An Introduction To C

by

Ged Firth

Section 1 Introduction

1.1 Where Did C Come From ?

C evolved from CPL via B & BCPL. CPL (Combined Programming Language) was created by a joint committee from the universities of London and Cambridge in 1963 who required a language that plugged the hole left by high-level languages that lacked practicality and low-level languages that lacked structure. In 1967, Martin Richards of Cambridge University developed BCPL (Basic CPL) as CPL was considered too large for most applications. CPL was still considered too large by Ken Thompson of Bell Laboratories in 1970 so he created B. In 1972 Dennis Ritchie (also at Bell Labs) improved B and re-wrote most of the UNIX operating system in a new language called C.

1.2 Advantages Of C

C is a relatively small language containing only some 30 or so key-words and combines the benefit of a high-level language with the low-level features of an assembly language. Although intended as a systems programming language, C has proved to be a good all-round general-purpose language. C supports the facilities expected in a modern imperative language such as structured control statements, recursion, records, dynamic data structures etc. as well as low-level bit-wise operations such as AND, OR, XOR, shifts etc.

1.3 Disadvantages Of C

Although an ANSI standard for C was drafted in the late 1980's, some earlier versions are still in common use which can cause problems with portability.

The flexibility of C combined with the wide range of operators means that it is possible to write compact and efficient code which can be far from readable to other than the author. Further to this, the fact that C gives access to low-level facilities means that errors can often result in some strange behaviour such as re-booting a PC.

Section 2 Getting Started

2.1 Fundamentals Of C

All C programs can be seen as a group of building blocks called functions. A function is a group of one or more C statements (instructions) that performs a specific task. Your C compiler comes complete with a host of pre-written functions held in libraries for your use. If your libraries don't have a function that suits your particular needs, then of course you can write your own, and if you want to save it for future use, you can.

As you can see, after a while you'll have built up a library of your own common tools such as menu display and selection functions that can be called as you wish so you won't have to 're-invent the wheel' every time you want such a routine.

Perhaps the best way of learning a language is to get your sleeves rolled up and practice it, so let's start by writing our first C program.

2.2 Simple Output

```
/* program to display a simple message on screen */
#include <stdio.h>
void main()
{
   printf("This is my 1st C program\n");
}
```

Let's look at the code in detail to see what's happening.

/* program to display a simple message on screen */

This line is a comment. In C, comments (or REMarks to Basic programmers) are inserted in a program by using the characters **/*** to start and ***/** to end, with everything in between (including new lines) ignored by the compiler.

#include <stdio.h>

This line tells the compiler to **include** the contents of the library file **stdio.h**, which contains the definitions of the standard I/O functions such as **printf**.

void main()

As the name implies, this is the **main** function in the program which tells the compiler where to start ({) and stop (}) execution and as such, must appear in every C program. The **void** states that the function does not return a value (more on this later).

```
{
```

The opening brace signifies the start of the program.

```
printf("This is my 1st C program\n");
```

This line calls the function **printf** (print formatted) with the text to be output passed as an argument within the parenthesis. The example shows a string of characters enclosed in double quotes. The **\n** is a newline character which will be explained later along with a more in-depth look at printf. Note that a C statement is terminated by a semi-colon (**;**).

```
}
```

This brace matches the opening brace of main() and therefore marks the end of the program.

2.3 Variables & printf

Consider a variation of our first C program

```
/* example program to introduce variables */
#include <stdio.h>
void main()
{
   int num;
   num = 2;
   printf("This is C program No. %d\n", num);
}
```

Let's now look at the key points of the program.

int num;

A variable of type int is declared. There are three basic types of variable in C which are **int** (integer), **float** (floating-point or real number) and **char** (character). Before a variable can be used, it must be declared.

num = 2;

The variable **num** is assigned a value. C allows you to declare and initialise a variable at the same time therefore the first two lines of the program could have been the single line **int num = 2;**.

printf("This is C program No. %d\n", num);

The printf function now contains two arguments separated by a comma. The first argument to printf is the format string and is always enclosed within double quotes. As a variable is to be printed within the text, printf must know which variable and how it is to be displayed. How it is displayed is determined by the conversion specification (**%d**) which in this case informs printf that the data is to be displayed in decimal form. Which variable is to be displayed is determined by the second argument to printf (num). If more than one variable is to be printed, then the arguments that follow the format string match the conversion specifications in order from left to right.

When a backslash character (\) is encountered by the compiler, it signifies the start of an 'escape sequence' which reads the next character and substitutes the appropriate numeric value for that character e.g. on a PC system, **\t** becomes decimal 9 which is the ASCII value for the **tab**. The most common escape sequences used in C are :

\n	newline (linefeed)
\t	tab

\b backspace
\a bell (alarm)

You may be wondering what all the fuss is about in displaying an integer as a decimal, well we may have wanted to display the integer in hex (%x) or octal (%o) which is often the case in computing.

2.4 Simple Input

Let's now look at how we get input from the keyboard and at the same time see how printf handles floats.

```
/* program to accept 2 numbers from the user, divide
   the first by the second and display the result */
#include <stdio.h>
void main()
{
   float num1, num2, result;
   printf("Enter 1st Number : ");
   scanf("%f", &num1);
   printf("Enter 2nd Number : ");
   scanf("%f", &num2);
   result = num1 / num2;
   printf("%.2f divided by %.2f = %.2f\n", num1, num2, result);
}
```

The key lines of code in this program are :

float num1, num2, result;

More than one variable may be declared on the same line.

scanf("%f", &num1);

This statement calls the **scanf** function to read the input into the variable **num1**. The program will halt until a value has been entered by the user. The format string **"%f"** informs scanf that the incoming value is a float, the second argument is the variable into which the number is to be stored. Note that the variable name is preceded by an ampersand (**&**) character which gives scanf the **address** of the variable. The technicalities behind this will be covered later in the course but for now accept that when reading numbers (or single characters) with scanf, the ampersand should be used, but not when accessing the contents of variables.

scanf("%f", &num2);

The value of num2 is read from the keyboard.

result = num1 / num2;

The variable result is assigned the value of num1 divided by num2.

printf("%.2f divided by %.2f = %.2f\n", num1, num2, result);

This statement uses printf to output the result in a user friendly manner. The conversion specification **%.2f** outputs a float to 2 decimal places.

Some examples of conversion specifications :

%d	print an integer, no field width specified
%5d	print an integer in a 5 character field
%f	print a float, no format or field width specified
%6.2f	print a float in a 6 character field, output to two decimal places (rounding as appropriate)

Note that if a specified field width is too small for the required data, then printf will print the full value.

2.5 Simple Arithmetic

Let's start by looking at the binary arithmetic operators and their meanings.

+	add
-	subtract
*	multiply
/	divide
%	modulus (remainder of integer division)

Although perhaps straightforward, you must remember that integers deal with whole numbers and that integer division will result in any fractional part being lost i.e.

5 / 2 = 2

12 / 25 = 0.

