Next Up Previous Contents
**Next:** Contents

# C Programming

**Steve Holmes**
**University of Strathclyde Computer Centre**
**Curran Building**
**100 Cathedral Street**
**Glasgow**

Please note: Steve no longer works for the University of Strathclyde, and we are unable to answer queries relating to this course. You are welcome to make links to it however, but please bear in mind that it was written for students within the University and so some parts may not be relevant to external readers.


This course was awarded a NetGuide Gold Award during the 1990s.

---

*January 1995*


**Next:** Copyright Notice and **Up:** C Programming **Previous:** C Programming

# Contents

---

January 1995

Next Up Previous Contents
**Next:** Introduction **Up:** C Programming **Previous:** Contents

# Copyright Notice and Credits



These notes were written by Steve Holmes of the University of Strathclyde Computer Centre. They originally formed the basis of the Computer Centre's C programming course. Steve no longer works for the University of Strathclyde, and we are unable to answer queries relating to this course. You are welcome to make links to it however, but please bear in mind that it was written for students within the University and so some parts may not be relevant to external readers.

This page has nothing to do with C programming. The rest of this document does, read on and good luck.

January 1995

Next Up Previous Contents
**Next:** About C **Up:** C Programming **Previous:** Copyright Notice and

# Introduction



January 1995

Next Up Previous Contents
**Next:** C and UNIX **Up:** Introduction **Previous:** Introduction

# About C

As a programming language, C is rather like Pascal or Fortran. Values are stored in variables. Programs are structured by defining and calling functions. Program flow is controlled using loops, if statements and function calls. Input and output can be directed to the terminal or to files. Related data can be stored together in arrays or structures.

Of the three languages, C allows the most precise control of input and output. C is also rather more terse than Fortran or Pascal. This can result in short efficient programs, where the programmer has made wise use of C's range of powerful operators. It also allows the programmer to produce programs which are impossible to understand.

Programmers who are familiar with the use of pointers (or indirect addressing, to use the correct term) will welcome the ease of use compared with some other languages. Undisciplined use of pointers can lead to errors which are very hard to trace. This course only deals with the simplest applications of pointers.

It is hoped that newcomers will find C a useful and friendly language. Care must be taken in using C. Many of the extra facilities which it offers can lead to extra types of programming error. You will have to learn to deal with these to successfully make the transition to being a C programmer.

January 1995

Next Up Previous Contents
**Next:** This Course **Up:** Introduction **Previous:** About C

# C and UNIX

This course teaches C under the UNIX operating system. C programs will look similar under any other system (such as VMS or DOS), some other features will differ from system to system. In particular the method of compiling a program to produce a file of runnable code will be different on each system.

The UNIX system is itself written in C. In fact C was invented specifically to implement UNIX. All of the UNIX commands which you type, plus the other system facilities such as password checking, lineprinter queues or magnetic tape controllers are written in C.

In the course of the development of UNIX, hundreds of functions were written to give access to various facets of the system. These functions are available to the programmer in libraries. By writing in C and using the UNIX system libraries, very powerful system programs can be created. These libraries are less easy to access using other programming languages. C is therefore the natural language for writing UNIX system programs.

---

January 1995

Next Up Previous Contents
**Next:** Dialects of C **Up:** Introduction **Previous:** C and UNIX

# This Course

The course aims to introduce programmers to the C language. Previous programming experience is assumed, so we can quickly progress to looking at the features of C and their uses. Students with little programming experience will need to do some homework in order to keep up with the lectures.

Teaching will emphasise the use of supervised practical sessions, giving the student hands on programming experience. The student will collect a number of working practical programs which will be useful reference material for the future.

The notes will include examples and explanation as far as possible. We will try to avoid involved discussion of the syntax of the language. This subject is exhaustively covered in a range of books which are available from bookshops or the University Library.

We aim to introduce C in a structured manner, beginning with the simpler aspects of the language, and working up to more complex issues. Simple aspects will be dealt with rather quickly in order to leave more time for the more powerful features.

---

January 1995

Next Up Previous Contents
**Next:** Common C **Up:** Introduction **Previous:** This Course

# Dialects of C



January 1995

Next Up Previous Contents
**Next:** ANSI C **Up:** Dialects of C **Previous:** Dialects of C

## Common C

Until recently there was one dominant form of the C language. This was the native UNIX form, which for historical reasons is known as either Bell Labs C, after the most popular compiler, or K. &R. C, after the authors of the most popular textbook on the language. It is now often called "Classic C"

January 1995

Next Up Previous Contents
**Next:** A Quick Overview **Up:** Dialects of C **Previous:** Common C

## ANSI C

The American National Standards Institute defined a standard for C, eliminating much uncertainty about the exact syntax of the language. This newcomer, called ANSI C, proclaims itself the standard version of the language. As such it will inevitably overtake, and eventually replace common C.

ANSI C does incorporate a few improvements over the old common C. The main difference is in the grammar of the language. The form of function declarations has been changed making them rather more like Pascal procedures.

This course introduces ANSI C since it is supported by the SUN workstation compilers. Most C programming texts are now available in ANSI editions.

---

January 1995

Next Up Previous Contents
**Next:** A Very Simple **Up:** C Programming **Previous:** ANSI C

# A Quick Overview of C

It is usual to start programming courses with a simple example program. This course is no exception.



January 1995

Next Up Previous Contents
**Next:** A Weight Conversion **Up:** A Quick Overview **Previous:** A Quick Overview

# A Very Simple Program

This program which will print out the message This is a C program

```
#include <stdio.h>

main()
{
        printf("This is a C program\n");
}
```

Though the program is very simple, a few points are worthy of note.

Every C program contains a function called main. This is the start point of the program.

#include <stdio.h> allows the program to interact with the screen, keyboard and filesystem of your computer. You will find it at the beginning of almost every C program.

main() declares the start of the function, while the two curly brackets show the start and finish of the function. Curly brackets in C are used to group statements together as in a function, or in the body of a loop. Such a grouping is known as a compound statement or a block.

printf("This is a C program\n");
prints the words on the screen. The text to be printed is enclosed in double quotes. The \n at the end of the text tells the program to print a newline as part of the output.

Most C programs are in lower case letters. You will usually find upper case letters used in preprocessor definitions (which will be discussed later) or inside quotes as parts of character strings. C is case sensitive, that is, it recognises a lower case letter and it's upper case equivalent as being different.

While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. The following program will print a conversion table for weight in pounds (U.S.A. Measurement) to pounds and stones (Imperial Measurement) or Kilograms (International).

January 1995

Next Up Previous Contents
**Next:** Weight Conversion Table **Up:** A Quick Overview **Previous:** A Very Simple

# A Weight Conversion Program

```
#include <stdio.h>
#define KILOS_PER_POUND .45359
main()
{       int pounds;

        printf(" US lbs      UK st. lbs       INT Kg\n");

        for(pounds=10; pounds < 250; pounds+=10)
        {       int stones = pounds / 14;
                int uklbs = pounds % 14;
                float kilos = pounds * KILOS_PER_POUND;
                printf("   %d           %d   %d        %f\n",
                                pounds, stones, uklbs, kilos);
        }
}
```

Again notice that the only function is called main.

int pounds; Creates a variable of integer type called pounds.

float kilos; Creates a floating point variable (real number) called kilos.

#define KILOS_PER_POUND .45359
defines a constant called KILOS_PER_POUND. This definition will be true throughout the program. It is customary to use capital letters for the names of constants, since they are implemented by the C preprocessor.

`for(pounds=10; pounds < 250; pounds+=10)`
This is the start of the loop. All statements enclosed in the following curly brackets will be repeated. The loop definition contains three parts separated by semi-colons.

- The first is used to initialise variables when the loop is entered.
- The second is a check, when it proves false, the loop is exited.

- The third is a statement used to modify loop counters on each loop iteration after the first.

The effect of `pounds += 10` is to add 10 to the value of the variable pounds. This is a shorthand way of writing `pounds = pounds + 10`.

The printf statement now contains the symbols %d and %f. These are instructions to print out a decimal (integer) or floating (real) value. The values to be printed are listed after the closing quote of the printf statement. Note also that the printf statement has been split over 2 lines so it can fit onto our page. The computer can recognise this because all C statements end with a semicolon.

