

**LECTURER NOTES**

**ICS 2175 AND SMA 2175**

**COMPUTER PROGRAMING 1**

**BSC. MECHANICAL AND**

**MECHATRONIC ENGINEERING**

**YEAR 2 SEM 1**

**MAY AUGUST 2012**

# CONTENTS

<b>INTRODUCTION .....</b>	<b>Error! Bookmark not defined.</b>
<b>1 BASIC CONCEPTS .....</b>	<b>1</b>
Objectives.....	1
Introduction .....	1
Programming defined.....	1
Programming language defined.....	2
Structured Programming defined .....	2
Steps in program development.....	2
C language basic features .....	3
Advantages of C Language.....	4
C Program components.....	5
Program errors and debugging .....	11
Guidelines to good C programming .....	12
Revision Exercise .....	Error! Bookmark not defined.
<b>2 DATA HANDLING.....</b>	<b>14</b>
Objectives.....	14
Introduction .....	14
Variables.....	14
Basic data types .....	15
Using printf( ) to output values .....	16
Inputting values from the keyboard using scanf( ) .....	18
Types of variables .....	20
Storage classes.....	20
Constants.....	23
Revision Exercise .....	Error! Bookmark not defined.
<b>3 OPERATORS.....</b>	<b>27</b>
Objectives.....	27
Introduction .....	27
Operators versus operands .....	27
Arithmetic operators .....	28
The assignment operator .....	31
Relational operators.....	32
Logical operators.....	33
The Conditional operator .....	33
Unary operators .....	34
The sizeof operator .....	34
Revision Exercise .....	Error! Bookmark not defined.

<b>4 CONTROL STRUCTURES .....</b>	<b>37</b>
Objectives.....	37
Introduction .....	37
The if statement .....	38
The if - else statement .....	39
The else .. if statement.....	39
The switch ..case statements .....	41
The 'while' loop .....	44
The 'do .. while' loop .....	45
The 'for' loop .....	46
Using break and continue statements in loops.....	48
The 'goto' statement .....	49
Nesting statements .....	49
Revision Exercise .....	Error! Bookmark not defined.
<b>5 FUNCTIONS.....</b>	<b>53</b>
Objectives.....	53
Introduction .....	53
Why use functions? .....	53
Types of functions .....	54
Defining a function .....	55
Accessing a function.....	57
Function prototypes .....	59
Recursion.....	61
Revision Exercise .....	Error! Bookmark not defined.
<b>6 ARRAYS .....</b>	<b>67</b>
Objectives.....	67
Introduction .....	67
Array defined .....	67
Declaring arrays .....	68
One-dimensional arrays .....	69
Two – dimensional arrays .....	71
Initialising arrays .....	73
Strings .....	80
Revision Exercise .....	Error! Bookmark not defined.
<b>7 POINTERS .....</b>	<b>87</b>
Objectives.....	87
Introduction .....	87
Pointer defined.....	87
Pointer declaration.....	87
Pointer Operators.....	88
Pointers operations .....	89

Pointer precautions .....	91
Pointers and arrays .....	92
String variables as pointers .....	93
Revision Exercise .....	Error! Bookmark not defined.
<b>8 STRUCTURES .....</b>	<b>96</b>
Objectives.....	96
Introduction .....	96
Structure defined .....	96
Declaring a structure.....	96
Initializing a structure .....	99
Accessing structure members .....	99
Arrays of structures.....	101
Uses of structures .....	102
User defined types (typedef) .....	103
Unions.....	104
Revision Exercise .....	Error! Bookmark not defined.
<b>9 FILE HANDLING .....</b>	<b>109</b>
Objectives.....	109
Introduction .....	109
The streams concept .....	109
Opening a file .....	110
Closing a file .....	112
Formatted disk I/O functions: fprintf( ) and fscanf( ) .....	112
Using fread( ) and fwrite( ) functions.....	116
Simple database management application .....	119
Revision Exercise .....	Error! Bookmark not defined.
<b>MODEL EXAMINATION PAPER .....</b>	<b>125</b>
<b>BIBLIOGRAPHY .....</b>	<b>Error! Bookmark not defined.</b>
<b>INDEX .....</b>	<b>Error! Bookmark not defined.</b>

## BASIC CONCEPTS

### Objectives

At the end of this chapter, the reader should be able to:

- Understand what programming is
- Define a programming language.
- Understand what is structured programming
- Identify the steps of developing a program
- Explain the characteristics of C language.
- Understand the components of a C program
- Identify various types of program errors
- Create and compile a program
- Add comments to a program.

### Introduction

Literally speaking, a language is a vehicle of communication amongst human beings. To communicate, a given procedure is followed which may normally be dictated by grammar.

A computer language is not very different. Computers do not understand any of the natural languages for transfer of data and instructions. So there are languages specially developed so that you could pass your data and instructions to the computer to do a specific job.

This chapter introduces structured approach to programming which the rest of this guide is all about. The chapter also presents a quick overview of C language. The goal is to give you sufficient working knowledge of C language so that you can understand more concrete concepts in later chapters.

### Programming defined

Programming is the translation of user ideas into a representation or form that can be understood by the computer. The tools of writing programs are called programming languages.

## Programming language defined

A programming language is a set of rules used to write computer programs. Like human languages, computer languages have a syntax which defines the language grammar.

## Structured Programming defined

Structured programming is an approach to writing programs that are easier to read, test, debug and modify. The approach assists in the development of large programs through stepwise refinement and modularity. Programs designed this way can be developed faster. When modules are used to develop large programs, several programmers can work on different modules, thereby reducing program development time.

In short, structured programming serves to increase programmer productivity, program reliability (readability and execution time), program testing, program debugging and serviceability.

## Steps in program development

- *Design program objectives*

It involves forming a clear idea in terms of the information you want to include in the program, computations needed and the output. At this stage, the programmer should think in general terms, not in terms of some specific computer language.

- *Design program*

The programmer decides how the program will go about its implementation, what should the user interface be like, how should the program be organized, how to represent the data and what methods to use during processing. At this point, you should also be thinking generally although some of your decisions may be based on some general characteristics of the C language.

- *Develop the program*

Developing a program in a compiled language such as C requires at least four steps:

- Editing (or writing) the program
- Compiling it
- Linking it
- Executing it

### Editing

You write a computer program with words and symbols that are understandable to human beings. This is the *editing* part of the development cycle. You type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the *source file*. The custom is that the text of a C program is stored in a file with the **extension .c** for C programming language.

### Compiling

You cannot directly execute the source file. To run on any computer system, the source file must be translated into binary numbers understandable to the computer's Central Processing Unit. This process produces an intermediate object file - with the **extension .obj**.

### Linking

The main reason for linking is that many compiled languages come with library routines which can be added to your program. These routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. In the case of C the standard input and output functions are contained in a library (stdio.h) so even the most basic program will require a library function. After linking, the file **extension is .exe** which is an executable file.

### Executable files

The text editor produces .c source files, which go to the compiler, which produces .obj object files, which go to the linker, which produces .exe executable file. You can then run .exe files as you run applications, simply by typing their names at the **DOS** prompt or run using Windows menu.

- *Test the program*

This involves checking whether the system does what it is supposed to do. Programs may have bugs (errors). Debugging involves the finding and fixing of program errors.

- *Maintain the program*

Occasionally, changes become necessary to make to a given program. You may think of a better way to do something in a program, a clever feature or you may want to adapt the program to run in a different machine. These tasks are simplified greatly if you document the program clearly and follow good program design practices.

## **C language basic features**

C is a general-purpose language which has been closely associated with the Unix operating system for which it was developed - since the system and most of the programs that run it are written in C.

C supports the traditional programming style of structured programming and is one of the most popular and certainly the most powerful language in this class.

C is often referred to as a *middle level* language. This refers to the fact that C can be used to write low level programs as well as high level languages. Low level languages are machine oriented but provide greater efficiency than high level languages. High level languages provide various control structures, I/O commands and so on which make programming easier and faster.

Before C, assembly languages were used to write computer programs. The shortcomings of the assembly languages were one reason that C was initially designed. It successfully combines the features of a high-level language and the power and efficiency of assembly language.

Initially C was used for creating *systems software*. Systems software consists of those programs which help run the computer. These include operating systems, compilers and editors. However, as C gained popularity, it began to be used for general purpose programming. Today, C is used by programmers for virtually any programming task.