The **+** and **-** operators have equal precedence which is lower than *, **/** and **%** which have equal precedence. In the absence of precedence, statements are evaluated from left to right. Parenthesis may be used to alter the order of evaluation and should be used if you are unsure. Let's look at some examples.

```
x = 9 / 5              x evaluates to 1
x = 9.0 / 5.0          x evaluates to 1.8
x = 12 / 2 * 4         x evaluates to 24
x = 12 / (2 * 4)       x evaluates to 1
x = 9 - 5 + 3          x evaluates to 7
x = 9 - (5 + 3)        x evaluates to 1
x = 2 + 3 * 4          x evaluates to 14
x = (2 + 3) * 4        x evaluates to 20
```

```
#include <stdio.h>   /* prog to add VAT to cost price and output sale price */
void main()
{
   float cost, vat, sell;
   printf("Enter Cost Price : ");
   scanf("%f", &cost);
   vat = 15.0 / 100.0 * cost;
   sell = cost + vat;
   printf("Cost          - %7.2f\n", cost);
   printf("VAT           - %7.2f\n", vat);
   printf("Sale Price    - %7.2f\n\n", sell);
}
```

The key statements in the program are :

vat = 15.0 / 100.0 * cost;

Had the statement been - **vat = 15 / 100 * cost;,** then integer division of 15 / 100 would have produced zero, which of course was not intended. Note that unlike some languages, variables or constants of different types may be mixed in C and in such expressions automatic type conversion takes place. More on this later in the course, but for now you should take care and be aware of the problem.

Exercises

2.1 Produce a program to calculate the m.p.g. of a vehicle. The miles covered and the amount of fuel used to be entered by the user into suitable variables.

2.2 Produce a program that takes a temperature entered in degrees Fahrenheit by the user and displays the equivalent temperature in degrees Celsius. Use the formula deg C = (5/9) * (deg F-32).

Produce a similar program to convert from Celsius to Fahrenheit.

2.3 Given the radius of a circle by the user, produce a program that will give the following (formulae in brackets) :

The circumference of the circle ($2\pi r$)
The area of the circle (πr^2)
The volume of the sphere ($^4/_3\pi r^3$)
The surface area of a sphere ($4\pi r^2$)

2.4 Write a program which accepts the lengths of the adjacent and opposite sides of a right-angled triangle from a user and displays the length of the hypotenuse. (The square of the hypotenuse is equal to the sum of the squares of the other two sides) Look in the help index for a function to find the square root of a number.

2.5 There are 14 pounds in a stone and 16 ounces in a pound. Write a program to accept a weight in ounces and convert it to stones, pounds and ounces.

Produce a second program to convert from stones pounds and ounces to ounces.

2.6 There used to be 12 pennies in shilling and 20 shillings in a pound. Write a program to accept a value in old pence and convert it to pounds, shillings and pence.

Modify the program to further show the value in today's currency (pounds and pence).

Produce a second program to convert from pounds, shillings and pence to today's currency.

Section 3 Characters & Strings

3.1 Characters

Declaring a character variable is similar to numeric types i.e.

```
char letter;
```

this reserves one byte of memory for the character 'letter'.

Characters may be processed similarly to numeric types by using their number in the character set or by writing a character constant within single quotes i.e. in the ASCII character set, the letter '**A**' has the value 65, therefore these statements are similar.

```
letter = 65;
letter = 'A';
```

The input and output of single characters is supported by the functions **getchar**, **getche**, and **putchar** with **scanf** and **printf** available with the **%c** format. **getchar** returns the next character from standard input i.e.

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

getche (and getch – try it to see the difference) reads a character from the keyboard when pressed. getchar reads a character from 'standard input' which is normally the keyboard, but this can be redefined.

```
int c;
c = getchar();
```

Note that when reading an indeterminate number of characters either from keyboard or file, the variable must be large enough to hold the end of file marker (EOF, ^D, ^Z) in order to recognise the end of input. The receiving variable therefore will have to be declared as an integer. This makes no difference to normal character processing as the incoming character is held in the lower order byte of the integer and is treated as a character variable.

The function **putchar** outputs a single character to the standard output i.e.

```
putchar(c);
putchar(letter);
putchar(65);  /* but take care */
putchar('A');
```

3.2 Strings

There is no built-in data type for strings in C. Strings are of course, a number of characters held together under one identity in a data structure called an **array**.

To declare a string in C we use an array of characters i.e.

```
char name[10];
```

this reserves space for a 10 character string called **name**. C does not check the boundary of the array (this is your responsibility as the programmer) instead it uses a marker to determine the end of the string. This marker is called **NULL** which is a byte with all bits set to zero and should be considered when determining the required length of a string i.e. ensure the size of the declared string is one character more than the number required. The characters in the string can be referenced as name[0] ... name[9]. The number enclosed within the brackets is known as a subscript. Subscripts start at zero and are integers, integer variables or integer expressions i.e.

```
name[9] = name[pos];
name[pos-1] = name[0];
```

Elements within a string can be processed the same as characters i.e.

```
name[0] = 'A';
letter  =  name[pos];
putchar(name[pos - offset];
```

As the string boundary is not checked, the statement :

```
name[50] = 'x';
```

is quite legal and would put 'x' in a part of memory not declared for 'name'.

3.3 Input & Output Of Strings

The scanf and printf functions support string I/O i.e.

```
#include <stdio.h>
char prompt[] = "What is your name ? : ";
void main()
{
   char name[21];
   printf("\n%s", prompt);
   scanf("%s", &name[0]);
   printf("Hello %s, Nice To Meet You\n", &name[0]);
}
```

The key points of the program are :

char prompt[] = "What is your name ? : ";

This declaration also initialises the string, the size of the array is determined by the number of elements in the initialisation string plus the NULL character (i.e. 23).

Note that the position of the declaration outside of main is purely arbitrary. In ANSI C, the declaration may be within functions but pre-ANSI compilers can 'complain'.

scanf("%s", &name[0]);

The scanf function reads strings using the **%s** format specification. The **address** of the 1st element of the array is given as the starting location in which to store the string. The input could have been read into any position i.e.

scanf("%s", &name[3]);

This would read the input starting at the 4th element of the array, name[0] ... name[2] would remain unchanged. The scanf function automatically terminates the entered string with the NULL character when a **white space** (carriage return, space or newline) is reached and is therefore unsuitable for reading sentences.

printf("Hello %s, Nice To Meet You\n", &name[0]);

Like scanf, the printf function uses the **%s** format specification.

Note that unlike numeric and character variables, a copy of the data (the string) is not passed as an argument to printf, instead the address of the memory location is passed for printf to access the data directly.

The **identifier** of an array alone (i.e. its name without the ampersand and subscript) is in fact the address of the leading memory location of the array. As most string processing involves the full array contents, it is the norm to see strings referenced this way as in line 6 of the program i.e.

printf("\n%s", prompt);

Likewise lines 7 & 8 could have been written :

scanf("%s", name);
printf("Hello %s, Nice To Meet You\n", name);

Another function for reading strings from the keyboard is **gets**. This function is perhaps more popular with C programmers as it is simpler to use i.e.

gets(name);

compared with the scanf equivalent :

```
scanf("%s", name);
```

An option to using **printf** to output a string is **puts**. This function simply outputs each character within the string until the NULL is reached i.e.

```
puts(name);
```

3.4 Beware of 'Hanging New Lines'

When using **scanf** to read input, the newline that is entered on input is left hanging in the input stream. The next time you read a character, this 'hanging new line' may be presented as input (which you probably didn't want).

There are two commonly used methods of dealing with this problem :

```
scanf("%d", &value);
fflush(stdin);
```

This use of the **fflush** function empties the standard input buffer.

```
getchar();
```

The use of **getchar** with or without assigning it to a variable will also discard a hanging newline. Note that if there is no character to read then the program will halt until a character is entered at the keyboard.