---

January 1995

Next Up Previous Contents
**Next:** Weight Conversion with **Up:** A Quick Overview **Previous:** A Weight Conversion

# Weight Conversion Table Using a Function

The previous program could be better structured by defining a function to convert the weights and print out their values. It will then look like this.

```
#include <stdio.h>

void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

main()
{       int us_pounds;

        printf(" US lbs      UK st. lbs       INT Kg\n");

        for(us_pounds=10; us_pounds < 250; us_pounds+=10)
                print_converted(us_pounds);
}
```

void print_converted(int pounds) is the beginning of the function definition. The line within the loop reading print_converted(us_pounds) is a call to that function. When execution of the main function reaches that call, print_converted is executed, after which control returns to main.

The text enclosed by symbols /* and */ is a comment. These are C's way of separating plain text comments from the body of the program. It is usually good practice to have a short comment to explain the purpose of each function.

Defining a function has made this program larger, but what have we gained? The structure has been improved. This may make little difference to the readability of such a small program. In a larger program, such structuring makes the program shorter, easier to read, and simplifies future maintenance of the program. Another benefit of defining a function, is that the function can easily be re-used as part of another program.

---

January 1995

Next Up Previous Contents
**Next:** Weight Conversion with **Up:** A Quick Overview **Previous:** Weight Conversion Table

# Weight Conversion with a Prompt

To illustrate this, our next program will re-use the function from 2.3. The program is similar to the last one, but instead of printing a table of weights, the user enters a weight, and this value is converted. Such a program requires user input. This can be done in two ways, both of which will be shown.

The simpler implementation is to prompt for a weight, and then read the user's keyboard input. This would be done as follows

```
#include <stdio.h>

void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

main()
{       int us_pounds;

        printf("Give an integer weight in Pounds : ");
        scanf("%d", &us_pounds);

        printf(" US lbs      UK st. lbs       INT Kg\n");
        print_converted(us_pounds);
}
```

A printf statement is used to prompt for input. scanf is an equivalent input statement, note that the variable to be read us_pounds is written as &us_pounds here. This is very important and it will be

explained later.

---

January 1995

Next Up Previous Contents
**Next:** Fibonacci Series Using **Up:** A Quick Overview **Previous:** Weight Conversion with

# Weight Conversion with Command Line Argument

In this example, the number to be converted is supplied as part of the command to run the program. This can be a useful way of supplying a limited number of names or numbers to a program. We can run the program by typing program 1200 to convert 1200 lbs.

```
#include <stdio.h>

void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

main(int argc,char *argv[])
{       int pounds;

        if(argc != 2)
        {        printf("Usage: convert weight_in_pounds\n");
                exit(1);
        }

        sscanf(argv[1], "%d", &pounds); /* Convert String to int */

        printf(" US lbs      UK st. lbs       INT Kg\n");
        print_converted(pounds);
}
```

The main function definition has changed so that it takes two arguments, argc is a count of the number

of arguments, and argv is an array of strings containing each of the arguments. The system creates these when the program is run.

Some other new concepts are introduced here.

if(argc != 2) Checks that the typed command has two elements, the command name and the weight in pounds.

exit(1); Leave the program. The argument 1 is a way of telling the operating system that an error has occurred. A 0 would be used for a successful exit.

sscanf(argv[1], "%d", &pounds)
Converts a string like "100" into an integer value stored in pounds. The argument is stored in argv[1] as a string. sscanf works like scanf, but reads from a string instead of from the terminal.

The rest of the program is the same as the previous one.

---

January 1995

Next Up Previous Contents
**Next:** Using C with **Up:** A Quick Overview **Previous:** Weight Conversion with

# Fibonacci Series Using an Array

We have now used a variety the features of C. This final example will introduce the array. The program prints out a table of Fibonacci numbers. These are defined by a series in which any element is the sum of the previous two elements. This program stores the series in an array, and after calculating it, prints the numbers out as a table.

```
#include <stdio.h>

main()
{       int fib[24];
        int i;

        fib[0] = 0;
        fib[1] = 1;

        for(i = 2; i < 24; i++)
                fib[i] = fib[i-1] + fib[i-2];

        for (i = 0; i < 24; i++)
                printf("%3d   %6d\n", i, fib[i]);
}
```

One new construct has been introduced here.

int fib[24];
This defines an array called fib of 24 integers. In C, the array index is bounded by square brackets (this avoids confusion with a function call). Also, the first element of the array has index zero, and for this 24 element array, the last element is index 23.

Following this brief scan through the language, we shall introduce the components of C in rather more detail.

January 1995

Next Up Previous Contents
**Next:** Writing the Program **Up:** C Programming **Previous:** Fibonacci Series Using

# Using C with UNIX

A little knowledge is necessary before you can write and compile programs on the UNIX system. Every programmer goes through the same three step cycle.

1. Writing the program into a file
2. Compiling the program
3. Running the program

During program development, the programmer may repeat this cycle many times, refining, testing and debugging a program until a satisfactory result is achieved. The UNIX commands for each step are discussed below.



January 1995

Next Up Previous Contents
**Next:** Compiling the Program **Up:** Using C with **Previous:** Using C with

# Writing the Program

UNIX expects you to store your program in a file whose name ends in .c This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. To edit a file called testprog.c using vi type

```
vi testprog.c
```

The editor is also used to make subsequent changes to the program.

January 1995

Next Up Previous Contents
**Next:** The C Compiler **Up:** Using C with **Previous:** Writing the Program

# Compiling the Program

There are a number of ways to achieve this, though all of them eventually rely on the compiler (called cc on our system).



January 1995

Next Up Previous Contents
**Next:** Makea Program **Up:** Compiling the Program **Previous:** Compiling the Program

## The C Compiler (cc)

The simplest method is to type

```
cc testprog.c
```

This will try to compile testprog.c, and, if successful, will produce a runnable file called a.out. If you want to give the runnable file a better name you can type

```
cc testprog.c -o testprog
```

This will compile testprog.c, creating runnable file testprog.

January 1995

Next Up Previous Contents
**Next:** Improved Type Checking **Up:** Compiling the Program **Previous:** The C Compiler

## Make, a Program Builder

UNIX also includes a very useful program called make. Make allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called Makefile). If your C program is a single file, you can usually use make by simply typing

```
make testprog
```

This will compile testprog.c and put the executable code in testprog.

January 1995

Next Up Previous Contents
**Next:** Running the Program **Up:** Compiling the Program **Previous:** Makea Program

## Improved Type Checking Using Lint

The C compiler is rather liberal about type checking function arguments, it doesn't check bounds of array indices. There is a stricter checker called lint which won't generate any runnable code. It is a good idea to use lint to check your programs before they are completed. This is done by typing

```
lint testprog.c
```

Lint is very good at detecting errors which cause programs to crash at run time. However, lint is very fussy, and generally produces a long list of messages about minor problems with the program. Many of these will be quite harmless. Experience will teach you to distinguish the important messages from those which can be ignored.

---

January 1995

Next Up Previous Contents
**Next:** Constant and Variable **Up:** Using C with **Previous:** Improved Type Checking

# Running the Program

To run a program under UNIX you simply type in the filename. So to run program testprog, you would type

```
testprog
```

or if this fails to work, you could type

```
./testprog
```

You will see your prompt again after the program is done.

January 1995

Next Up Previous Contents
**Next:** Variables **Up:** C Programming **Previous:** Running the Program

# Constant and Variable Types



January 1995

Next Up Previous Contents
**Next:** Variable Names **Up:** Constant and Variable **Previous:** Constant and Variable

# Variables

In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.

A declaration begins with the type, followed by the name of one or more variables. For example,

```
int high, low, results[20];
```

Declarations can be spread out, allowing space for an explanatory comment. Variables can also be initialised when they are declared, this is done by adding an equals sign and the required value after the declaration.

```
int high = 250;     /* Maximum Temperature */
int low = -40;      /* Minimum Temperature */
int results[20];    /* Series of temperature readings */
```

C provides a wide range of types. The most common are

| | |
|---|---|
| `int` | An Integer |
| `float` | A floating point (real) number |
| `char` | A single byte of memory, enough to hold a character |

There are also several variants on these types.

| | |
|---|---|
| `short` | An integer, possibly of reduced range |
| `long` | An integer, possibly of increased range |
| `unsigned` | An integer with no negative range, the spare capacity being used to increase the positive range |
| `unsigned long` | Like unsigned, possibly of increased range |
| `double` | A double precision floating point number. |

All of the integer types plus the char are called the integral types. float and double are called the real types.