### **Advantages of C Language**

- *C Supports structured programming design features*

It allows programmers to break down their programs into functions. It also supports the use of comments, making programs readable and easily maintainable.

- *Efficiency*

C is a concise language that allows you to say what you mean in a few words. The final code tends to be more compact and runs quickly.

- 
- *Portability*

C programs written for one system can be run with little or no modification on other systems.



- *Power and flexibility*

- Power

C has been used to write operating systems (such as Unix and Windows), Language Compilers, Assemblers, Text Editors, Print Spoolers, Network Drivers, Application packages (such as WordPerfect and Dbase), Language Interpreters, Utilities, et al.

- Flexibility

It has (and still is) been used to solve problems in areas such as physics and engineering.

- *Programmer orientation*

C is oriented towards the programmer's needs. It gives access to the hardware. It lets you manipulate individual bits of memory. It also has a rich selection of operators that allow you to expand programming capability.

## **C Program components**

### *Keywords*

These are reserved words that have special meaning in a language. The compiler recognizes a keyword as part of the language's built – in syntax and therefore it cannot be used for any other purpose such as a variable or a function name. C keywords **must be used in lowercase** otherwise they will not be recognized.

### Examples of keywords

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

A typical C program is made of the following components:

- Preprocessor directives
- Functions
- Declaration statements
- Comments
- Expressions
- Input and output statements

## Example

This program will print out the message: **This is a C program.**

```
#include<stdio.h>
main()
{
    printf("This is a C program \n");
    return 0;
}
```

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 file system 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 new line as part of the output.

Most C programs are written 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.

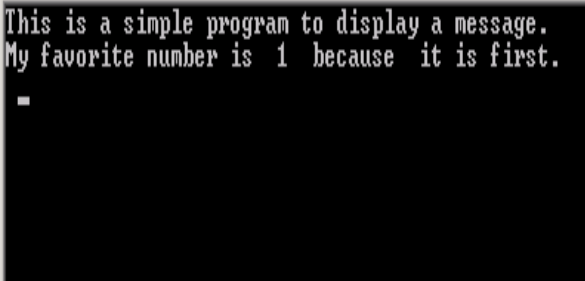
C is case sensitive, that is, it recognises a lower case letter and it's upper case equivalent as being different.

The following program demonstrates the C program components.

```
#include<stdio.h>
main()
{
    int num;      /* define a variable called num */
    num = 1;      /* assignment */
    printf("This is a simple program ");
    printf("to display a message. \n");
    printf("My favorite number is %d because ", num);
    printf("it is first.\n ");
    return 0;
}
```

```
}
```

On running the above program, you get the following output.



```
This is a simple program to display a message.  
My favorite number is 1 because it is first.  
-
```

### *Functions*

All C programs consist of one or more functions, each of which contains one or more **statements**. In C, a function is a named subroutine that can be called by other parts of the program. Functions are the building blocks of C.

A *statement* specifies an action to be performed by the program. In other words, statements are parts of your program that actually perform operations.

All C statements must end with a semicolon. C does not recognize the end of a line as a terminator. This means that there are no constraints on the position of statements within a line. Also you may place two or more statements on one line.

Although a C program may contain several functions, the only function that it must have is **main( )**.

The **main( )** function is the point at which execution of your program begins. That is, when your program begins running, it starts executing the statements inside the **main( )** function, beginning with the first statement after the opening curly brace. Execution of your program terminates when the closing brace is reached.

Another important component of all C programs is *library functions*. The ANSI C standard specifies a minimal set of library functions to be supplied by all C compilers, which your program may use. This collection of functions is called the *C standard library*. The standard library contains functions to perform disk I/O (input / output), string manipulations, mathematics, and much more. When your program is compiled, the code for library functions is automatically added to your program.

One of the most common library functions is called **printf( )**. This is C's general purpose output function. Its simplest form is

```
printf("string - to - output");
```

The `printf( )` outputs the characters that are contained between the beginning and ending double quotes.

For example, `printf(" This is a C program ");`

The double quotes are not displayed on the screen. In C, one or more characters enclosed between double quotes is called a *string*. The quoted string between `printf( )`'s parenthesis is called an *argument* to `printf( )`. In general, information passed to a function is called an argument. In C, calling a library function such as `printf( )` is a statement; therefore it must end with a semicolon.

In the above program, line 6 causes the message enclosed in speech marks “ ” to be printed on the screen. Line 7 does the same thing.

The `\n` in line 7 tells the computer to insert a new line after printing the message. `\n` is an example of an escape sequence.

Line 8 prints the value of the variable `num` (1) embedded in the phrase. The `%d` instructs the computer where and in what form to print the value. `%d` is a **type specifier** used to specify the output format for integer numbers.

Line 9 has the same effect as line 7.

Line 11 indicates the value to be returned by the function `main( )` when it is executed. By default any function used in a C program returns an integer value (when it is called to execute). Therefore, line 2 could also be written `int main( )`. If the `int` keyword is omitted, still an integer is returned.

Then, why `return (0);` ? Since all functions are subordinate to `main( )`, the function does not return any value.

### **Note**

- (i) Since the main function does not return any value, line 3 can alternatively be written as : **`void main( )`** – void means valueless. In this case, the statement **`return 0;`** is not necessary.
- (ii) While omitting the keyword **`int`** to imply the return type of the **`main( )`** function does not disqualify the fact that an integer is returned (since **`int`** is default), you should explicitly write it in other functions, especially if another value other than zero is to be returned by the function.

### *Preprocessor directives and header files*

A preprocessor directive performs various manipulations on your source file before it is actually compiled. Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler

The preprocessor directive `#include` is an instruction to read in the contents of another file and include it within your program. This is generally used to read in header files for library functions ( See line 1 in sample program above).

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.

### *Comments*

Comments are non – executable program statements meant to enhance program readability and allow easier program maintenance, i.e. they document the program. They can be used in the same line as the material they explain (see lines 4 and 5 in sample program).

A long comment can be put on its own line or even spread on more than one line. Comments are however optional in a program. The need to use too many comments can be avoided by good programming practices such as use of sensible variable names, indenting program statements, and good logic design. Everything between the opening `/*` and closing `*/` is ignored by the compiler.

### *Declaration statements*

In C, all variables must be declared before they are used. Variable declarations ensure that appropriate memory space is reserved for the variables, depending on the data types of the variables. Line 4 is a declaration for an integer variable called `num`.

### *Assignment and expression statements*

An assignment statement uses the assignment operator `=` to give a variable on the operator's left side the value to the operator's right or the result of the expression on the right. The statement `num = 1;` (Line 5) is an assignment statement.

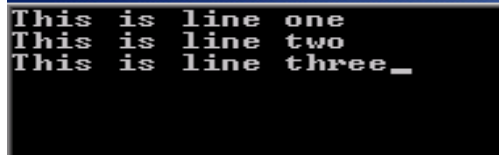
### *Escape sequences*

Escape sequences (also called back slash codes) are character combinations that begin with a backslash symbol (`\`) used to format output and represent difficult-to-type characters.

One of the most important escape sequences is `\n`, which is often referred to as the new line character. When the C compiler encounters `\n`, it translates it into a carriage return.

### Example

The program below displays the following output on the screen.



```
This is line one
This is line two
This is line three_
```

```
#include<stdio.h>
main()
{
    printf("This is line one  \n");
    printf("This is line two \n");
    printf("This is line three");
    return 0;
}
```

### Example

The program below sounds the bell.

```
#include<stdio.h>
main()
{
    printf("\a");
    return 0;
}
```

Remember that the escape sequences are character constants. Therefore to assign one to a character variable, you must enclose the escape sequence within single quotes, as shown in this fragment.

```
char ch;
ch = '\t '          /*assign ch the tab character */
```

Below are other escape sequences:

Escape sequence	Meaning
\a	alert/bell
\b	backspace
\n	new line

<code>\v</code>	vertical tab
<code>\t</code>	horizontal tab
<code>\\</code>	back slash
<code>\'</code>	Single quote (')
<code>\"</code>	Double quote (")
<code>\0</code>	null

## Program errors and debugging

There are three types of errors: Syntax, Semantic and Logic errors.