3.5 String Handling Routines

Functions to process strings are defined in the header file <string.h>. Some commonly used string functions are :

strcpy(s1, s2)	copies s2 to s1
strncpy(s1, s2, n)	copies n characters from s2 to s1 (and no more)
strcat(s1, s2)	appends s2 to s1
strlen(s)	returns the length of s (excluding NULL)

To familiarise yourself with strings and these functions, compile & run the following example program :

```
/* a meaningless program that uses string handling functions */
#include <stdio.h>
#include <string.h>
#define  NAMELEN  30        /* constant definition */
void main()
{
   char str1[NAMELEN], str2[NAMELEN], str3[NAMELEN];
   int n;
   printf("Enter String 1 : ");
   gets(str1);
   printf("Enter String 2 : ");
   gets(str2);
   n = strlen(str1);
   printf("\nStr1 is %d chars long\n", n);
   n = strlen(str2);
   printf("\nStr2 is %d chars long\n", n);
   strcpy(str3, str1);
   strcat(str3, str2);
   n = strlen(str3);
   printf("Str3 = %s and is %d chars long\n", str3, n);
}
```

Exercises

3.1 Write a program to read a string **s** and an integer **n** from the keyboard and output the character at the **n**th position of **s**.

i.e. the string "programmer" and the integer 4 entered, output 'g'.

3.2 Modify the above program to output the **n**th character from the end of **s**.

3.3 Re-write the program to output the first, last and middle characters of **s**. (either middle character if an even no.)

3.4 Write a program to read a string **s** from the keyboard, split **s** into two halves and output each half separately.

3.5 Modify the above program to re-join the two halves in reverse order from the original string in a different variable.

i.e. input halftime
output timehalf

3.6 Write a program to read the name of a user (firstname & surname) and output their initials.

Section 4 Iteration

Loops allow us to repeat a section of code a number of times. C supports three types of loop : **while**, **do - while** and **for**.

4.1 while

The format of a while loop is

```
while (condition)
   statement;
```

The program will repeat the statement until the condition becomes false. The condition is tested before the first execution of the statement therefore if false at the start of the loop, the statement will not be executed at all. If more than one statement is required then braces should be used to group the statements into one block.

```
#include <stdio.h>      /* simple  check-out program to add up a number */
void main()                     /* of prices and output the total */
{
   float item, total = 0;
   printf("Item Price : ");
   scanf("%f", &item);
   while ( item > 0 )
   {
      total = total + item;
      printf("Item Price : ");
      scanf("%f", &item);
   }
   printf("Total = %.2f", total);
}
```

The while loop tests the condition on each iteration and continues until the condition becomes false. A conditional expression returns a value which equates to false if the value is zero or true otherwise. The relational operators are :

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

4.2 do - while

The format for a do - while loop is

```
do
{
   statement(s);
} while (condition);
```

The do - while loop is similar to the while loop except that the condition is tested at the bottom of the loop ensuring that the statement is executed at least once. Although the position of the closing brace is arbitrary, it is felt that readability is enhanced by keeping it on the same line as the while statement.

```
/* example program to print the numeric (ascii)  value of a character */
#include <stdio.h>
void main()
{
   char inchar;
   do
   {
      printf("Enter Character : ");
      scanf("%c%*c", &inchar);          /* get char and newline */
      printf("%c = %d\n", inchar, inchar);
   } while ( inchar != 'q' );           /* 'q' to quit */
}
```

The difference in the **while** and the **do - while** loops is that the **while** loop is a pre-test loop and the **do - while** a post-test loop. Some programmers often use a while loop instead of a do - while loop by simply making the condition true before the start of the loop. Which loop you select is of course your preference as the programmer. As an exercise, re-write the two example programs in the other loop format and see which you prefer.

4.3 for

You normally use a **for** loop when you know how many times you require the statement(s) to be repeated i.e.

```
int count;
for (count = 0; count < 10; count = count + 1)
{
   statement(s);
}
```

A for loop has three arguments separated by semi-colons, which using the example are:

count = 0

The first argument is a statement that is done once at the beginning of the loop, usually, as in this case to initialise the control variable.

count < 10

The second argument is a conditional expression which is evaluated before each iteration. When the expression is false, the loop terminates.

count = count + 1

The third argument is a statement that is done at the end of each iteration. In this case the variable **count** is incremented.

The example could have been written using a while loop:

```
count = 0;
while (count < 10)
{
   statement 1;
   statement 2;
      "   "
   statement n;
   count = count + 1;
}
```

Example

```
/* program to print the integers from 1 to 100 */
#include <stdio.h>
#define  LIMIT  100
void main()
{
   int count;
   for (count=1; count <= LIMIT; count = count + 1)
      printf("%d\n", count);
}
```

The statement **#define LIMIT 100** is the method of defining a constant in C. This is a pre-processor directive (it starts with a '#' and does not terminate with a semi-colon) which substitutes all occurrences of 'LIMIT' with the value 100. It is good programming practice to avoid burying literal values within a program. Although the example perhaps doesn't warrant a constant, the advantages become more

apparent as your programs become larger. Consider an accounting program with several references to VAT at 17.5 %, when the rate changes, the program can be updated simply if the programmer has used a constant for the VAT rate, otherwise each occurrence of 17.5 will need to be found and altered.

4.4 Incrementing & Decrementing

Adding or subtracting values from a variable is so common, especially when using loops, that C offers shortened alternatives i.e.

count = count + 1;

could equally have been written

count++;

Similarly when decrementing :

count--;

In the examples given (count++ & count--), the **++** and **--** are implemented as post-increment/decrement operators, that is, they are incremented/decremented **after** the variable has been used. When used as pre-increment/decrement operators, the increment/decrement operation occurs **before** the variable is used i.e.

if the variable **y** has a value of 99:

x = y++;	x = 99, y = 100
x = ++y;	x = 100, y = 100

Further to the increment & decrement operators, C offers a range of assignment operators i.e.

Operator	Example	equivalent
+=	x += y;	x = x + y;
-=	x -= y;	x = x - y;
*=	x *= y;	x = x * y;
/=	x /= y;	x = x / y;
%=	x %= y;	x = x % y;

Exercises

4.1 Write a program that counts the number of characters in a string entered by a user without using the strlen function.

4.2 Write a program to give the factorial of a number entered by the user. e.g. factorial of 6 = 6 * 5 * 4 * 3 * 2 * 1 = 720.

4.3 Write a program that displays the 'multiplication table' of a number entered by the user. The table should range from 1 to 12.

4.4 Write a program to read a series of integers (in the range 32 - 127) from the user and output the character represented by the number. The program should terminate on an invalid entry.

4.5 Write a program to accept the name of a user and output their initials.
i.e.

Ged Firth	GF
Albert Ian Tatlock	AIT

4.6 Write a program to read a sentence (up to 80 characters) and output it to the screen in reverse. Use a loop to copy each character from the ‘in string’ to the ‘out string’.

Section 5 Arrays

Earlier we looked at strings, which are a number of characters held in a data structure called an array. An array can be defined as a collection of variables of the same **type** held together in a linear sequence in memory.

