameter. The formatted string generated by *asctime()* (and *ctime()*) includes a new-line character. Most of these quirks are for backwards compatability with very early versions of C and Unix.

The next example reads in an integer and displays the date the relevant number of days into the future or past.

```
#include        <stdio.h>
#include        <time.h>
main()
{
        int     diff;
        time_t  now;
        time_t  new;
        time(&now);
        now = (now/86400)*86400;
        do
        {
                printf("Enter Difference ");
                scanf("%ld",&diff);
                if(!diff)break;
                new = now+diff*86400;
                if(new<0)
                        printf("Out of range\n");
                else
                        printf("Date is %s",ctime(&new));
        } while(1);
}
```

A typical log is listed below

```
Enter Difference 100
Date is Sat Apr 17 01:00:00 1993
Enter Difference 200
Date is Mon Jul 26 01:00:00 1993
Enter Difference 2609
Date is Tue Feb 29 00:00:00 2000
Enter Difference 1148
Date is Thu Feb 29 00:00:00 1996
Enter Difference 5000
Date is Sat Sep 16 01:00:00 2006
Enter Difference -200
Date is Sun Jun 21 01:00:00 1992
Enter Difference -1000
Date is Fri Apr 13 01:00:00 1990
Enter Difference 0
```

The variation in the displayed times has arisen because the functions are making some attempt to allow for daylight saving time.