### *Syntax errors*

They result from the incorrect use of the rules of programming. The compiler detects such errors as soon as you start compiling. A program that has syntax errors can produce no results. You should look for the error in the line suggested by the compiler.

Syntax errors include;

- Missing semi colon at the end of a statement e.g. `Area = Base * Length`
- Use of an undeclared variable in an expression
- Illegal declaration e.g. `int x, int y, int z;`
- Use of a keyword in uppercase e.g. `FLOAT`, `WHILE`
- Misspelling keywords e.g. `init` instead of `int`

### **Note**

The compiler may suggest that other program line statements have errors when they may not. This will be the case when a syntax error in one line affects directly the execution of other statements, for example multiple use of an undeclared variable. Declaring the variable will remove all the errors in the other statements.

### *Logic Errors*

These occur from the incorrect use of control structures, incorrect calculation, or omission of a procedure. Examples include: An indefinite loop in a program, generation of negative values instead of positive values. The compiler will not detect such errors since it has no way of knowing your intentions. The programmer must dry run the program so that he/she can compare the program's results with already known results.

### *Semantic errors*

They are caused by illegal expressions that the computer cannot make meaning of. Usually no results will come out of them and the programmer will find it difficult to debug such errors. Examples include a data overflow caused by an attempt to assign a value to a field or memory space smaller than the value requires, division by zero, etc.

### **Guidelines to good C programming**

- Ensure that your program logic design is clear and correct. Avoid starting to code before the logic is clearly set out. Good logic will reduce coding time and result in programs that are easy to understand, error free and easily maintainable.
- Declare all variables before using them.
- Use sensible names for variables. Use of general names such as **n** instead of **net\_sal** for net salary makes variables vague and may make debugging difficult. This however depends on the programmer's ingenuity.
- Use a variable name in the case that it was declared in.
- Never use keywords in uppercase.
- Terminate C declarations, expressions and input/output statements with a semi colon.
- Always save your program every time you make changes.
- Proof read your program to check for any errors or omissions
- Dry run you design with some simple test data before running the code, then compare the two.

### **Revision Exercise**

1. Outline the logical stages of C programs' development.
2. From the following program, suggest the syntax and logical errors that may have been made.

The program is supposed to find the square and cube of 5, then output 5, its square and cube.

```
#include<stdio.h>
main()
{
```



```
int , int n2, n3;  
n = 5;  
n2 = n *n  
n3 = n2 * n2;  
printf(" n = %d, n squared = %d, n cubed = %d \ n", n, n2, n3);  
return 0;  
}
```

3. Give the meaning of the following, with examples
  - (i) Preprocessor command
  - (ii) Keyword
  - (iii) Escape sequence
  - (iv) Comment
  - (v) Linking
  - (vi) Executable file
4. C is both 'portable' and 'efficient'. Explain.
5. C is a 'case sensitive' language. Explain.
6. The use of comments in C programs is generally considered to be good programming practice. Why?

## DATA HANDLING

### Objectives

At the end of this chapter, the reader should be able to:

- Declare variables and assign values
- Describe basic data types used to declare variables
- Set up constants and apply them in a program
- Input numbers from the keyboard using the **scanf()** function
- Explain **printf()** function capabilities.

### Introduction

Almost every program you write performs some sort of data manipulation, however basic. These data are stored and understood differently in your program depending on how you have defined them.

The aim of this chapter is to help you understand what is required to use data items in a C program.

### Variables

#### ***Variable defined***

A variable is a memory location whose value can change during program execution. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code.

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

That is:

*datatype variable; or*

*datatype variable1,variable2,...variablen; (where there are  $n$  variables)*

For example,

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

Declarations can be spread out, allowing space for an explanatory comment. That is: 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 */
```

### **Variable names**

Every variable has a name and a value. The name identifies the variable and 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 recognizes upper and lower case characters as being different (C is case- sensitive). Finally, you cannot use any of C's keywords like **main**, **while**, **switch** etc as variable names.

### **Examples of legal variable names**

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 variablenames. 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, to be covered in chapter 5.

### **Basic data types**

C supports five basic data types. The table below shows the five types, along with the C keywords that represent them. Don't be confused by *void*. This is a special purpose data type used to explicitly declare functions that return no value.

Type	Meaning	Keyword
Character	Character data	Char
Integer	Signed whole number	Int
Float	floating-point numbers	Float
Double	double precision floating-point numbers	Double
Void	Valueless	Void

### ***The 'int' specifier***

It is a type specifier used to declare integer variables. For example, to declare count as an integer you would write:

```
int count;
```

Integer variables may hold signed whole numbers (numbers with no fractional part). Typically, an integer variable may hold values in the range –32,768 to 32,767 and are 2 bytes long.

### ***The 'char' specifier***

A variable of type char is 1 byte long and is mostly used to hold a single character. For example to declare **ch** to be a character type, you would write:

```
char ch;
```

### ***The 'float' specifier***

It is a type specifier used to declare floating-point variables. These are numbers that have a whole number part and a fractional or decimal part for example 234.936. To declare **f** to be of type float, you would write:

```
float f;
```

Floating point variables typically occupy 4 bytes.

### ***The 'double' specifier***

It is a type specifier used to declare double-precision floating point variables. These are variables that store float point numbers with a precision twice the size of a normal float value. To declare **d** to be of type double you would write:

```
double d;
```

Double-type variables typically occupy 8 bytes.

### **Using printf( ) to output values**

You can use printf( ) to display values of characters, integers and floating - point values. To do so, however, requires that you know more about the **printf( )** function.

For example:

```
printf("This prints the number %d ", 99);
```

displays **This prints the number 99** on the screen. As you can see, this call to the `printf( )` function contains two arguments. The first one is the quoted string and the other is the constant 99. Notice that the arguments are separated from each other by a comma.

In general, when there is more than one argument to a function, the arguments are separated from each other by commas. The first argument is a quoted string that may contain either normal characters or format specifiers that begin with a percent (%) sign.

Normal characters are simply displayed as is on the screen in the order in which they are encountered in the string (reading left to right). A format specifier, on the other hand informs **printf( )** that a different type item is being displayed. In this case, the **%d**, means that an integer is to be output in decimal format. The value to be displayed is to be found in the second argument. This value is then output at the position at which the format specifier is found on the string.

If you want to specify a character value, the format specifier is **%c**. To specify a floating-point value, use **%f**. The **%f** works for both **float** and **double**. Keep in mind that the values matched with the format specifier need not be constants (such as 99 in the **printf** statement above). They may be variables too.

Code	Format
<b>%c</b>	Character
<b>%d</b>	Signed decimal integers
<b>%i</b>	Signed decimal integers
<b>%e</b>	Scientific notation (lowercase 'e')
<b>%E</b>	Scientific notation (lowercase 'E')
<b>%f</b>	Decimal floating point
<b>%s</b>	String of characters
<b>%u</b>	Unsigned decimal integers
<b>%x</b>	Unsigned hexadecimal (lowercase letters)
<b>%X</b>	Unsigned hexadecimal (Uppercase letters)

### **Examples**

1. The program shown below illustrates the above concepts. First, it declares a variable called **num**. Second, it assigns this variable the value 100. Finally, it uses **printf( )** to display **the value is 100** on the screen. Examine it closely.

```
#include<stdio.h>
main()
{
    int num;
    num = 100;
    printf(" The value is %d ", num);
    return 0;
}
```

2. This program creates variables of types **char**, **float**, and **double** assigns each a value and outputs these values to the screen.

```
#include<stdio.h>
main()
{
    char ch;
    float f;
    double d;

    ch = 'X';
    f = 100.123;
    d = 123.009;

    printf(" ch is %c ", ch);
    printf(" f  is %f ", f);
    printf(" d  is %f ", d);
    return 0;
}
```

### **Exercise**

Enter, compile, and run the two programs above.

### **Inputting values from the keyboard using scanf( )**

There are several ways to input values through the keyboard. One of the easiest is to use another of C's standard library functions called **scanf( )**.

To use **scanf( )** to read an integer value from the keyboard, call it using the general form:

```
scanf("%d", &int_var-name);
```

Where *int-var-name* is the name of the integer variable you wish to receive the value. The first argument to **scanf( )** is a string that determines how the second argument will be treated. In this case the %d specifies that the second argument will be receiving an integer value entered in decimal format. The fragment below, for example, reads an integer entered from the keyboard.