5.1 One-dimensional Arrays

A one-dimensional array has the form :

```
array_type  array_name[array_size]
```

where

array_type is any valid type supported by C,
array_name is the identifier given by the programmer
array_size is a positive integer of the no. of array elements

for example the declaration

```
int score[6];
```

declares an array of 6 integers named 'score'. Like strings (character arrays), other arrays may be initialised during declaration :

```
int score[] = { 1, 2, 1, 2, 0, 6 };
```

Remembering that arrays in C start from zero, the above declaration has assigned the integer array as follows :

score[0] has a value 1
score[1] " " " 2
score[2] " " " 1
score[3] " " " 2
score[4] " " " 0
score[5] " " " 6

The elements within the array are processed directly by giving the array name with the element position (subscript) in brackets i.e.

```
goals_for[count] += score[count];

score[x] = score[y];

score[x-1] = score[y+1];        etc.
```

Example program

```
/* declare an array of ints (team scores), initialise it,
   read in values (goals) & output total goals scored */
#include <stdio.h>
#define  NO_OF_TEAMS   10
void main()
{
   int score[NO_OF_TEAMS];
   int count, sum = 0;
   for (count = 0; count < NO_OF_TEAMS; count++)
   {
      printf("Enter Score %d :- ", count+1);    /* 1 - 10 not 0 - 9 */
      scanf("%d", &score[count]);
   }
   for (count = 0; count < NO_OF_TEAMS; count++)
      sum+=score[count];
   printf("Total Goals Scored = %d\n", sum);
}
```

5.1 Multi-dimensional Arrays

The simplest form of a multi-dimensional array is a two-dimensional array. This is in effect a one-dimensional array, whose elements are also a one-dimensional array.

A two-dimensional array can be viewed as rows and columns i.e.

```
int matrix[3][4] = {1, 2, 3, 4,  /* note the layout to  */
                    2, 4, 6, 8,   /* improve readability */
                    1, 3, 5, 7 };
```

This declares a two dimensional array - 3 rows of 4 elements.

To process the above two-dimensional array, it is common to use **nested** loops, for example to multiply each element by two :

```
int row, col;
for (row = 0; row < 3; row++)
   for (col = 0; col < 4; col++)
      matrix[row][col] *= 2;
```

Exercises

5.1 Write a program that will read in a string (character array) from the user and store each character of the initial string in reverse order in a second array.

i.e. input - “hello Mum”
output - “muM olleh”

Print out each string to prove your program works.

5.2 Write a program in which a one-dimensional array of 6 floats is declared. Read a value entered by the user into each element, then calculate and display the average value (total / 6).

Modify the program to process an indeterminate number of values i.e terminate on zero input.

5.3 Write a program that declares a data table in the form of a two-dimensional array of ints (12 x 12), then by using nested loops : record a matrix of the multiplication table data for each combination i.e.

3 x 4 = 12
4 x 3 = 12

1	2	3	4	etc.
2	4	6	8	
3	6	9	**12**	
4	8	**12**	16	
etc.				

Having stored the table in memory, produce a routine to accept 2 integers from the user and reference the data table and display the result.

This routine should repeat until the first integer entered by a user is 0.

Section 6 Selection

C supports selection with the if, if - else and case constructs.

6.1 if

The **if** statement evaluates a conditional expression (remember conditional expressions in section 4) and only if this condition is true will the next statement(s) be executed. The format of an if statement is:

```
if (condition)
   statement;
```

example

```
/* program to notify user of a negative input */
#include <stdio.h>
void main()
{
   int num;
   printf("Enter Number : ");
   scanf("%d", &num);
   if ( num < 0 )
      printf("Negative\n");
}
```

Where multiple statements are made, braces should be used to **block** the statements together.

6.2 if - else

Where an alternative statement is required should an **if** statement be false, then the if - else construct is used which has the following format:

```
if (condition)
   statement1;
else
   statement2;
```

example

```
if ( num == 0 )
   printf("Zero Entered\n");
else
   printf("Value = %d\n", num);
```

6.3 else if

Although not considered a construct in the true sense, **else if** is worthy of note due to its common usage in C. This is used where a number of tests are made that are associated with the same **if** statement i.e.

```
if (condition)
   statement1;
else if (condition)
   statement2;
else if (condition)
   statement3;
else
   statement4;
```

Traditionalists would write this :

```
if (condition)
   statement1;
else
   if (condition)
      statement2;
   else
      if (condition)
         statement3;
      else
         statement4;
```

As previously stated, layout in C is arbitrary as white spaces (tabs, newlines & spaces) are ignored by the compiler. This enables the **else** and the **if** to be separated only by the minimum one space.

example

```
/* prog to respond to a selection */
#include <stdio.h>
int choice;
void main()
{
   printf("What Do You Want For Breakfast ? ");
   printf("\n\t1 .. Eggs\n");
   printf("\n\t2 .. Toast\n");
   printf("\n\t3 .. Corn Flakes\n");
   printf("\n\t4 .. Full English Breakfast\n");
   printf("\n\tEnter Selection -> ");
```

```
    scanf("%d", &choice);
    if ( choice == 1 )
       printf("Eggs ready in ten minutes\n");
    else if ( choice == 2 )
       printf("Toast ready in five minutes\n");
    else if ( choice == 3 )
       printf("Cornflakes ready when you are\n");
    else if ( choice == 4 )
       printf("You can make that yourself\n");
    else
       printf("Invalid Entry\a\n");
}
```

6.4 switch

The previous example shows that multi-way selection with **if** & **else if** can be tedious. The switch statement gives a more structured approach.

```
switch(choice)
{
   case 1 :
      printf("Eggs ready in ten minutes\n");
      break;
   case 2 :
      printf("Toast ready in five minutes\n");
      break;
   case 3 :
      printf("Cornflakes ready when you are\n");
      break;
   case 4 :
      printf("You can make that yourself\n");
      break;
   default :
      printf("Invalid Entry\a\n");
}
```

The keyword **switch** is followed by an integer expression in parenthesis which is the value to be compared to the value following the keyword **case**. When a case statement matches, then statements following that case will be executed. The statement(s) associated with the case should be terminated with the keyword **break**

otherwise subsequent case statements will be executed until a break is encountered or the end of the switch statement is reached. This allows more than one label to be attached to a statement i.e.

```
case 2 :
   printf("Toast ready in five minutes\n");
   break;
case 3 :
   printf("Cornflakes ready when you are\n");
   break;
case 4 :
case 1 :
   printf("You can make that yourself\n");
   break;
default :
   printf("Invalid Entry\a\n");
```

6.5 Tidying Up Loops & Selection

Before closing this section there are two things that I feel need covering to enable you to get the most from loops and selection.

6.5.1 Logical Operations

First a closer look at a simple conditional expression. Remember that if a conditional expression evaluates to zero, then that expression is considered **false**. Any non-zero value (positive or negative) is therefore **true**. As programmers, we are not necessarily concerned with the value produced when a test criterion is evaluated, rather our only concern is whether it is zero (logic false) or non-zero (logic true).

Consider the following section of code:

```
if ( x == 0 )
   statement1;
else
   statement2;
```

If the test criterion is logically true, a non-zero value is evaluated from the expression and statement1 is executed. If the test criterion is replaced with a simple value, then any non-zero value is logic true and zero is logic false. The routine could therefore be written as follows:

```
if(x)
   statement2;
else
   statement1;
```

This may be interpreted as, if x is true (non-zero).

So what's happening ? The contents of the parenthesis are evaluated, which in this case it is not a logical expression but a simple variable which is evaluated. This of course reverses the statements of the previous routine. To keep the routine in its original form, the condition should be negated i.e.

```
if(!x)
   statement1;
else
   statement2;
```

The '**!**' is the logical negation operator which negates the value of an expression.