---

January 1995

Next Up Previous Contents
**Next:** Global Variables **Up:** Constant and Variable **Previous:** Variables

# Variable Names

Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognises upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names include

```
x          result          outfile          bestyet
x1         x2              out_file         best_yet
power      impetus         gamma            hi_score
```

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name. Most modern ones don't apply this limit though. 

The rules governing variable names also apply to the names of functions. We shall meet functions later on in the course.

January 1995

Next Up Previous Contents
**Next:** External Variables **Up:** Constant and Variable **Previous:** Variable Names

# Global Variables

Local variables are declared within the body of a function, and can only be used within that function. This is usually no problem, since when another function is called, all required data is passed to it as arguments. Alternatively, a variable can be declared globally so it is available to all functions. Modern programming practice recommends against the excessive use of global variables. They can lead to poor program structure, and tend to clog up the available name space.

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

- External Variables

January 1995

Next Up Previous Contents
**Next:** Static Variables **Up:** Global Variables **Previous:** Global Variables

## External Variables

Where a global variable is declared in one file, but used by functions from another, then the variable is called an external variable in these functions, and must be declared as such. The declaration must be preceeded by the word extern. The declaration is required so the compiler can find the type of the variable without having to search through several source files for the declaration.

Global and external variables can be of any legal type. They can be initialised, but the initialisation takes place when the program starts up, before entry to the main function.

January 1995

Next Up Previous Contents
**Next:** Constants **Up:** Constant and Variable **Previous:** External Variables

# Static Variables

Another class of local variable is the static type. A static can only be accessed from the function in which it was declared, like a local variable. The static variable is not destroyed on exit from the function, instead its value is preserved, and becomes available again when the function is next called. Static variables are declared as local variables, but the declaration is preceeded by the word static.

```
static int counter;
```

Static variables can be initialised as normal, the initialisation is performed once only, when the program starts up.

January 1995

Next Up Previous Contents
**Next:** Arrays **Up:** Constant and Variable **Previous:** Static Variables

# Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence.

| | |
|---|---|
| '\n' | newline |
| '\t' | tab |
| '\\ ' | backslash |
| '\'' | single quote |
| '\0' | null (used automatically to terminate character strings). |

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes eg "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

A character is a different type to a single character string. This is important.

We will meet strings and characters again when we deal with the input / output functions in more detail.

---

January 1995

Next Up Previous Contents
**Next:** Expressions and Operators **Up:** Constant and Variable **Previous:** Constants

# Arrays

An array is a collection of variables of the same type. Individual array elements are identified by an integer index. In C the index begins at zero and is always written inside square brackets.

We have already met single dimensioned arrays which are declared like this

```
int results[20];
```

Arrays can have more dimensions, in which case they might be declared as

```
int results_2d[20][5];
int results_3d[20][5][3];
```

Each index has its own set of square brackets.

Where an array is declared in the main function it will usually have details of dimensions included. It is possible to use another type called a pointer in place of an array. This means that dimensions are not fixed immediately, but space can be allocated as required. This is an advanced technique which is only required in certain specialised programs.

When passed as an argument to a function, the receiving function need not know the size of the array. So for example if we have a function which sorts a list (represented by an array) then the function will be able to sort lists of different sizes. The drawback is that the function is unable to determine what size the list is, so this information will have to be passed as an additional argument.

As an example, here is a simple function to add up all of the integers in a single dimensioned array.

```
int add_array(int array[], int size)
{       int i;
        int total = 0;

        for(i = 0; i < size; i++)
                total += array[i];
```

```
        return(total);
}
```

---

January 1995

Next Up Previous Contents
**Next:** Assignment Statement **Up:** C Programming **Previous:** Arrays

# Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /.

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.



January 1995

Next Up Previous Contents
**Next:** Arithmetic operators **Up:** Expressions and Operators **Previous:** Expressions and Operators

# Assignment Statement

The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like

```
y = (m * x) + c
```

This assignment will save the value of the expression in variable y.

January 1995

Next Up Previous Contents
**Next:** Type conversion **Up:** Expressions and Operators **Previous:** Assignment Statement

# Arithmetic operators

Here are the most common arithmetic operators

| | |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulo Reduction (Remainder from integer division) |

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.

Here are some arithmetic expressions used within assignment statements.

```
velocity = distance / time;

force = mass * acceleration;

count = count + 1;
```

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

| Shorthand | Equivalent |
|---|---|
| `i++;` or `++i;` | `i = i + 1;` |
| `i--;` or `--i;` | `i = i - 1;` |

These operations are usually very efficient. They can be combined with another expression.

```
x = a * b++;   is equivalent to   x = a * b;
                                  b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

```
x = --i * (a + b);   is equivalent to   i = i - 1;
                                        x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - to ensure that your programs stay readable.

Another shorthand notation is listed below

| Shorthand | Equivalent |
|---|---|
| `i += 10;` | `i = i + 10;` |
| `i -= 10;` | `i = i - 10;` |
| `i *= 10;` | `i = i * 10;` |
| `i /= 10;` | `i = i / 10;` |

These are simple to read and use.

---

January 1995

Next Up Previous Contents
**Next:** Comparison **Up:** Expressions and Operators **Previous:** Arithmetic operators

# Type conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size, so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.

There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;

- The variable is too small to hold the value. In this case it will be corrupted (this is bad).
- The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.

Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting which temporarily disguises a value as a different type.

eg. The function sqrt finds the square root of a double.

```
int i = 256;
int root

root = sqrt( (double) i);
```

The cast is made by putting the bracketed name of the required type just before the value. (double) in this example. The result of sqrt( (double) i); is also a double, but this is automatically converted to an int on assignment to root.

January 1995

Next Up Previous Contents
**Next:** Logical Connectors **Up:** Expressions and Operators **Previous:** Type conversion

# Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

| C notation | Meaning |
|---|---|
| == | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| != | Not equal to |

Note that == is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

```
x == y

i > 10

a + b != c
```

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement or a for or a while loop. These will be introduced in a later chapter.

January 1995

Next Up Previous Contents
**Next:** Summary **Up:** Expressions and Operators **Previous:** Comparison

# Logical Connectors

These are the usual And, Or and Not operators.

| Symbol | Meaning |
|---|---|
| && | And |
| \|\| | Or |
| ! | Not |

They are frequently used to combine relational operators, for example

```
x < 20 && x >= 10
```

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
        printf("Not Acceptable !!\n");
```

---

January 1995

Next Up Previous Contents
**Next:** Control Statements **Up:** Expressions and Operators **Previous:** Logical Connectors

# Summary

Three types of expression have been introduced here;

- Arithmetic expressions are simple, but watch out for subtle type conversions. The shorthand notations may save you a lot of typing.
- Comparison takes two numbers and produces a logical result. Comparisons are usually found controlling if statements or loops.
- Logical connectors allow several comparisons to be combined into a single test. Lazy evaluation can improve the efficiency of the program by reducing the amount of calculation required.

C also provides bit manipulation operators. These are too specialised for the scope of this course.

---

January 1995

Next Up Previous Contents
**Next:** The if else **Up:** C Programming **Previous:** Summary

# Control Statements

A program consists of a number of statements which are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.

Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

This section will discuss the use of control statements in C. We will show how they can be used to write powerful programs by;

- Repeating important sections of the program.
- Selecting between optional sections of a program.

---



---

January 1995

Next Up Previous Contents
**Next:** The switch Statement **Up:** Control Statements **Previous:** Control Statements

# The if else Statement

This is used to decide whether to do something at a special point, or to decide between two courses of action.

The following test decides whether a student has passed an exam with a pass mark of 45

```
if (result >= 45)
        printf("Pass\n");
else
        printf("Fail\n");
```

It is possible to use the if part without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If is is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{       printf("Passed\n");
        printf("Congratulations\n")
}
else
{       printf("Failed\n");
        printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several

comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
        printf("Passed: Grade A\n");
else if (result >= 60)
        printf("Passed: Grade B\n");
else if (result >= 45)
        printf("Passed: Grade C\n");
else
        printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

---

January 1995

Next Up Previous Contents
**Next:** Loops **Up:** Control Statements **Previous:** The if else

# The switch Statement

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
        }
```

```
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

---

January 1995

Next Up Previous Contents
**Next:** The while Loop **Up:** Control Statements **Previous:** The switch Statement

# Loops

C gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loops is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.

January 1995

Next Up Previous Contents
**Next:** The do while **Up:** Control Statements **Previous:** Loops

# The while Loop

The while loop repeats a statement until the test at the top proves false.

As an example, here is a function to return the length of a string. Remember that the string is represented as an array of characters terminated by a null character '\0'.