```
int num;
scanf("%d", &num);
```

The **&** preceding the variable name means 'address of'. The values you enter are put into variables using the variables' location in memory.

When you enter a number at the keyboard, you are simply typing a string of digits. The **scanf()** function waits until you have pressed **<ENTER>** before it converts the string into the internal format used by the computer.

The table below shows format specifiers or codes used in the **scanf()** function and their meaning.

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer
%e	Read a floating point number
%f	Read a floating point number
%lf	Read a double
%s	Read a string
%u	Reads an unsigned integer

### Examples

1. This program asks you to input an integer and a floating-point number and displays the value.

```
#include<stdio.h>
main()
{
    int num;
    float f;
    printf(" \nEnter an integer: ");
    scanf( "%d ", &num);
    printf("\n Enter a floating point number: ");
    scanf( "%f ", &f);
    printf( "%d  ", num);
    printf( "\n %f ", f);
    return 0;
}
```

2. This program computes the area of a rectangle, given its dimensions. It first prompts the user for the length and width of the rectangle and then displays the area.

```
#include<stdio.h>
main()
{
    int len, width;
    printf("\n Enter length:  ");
    scanf ("%d ", &len);
    printf("\n Enter width :  ");
    scanf( " %d ", &width);
    printf("\n The area is %d ", len  * width);
    return 0;
}
```

### **Exercise**

Enter, compile and run the example programs.

### **Types of variables**

There are two places where variables are declared: inside a function or outside all functions.

#### **Global variables**

Variables declared outside all functions are called **global variables** and they may be accessed by any function in your program. Global variables exist the entire time your program is executing.

#### **Local variables**

Variables declared inside a function are called **local variables**. A local variable is known to and may be accessed by only the function in which it is declared. You need to be aware of two important points about local variables.

- (i) The local variables in one function have no relationship to the local variables in another function. That is, if a variable called **count** is declared in one function, another variable called **count** may also be declared in a second function – the two variables are completely separate from and unrelated to one another.
- (ii) Local variables are created when a function is called, and they are destroyed when the function is exited. Therefore local variables do not maintain their values between function calls.

### **Storage classes**

C storage classes determine how a variable is stored. The storage class specifiers are



- auto
- extern
- register
- static

These specifiers precede the type name.

### *auto*

**auto** is the default storage class for local variables. The example below defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

```
{
    int Count;
    auto int Month;
}
```

### *extern*

As the size of a program grows, it takes longer to compile. C allows you to break down your program into two or more files or functions. You can separately compile these files and then link them together. In general, global data may only be declared once. Because global data may need to be accessed by two or more functions that form the program, there must be a way of informing the compiler about the global data used by the program.

Consider the following;

#### **File 1**

```
#include<stdio.h>
int count;
void f1 (void);
main()
{
    int i;
    f1 ( );          /* Set count's value */
    for( i =0; i <count; i++)
        printf("%d");
    return 0;
}
```

## File 2

```
#include<stdlib.h>
void f1(void)
{
    count = rand ( );    /* Generates a random number */
}
```

If you try to compile File 2, an error will be reported because **count** is not defined. However, you cannot change File 2 as follows:

```
#include<stdlib.h>
int count;
void f1(void)
{
    count = rand ( );    /* Generates a random number */
}
```

But there is still a problem, If you declare **count** a second time, the linker will report a duplicate-symbol error, which means that count is defined twice, and the linker doesn't know which to use.

The solution to this problem is C's **extern** specifier. By placing **extern** in front of **count's** declaration in File 2, you are telling the compiler that **count** is an integer declared elsewhere. In other words, using extern informs the compiler about the existence and type of the variable it precedes but does not cause storage for that variable to be allocated. The correct version for File 2 is:

```
#include<stdlib.h>
extern int count;
void f1(void)
{
    count = rand ( );    /* Generates a random number */
}
```

**Note:** `stdlib.h` is a header file that contains certain standard library functions. `rand()` function is one of them and is used to generate a random number between . Others are `abort()` – to abort a program, `abs()` – to get the absolute value ,`malloc()` for dynamic memory allocation ,`free()` to free memory allocated with `malloc()`, `qsort()`to sort an array, `realloc()` to reallocate memory, et al.

### *register*

**register** is used to define local variables that should be stored in a register instead of Random Access Memory (RAM). For example:

```
register int Miles;
```

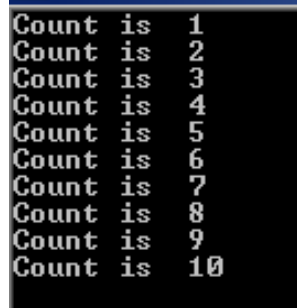
register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it might be stored in a register - depending on hardware and implementation restrictions.

### *static*

The static modifier causes the contents of a local variable to be preserved between function calls. Also, unlike normal local variables, which are initialized each time a function is entered a **static** variable is initialized only once. For example, take a look at the following program:

```
#include<stdio.h>
void f(void);
main()
{
    int i;
    for (i =0; i < 10; i ++ )
        f( );
    return 0;
}
void f (void)
{
    static int count = 0;
    count ++;
    printf("Count is  %d  \n", count);
}
```

which displays the following output.



```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
Count is 6
Count is 7
Count is 8
Count is 9
Count is 10
```

Visibly from above, **count** retains its value between function calls.

### **Constants**

A constant is a value that does not change during program execution. In other words, constants are fixed values that may not be altered by the program.

Integer constants are specified as numbers without fractional components. For example -10, 1000 are integer constants.

Floating - point constants require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating point constant. C allows you to use scientific notation for floating point numbers. Constants using scientific notation must follow this general form:

***number E sign exponent***

The number is optional. Although the general form is shown with spaces between the component parts for clarity, there may be no spaces between parts in an actual number. For example, the following defines the value 1234.56 using scientific notation.

```
123.456E1
```

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. For example:

```
ch = 'z';
```

There is nothing in C that prevents you from assigning a character variable a value using a numeric constant. For example the ASCII Code for 'A' is 65. Therefore, give the declaration:

```
char ch;
```

then these two assignments are equivalent.

```
ch = 'A';  
ch = 65;
```

### ***Types of constants***

Constants can be used in C expressions as:

- Direct constants
- Symbolic constants

#### ***Direct constants***

Here the constant value is inserted in the expression, as it should typically be.

For example:

```
Area = 3.14 * Radius * Radius;
```

The value **3.14** is used directly to represent the value of **PI** which never requires changes in the computation of the area of a circle

### *Symbolic constant*

This involves the use of another C preprocessor, `#define`.

For example, `#define SIZE 10`

A symbolic constant is an identifier that is replaced with replacement text by the C preprocessor before the program is compiled. For example, all occurrences of the symbolic constant **SIZE** are replaced with the replacement text 10.

This process is generally referred to as *macro substitution*. The general form of the `#define` statement is;

```
#define macro-name string
```

Notice that this line does not end in a semi colon. Each time the **macro - name** is encountered in the program, the associated **string** is substituted for it. For example, consider the following program.

### **Example: Area of a circle**