More than one condition may be applied to an expression i.e.

```
if(x < 100 && x > 0)
   printf("Value Within Range\n");
```

The **&&** is the logical **AND** operator which makes the condition read:

x is less than 100 **AND** x is greater than zero

Both conditions must be true in order to return a true condition for the whole expression. If the first condition is false, then the second condition is not evaluated.

Where only one of a series of conditions needs to be true for the condition as a whole to be true, the **OR** operator is used:

```
if(x < 1 || x > 99)
   printf("Value Out Of Range\n");
```

The || is the logical **OR** operator which makes the condition read:

x is less than 1 **OR** x is greater than 99

Either condition must be true in order to return a true condition for the whole expression. If the first condition is true then the second condition is not evaluated.

6.5.2 break and continue

It is often convenient to exit a loop from within the loop body and not process the remainder of the statement block. It is also often required that the remainder of the statement block does not require execution and that the remaining statements should be skipped to the next iteration of the loop. C supports these two situations with the **break** and **continue** statements:

```
#include <stdio.h>      /* prog to total up a number of entries */
void main()                         /* ignoring negative values */
{
   int count, num, tot;
   count = tot = 0;  /* alternative way of initialisation */
   while(1)          /* endless loop */
   {
      printf("Enter Value : ");
      scanf("%d", &num);
      if (num == 0)
         break;
      if (num < 0)
         continue;
      count++;
      tot += num;
      printf("Running Total = %d\n", tot);
   }
   printf("\nNo. Of Valid Entries   = %d\n", count);
}
```

The break statement causes the innermost loop (or indeed switch) to be exited immediately. The continue statement restarts the loop sequence immediately following the final statement in the loop.

6.5.3 Nested if's

Because the else part of an if statement is optional, care is necessary when nesting i.e.

```
if(x > 0)
   if(x > y)
      a = b;
   else
      a = c;
```

An **else** in C is always associated with the closest else-less **if**, therefore the else is associated with if(x > y). Note that the bottom four lines of code do not need blocking within braces because the if - else is **one** statement. Consider the following:

```
if(x > 0)
   if(x > y)
      a = b;
else
   y = c;
```

It is clear by the indentation what the programmer wants, but as C does not consider program layout, the else is associated with the previous if. To force the correct association then braces should have been used to block the 2nd if statement i.e.

```
if(x > 0)
{
   if(x > y)
      a = b;
}
else
   y = c;
```

6.5.4 The Ternary Operator

An alternative to if - else is the ternary operator which has the format:

```
expression1 ? expression2 : expression3
```

The first expression (expression1) is evaluated first, **if** that expression is logic true, then expression2 is evaluated **else** expression3 is evaluated.

Consider this **if** statement

```
if(x > y)
   a = b;
else
   a = c;
```

this could be written

```
a = ( x > y ) ? b : c;
```

The statement reads : a is assigned the value of the ternary expression. The value of the ternary expression is either b or c, depending on the condition (x > y).

The above statements can make code almost unreadable to the novice. The inclusion of the ternary operator in this text is to ensure you to recognise it when you come across it in the future.

Exercises

6.1 Write a program that accepts 3 integers (in any order) from the user and outputs the highest and lowest values.

Produce a second program to output the 3 values in ascending order (not for the faint hearted).

6.2 Write a program that accepts an undefined number of integers (max 20) from the user and outputs the highest and lowest numbers entered. Entry is terminated by a zero being entered.

6.3 Write a program to read a sentence from a user and count the number of spaces entered.

6.4 Write a program that will continually read characters from the keyboard and display the numeric value (ASCII) of that character. Only display letters and numeric digits (beep when other characters are entered) and terminate the program when a 'Q' or 'q' is entered.

Section 7 Functions

Functions in C are the sub-programs, procedures and sub-routines of other high-level languages. Functions may accept parameters as arguments and may also return a value to the calling function.

The format of a function declaration is:

```
return_type  function_name (parameters)
{
   statements;
}
```

If the return type is absent then it is implicitly declared as an **int**.

7.1 Void function - no parameters

The simplest function is one where no value is returned and no parameters are passed i.e.

```
#include <stdio.h>      /* program to total up & output a no.*/
                        /* of ints entered by a user */
void display_prompt()
{
   printf("\n\n\t\tZero To Quit");
   printf("\n\n\t\tEnter Value : ");
}
void main()
{
   int num, total = 0;
   do
   {
      display_prompt();
      scanf("%d", &num);
      total += num;
   } while ( num );
   printf("Total = %d\n", total);
}
```

The function display_prompt returns no value to the calling function (main) therefore its return type is void. There are no values passed as parameters therefore the parenthesis are empty. Whenever the function is called in main, the statements within the function are performed.

Functions may be declared in any order with the exception of void functions like display_prompt which must have a function prototype if it is called before the function declaration (more on this later).

7.2 return value - no parameters

The above program could have used the function display_prompt to also get the input from the user and **return** this input to the calling function i.e.

```
#include <stdio.h>
int get_input()          /* display prompt & read input */
{
   int input;
   printf("\n\n\t\tZero To Quit");
   printf("\n\n\t\tEnter Value : ");
   scanf("%d", &input);
   return input;
}
void main()
{
   int num, total = 0;
   do
   {
      num = get_input();
      total += num;
   } while ( num );
   printf("Total = %d\n", total);
}
```

The function **get_input** reads the input from the user into the **local** variable **input**. The **value** of input is returned to the calling function by the statement **return input;**. Note that the variable **input** is only known within the function get_input because it is declared locally within the function, therefore the function main has no knowledge of or access to this variable. Likewise, get_input has no access to the variables **num** & **total** declared in main.

7.3 Passing Parameters

Suppose the above program required a count of the number of entries to be taken and a running total to be displayed. This gives an opportunity to demonstrate the passing of parameters.

```
#include <stdio.h>
int get_input()
{
   int input;
   printf("\n\n\t\tZero To Quit");
   printf("\n\n\t\tEnter Value : ");
   scanf("%d", &input);
   return input;
}
void tot_up(int entry, int tot)
{
   printf("\n\t\tEntry No. %d .. Total = %d\n", entry, tot);
}
void main()
{
   int num, count, total;
   total = count = 0;
   do
   {
      num = get_input();
      count++;
      total += num;
      tot_up(count, total);
   } while (num);
}
```

The function tot_up receives two arguments (parameters) from main which are the integer variables **count** and **total**. The types of these parameters must be declared in the function declaration. In C all parameters are **passed by value**, i.e. functions receive **COPIES** of the variables passed as parameters and **NOT** the variables themselves. This means that the **VALUES** of **count** & **total** are assigned to **entry** & **tot**. The variables **count** & **total** will be unaffected by any operations on **entry** & **tot** from within the function tot_up.

7.4 Passing Strings

Passing arrays as parameters is slightly different in that the declaration of the function parameter does not contain the size of the array being passed as strings are processed using addresses (pointers). A variation of the example program in section 3.3 could have been written as follows using a function:

```
#include <stdio.h>
void greet(char name[])
{
   printf("Hello %s, Nice To Meet You\n", name);
}
void main()
{
   char name[21];
   printf("\nEnter Your Name ");
   gets(name);
   greet(name);
}
```

Note that the variable **name** in main and the parameter **name** both share the same identifier. Although confusing and thought not to be good practice, it is acceptable because as neither function (**main** or **greet**) has access to variables in the other function as they are local and therefore accessible only to that function.