```
int string_length(char string[])
{       int i = 0;

        while (string[i] != '\0')
                i++;

        return(i);
}
```

The string is passed to the function as an argument. The size of the array is not specified, the function will work for a string of any size.

The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't null, the index is incremented and the test is repeated.

---

January 1995

Next Up Previous Contents
**Next:** The for Loop **Up:** Control Statements **Previous:** The while Loop

# The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

January 1995

Next Up Previous Contents
**Next:** The break Statement **Up:** Control Statements **Previous:** The do while

# The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialisation of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

The example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{       float total = 0.0;
        int i;

        for(i = 0; i < count; i++)
                total += array[i];

        return(total / count);
}
```

The for loop ensures that the correct number of array elements are added up before calculating the average.

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialisation or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behaviour to be specified simply and quickly.

---

January 1995

Next Up Previous Contents
**Next:** The continue Statement **Up:** Control Statements **Previous:** The for Loop

# The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

---

January 1995

Next Up Previous Contents
**Next:** The goto Statement **Up:** Control Statements **Previous:** The break Statement

# The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

January 1995

Next Up Previous Contents
**Next:** Functions in C **Up:** Control Statements **Previous:** The continue Statement

# The goto Statement

C has a goto statement which permits unstructured jumps to be made. Its use is not recommended, so we'll not teach it here. Consult your textbook for details of its use.

January 1995

Next Up Previous Contents
**Next:** Scope of Function **Up:** C Programming **Previous:** The goto Statement

# Functions in C

Almost all programming languages have some equivalent of the function. You may have met them under the alternative names subroutine or procedure.

Some languages distinguish between functions which return variables and those which don't. C assumes that every function will return a value. If the programmer wants a return value, this is achieved using the return statement. If no return value is required, none should be used when calling the function.

Here is a function which raises a double to the power of an unsigned, and returns the result.

```
double power(double val, unsigned pow)
{       double ret_val = 1.0;
        unsigned i;

        for(i = 0; i < pow; i++)
                ret_val *= val;

        return(ret_val);
}
```

The function follows a simple algorithm, multiplying the value by itself pow times. A for loop is used to control the number of multiplications, and variable ret_val stores the value to be returned. Careful programming has ensured that the boundary condition is correct too. ie .

Let us examine the details of this function.

double power(double val, unsigned pow)
This line begins the function definition. It tells us the type of the return value, the name of the function, and a list of arguments used by the function. The arguments and their types are enclosed in brackets, each pair separated by commas.

The body of the function is bounded by a set of curly brackets. Any variables declared here will be treated as local unless specifically declared as static or extern types.

return(ret_val);
On reaching a return statement, control of the program returns to the calling function. The bracketed value is the value which is returned from the function. If the final closing curly bracket is reached before any return value, then the function will return automatically, any return value will then be meaningless.

The example function can be called by a line in another function which looks like this

```
result = power(val, pow);
```

This calls the function power assigning the return value to variable result.

Here is an example of a function which does not return a value.

```
void error_line(int line)
{      fprintf(stderr, "Error in input data: line %d\n", line);
}
```

The definition uses type void which is optional. It shows that no return value is used. Otherwise the function is much the same as the previous example, except that there is no return statement. Some void type functions might use return, but only to force an early exit from the function, and not to return any value. This is rather like using break to jump out of a loop.

This function also demonstrates a new feature.

fprintf(stderr, "Error in input data: line %d\n", line);
This is a variant on the printf statement, fprintf sends its output into a file. In this case, the file is stderr. stderr is a special UNIX file which serves as the channel for error messages. It is usually connected to the console of the computer system, so this is a good way to display error messages from your programs. Messages sent to stderr will appear on screen even if the normal output of the program has been redirected to a file or a printer.

The function would be called as follows

```
error_line(line_number);
```

---


- Scope of Function Variables
- Modifying Function Arguments
- Pointers in C

- Arrays and Pointers
- Recursive Functions

---

Next Up Previous Contents
**Next:** Scope of Function **Up:** C Programming **Previous:** The goto Statement

---

---

January 1995

Next Up Previous Contents
**Next:** Modifying Function Arguments **Up:** Functions in C **Previous:** Functions in C

# Scope of Function Variables

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

Local variables are declared within a function. They are created anew each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

Static variables are slightly different, they don't die on return from the function. Instead their last value is retained, and it becomes available when the function is called again.

Global variables don't die on return from a function. Their value is retained, and is available to any other function which accesses them.

---

January 1995

Next Up Previous Contents
**Next:** Pointers in C **Up:** Functions in C **Previous:** Scope of Function

# Modifying Function Arguments

Some functions work by modifying the values of their arguments. This may be done to pass more than one value back to the calling routine, or because the return value is already being used in some way. C requires special arrangements for arguments whose values will be changed.

You can treat the arguments of a function as variables, however direct manipulation of these arguments won't change the values of the arguments in the calling function. The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.

There is a way to change the values of variables declared outside the function. It is done by passing the addresses of variables to the function. These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer. It is what lies at the address which the pointer references which interests us.

To get back to our original function, we pass it the address of a variable whose value we wish to change. The function must now be written to use the value at that address (or at the end of the pointer). On return from the function, the desired value will have changed. We manipulate the actual value using a copy of the pointer.

---

January 1995

Next Up Previous Contents
**Next:** Arrays and Pointers **Up:** Functions in C **Previous:** Modifying Function Arguments

# Pointers in C

Pointers are not exclusive to functions, but this seems a good place to introduce the pointer type.

Imagine that we have an int called i. Its address could be represented by the symbol &i. If the pointer is to be stored as a variable, it should be stored like this.

```
int *pi = &i;
```

int * is the notation for a pointer to an int. & is the operator which returns the address of its argument. When it is used, as in &i we say it is referencing i.

The opposite operator, which gives the value at the end of the pointer is *. An example of use, known as de-referencing pi, would be

```
i = *pi;
```

Take care not to confuse the many uses of the * sign; Multiplication, pointer declaration and pointer de-referencing.

This is a very confusing subject, so let us illustrate it with an example. The following function fiddle takes two arguments, x is an int while y is a pointer to int. It changes both values.

```
fiddle(int x, int *y)
{   printf(" Starting fiddle: x = %d, y = %d\n", x, *y);
    x ++;
    (*y)++;
    printf("Finishing fiddle: x = %d, y = %d\n", x, *y);
}
```

since y is a pointer, we must de-reference it before incrementing its value.

A very simple program to call this function might be as follows.

```
main()
{   int i = 0;
    int j = 0;

    printf(" Starting main  : i = %d, j = %d\n", i, j);
    printf("Calling fiddle now\n");.
    fiddle(i, &j);
    printf("Returned from fiddle\n");
    printf("Finishing main  : i = %d, j = %d\n", i, j);
}
```

Note here how a pointer to int is created using the & operator within the call fiddle(i, &j);.

The result of running the program will look like this.

```
 Starting main  : i =   0 ,j =   0
Calling fiddle now
 Starting fiddle: x =   0, y =   0
Finishing fiddle: x =   1, y =   1
Returned from fiddle
Finishing main  : i =   0, j =   1
```

After the return from fiddle the value of i is unchanged while j, which was passed as a pointer, has changed.

To summarise, if you wish to use arguments to modify the value of variables from a function, these arguments must be passed as pointers, and de-referenced within the function.

Where the value of an argument isn't modified, the value can be passed without any worries about pointers.

---

January 1995

Next Up Previous Contents
**Next:** Recursive Functions **Up:** Functions in C **Previous:** Pointers in C

# Arrays and Pointers

To fully understand the workings of C you must know that pointers and arrays are related.