```
#include<stdio.h>
#define PI  3.14
main()
{
    float radius, area;
    printf("Enter the radius of the circle \n");
    scanf("%f", &radius);
    area = PI * radius * radius; /* PI is a symbolic constant */
    printf("Area is %.2f cm squared ",area);
    return 0;
}
```

At the time of the substitution, the text such as 3.14 is simply a string of characters composed of 3, ., 1 and 4. The preprocessor does not convert a number into any sort of internal format. This is left to the compiler.

The macro name can be any valid C identifier. Although macro names can appear in either uppercase or lowercase letters, most programmers have adopted the convention of using uppercase for macro names to distinguish them from variable names. This makes it easy for anyone reading your program to know when a macro name is being used.

Macro substitutions are useful in that they make it easier to maintain programs. For example, if you know that a value, such as array size, is going to be used in several places in your program, it is better to create a macro for this value. Then, if you ever need to change this value, you simply change the macro definition. All references will be automatically changed when the program is recompiled.

### Revision Exercise

1. Discuss four fundamental data types supported by C, stating how each type is stored in memory.
2. Distinguish between a variable and a constant.
3. Suggest, with examples two ways in which constant values can be used in C expression statements.
4. Give the meaning of the following declarations;  
(i) `char name[20];`  
(ii) `int num_emp;`  
(iii) `double tax, basicpay;`  
(iv) `char response;`
5. What is the difference between a local and a global variable?
6. Write a program that computes the number of seconds in a year.  
The mass of a single molecule of water is about  $3.0 \times 10^{-23}$  grams. A quart of water is about 950 grams. Write a program that requests an amount of water in quarts and displays the number of water molecules in that amount.
7. Write a program that declares one integer variable called **num**. Give this variable the 1000 and then, using one **printf ( )** statement, display the value on the screen like this:  
**1000 is the value of num**
8. Write a program that inputs two floating-point numbers (use type **float**) and then displays their sum.
9. Write a program that computes the volume of a cube. Have the program prompt the user for each dimension.
10. Write a program that inputs an integer from the keyboard and displays its square
11. Write a program that reads your first name and surname when you enter them. Each part of your name should not be more than 12 characters. Finally, have the program redisplay your full name.

## OPERATORS

### Objectives

At the end of this chapter, the reader should be able to:

- Distinguish between operators and operands
- Use arithmetic operators
- Construct C arithmetic statements from simple formulae
- Appreciate the role of operator precedence
- Use unary increment, unary decrement and sizeof operator
- Write comparison expressions using the relational and logical operators.

### Introduction

How does a C programmer tell a program to perform a calculation, compare values and so on. This requires one to know the symbols associated with these tasks, for example '+' for addition, '>' for checking whether one value is greater than another value, '=' for assignment a value to a variable and so on.

This chapter takes you through the basic symbols so that you may be able to construct various expressions, for example writing a Mathematical formula as a C statement, in the programs you will be developing.

### Operators versus operands

An **operator** is a component or symbol of any expression that joins individual constants, variables, array elements and function references.

An **operand** is a data item that is acted upon by an operator. Some operators act upon two operands (binary operators) while others act upon only one operand (unary operators).

An operand can be a constant value, a variable name or a symbolic constant.

An expression is a combination of operators and operands.

### Examples

- (i)  $x + y$ ;  $x, y$  are operands,  $+$  is an addition operator.
- (ii)  $3 * 5$ ; 3, 5 are constant operands,  $*$  is a multiplication operator.
- (iii)  $x \% 2.5$ ;  $x, 2.5$  are operands,  $\%$  is a modulus (remainder) operator.

(iv) sizeof (int); sizeof is an operator (unary), int is an operand.

### Arithmetic operators

There are five arithmetic operators in C.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

#### Note:

- (i) There exists no exponential operators in C.
- (ii) The operands acted upon by arithmetic operators must represent numeric values, that is operands may be integers, floating point quantities or characters (since character constants represent integer values).
- (iii) The % (remainder operator) requires that both operands be integers. Thus;
  - 5 % 3
  - int x = 8;
  - int y = 6 ; x % y are valid while;
  - 8.5 % 2.0 and
  - float p = 6.3, int w = 7 ; 5 %p , p % w are invalid.
- (iv) Division of one integer quantity by another is known as an integer division. If the quotient (result of division) has a decimal part, it is truncated.
- (v) Dividing a floating point number with another floating point number, or a floating point number with an integer results to a floating point quotient .

If one or both operands represent negative values, then the addition, subtraction, multiplication, and division operators will result in values whose signs are determined by their usual rules of algebra. Thus if a, b, and c are 11, -3 and -11 respectively, then

a + b = 8  
a - b = 14  
a \* b = -33  
a / b = -3  
a % b = -2  
c % b = -2  
c / b = 3

### Examples of floating point arithmetic operators



$r1 = -0.66$ ,  $r2 = 4.50$  (operands with different signs)  
 $r1 + r2 = 3.84$   
 $r1 - r2 = -5.16$   
 $r1 * r2 = -2.97$   
 $r1 / r2 = -0.1466667$

### **Note**

- (i) If both operands are floating point types whose precision differ (e.g. a float and a double) the lower precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. (Thus if an expression has a float and a double operand, the result will be a double).
- (ii) If one operand is a floating-point type (e.g. float, double or long double) and the other is a character or integer (including short or long integer), the character or integer will be converted to the floating point type and the result will be expressed as such.
- (iii) If neither operand is a floating-point type but one is long integer, the other will be converted to long integer and the result is expressed as such. (Thus between an int and a long int, the long int will be taken).
- (iv) If neither operand is a floating type or long int, then both operands will be converted to int (if necessary) and the result will be int (compare short int and long int)

From the above, evaluate the following expressions given:

$i = 7$ ,  $f = 5.5$ ,  $c = 'w'$ . State the type of the result.

- (i)  $i + f$
- (ii)  $i + c$
- (iii)  $i + c - 'w'$
- (iv)  $(i + c) - (2 * f / 5)$

('w' has ASCII decimal value of 119)

### **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.

For example, 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.

### **Operator precedence**

The order of executing the various operations makes a significant difference in the result. C assigns each operator a precedence level. The rules are;

- (i) Multiplication and division have a higher precedence than addition and subtraction, so they are performed first.
- (ii) If operators of equal precedence; (\*, /), (+, -) share an operand, they are executed in the order in which they occur in the statement. For most operators, the order (associativity) is from left to right with the exception of the assignment (=) operator.

Consider the statement:

```
butter = 25.0 + 60.0 * n / SCALE;
Where n = 6.0 and SCALE = 2.0.
```

The order of operations is as follows;

First:         $60.0 * n = 360.0$   
              (Since \* and / are first before + but \* and / share the operand n with \* first)

Second:      $360.0 / SCALE = 180$   
              (Division follows)

Third:  $25.0 + 180 = 205.0$  (Result)  
              (+ comes last)

Note that it is possible for the programmer to set his or her own order of evaluation by putting, say, parenthesis. Whatever is enclosed in parenthesis is evaluated first.

What is the result of the above expression written as:

(25+ 60.0 \* n) / SCALE?

### ***Example: Use of operators and their precedence***

```
/* Program to demonstrate use of operators and their precedence */
#include<stdio.h >
main()
{
    int score,top;
    score = 30;
    top = score - (2*5) + 6 * (4+3) + (2+3);
    printf ("top = %d \ n" , top);
    return 0;
}
```

Try changing the order of evaluation by shifting the parenthesis and note the change in the top score.

### ***Example: Converting seconds to minutes and seconds using the % operator***

```
#include<stdio.h>
#define SEC_PER_MIN 60
main()
{
    int sec, min, sec_left;
    printf("=== CONVERTING SECONDS TO MINUTES AND SECONDS === \n\n") ;
    printf("Enter number of seconds you wish to convert\n") ;
    scanf("%d",&sec ) ;          /* Read in number of seconds */
    min = sec / SEC_PER_MIN ; / * Truncate number of seconds */
    sec_left = sec % SEC_PER_MIN ;
    printf("\n%d seconds is % d minutes,% seconds\n" ,sec,min,sec_left);
    return 0;
}
```

Analyse the sample output below. Run the program with different values.

```
=== CONVERTING SECONDS TO MINUTES AND SECONDS ===
Enter number of seconds you wish to convert
310
310 seconds is 5 minutes,10 seconds
_
```

### **The assignment operator**

The Assignment operator ( = ) is a value assigning operator. Assignment expressions take the form;

*identifier* = *expression*;

where *identifier* generally represents a variable, constant or a larger expression.

### **Examples of assignment:**