Exercises

7.1 Re-write exercise 2.2 so that **main** contains only one variable declaration and two calls to functions i.e.

```
float degf;
degf = read_fahr();
display_cent(degf);
```

7.2 Re-write exercise 2.3 using functions to:

read the radius from the user
calculate the circumference
calculate the area
calculate the volume
display all three results

7.3 Write a simple calculator program which reads two numbers separated by an operator (i.e. 3 * 5) and calls one of four functions to calculate the correct answer and a further function to display the result.

7.4 Modify exercise 3.4 to use two functions **first_half** & **last_half** to display the output.

Section 8 Files

Until now we have read in our data from the keyboard and displayed our output on screen. We often need to read and write data from/to files which in C is done through functions defined in **stdio.h**.

Files are accessed through file pointers. As pointers have not yet been discussed, you may view the file pointer as a variable that points to the file control block for the file which holds information required by the file access functions. A file must be opened before you can access it and you should close it after use.

8.1 Reading From Files

Consider the following program which reads one character at a time from a file **myfile** and displays that character on screen.

```
#include <stdio.h>
void main()
{
   FILE *fp;
   int c;            /* use int to recognise EOF (see section 3.1) */
   fp = fopen("myfile","r");
   if(fp == NULL)
   {
      puts("Cannot Open File");
      exit(1);
   }
   c = fgetc(fp);
   while(c != EOF)
   {
      putchar(c);
      c = fgetc(fp);
   }
   fclose(fp);
}
```

Let's now look at the key points of the program.

FILE *fp;

The file pointer is declared. The asterisk signifies that it is a pointer to an object which in this case is a file.

fp = fopen("myfile","r");

The **fopen** function returns a valid pointer on successful opening of the file. If fopen could not open the file a NULL pointer is returned. The first argument to **fopen** is a string containing the name of the file as it is held on disk. The second argument is the **mode** of access which in this case is **"r"** to **read** from the file. The file pointer **fp** can now be used to access the file.

if(fp == NULL)

As **fopen** returns **NULL** on unsuccessful opening of a file, the pointer can be tested. If **fp** contains **NULL** we would know that the file was not opened successfully.

exit(1);

The **exit** function terminates execution of the program and passes back a value to the calling program which is generally 0 for a valid termination or a non-zero for error..

c = fgetc(fp);

The **fgetc** function reads a character from the file and returns it to the calling code. The variable **c** in this case is assigned the value of the character **pointed to** in the file. The next call to **fgetc** will return the next character in the file.

fclose(fp);

The fclose function closes the file freeing the pointer **fp** for another file.

8.2 Writing To Files

Consider a variation of the previous example program which copies a file.

```
#include <stdio.h>
void main()
{
   FILE *infile, *outfile;
   int c;
   infile = fopen("myfile","r");
   if(infile == NULL)
   {
      puts("Cannot Read File");
      exit(1);
   }
   outfile = fopen("newfile","w");
   if(outfile == NULL)
```

```
    {
       puts("Cannot Create File");
       exit(1);
    }
    c = fgetc(infile);
    while(c != EOF)
    {
       fputc(c, outfile);
       c = fgetc(infile);
    }
    fclose(infile);
    fclose(outfile);
}
```

Let's now look at the key points of the program.

outfile = fopen("newfile","w");

The **fopen** function opens the file for writing (**"w"**). A **NULL** will be returned should the file not be successfully opened i.e. write protected. If the named file already exists it will be **over-written** with the new data. Should you wish to append to an existing file then the mode **"a"** should be used.

fputc(c, outfile);

The **fputc** function writes a character to the file which in the above example is the character held by the variable **c** being written to the file pointed to by **outfile**.

8.3 Files & Strings

It is often the case that lines of text need to be processed when reading files. Consider the following program which reads data, a line at a time from a file. The file contents could be :

John 13
Eric 25
Joan 32
Peter 65
Harry 76
Tony 19

where the names and their ages are stored. The program prints all lines where the age of the person is less than **20** to a file called **teens** and all others to a file called **adults**.

```
#include <stdio.h>
#define LINELEN 82
void main()
{
   FILE *infile, *oldfile, *teenfile;
   char line[LINELEN];
   int age;
   infile = fopen("ages", "r");
   if(infile == NULL)
   {
      printf("CANNOT READ FILE\n");
      exit(1);
   }
   oldfile = fopen("adults","w");
   if(oldfile == NULL)
   {
      printf("CANNOT CREATE FILE\n");
      exit(1);
   }
   teenfile = fopen("teens","w");
   if(teenfile == NULL)
   {
      printf("CANNOT CREATE FILE\n");
      exit(1);
   }
   while(fgets(line, LINELEN, infile))
   {
      sscanf(line,"%*s%d", &age);
      if(age >= 20)
        fprintf(oldfile,"%s",line);
      else
        fprintf(teenfile,"%s",line);
   }
   fclose(infile);
   fclose(oldfile);
   fclose(teenfile);
}
```

Let's now look at the key points of the program.

while(fgets(line, LINELEN, infile))

The while loop has the function **fgets** as its condition. Although this may appear unreadable, it is quite common to encounter such statements in C, indeed it won't be long before you are doing it yourself so let's see what's happening. The function **fgets** returns a string (a pointer actually) or **NULL** when an error has occurred. This program uses t that is returned when no more data can be read to terminate the loop.

sscanf(line,"%*s%d", &age);

The sscanf function reads data from a string rather than standard input. It is often used to extract data from a string while leaving the original string intact.

fprintf(oldfile,"%s",line);

The **fprintf** function is the same as **printf** except that the output is to a file instead of standard output.

Exercises

8.1 Modify exercise 4.1 to accept input from a text file

8.2 Write a program that reads a text file and converts all text to uppercase.

8.3 Write a program that compares the contents of two text files. When the first difference is observed the program should report the line number, followed by the two lines and then exit. If no differences are found, the program should report such.

Section 9 Command Line Arguments

It is often required to pass data as arguments when calling a program i.e. the DOS command **type** displays the contents of a text file and is used:

type file.txt

where **type** is the command and **file.txt** is the argument. Let's write our own version of the DOS type command.

```
#include <stdio.h>
void main(int argc, char *argv[])       /* you may see this declared **argv   */
{
   FILE *infile;
   int ch;
   if(argc != 2)
   {
      printf("Usage .. type filename\n");
      exit();
   }
   if((infile = fopen(argv[1],"r")) == NULL)
   {
      printf("Can't Open %s\n", argv[1]);
      exit();
   }
   while((ch = fgetc(infile)) != EOF)
      putchar(ch);
   fclose(infile);
}
```

Let's look at the key points of the program.

void main(int argc, char *argv[])

Our main is given two parameters usually named **argc** & **argv**. argc is an integer representing the number of arguments in the command. argv is a two dimensional array of the characters in the command line (including the command itself).

if((infile = fopen(argv[1],"r")) == NULL)

argv[0] holds the command itself and argv[1] holds the first argument after the command which is the name of the file.

Exercises: Modify exercises in section 8 to pass the names of files as arguments.

Section 10 Bit-wise Operations

C allows us access to the individual bits within variables and provides the following 6 operators for bit manipulation.