An array is actually a pointer to the 0th element of the array. Dereferencing the array name will give the 0th element. This gives us a range of equivalent notations for array access. In the following examples, arr is an array.

| Array Access | Pointer Equivalent |
|---|---|
| `arr[0]` | `*arr` |
| `arr[2]` | `*(arr + 2)` |
| `arr[n]` | `*(arr + n)` |

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the values in the array, but not the array itself, so statements like `arr ++` are illegal, but `arr[n] ++` is legal.

Since an array is like a pointer, we can pass an array to a function, and modify elements of that array without having to worry about referencing and de-referencing. Since the array is implemented as a hidden pointer, all the difficult stuff gets done automatically.

A function which expects to be passed an array can declare that parameter in one of two ways.

```
int arr[];  or int *arr;
```

Either of these definitions is independent of the size of the array being passed. This is met most frequently in the case of character strings, which are implemented as an array of type char. This could be declared as char string[]; but is most frequently written as char *string; In the same way, the argument vector argv is an array of strings which can be supplied to function main. It can be declared as one of the following.

```
char **argv;  or  char *argv[];
```

Don't panic if you find pointers confusing. While you will inevitably meet pointers in the form of

strings, or as variable arguments for functions, they need not be used in most other simple types of programs.

---

January 1995

Next Up Previous Contents
**Next:** Input and Output **Up:** Functions in C **Previous:** Arrays and Pointers

# Recursive Functions

A recursive function is one which calls itself. This is another complicated idea which you are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{       switch(num) {
        case 0:
                return(0);
                break;
        case 1:
                return(1);
                break;
        default:  /* Including recursive calls */
                return(fib(num - 1) + fib(num - 2));
                break;
        }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
        if(pow == 0)  /* pow(x, 0) returns 1 */
                return(1.0);
        else
                return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly, otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the fibonacci function of a moderate size number.

| Input Value | Number of times `fib` is called |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 3 |
| 3 | 5 |
| 4 | 9 |
| 5 | 15 |
| 6 | 25 |
| 7 | 41 |
| 8 | 67 |
| 9 | 109 |
| 10 | 177 |

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

---

January 1995

Next Up Previous Contents
**Next:** The Standard Input **Up:** C Programming **Previous:** Recursive Functions

# Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analysing them and storing results in variables. These variants of the basic input and output commands are discussed in the next section



January 1995

Next Up Previous Contents
**Next:** Character Input / **Up:** Input and Output **Previous:** Input and Output

# The Standard Input Output File

UNIX supplies a standard package for performing input and output to files or the terminal. This contains most of the functions which will be introduced in this section, along with definitions of the datatypes required to use them. To use these facilities, your program must include these definitions by adding the line This is done by adding the line

```
#include <stdio.h>
```

near the start of the program file.

If you do not do this, the compiler may complain about undefined functions or datatypes.

---

January 1995

Next Up Previous Contents
**Next:** getchar **Up:** Input and Output **Previous:** The Standard Input

# Character Input / Output

This is the lowest level of input and output. It provides very precise control, but is usually too fiddly to be useful. Most computers perform buffering of input and output. This means that they'll not start reading any input until the return key is pressed, and they'll not print characters on the terminal until there is a whole line to be printed.



January 1995

Next Up Previous Contents
**Next:** putchar **Up:** Character Input / **Previous:** Character Input /

## getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>

main()
{   int ch, i = 0;

    while((ch = getchar()) != EOF)
        i ++;

    printf("%d\n", i);
}
```

January 1995

Next Up Previous Contents
**Next:** Formatted Input / **Up:** Character Input / **Previous:** getchar

## putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function toupper from the character conversion library ctype.h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

main()
{   int ch;

    while((ch = getchar()) != EOF)
        putchar(toupper(ch));
}
```

January 1995

Next Up Previous Contents
**Next:** printf **Up:** Input and Output **Previous:** putchar

# Formatted Input / Output

We have met these functions earlier in the course. They are closest to the facilities offered by Pascal or Fortran, and usually the easiest to use for input and output. The versions offered under C are a little more detailed, offering precise control of layout.



January 1995

Next Up Previous Contents
**Next:** scanf **Up:** Formatted Input / **Previous:** Formatted Input /

## printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string.

| Control String Entry | What Gets Printed |
|---|---|
| %d | A Decimal Integer |
| %f | A Floating Point Value |
| %c | A Character |
| %s | A Character String |

There are several more types available. For full details type

```
man printf
```

on your UNIX system.

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field.

January 1995

Next Up Previous Contents
**Next:** Whole Lines of **Up:** Formatted Input / **Previous:** printf

## scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceeded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceeded by the percentage sign in a similar way to their printf equivalents.

Type man scanf for details of all options on your system.

---

January 1995

Next Up Previous Contents
**Next:** gets **Up:** Input and Output **Previous:** scanf

# Whole Lines of Input and Output

Where we are not too interested in the format of our data, or perhaps we cannot predict its format in advance, we can read and write whole lines as character strings. This approach allows us to read in a line of input, and then use various string handling functions to analyse it at our leisure.



January 1995

Next Up Previous Contents
**Next:** puts **Up:** Whole Lines of **Previous:** Whole Lines of

## gets

gets reads a whole line of input into a string until a newline or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in stdio.h is returned.

---

January 1995

Next Up Previous Contents
**Next:** Handling Files in **Up:** Whole Lines of **Previous:** gets

## puts

puts writes a string to the output, and follows it with a newline character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>

main()
{       char line[256]; /* Define string sufficiently large to
                           store a line of input */

        while(gets(line) != NULL)   /* Read line        */
        {       puts(line);         /* Print line       */
                printf("\n");       /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together. So can getchar, scanf and gets.

January 1995

Next Up Previous Contents
**Next:** UNIX File Redirection **Up:** C Programming **Previous:** puts

# Handling Files in C

This section describes the use of C's input / output facilities for reading and writing files. There is also a brief description of string handling functions here.

The functions are all variants on the forms of input / output which were introduced in the previous section.



January 1995

Next Up Previous Contents
**Next:** C File Handling **Up:** Handling Files in **Previous:** Handling Files in

# UNIX File Redirection

UNIX has a facility called redirection which allows a program to access a single input file and a single output file very easily. The program is written to read from the keyboard and write to the terminal screen as normal.

To run prog1 but read data from file infile instead of the keyboard, you would type

```
prog1 < infile
```

To run prog1 and write data to outfile instead of the screen, you would type

```
prog1 > outfile
```

Both can also be combined as in

```
prog1 < infile > outfile
```

Redirection is simple, and allows a single program to read or write data to or from files or the screen and keyboard.

Some programs need to access several files for input or output, redirection cannot do this. In such cases you will have to use C's file handling facilities.

---

January 1995

Next Up Previous Contents
**Next:** Opening a file **Up:** Handling Files in **Previous:** UNIX File Redirection

# C File Handling - File Pointers

C communicates with files using a new datatype called a file pointer. This type is defined within stdio.h, and written as FILE *. A file pointer called output_file is declared in a statement like

```
FILE *output_file;
```



January 1995

Next Up Previous Contents
**Next:** Standard file pointers **Up:** C File Handling **Previous:** C File Handling

## Opening a file pointer using fopen

Your program must open a file before it can access it. This is done using the fopen function, which returns the required file pointer. If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows

```
if ((output_file = fopen("output_file", "w")) == NULL)
        fprintf(stderr, "Cannot open %s\n", "output_file");
```

fopen takes two arguments, both are strings, the first is the name of the file to be opened, the second is an access character, which is usually one of:

"r" Open file for reading
"w" Create file for writing
"a" Open file for appending

As usual, use the man command for further details by typing man fopen.

January 1995

Next Up Previous Contents
**Next:** Closing a file **Up:** C File Handling **Previous:** Opening a file

## Standard file pointers in UNIX

UNIX systems provide three file descriptors which are automatically open to all C programs. These are

| | |
|---|---|
| `stdin` | The standard input. The keyboard or a redirected input file. |
| `stdout` | The standard output. The screen or a redirected output file. |
| `stderr` | The standard error. This is the screen, even when output is redirected. This is the conventional place to put any error messages. |

Since these files are already open, there is no need to use fopen on them.

January 1995

Next Up Previous Contents
**Next:** Input and Output **Up:** C File Handling **Previous:** Standard file pointers

## Closing a file using fclose

The fclose command can be used to disconnect a file pointer from a file. This is usually done so that the pointer can be used to access a different file. Systems have a limit on the number of files which can be open simultaneously, so it is a good idea to close a file when you have finished using it.