```
a = 3 ;
x = y ;
pi = 3.14;
sum = a + b ;
area_circle = pi * radius * radius;
```

### **Note**

- (i) You cannot assign a variable to a constant such as 3 = a ;
- (ii) The assignment operator = and equality operator (==) are distinctively different. The = operator assigns a value to an identifier. The equality operator (==) tests whether two expressions have the same value.
- (iii) Multiple assignments are possible e.g. a = b = 5 ; assigns the integer value 5 to both a and b.
- (iv) Assignment can be combined with +, -, /, \*, and %

### **Relational operators**

There are four relational operators in C.

- <            Less than
- <=          Less than or equal to
- >            Greater than
- > =        Greater than or equal to

Closely associated with the above are two equality operators;

- ==          Equal to
- !=          Not equal to

The above six operators form **logical expressions**.

A logical expression represents conditions that are either true (represented by integer 1) or false (represented by 0).

### **Example**

Consider a, b, c to be integers with values 1, 2,3 respectively. Note their results with relational operators below.

Expression	Result
$a < b$	1 (true)
$(a + b) > = c$	1 (true)
$(b + c) > (a + 5)$	0 (false)
$c := 3$	0 (false)
$b == 2$	1 (true)

## Logical operators

&&	Logical AND
	Logical OR
!	NOT

The two operators act upon operands that are themselves logical expressions to produce more complex conditions that are either true or false.

### Example

Suppose  $i$  is an integer whose value is 7,  $f$  is a floating point variable whose value is 5.5 and  $C$  is a character that represents the character 'w', then;

$(i > = 6) \&\& (C == 'w')$  is 1 (true)  
 $(C > = 6) || (C = 119)$  is 1 (true)  
 $(f < 11) \&\& (i > 100)$  is 0 (false)  
 $(C != 'p') || ((i + f) < = 10)$  is 1 (true)

## The Conditional operator

Conditional tests can be carried out with the conditional operator (?). A conditional expression takes the form:

**expression1 ? expression2 : expression3** and implies;

evaluate expression1. If expression1 evaluates to true ( value is 1 or non zero) then evaluate expression 2, otherwise (i.e. if expression 1 is false or zero ) , evaluate expression3.

Consider the statement  $(i < 0) ? 0 : 100$

Assuming  $i$  is an integer, the expression  $(i < 0)$  is evaluated and if it is true, then the result of the entire conditional expression is zero (0), otherwise, the result will be 100.

## Unary operators

These are operators that act on a single operand to produce a value. The operators may precede the operand or be after an operand.

### Examples

- (i) Unary minus e.g. - 700 or -x
- (ii) Incrementation operator e.g. c++
- (iii) Decrementation operator e.g. f - -
- (iv) sizeof operator e.g. sizeof( float)

### The sizeof operator

sizeof returns the size in bytes, of its operand. The operand can be a data type e.g. *sizeof (int)*, or a specific data object e.g. *sizeof n*.

If it is a name type such as int, float etc. The operand should be enclosed in parenthesis.

### Example : Demonstrating 'sizeof' operator

```
#include <stdio.h>
main()
{
    int n;
    printf("n has % d bytes; all ints have % d bytes\n" ,sizeof
n,sizeof(int)) ;
    return 0;
}
```

Run the program and analyse the results. You can later modify program to make the variable **n** a different data type and other appropriate changes, then run it.

## Revision Exercise

1. Describe with examples, four relational operators.
2. What is 'operator precedence'? Give the relative precedence of arithmetic operators.
3. Suppose a, b, c are integer variables that have been assigned the values a =8, b = 3 and c = - 5, x, y, z are floating point variables with values x =8.8, y = 3.5, z = -5.2.

Further suppose that c1, c2, c3 are character-type variables assigned the values E, 5 and ? respectively.

Determine the value of each of the following expressions:

- (i)  $a / b$
- (ii)  $2 * b + 3 * (a - c)$
- (iii)  $(a * c) \% b$
- (iv)  $(x / y) + z$
- (v)  $x \% y$
- (vi)  $2 * x / (3 * y)$
- (vii)  $c1 / c3$
- (viii)  $(c1 / c2) * c3$

4. The roots of a quadratic equation  $ax^2 + bx + c = 0$  can be evaluated as:

$$x1 = (-b + \sqrt{(b^2 - 4ac)})/2a$$

$$x2 = (-b - \sqrt{(b^2 - 4ac)})/2a$$

where  $a, b, c$  are double type variables and  $b^2 = b * b$ ,  $4ac = 4 * a * c$ ,  $2a = 2 * a$ .

Write a program that calculates the two roots  $x_1, x_2$  with double precision, and displays the roots on the screen.

Hint: We use the math function **sqrt( )** to get the square root of a value, e.g. if you want to get the square root of a value  $x$ , you write **sqrt(x)**. However, for the function **sqrt** to work, you need to have contents of the header file **math.h** in your source program. Therefore you will need to have the statement **#include<math.h>**

5. The total mechanical energy of a particle is given by:

$$\text{Energy} = mgh + \frac{1}{2}mv^2$$

Where  $m$  = mass

$g$  = acceleration due to gravity.

$h$  = height

$v$  = velocity

Write a program to calculate the energy using the formula above. The data is to be input from the keyboard.

6. Identify errors in the following programs and write error free programs.

(i)

```

#include <stdio.h>
#include <math.h>
#define PI = 3.142

VOID main[ ]
{
    /* Radius, area and perimeter are declared to be floats */
    Float r, area, perimeter,
    printf("\n "Enter the radius of the circle - > \n");
    scanf("%f",&r);

    area = PI * pow(r, 2)
    perimeter = 2 * PI * r;

    /* Result is printed after the computation */
    printf(" Area = %f Perimeter = %d /n", area, perimeter);
}

```

(ii)

```

#include <stdio.h>
Main( )
{
    cows, legs, integer;
    printf("How many cow legs did you count ? \n;
    Scanf("%c", legs);
    cows = legs/4;
    printf("That implies that there are %f cows. \n", cows)
}

```



## CONTROL STRUCTURES

### Objectives

At the end of this chapter, the reader should be able to:

- Understand the *sequence* structure
- Understand selection structures; *if*, *if..else* statements
- Select paths with the *switch..case* statement
- Understand loop structures; *for* loop, *do..while* loop and *do..while* loop
- Use *break* to exit a loop
- Know when to use the *continue* statement
- Understand the *goto* statement
- Nest *if* statements
- Create nested loops

### Introduction

Control structures represent the forms by which statements in a program are executed.

Three structures control program execution:

- Sequence
- Selection or decision structure
- Iteration or looping structure

Basically, program statements are executed in the sequence in which they appear in the program.

In reality, a logical test using logical and relational operators may require to be used in order to determine which actions to take (subsequent statements to be executed) depending on the outcome of the test. This is **selection**. For example:

```
if (score >= 50)
    printf("Pass");
else
    printf("Fail");
```

In addition, a group of statements in a program may have to be executed repeatedly until some condition is satisfied. This is known as **looping**. Suppose you were asked to display "Hello World!" five times on the screen. You would probably write the following program.

```
#include <stdio.h>
main()
{
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
    printf("Hello World!\n");
}
```

Indeed, this does exactly what was asked. But suppose you were asked to do the same job for 100 Hello Worlds or, if you're still willing to type that much code, maybe 1,000 Hello Worlds. You will realise that that this is not a sensible method!

What you really need is some way of repeating the printf statements without having to write it out each time. This is the purpose of a loop.

### The if statement

The *if* statement provides a junction at which the program has to select which path to follow. The general form is :

```
if(expression)
    statement;
```

If *expression* is true (i.e. non zero) , the *statement* is executed, otherwise it is skipped. Normally the expression is a relational expression that compares the magnitude of two quantities ( For example  $x > y$  or  $c == 6$ )

### Examples

- (i) 

```
if (x<y)
    printf("x is less than y");
```
- (ii) 

```
if (salary >500)
    tax-amount = salary * 1.5;
```
- (iii) 

```
if(balance<1000 || status =='R')
    print ("Balance = %f", balance);
```

The statement in the if structure can be a single statement or a block (compound statement).

If the statement is a block (of statements), it must be marked off by braces.