&	AND
|	OR
^	XOR
~	NOT
>>	right shift
<<	left shift

The following example program demonstrates each operator.

```
#include <stdio.h>
void showBits(char c) /* display bit settings of a byte */
{
   int i;
   for(i=0 ; i<8 ; c <<= 1, i++)
      putchar((c & 0x80 ) ? '1' : '0');
   putchar('\n');
}
void logAND(char a, char b)
{
   puts("\nLogical AND");
   showBits(a);
   showBits(b);
   showBits(a & b);
}
void logOR(char a, char b)
{
   puts("\nLogical OR");
   showBits(a);
   showBits(b);
   showBits(a | b);
}
void logXOR(char a, char b)
{
   puts("\nLogical XOR");
   showBits(a);
```

```
    showBits(b);
    showBits(a ^ b);
}
void logNOT(char a)
{
    puts("\nLogical NOT");
    showBits(a);
    showBits( ~ a);
}
void lshift(char a)
{
    puts("\nLeft Shift One");
    showBits(a);
    showBits( a << 1);
}
void rshift(char a)
{
    puts("\nRight Shift One");
    showBits(a);
    showBits( a >> 1);
}
void main()
{
    char x, y;
    x = 0xb4;    /* x = 1011 0100 */
    y = 0x6c;    /* y = 0110 1100 */
    logAND(x,y);
    logOR(x,y);
    logXOR(x,y);
    logNOT(x);
    lshift(x);
    rshift(y);
}
```

Compile and run the program to view the results.

Let's now look at a practical example where from a string, a vertical and horizontal parity check character is produced (even parity) to transmit within the transmission frame for error checking in a simple communications network. The program does a ParityCheck on each character in the string, setting the high-order bit as necessary to

produce an even no. of bits set. Each character is then XORed to produce the check character. This bit settings of this check character is displayed.

```
#include <stdio.h>
#define BYTE unsigned char
#define BUFLEN 30
void showBits(BYTE c)        /* display bit settings of a byte */
{
   int i;
   for(i=0 ; i<8 ; c <<= 1, i++)
      putchar((c & 0x80 ) ? '1' : '0');
   putchar('\n');
}
char ParityCheck(BYTE c)
{
   int i;
   char temp = c;                  /* save original char */
   for( i = 0; temp != 0; temp >>= 1)
      if( temp & 01 )
         i++;
   if(i%2)                         /* odd bits ? */
      c |= 0x80;           /* set parity bit */
   return c;
}
void main()
{
   BYTE pchar = 0;
   char frame[BUFLEN];
   int i;
   for ( i = 0; i < BUFLEN ; i++ )
      frame[i] = 0;                /* initialise frame */
   printf("\nEnter Word - ");
   gets(frame);
   for ( i = 0; i < BUFLEN-1 && frame[i]; i++ )
      pchar ^= ParityCheck(frame[i]);
   showBits(pchar);
}
```

Let's see what's happening with the simple input string "fred".

This example uses the **ASCII** code which represents the string "fred" as shown. The parity column shows that only the letter 'd' has an odd number of bits set and therefore only this character requires the parity bit set for the horizontal parity check.

char	ASCII	Binary	Parity Char
'f'	102	0110 0110	0110 0110
'r'	114	0111 0010	0111 0010
'e'	101	0110 0101	0110 0101
'd'	100	0110 0100	**1**110 0100

The variable **pchar** is initialised to zero, so each stage of the vertical parity check (XORing with the previous value) is shown as follows.

pchar	0000 0000	
	0110 0110	parity char for 'f'
pchar	0110 0110	pchar XORed with zero gives same bit setting
	0111 0010	parity char for 'r'
pchar	0001 0100	pchar XORed with 'r'
	0110 0101	parity char for 'e'
pchar	0111 0001	pchar XORed with 'e'
	1110 0100	parity char for 'd'
pchar	1001 0101	

Exercise Write a program that accepts an integer as input from a user, copies the high and low-order bytes of the integer into two character variables, then re-assembles the integer into another integer variable.

Section 11 Structures

Structures are a way of holding related data together in one variable. Consider the following program reading records from a file and outputs to screen and another file.

```
#include <stdio.h>
struct data
      {
         int age;
         char name[20];
      };
void main()
{
   int i, count = 0;
   char line[80];
   FILE *f;
   struct data emp[10];                    /* an array of data */
   f = fopen("a:\\datain.txt","r");
   if ( f == NULL)
      exit(1);    /* file not opened */
   while ((fgets(line,80,f)) != NULL)           /* fgets returns NULL if EOF reached */
   {
      sscanf(line,"%d %s",&emp[count].age,emp[count].name);
      printf("\nData - %d %s",emp[count].age,emp[count].name);    /* output to screen */
      count++;
   }
   fclose(f);
   printf("%d Records In The File",count);
   f = fopen("dataout.txt","w");
   for (i = 0; i < count;i++)     /* display all records */
      fprintf(f,"%d %s\n",emp[i].age,emp[i].name);
   fclose(f);
}
```

The input file (datain.txt) has the following format

```
23 Ged
66 Albert
12 Johnny
16 Jane
```

Let's look at the code to see what's happening.

```
struct data
      {
         int age;
         char name[20];
      };
```

The record structure is defined to have two elements, an integer and a string..

struct data emp[10];

An array of the structure type is declared. All variables of this type contain two items of data:

an integer *varname***.age**
a string *varname***.name**

Any variable of this type can accessed by using the **.** operator with the variable name followed by its sub identifier and can be treated the same as any other variable of the same, standard types already present in C i.e. int, char, float, etc.

```
sscanf(line,"%d %s",&emp[count].age,emp[count].name);
printf("\nData - %d %s",emp[count].age,emp[count].name);
```

sscanf (string scanf) reads data from the array **line** and assigns the values to appropriate fields. Access to elements of the structure is by use of the **.** operator. printf similarly outputs elements of the data.

Exercises

11.1 Write a program to read records in the following format (age, salary, id, name) into a suitable record structure into memory from a text file. Each field in the file is separated by one space.

Age	Salary	ID No.	Name
21	14500	1421	Albert
43	12125	4182	Frank
etc.			

Create an enquiry routine that will accept an ID from the user then search the data for a match, outputting the name and age if found or a suitable message if the ID is not on record. Repeat this routine until zero is entered.

11.2 Write a program that reads data from the keyboard into the above structure and writes this data to a file. Check this by running exercise 11.1 on the created file.

Section 12 Memory Management & Pointers

Considering the example in section 11, we can see that the structure

```
struct data
    {
      int age;
      char name[20];
    };
```

will take up 22 bytes of memory for each record. As it is only an example, the total amount of memory allocated by the declaration

```
struct data emp[10];
```

is 10 x 22 = 220 bytes is hardly excessive but considering a more realistic example may put this into perspective. Consider the following data structure of student information:

```
struct student
    {
      char id[10];
      char surname[25];
      char forenames[25];
      char addrstreet[25];
      char addrtown[25];
      char addrcounty[20];
      char postcode[10];
      char telno[10];
    };
```

This information equates to 150 bytes for one record and considering that more data would actually be recorded such as next of kin details, courses taken etc. this is still only a modest figure.

Taking a modest figure of 10,000 students (large colleges and Universities have far more) then 150 x 10,000 = 1,500,000 or 1.5 Mbytes of memory would be required to hold the data.