This would be done using a statement like

```
fclose(output_file);
```

If files are still open when a program exits, the system will close them for you. However it is usually better to close the files properly.

---

January 1995

Next Up Previous Contents
**Next:** Character Input and **Up:** Handling Files in **Previous:** Closing a file

# Input and Output using file pointers

Having opened a file pointer, you will wish to use it for either input or output. C supplies a set of functions to allow you to do this. All are very similar to input and output functions that you have already met.



January 1995

Next Up Previous Contents
**Next:** Formatted Input Output **Up:** Input and Output **Previous:** Input and Output

## Character Input and Output with Files

This is done using equivalents of getchar and putchar which are called getc and putc. Each takes an extra argument, which identifies the file pointer to be used for input or output.

```
putchar(c)  is equivalent to  putc(c, stdout)
getchar()   is equivalent to  getc(stdin)
```

January 1995

Next Up Previous Contents
**Next:** Formatted Input Output **Up:** Input and Output **Previous:** Character Input and

## Formatted Input Output with File Pointers

Similarly there are equivalents to the functions printf and scanf which read or write data to files. These are called fprintf and fscanf. You have already seen fprintf being used to write data to stderr.

The functions are used in the same way, except that the fprintf and fscanf take the file pointer as an additional first argument.

January 1995

Next Up Previous Contents
**Next:** Whole Line Input **Up:** Input and Output **Previous:** Formatted Input Output

## Formatted Input Output with Strings

These are the third set of the printf and scanf families. They are called sprintf and sscanf.

sprintf
: puts formatted data into a string which must have sufficient space allocated to hold it. This can be done by declaring it as an array of char. The data is formatted according to a control string of the same form as that for p rintf.

sscanf
: takes data from a string and stores it in other variables as specified by the control string. This is done in the same way that scanf reads input data into variables. sscanf is very useful for converting strings into numeric v values.

January 1995

Next Up Previous Contents
**Next:** Special Characters **Up:** Input and Output **Previous:** Formatted Input Output

## Whole Line Input and Output using File Pointers

Predictably, equivalents to gets and puts exist called fgets and fputs. The programmer should be careful in using them, since they are incompatible with gets and puts. gets requires the programmer to specify the maximum number of characters to be read. fgets and fputs retain the trailing newline character on the line they read or write, wheras gets and puts discard the newline.

When transferring data from files to standard input / output channels, the simplest way to avoid incompatibility with the newline is to use fgets and fputs for files and standard channels too.

For Example, read a line from the keyboard using

```
fgets(data_string, 80, stdin);
```

and write a line to the screen using

```
fputs(data_string, stdout);
```

---

January 1995

Next Up Previous Contents
**Next:** NULLThe Null **Up:** Handling Files in **Previous:** Whole Line Input

# Special Characters

C makes use of some 'invisible' characters which have already been mentioned. However a fuller description seems appropriate here.



January 1995

Next Up Previous Contents
**Next:** EOFThe End **Up:** Special Characters **Previous:** Special Characters

## NULL, The Null Pointer or Character

NULL is a character or pointer value. If a pointer, then the pointer variable does not reference any object (i.e. a pointer to nothing). It is usual for functions which return pointers to return NULL if they failed in some way. The return value can be tested. See the section on fopen for an example of this.

NULL is returned by read commands of the gets family when they try to read beyond the end of an input file.

Where it is used as a character, NULL is commonly written as '\0'. It is the string termination character which is automatically appended to any strings in your C program. You usually need not bother about this final \0', since it is handled automatically. However it sometimes makes a useful target to terminate a string search. There is an example of this in the string_length function example in the section on Functions in C.

---

January 1995

Next Up Previous Contents
**Next:** Other String Handling **Up:** Special Characters **Previous:** NULLThe Null

## EOF, The End of File Marker

EOF is a character which indicates the end of a file. It is returned by read commands of the getc and scanf families when they try to read beyond the end of a file.

January 1995

Next Up Previous Contents
**Next:** Conclusion **Up:** Handling Files in **Previous:** EOFThe End

# Other String Handling Functions

As well as sprintf and sscanf, the UNIX system has a number of other string handling functions within its libraries. A number of the most useful ones are contained in the <strings.h> file, and are made available by putting the line

```
#include <strings.h>
```

near to the head of your program file.

A couple of the functions are described below.

| | |
|---|---|
| `strcpy(str1, str2)` | copies `str2` into `str1`. |
| `strcmp(str1, str2)` | compares the contents of `str1` and `str2`. Returns 0 (false) if both are equal. |

A full list of these functions can be seen using the man command by typing

```
man 3 strings
```

January 1995

Next Up Previous Contents
**Next:** Structures in C **Up:** Handling Files in **Previous:** Other String Handling

# Conclusion

The variety of different types of input and output, using standard input or output, files or character strings make C a very powerful language. The addition of character input and output make it highly suitable for applications where the format of data must be controlled very precisely.

January 1995

Next Up Previous Contents
**Next:** Defining a Structure **Up:** C Programming **Previous:** Conclusion

# Structures in C

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.



January 1995

Next Up Previous Contents
**Next:** Accessing Members of **Up:** Structures in C **Previous:** Structures in C

# Defining a Structure

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
} student;
```

This defines a new type student variables of type student can be declared as follows.

```
student st_rec;
```

Notice how similar this is to declaring an int or float.

The variable name is st_rec, it has members called name, course, age and year.

---

January 1995

Next Up Previous Contents
**Next:** Structures as Function **Up:** Structures in C **Previous:** Defining a Structure

# Accessing Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure st_rec will behave just like a normal array of char, however we refer to it by the name

```
st_rec.name
```

Here the dot is an operator which selects a member from a structure.

Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own operator -> which acts as follows. Assume that st_ptr is a pointer to a structure of type student We would refer to the name member as

```
st_ptr -> name
```

---

January 1995

Next Up Previous Contents
**Next:** Further Uses of **Up:** Structures in C **Previous:** Accessing Members of

# Structures as Function Arguments

A structure can be passed as a function argument just like any other variable. This raises a few practical issues.

Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.

If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

---

January 1995

Next Up Previous Contents
**Next:** The C Preprocessor **Up:** Structures in C **Previous:** Structures as Function

# Further Uses of Structures

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct {
        double real;
        double imag;
} complex;
```

doubles have been used for each field because their range is greater than floats and because the majority of mathematical library functions deal with doubles by default.

In a similar way, structures could be used to hold the locations of points in multi-dimensional space. Mathematicians and engineers might see a storage efficient implementation for sparse arrays here.

Apart from holding data, structures can be used as members of other structures. Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases.

Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other connected structures. These are rather daunting to the new programmer, so we won't deal with them here.

---

January 1995

Next Up Previous Contents
**Next:** Using #define to **Up:** C Programming **Previous:** Further Uses of

# The C Preprocessor

The C preprocessor is a tool which filters your source code before it is compiled. The preprocessor allows constants to be named using the #define notation. The preprocessor provides several other facilities which will be described here. It is particularly useful for selecting machine dependent pieces of code for different computer types, allowing a single program to be compiled and run on several different computers.

The C preprocessor isn't restricted to use with C programs, and programmers who use other languages may also find it useful, however it is tuned to recognise features of the C language like comments and strings, so its use may be restricted in other circu mstances.

The preprocessor is called cpp, however it is called automatically by the compiler so you will not need to call it while programming in C.



January 1995

Next Up Previous Contents
**Next:** Using #define to **Up:** The C Preprocessor **Previous:** The C Preprocessor

# Using #define to Implement Constants

We have already met this facility, in its simplest form it allows us to define textual substitutions as follows.

```
#define MAXSIZE 256
```

This will lead to the value 256 being substituted for each occurrence of the word MAXSIZE in the file.

January 1995

Next Up Previous Contents
**Next:** Reading in Other **Up:** The C Preprocessor **Previous:** Using #define to

# Using #define to Create Functional Macros

#define can also be given arguments which are used in its replacement. The definitions are then called macros. Macros work rather like functions, but with the following minor differences.

- Since macros are implemented as a textual substitution, there is no effect on program performance (as with functions).
- Recursive macros are generally not a good idea.
- Macros don't care about the type of their arguments. Hence macros are a good choice where we might want to operate on reals, integers or a mixture of the two. Programmers sometimes call such type flexibility polymorphism.
- Macros are generally fairly small.