```
if(expression)
```

```

{
    block of statements;
}

```

### **Example**

```

if(salary>5000)
{
    tax_amt = salary *1.5;
    printf("Tax charged is %f", tax_amt);
}

```

### **The if - else statement**

The if else statement lets the programmer choose between two statements as opposed to the simple if statement which gives you the choice of executing a statement (possibly compound) or skipping it.

The general form is:

```

if (expression)
    statement1;
else
    statement2;

```

If expression is true, statement1 is executed. If expression is false, the single statement following the else (statement2) is executed. The statements can be simple or compound.

**Note:** Indentation is not required but it is a standard style of programming.

### **Example**

```

if(x >=0)
{
    printf("let us increment x:\n");
    x++;
}
else
    printf("x < 0 \n");

```

### **The else .. if statement**

This is use when many choices are involved.

The general form is:

```
        if (expression)
            statement;
        else if (expression)
            statement;
        else if (expression)
            statement;
        else
            statement;
```

(Braces still apply for block statements)

### **Example**

```
if(sale_amount>=10000)
    Disc=sal_amt*0.10;                                /*ten percent/
else if (sal_amt>=5000&&sal_amt<1000 )
    printf("The discount is %f",sal_amt*0.07 ); /*seven percent */
else if(sal_amt=3000&&sal_amt<5000)
{
    Disc = sal_amt * 0.05;                            /* five percent */
    printf ( " The discount is %f " , Disc ) ;
}
else
    printf("The discount is 0") ;
```

### **Example : Determining grade category**

```
#include<stdio.h >
#include<string.h >
main()
{
    int marks;
    char grade [15];
    printf ("Enter the students marks  \n");
    scanf( "%d ",&marks ) ;
```

```

if ( marks > =75  &&  marks <=100)
{
    strcpy(grade, "Distinction"); /*Copy the string to the grade */
    printf("The grade is %s" , grade);
}

else if( marks > = 60 &&  marks < 75 )
{
    strcpy(grade, "Credit");
    printf("The grade is %  s" , grade );
}

else if(marks>=50  &&  marks<60)
{
    strcpy(grade, "Pass");
    printf("The grade is %  s" , grade );
}

else if (marks>=0 && marks<50)
{
    strcpy(grade, "Fail");
    printf ("The grade is %  s" , grade)  ;
}
else
    printf("The mark is impossible!" );
return 0;
}

```

## The switch ..case statements

The *switch..case* statements can be used in place of the *if - else* statements when there are several choices to be made.

### **Example: Demonstrating the ‘switch’ structure**

```

#include<stdio.h>
main()
{
    int choice;
    printf("Enter a number of your choice  ");
    scanf(" %d", &choice);
    if (choice >=1 && choice <=9)          /* Range of choice numbers */
    switch (choice)
    {
        /* Begin of switch */
        case 1:                             /* label 1 */
            printf("\n You typed  1");
            break;
        case 2:                             /* label 2 */
            printf("\n You typed 2");

```

```

        break;
    case 3:                                /* label 3 */
        printf("\n You typed 3");
        break;
    case 4:                                /* label 4 */
        printf( " \n You typed 4");
        break;
    default:
        printf("There is no match in your choice");
}                                           /* End of switch */
else
    printf("Your choice is out of range");
return (0);
}                                           /* End of main */

```

### ***Explanation***

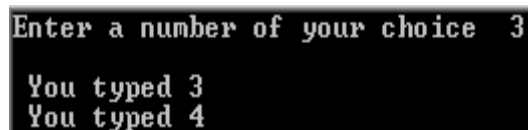
The expression in the parenthesis following the switch is evaluated. In the example above, it has whatever value we entered as our choice.

Then the program scans a list of labels (case 1, case 2,... case 4) until it finds one that matches the one that is in parenthesis following the switch statement.

If there is no match, the program moves to the line labeled default, otherwise the program proceeds to the statements following the switch.

The break statement causes the program to break out of the switch and skip to the next statement after the switch. Without the break statement, every statement from the matched label to the end of the switch will be processed.

For example if we remove all the break statements from the program and then run the program using the number 3 we will have the following exchange.



```

Enter a number of your choice 3
You typed 3
You typed 4

```

The structure of a switch..case statement is as follows:

```

switch (integer expression)
{
    case constant 1:
        statement; optional

```

```

    case constant 2:
        statement; optional
        .....

    default: (optional)
        statement; (optional)
}

```

### **Note**

- (i) The switch labels (case labels) must be type int (including char) constants or constant expression.
- (ii) You cannot use a variable for an expression for a label expression.
- (iii) The expressions in the parenthesis should be one with an integer value. (again including type char)

### **Example: Demonstrating the 'switch' structure**

```

#include<stdio.h>
main()
{
    char ch;
    printf("Give me a letter of the alphabet \n");
    printf("An animal beginning with letter");
    printf ("is displayed \n ");
    scanf("%c", &ch);
    if (ch>='a' && ch<='z')      /*lowercase letters only */
    switch (ch)
    {
        /*begin of switch*/
        case 'a':
            printf("Alligator , Australian aquatic animal \n");
            break;
        case 'b':
            printf("Barbirusa, a wild pig of Malaysia \n");
            break;
        case 'c':
            printf("Coati, baboon like animal \n");
            break;
        case 'd':
            printf("Desman, aquatic mole-like creature \n");
            break;
        default:
            printf(" That is a stumper! \n")
    }
    else
        printf("I only recognize lowercase letters.\n");
}

```

```
    return 0;
}    /* End of main */
```

## Looping

C supports three loop versions:

- *while* loop
- *do while* loop
- *for* loop.

### The 'while' loop

The while statement is used to carry out looping instructions where a group of instructions executed repeatedly until some conditions are satisfied.

General form:

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

The statement will be executed as long as the expression is true. The statement can be a single or compound.

```
/* counter.c */
/* Displays the digits 1 through 9 */
main()
{
    int digit=0;        /* Initialisation */
    while (digit<=9)
    {
        printf("%d \n", digit);
        digit++;
    }
    return 0;
}
```

### **Example: Calculating the average of *n* numbers using a 'while' loop**

Algorithm:

- (i) Initialise an integer count variable to 1. It will be used as a loop counter.
- (ii) Assign a value of 0 to the floating-point sum.
- (iii) Read in the variable for *n* (number of values)
- (iv) Carry out the following repeatedly (as long as the count is less or equal to *n*).
  - (a) Read in a number, say *x*.



- (b) Add the value of x to current value of sum.
- (c) Increase the value of count by 1.
- (v) Calculate the average: Divide the value of sum by n.
- (vi) Write out the calculated value of average.

**Program code:**

```
/* To add numbers and compute the average */
#include<stdio.h>
main()
{
    int n, count = 1;
    float x, average, sum=0.0;
    /* initialise and read in a value of n */
    printf("How many numbers? ");
    scanf("%d", &n);
    /*Read in the number */
    while (count<=n)
    {
        printf("x = ");
        scanf("%f", &x);
        sum+=x;
        count++;
    }
    /* Calculate the average and display the answer */
    average = sum/n;
    printf("\n The average is %f \n", average);
    return 0;
}
```

Note that using the while loop, the loop test is carried out at the beginning of each loop pass.

**The 'do .. while' loop**

It is used when the loop condition is executed at the end of each loop pass.

It takes the form:

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

The statement (simple or compound) will be executed repeatedly as long as the value of the expression is true. (i.e. non zero).

Notice that since the test comes at the end, the loop body (statement) must be executed at least once.

Rewriting the program that counts from 0 to 9, using the *do while* loop:

```
/* counter1.c */
/* Displays the digits 1 through 9 */
main()
{
    int digit=0; /* Initialisation */
    do
    {
        printf("%d \n", digit);
        digit++;
    } while (digit<=9);
    return 0;
}
```

### **Exercise**

Rewrite the program that computes the average of n numbers using the do while loop.

### **The 'for' loop**

This is the most commonly used looping statement in C.

It takes the form:

```
for (expression1;expression2;expression3)
    statement;
```

where:

*expression1* is used to initialize some parameter (called an index). The index controls the loop action. It is usually an assignment operator.

*expression2* is a test expression, usually comparing the initialised index in *expression1* to some maximum or minimum value.

*expression3* is used to alter the value of the parameter index initially assigned by *expression* and is usually a unary expression or assignment operator;

### **Example**

```
for(int k=0;k<=5; k++)
    printf(k = %d \n", k);
```

The code should print integers 0 through to 5, each on a different line.

**Example: Counting 0 to 9 using a 'for' loop**

```
/* Displays the digits 1 through 9 */
#include<stdio.h>
main()
{
    int digit;
    for(digit=0;digit<=9; digit++)
        printf("%d \n" , digit);
    return 0;
}
```

**Example: Averaging a set of numbers using a 'for' loop**

```
/* average.c */
/* To add numbers and compute the average */
#include<stdio.h>
main()
{
    int n, count;
    float x, average, sum=0.0;.