Now consider that it is unlikely that we would be processing all the students at once, we still have a decision to make when declaring the variable (and therefore allocating memory) of how many records are needed to give the right balance of:

1. enough storage to process all required records
2. not excessive as to unnecessarily waste memory

It makes sense therefore to get the memory when it is required. We can do this by using pointers which can be done in different ways.

12.1 An array of pointers

Consider the following

struct student studdata[1000];

An array of 1000 records is declared which using the earlier example allocates 150,000 bytes of memory. The following declaration of an array of pointers

struct student *studdata[1000];

declares 1000 pointers to the data without actually allocating the memory for the data. The pointer will be allocated the address of the data when it is known. As an address (depending on the system) is only 2 or so bytes, therefore the declaration in this case would only use

1000 x 2 = 2000 bytes

Now when each new record is encountered, we must allocate the memory to store the data and assign this memory address to the pointer. Consider a variation of the example in section 11:

```
#include <stdio.h>
#include <alloc.h>
#define MAX  100
typedef struct  /* alternative way of defining a record structure */
       {
          int age;
          char name[20];
       } mydata;
void err()        /* error report & exit function */
{
   printf("System Capacity Exceeded \n");
   printf("Or File Error ... Press Any Key To Exit");
   getch();
   exit(1);
}
void main()
{
   int i, count = 0;
   char line[80];
   FILE *f;
   mydata *data[MAX];         /* an array of MAX pointers */
   f = fopen("datain.txt","r");
   if(f == NULL)
      err();
   clrscr();
   while ((fgets(line,80,f)) != NULL)
   {
      data[count] = ( mydata *) malloc ( sizeof(mydata));
      if(data[count] == NULL)
         err();
      sscanf(line,"%d %s",&data[count]->age,data[count]->name);
      printf("%d %s\n",data[count]->age,data[count]->name);
      count++;
   }
   fclose(f);
   printf("\n%d Records In The File\n",count);
   f = fopen("dataout.txt","w");
   if(f == NULL)
      err();
   for (i = 0; i < count;i++)
   {
      fprintf(f,"%d %s\n",data[i]->age,data[i]->name);
      printf("%d %s\n",data[i]->age,data[i]->name);
   }
   fclose(f);
}
```

Let's look at the key points of the code.

```
typedef struct/* alternative way of defining a record structure */
    {
       int age;
        char name[20];
    } mydata;
```

As commented, this is an alternative method of declaring a data structure (defining a new type). This type is identified as **mydata**.

```
mydata *data[MAX];          /* an array of MAX pointers */
```

An array of pointers is declared, i.e. each array element (data[0], data[1], ... data[MAX]) can hold the memory address of a record of type **mydata**. As a pointer (an address) can refer to any type of data (int, char, float or user defined type in this case) the question may be raised as to why its type is necessary in the declaration. This is because pointers can be manipulated in C in different ways. This will be explained later in this section.

```
data[count] = ( mydata *) malloc ( sizeof(mydata));
```

The memory for each new record is allocated by **malloc** (memory allocation) and rather than stating explicitly the size of memory in bytes that is required, the function **sizeof** is used to do the counting for us. This makes for a more maintainable program as later modifications in the record structure can be catered for without the need for further action. As malloc does not return the specific pointer type, the type of pointer must be **cast** (forcing its type). Although the pointer type is not used in this example, it is recommended that it is used as any future use (modifications to the program) may cause unwanted side effects.

```
if(data[count] == NULL)
```

Malloc returns an address if memory is available or NULL otherwise. This can be tested to give our program reliability.

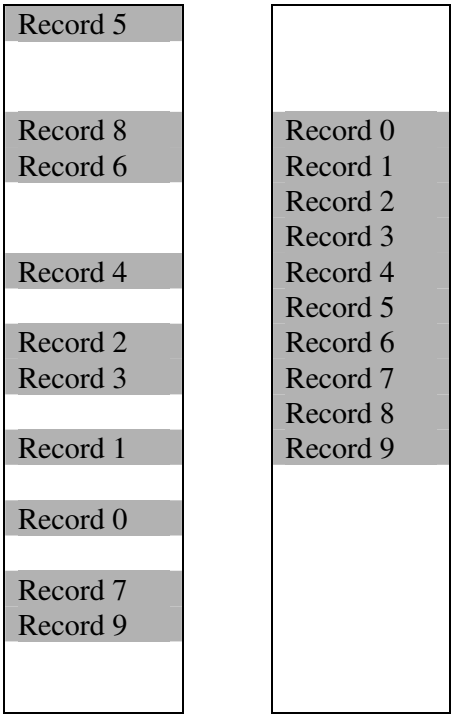
```
sscanf(line,"%d %s",&data[count]->age,data[count]->name);
```

sscanf (string scanf) reads data from the array line and assigns the values to appropriate fields similar to the previous example except access to elements of the structure is by use of the **->** operator to differentiate between data (with the . operator) and a pointer to data.

12.2 Blocks of Memory

In the previous example, the location of data may be scattered throughout memory. If all the data is held together, then we only need to hold one pointer to the data block, then simply step down memory during processing.

Consider the following memory models:



It can be seen that if our records are distributed throughout memory, individual pointers are necessary to process the data. If all records were located together, then only one pointer is needed with the position of subsequent location able to be calculated easily as we would know the size of a record. Consider a variation of the previous example program:

```
#include <stdio.h>
#include <alloc.h>
typedef struct
      {
         int age;
         char name[20];
      } mydata;
void err()
{
   printf("System Capacity Exceeded \n");
   printf("Or File Error ... Press Any Key To Exit");
   getch();
   exit(1);
}
void main()
{
   int i, count = 0;
   char line[80];
   FILE *f;
   mydata *current,*data;
   f = fopen("a:\\datain.txt","r");
   data = (mydata *) malloc (sizeof(mydata));
   if(data == NULL || f == NULL)
      err();
   while ((fgets(line,80,f)) != NULL)
   {
      current = data + count;
      sscanf(line,"%d%s",&current->age,current->name);
      printf("\nData - %d %s",current->age,current->name);
      count++;
      data = ( mydata *) realloc (data,(count + 1) * sizeof(mydata));
      if(data == NULL)
         err();
   }
   fclose(f);
   printf("\n%d Records In The File\n",count);
   f = fopen("dataout.txt","w");
   if(f == NULL)
      err();
   for (i = 0,current = data; i < count;i++,current++)
   {
      fprintf(f,"%d %s\n",current->age,current->name);
      printf("%d %s\n",current->age,current->name);
   }
   fclose(f);
}
```

Let's look at the key points of the code.

```
mydata *current,*data;
```

Although we only need one pointer to process the data, we will need another to save the memory location of the start of the memory block.

```
data = ( mydata *) realloc (data,(count + 1) * sizeof(mydata));
```

The realloc function attempts to re-size the original data block to the new specified size. If the current block cannot be expanded, realloc will attempt to find a new memory block and copy the original data into it.

```
for (i = 0,current = data; i < count;i++,current++)
{
   fprintf(f,"%d %s\n",current->age,current->name);
   printf("%d %s\n",current->age,current->name);
}
```

To process the data, the pointer current is first assigned the value of the start of the data block. Once the current record has been processed, the pointer can be incremented to the next record i.e.

```
current++;
```

This is the reason why the type is needed so that data blocks can be stepped up and down using one pointer, rather than an array of pointers or other methods.

Familiarise yourself with the above by modifying the exercises in section 11.