Macros are full of traps for the unwary programmer. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules.

Here is an example of a macro which won't work.

```
#define DOUBLE(x) x+x
```

Now if we have a statement

```
a = DOUBLE(b) * c;
```

This will be expanded to

```
a = b+b * c;
```

And since * has a higher priority than +, the compiler will treat it as.

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

The problem can be solved using a more robust definition of DOUBLE

```
#define DOUBLE(x) (x+x)
```

Here the brackets around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.

In general it is better to write a C function than risk using a macro.

---

January 1995

Next Up Previous Contents
**Next:** Conditional selection of **Up:** The C Preprocessor **Previous:** Using #define to

# Reading in Other Files using #include

The preprocessor directive #include is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions.

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is /usr/include for most UNIX systems.

For example

```
#include <stdio.h>
```

Another use for #include for the programmer is where multi-file programs are being written. Certain information is required at the beginning of each program file. This can be put into a file called globals.h and included in each program file. Local header file names are usually enclosed by double quotes, " ". It is conventional to give header files a name which ends in .h to distinguish them from other types of file.

Our globals.h file would be included by the following line.

```
#include "globals.h"
```

---

January 1995

Next Up Previous Contents
**Next:** Using #ifdef for **Up:** The C Preprocessor **Previous:** Reading in Other

# Conditional selection of code using #ifdef

The preprocessor has a conditional statement similar to C's if else. It can be used to selectively include statements in a program. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program which will run on either type.

The keywords for conditional selection are; #ifdef, #else and #endif.

#ifdef
: takes a name as an argument, and returns true if the the name has a current definition. The name may be defined using a #define, the -d option of the compiler, or certain names which are automatically defined by the UNIX environment.

#else
: is optional and ends the block beginning with #ifdef. It is used to create a 2 way optional selection.

#endif
: ends the block started by #ifdef or #else.

Where the #ifdef is true, statements between it and a following #else or #endif are included in the program. Where it is false, and there is a following #else, statements between the #else and the following #endif are included.

This is best illustrated by an example.



January 1995

Next Up Previous Contents
**Next:** Using #ifdef to **Up:** Conditional selection of **Previous:** Conditional selection of

## Using #ifdef for Different Computer Types

Conditional selection is rarely performed using #defined values. A simple application using machine dependent values is illustrated below.

```
#include <stdio.h>

main()
{
#ifdef vax
      printf("This is a VAX\n");
#endif
#ifdef sun
      printf("This is a SUN\n");
#endif
}
```

sun is defined automatically on SUN computers. vax is defined automatically on VAX computers.

January 1995

Next Up Previous Contents
**Next:** Programs with Several **Up:** Conditional selection of **Previous:** Using #ifdef for

## Using #ifdef to Temporarily Remove Program Statements

#ifdef also provides a useful means of temporarily `blanking out' lines of a program. The lines in question are preceeded by #ifdef NEVER and followed by #endif. Of course you should ensure that the name NEVER isn't defined anywhere.

The preprocessor has several other useful facilities. If you are interested in these you can read more by typing

```
man cpp
```

January 1995

Next Up Previous Contents
**Next:** Advantages of Using **Up:** C Programming **Previous:** Using #ifdef to

# Programs with Several Files

When writing a large program, you may find it convenient to split it several source files. This has several advantages, but makes compilation more complicated.

This section will discuss advantages and disadvantages of using several files in a program, and advise you on how to divide a program between several files, should you wish to do so.



January 1995

Next Up Previous Contents
**Next:** How to Divide **Up:** Programs with Several **Previous:** Programs with Several

# Advantages of Using Several Files

The main advantages of spreading a program across several files are:

- Teams of programmers can work on programs. Each programmer works on a different file.
- An object oriented style can be used. Each file defines a particular type of object as a datatype and operations on that object as functions. The implementation of the object can be kept private from the rest of the program. This makes for well structured programs which are easy to maintain.
- Files can contain all functions from a related group. For Example all matrix operations. These can then be accessed like a function library.
- Well implemented objects or function definitions can be re-used in other programs, reducing development time.
- In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
- When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multi-file programs in this way.

January 1995

Next Up Previous Contents
**Next:** Organisation of Data **Up:** Programs with Several **Previous:** Advantages of Using

# How to Divide a Program between Several Files

Where a function is spread over several files, each file will contain one or more functions. One file will include main while the others will contain functions which are called by others. These other files can be treated as a library of functions.

Programmers usually start designing a program by dividing the problem into easily managed sections. Each of these sections might be implemented as one or more functions. All functions from each section will usually live in a single file.

Where objects are implemented as data structures, it is usual to to keep all functions which access that object in the same file. The advantages of this are;

- The object can easily be re-used in other programs.
- All related functions are stored together.
- Later changes to the object require only one file to be modified.

Where the file contains the definition of an object, or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly.

The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file.

Whenever a function in another file calls a function from our C file, it can define the function by making a #include of the appropriate .h file.

---

January 1995

Next Up Previous Contents
**Next:** Compiling Multi-File Programs **Up:** Programs with Several **Previous:** How to Divide

# Organisation of Data in each File

Any file must have its data organised in a certain order. This will typically be:

1. A preamble consisting of #defined constants, #included header files and typedefs of important datatypes.
2. Declaration of global and external variables. Global variables may also be initialised here.
3. One or more functions.

The order of items is important, since every object must be defined before it can be used. Functions which return values must be defined before they are called. This definition might be one of the following:

- Where the function is defined and called in the same file, a full declaration of the function can be placed ahead of any call to the function.
- If the function is called from a file where it is not defined, a prototype should appear before the call to the function.

A function defined as

```
float find_max(float a, float b, float c)
{  /* etc ... ... */
```

would have a prototype of

```
float find_max(float a, float b, float c);
```

The prototype may occur among the global variables at the start of the source file. Alternatively it may be declared in a header file which is read in using a #include.

It is important to remember that all C objects should be declared before use.

January 1995

Next Up Previous Contents
**Next:** Separate Compilation **Up:** Programs with Several **Previous:** Organisation of Data

# Compiling Multi-File Programs

This process is rather more involved than compiling a single file program. Imagine a program in three files prog.c, containing main(), func1.c and func2.c. The simplest method of compilation (to produce a runnable file called a.out) would be

```
cc prog.c func1.c func2.c
```

If we wanted to call the runnable file prog we would have to type

```
cc prog.c func1.c func2.c -o prog
```

In these examples, each of the .c files is compiled, and then they are automatically linked together using a program called the loader ld.



January 1995

Next Up Previous Contents
**Next:** Using make with **Up:** Compiling Multi-File Programs **Previous:** Compiling Multi-File Programs

## Separate Compilation

We can also compile each C file separately using the cc -c option. This produces an object file with the same name, but ending in .o. After this, the object files are linked using the linker. This would require the four following commands for our current example.

```
cc -c prog.c
cc -c func1.c
cc -c func2.c
ld prog.o func1.o func2.o -o prog
```

Each file is compiled before the object files are linked to give a runnable file.

The advantage of separate compilation is that only files which have been edited since the last compilation need to be re-compiled when re-building the program. For very large programs this can save a lot of time.

The make utility is designed to automate this re-building process. It checks the times of modification of files, and selectively re-compiles as required. It is an excellent tool for maintaining multi-file programs. Its use is recommended when building multi-file programs.

January 1995

Next Up Previous Contents
**Next:** UNIX Library Functions **Up:** Programs with Several **Previous:** Separate Compilation

# Using make with Multi-File Programs

We have already used make to build single file programs. It was really designed to help build large multi-file programs. Its use will be described here.

Make knows about `dependencies' in program building. For example;

- We can get prog.o by running cc -c prog.c.
- This need only be done if prog.c changed more recently than prog.o.

make is usually used with a configuration file called Makefile which describes the structure of the program. This includes the name of the runnable file, and the object files to be linked to create it. Here is a sample Makefile for our current example

```
#  Sample Makefile for prog
#
# prog is built from prog.c func1.c func2.c
#

# Object files (Ending in .o,
# these are compiled from .c files by make)
OBJS       = prog.o func1.o func2.o

# Prog is generated from the object files
prog: $(OBJS)
        $(CC) $(CFLAGS) -o prog $(OBJS)
# ^^^ This space must be a TAB.
# Above line is an instruction to link object files
```

This looks cluttered, but ignore the comments (lines starting with #) andthere are just 3 lines.

When make is run, Makefile is searched for a list of dependencies. The compiler is involved to create .o files where needed. The link statement is then used to create the runnable file.

make re-builds the whole program with a minimum of re-compilation, and ensures that all parts of the

program are up to date. It has many other features, some of which are very complicated.

For a full description of all of these features, look at the manual page for make by typing

```
man make
```

---

January 1995

Next Up Previous Contents
**Next:** Finding Information about **Up:** C Programming **Previous:** Using make with

# UNIX Library Functions

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

Wise programmers will check whether a library function is available to perform a task before writing their own version. This will reduce program development time. The library functions have been tested, so they are more likely to be correct than any function which the programmer might write. This will save time when debugging the program.



January 1995

Next Up Previous Contents
**Next:** Use of Library **Up:** UNIX Library Functions **Previous:** UNIX Library Functions

# Finding Information about Library Functions

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

```
man 3 strcat
```

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

```
man 3 intro
```

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

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

```
#include <time.h>

char *ctime(time_t *clock)
```

This means that you must have

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

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

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

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

Further related reading is suggested in the SEE ALSO section.

---

January 1995

Next Up Previous Contents
**Next:** Some Useful Library **Up:** UNIX Library Functions **Previous:** Finding Information about

# Use of Library Functions

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

It is important to ensure that your arguments have the expected types, otherwise the function will probably produce strange results. lint is quite good at checking such things.

Some libraries require extra options before the compiler can support their use. For example, to compile a program including functions from the math.h library the command might be

```
cc mathprog.c -o mathprog -lm
```

The final -lm is an instruction to link the maths library with the program. The manual page for each function will usually inform you if any special compiler flags are required.

---

January 1995

Next Up Previous Contents
**Next:** Precedence of C **Up:** UNIX Library Functions **Previous:** Use of Library

# Some Useful Library Functions

The following functions may be useful to you. Each manual page typically describes several functions, so if you see something similar to what you want, try looking in that manual page.

| Function | Description |
|---|---|
| abs | integer absolute value |
| ctime | convert date and time |
| fopen | open a stream |
| printf | formatted output |
| fputc | put character or word on a stream |
| getwd | get current working directory path name |
| strcat | string concatenation |
| ctype | character classification and conversion macros and functions |
| mktemp | make a unique file name |
| puts | put a string on a stream |
| sleep | suspend execution for interval |
| stdio | standard buffered input/output package |

To get a full summary type

```
man 3 intro
```

at your terminal.

January 1995

Next Up Previous Contents
**Next:** Special Characters **Up:** C Programming **Previous:** Some Useful Library

# Precedence of C operators

The following table shows the precedence of operators in C. Where a statement involves the use of several operators, those with the lowest number in the table will be applied first.

| | Description | Represented By |
|---|---|---|
| 1 | Parenthesis | () [] |
| 1 | Structure Access | . -> |
| 2 | Unary | ! ~ ++ -- - * & |
| 3 | Mutiply, Divide, Modulus | * / % |
| 4 | Add, Subtract | + - |
| 5 | Shift Right, Left | >> << |
| 6 | Greater, Less Than, etc | > < = |
| 7 | Equal, Not Equal | == != |
| 8 | Bitwise AND | & |
| 9 | Bitwise Exclusive OR | ^ |
| 10 | Bitwise OR | \| |
| 11 | Logical AND | && |
| 12 | Logical OR | \|\| |
| 13 | Conditional Expression | ?: |
| 14 | Assignment | = += -= etc |
| 15 | Comma | , |

Some of these operators have not been described in this course, consult a textbook if you want details about them.

January 1995

Next Up Previous Contents
**Next:** Formatted Input and **Up:** C Programming **Previous:** Precedence of C

# Special Characters

The following special patterns are used to represent a single character in C programs. The leading backslash in the single quotes indicates that more information is to follow.

| C code | Meaning |
|---|---|
| '\014' | Bit pattern for Form Feed |
| '\n' | Newline |
| '\t' | Tab |
| '\\ ' | Backslash |
| '\'' | Single Quote |
| '\"' | Double Quote |
| '\b' | Backspace |
| '\r' | Carriage Return |
| '\f' | Form Feed |
| '\0' | NULL (String Terminator) |

January 1995

Next Up Previous Contents
**Next:** Some Recommended Books **Up:** C Programming **Previous:** Special Characters

# Formatted Input and Output Function Types

The following format strings can be used to specify data to be printed or read using the printf or scanf functions, or any of their variants; fprintf, sprintf, fscanf or sscanf .

Note that though all can be used with the printf variants, some are not available for scanf.

| FORMAT STRING | DESCRIPTION | scanf() |
|---|---|---|
| | Numeric Values | |
| %d | Decimal integer | y |
| %f | Floating point decimal(Real number) | y |
| %lf | Double (Long Real number) | y |
| %e | Exponential representation | n |
| %u | Unsigned (positive) integer | n |
| %x | Hexadecimal integer | y |
| %o | Octal integer | y |
| %g | Automatically Selects Shorter of %f and %e | n |
| | Character Types | |
| %c | Single character | y |
| %s | Character String, Defined as `char *` | y |

*craa27@strath.ac.uk*
*Tue Jan 17 11:40:37 GMT 1995*

Next Up Previous Contents
**Next:** C Language Keywords **Up:** C Programming **Previous:** Formatted Input and

# Some Recommended Books

Apart from these excellent notes, there are many C books to choose from. Try to browse a few in your library or bookshop before buying one. Consider the following when selecting a book:

Price:
: Can you afford it?

Style:
: Is the book aimed at your standard of programming skill? Do you understand the points which the author is trying to make?

Accuracy:
: The book should support the ANSI C standard. Check whether it mentions ANSI, and also the date of the "last revision" (Computer technology changes rapidly, printed textbooks don't change at all).

These are two books which I have found very useful.

The C Programming Language
By: Brian W. Kernighan, Dennis M. Ritchie
Publishers: Prentice Hall

> This book, by the authors of C and the original UNIX system, gives a full and economical description of the language. It is not cluttered with trivial examples, so will not suit the novice programmer. This book is regarded by many as the `standard' work on the C language.

A Book on C
By: Al Kelley, Ira Pohl
Publishers: Benjamin Cummings

> This book gives a good description of C and the UNIX environment. Its style is more tutorial than Kernighan and Ritchie, so it would make a better purchase for the less experienced programmer. Newcomers to programming will find the plentiful examples helpful.

January 1995

Next Up Previous Contents
**Next:** Usable SUN Systems **Up:** C Programming **Previous:** Some Recommended Books

# C Language Keywords

The following names are reserved by the C language. Their meaning is already defined, and they cannot be re-defined to mean anything else.

| | | | | | |
|---|---|---|---|---|---|
| auto | break | case | char | continue | default |
| do | double | else | enum | exterm | float |
| for | goto | if | int | long | register |
| return | short | sizeof | static | struct | switch |
| typedef | union | unsigned | void | while | |

Other than these names, you can choose any names of reasonable length for variables, functions etc. The names must begin with a letter or underscore (letters are better), and then can contain letters, numbers and underscores.

January 1995

Next Up Previous Contents
**Next:** About this document **Up:** C Programming **Previous:** C Language Keywords

# Usable SUN Systems

This section is no longer relevant.

January 1995

*About this document ...*

---

Next Up Previous Contents
**Up:** C Programming **Previous:** Usable SUN Systems

---

# *About this document ...*

This document was originally generated using the **LaTeX**2HTML translator Version 0.5.3 (Wed Jan 26 1994) Copyright © 1993, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:
**latex2html** `ccourse.tex`.

HTML documentation modified and tweaked to correct errors and tidy up presentation, IT Services, 2001.

---

January 1995

About IT Services
Service News
Services
Teaching Rooms
Help Desk
Quick Find
University Home
IT Services Home

# Information Technology Services

## Quick Find

### Find on the IT Services pages

Enter search term:

Find Clear

### Select a topic

**Official Information**

Policies & Procedures

Choose a topic and then: Find

**Useful Information**

Helpdesk

Choose a topic and then: Find

### Other sources of information

- Mailing lists: dialup-users, resnet



*Last modified: Oct99*