    /* initialise and read in a value of n */
    printf("How many numbers? ");
    scanf("%d", &n);
    /*Read in the number */
    for(count=1;count<=n;count++){
        printf("x = ");
        scanf("%f", &x);
        sum+=x;
    }
    /* Calculate the average and display the answer */
    average = sum/n;
    printf("\n The average is %f \n", average);
    return 0;
}
```

**Example: Table of cubes**

```
/ Using a loop to make a table of cubes */
#include<stdio.h>
main()
{
    int number;
    printf("\n          n cubed ");
    for(num=1; num<=6;num++)
        printf("%5d  %5d \n", num, num*num*num);
    return 0;
}
```

```
}
```

Also note the following points about the **for** structure.

- You can count down using the decrement operator
- You can count by any number you wish; two's threes, etc.
- You can test some condition other than the number of operators.
- A quantity can increase geometrically instead of arithmetically.

### Using break and continue statements in loops

The **break** statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the **break** statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. The **break** statement can be used with all three of C's loops. You can have as many statements within a loop as you desire. It is generally best to use the **break** for special purposes, not as your normal loop exit. **break** is also used with **case** statements as explained earlier.

The **continue** statement is somewhat the opposite of the **break** statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. In **while** and **do-while** loops, a **continue** statement will cause control to go directly to the test condition and then continue the looping process. In the case of the **for** loop, the increment part of the loop continues. One good use of **continue** is to restart a statement sequence when an error occurs.

Below is an example using the continue statements. Only even integers between 0 and 100 are printed.

```
#include <stdio.h>
main()
{
    int x ;
    for (x=0 ; x<=100 ; x++){
        if(x%2==0) continue;
        printf("%d\n" , x);
    }
}
```

Here's an example of use for the **break** statement. The program **for** loop is infinite. The loop is terminated when the break statement is executed, which happens upon the user inputting the value 10 to the variable t.

```

#include <stdio.h>

main()
{
    int t ;

    for ( ; ; )
    {
        printf("Value of t: ");
        scanf("%d" , &t) ;
        if ( t==10 )
            break ;
    }
    printf("End of an infinite loop...\n");
}

```

## The 'goto' statement

This is another control flow statement. It takes the form `goto labelname;`

### Example

```

goto part2;

part2: printf("programming in c"\n");

```

In principle you never need to use *goto* in a C statement. The *if* construct can be used in its place as shown below.

### Alternative 1

```

if (a>14)
    goto a;
sheds=2;
goto b;
a: sheds=3;
b: k=2 * sheds;

```

### Alternative 2

```

if (a>14)
    sheds=3;
else
    sheds=2;
k=2*sheds;

```

## Nesting statements

It is possible to embed (place one inside another) control structures, particularly the `if` and `for` statements.

### ***Nested 'if' statement***

It is used whenever choosing a particular selection leads to an additional choice.

#### ***Example***

```
if (number>6)
    if (number<12)
        printf("You are very close to the target!");
else
    printf("Sorry, you lose!");
```

### ***Nested 'for' statement***

Suppose we want to calculate the average of several consecutive lists of numbers, if we know in advance how many lists are to be averaged.

#### ***Example: Nested 'for' statements***

```
/* Calculate the averages of several different lists of number */
#include<stdio.h>
main()
{
    int n, count, loops, loopcount;
    float x, average, sum;
    /*Read in the number of loops */
    printf("How many lists? ");
    scanf("%d", &loops);
    /*Outer loop processes each list of numbers */
    for (loopcount=1; loopcount<=loops; loopcount++)
    {
        /* initialise sum and read in a value of n */
        sum=0.0;
        printf("List number %d \n How many numbers ? ",loopcount);
        scanf("%d", &n);
        /*Read in the numbers */
        for(count=1;count<=n; count++)
        {
```

```

        printf("x = ");
        scanf("%f", &x);
        sum+=x;
    }          /* End of inner loop */
    /* Calculate the average and display the answer */
    average = sum/n;
    printf("\n The average is %f \n", average);
}          /*End of outer loop */
return 0;
}

```

## Revision Exercise

1. A retail shop offers discounts to its customers according to the following rules:

Purchase Amount  $\geq$  Ksh. 10,000 - Give 10% discount on the amount.  
 Ksh. 5, 000  $\leq$  Purchase Amount  $<$  Ksh. 10,000 - Give 5% discount on the amount.  
 Ksh. 3, 000  $\leq$  Purchase Amount  $<$  Ksh. 5,000 - Give 3% discount on the amount.  
 0  $>$  Purchase Amount  $<$  Ksh. 3,000 - Pay full amount.

Write a program that asks for the customer's purchase amount, then uses *if* statements to recommend the appropriate payable amount. The program should cater for negative purchase amounts and display the payable amount in each case.

2. In what circumstance is the *continue* statement used in a C program?
3. Using a nested if statement, write a program that prompts the user for a number and then reports if the number is positive, zero or negative.
4. Write a *while* loop that will calculate the sum of every fourth integer, beginning with the integer 3 (that is calculate the sum 3 + 7 +11 + 15 + ...) for all integers that are less than 30.
5. Write a program that prints only the odd numbers between 1 and 100. Use a *for* loop that looks this:

```
for(i=1; i<101; i++)....
```

Use a **continue** statement to avoid printing even numbers.

6. Write a program that computes the area of a circle, rectangle or triangle using an *if..else if* ladder.
7. Write a program that requests two numbers and then displays their sum or product, depending upon what the user selects.
8. Write a program that asks the user for an integer and then tells the user if that number is even or odd. (Hint: Use C's modulus operator %)





## FUNCTIONS

### Objectives

At the end of this chapter, the reader should be able to:

- Explain why functions are important
- Distinguish between standard library functions and user-defined functions
- Create a function definition
- Make function calls in a program
- Pass values in function calls
- Use prototyping in function-based programs.
- Write programs that use recursive functions

### Introduction

A function is a self-contained program segment that carries out some specific well - defined task. Every C program consists of one or more functions. One of these functions must be called main. Execution of the program will always begin by carrying out the instructions in main. Additional functions will be subordinate to main, and perhaps to one another.

If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another.

### Why use functions?

The use of programmer-defined functions allows a large program to be broken down to a number of smaller, self-contained components each of which has some unique identifiable purpose. Thus a C program can be modularized through the intelligent use of functions.

There are several advantages to this modular approach to program development.

- *Elimination of code redundancy*

For example many programs require that a particular group of instructions be accessed repeatedly from several different places in the program. The repeated instructions can be placed within a single function which can then be accessed whenever it is needed. Moreover a different set of data can be transferred to the function each time it is

accessed . Thus the use of a function *eliminates the need for redundant programming of the same instructions*.

- *Clear program logic*

Equally important is the *logical clarity* resulting from the decomposition of a program into several concise functions where each function represents some well-defined part of the overall problem. Such programs are easier to write and debug and their logical structure is more clear than programs which lack this type of structure. This is especially true of lengthy, complicated programs. Most C programs are therefore modularised in this manner, even though they may not involve repeated execution of some task. In fact the decomposition of a program into individual program modules is generally considered to be good programming practice.

- *Program libraries*

This use of functions also enables a programmer to build a *customized library of frequently used routines* or of routines containing system-dependent features. Each routine can be programmed as a separate function and stored within a special library file. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process. Hence a single function can be utilised by many different programs.

## **Types of functions**

Functions can be classified into two:

- (i) Standard library functions and
- (ii) User defined functions.

### ***Standard library functions***

Every compiler comes with some standard predefined functions which are available for your use. These are mostly input/output functions, character and string manipulation functions, and math functions. Prototypes are defined for you by the compiler writer for all of the functions that are included with your compiler. A few minutes spent studying your compiler's Reference Guide will give you an insight into where the prototypes are defined for each of the functions.

In addition, most compilers have additional functions predefined that are not standard but allow the programmer to get the most out of a particular computer. One example of a library of this type is the *CONIO* library which is provided as part of the Turbo C package. The *CONIO* library provides functions that allow you to position output on the monitor, change the color of text written to the monitor and a number of other screen (console) based operations.

Most of these libraries can be used using the `#include` statement

### ***User – defined functions***

This is a function created and customized by a programmer for use within a given program.

### **Defining a function**

The general form of a C function is:

```
type function_name(parameter- list)
{
    statements
}
```

`type` specifies the return type of a function. A function can return any type of data. If no data type specifier is present, the C compiler assumes that the function is returning an integer. In other words *int* is the default type when no type specifier is present. However, when a function returns a type other than *int*, it must be explicitly declared.

For example, this function returns a *double*.

```
/* Compute the area of a rectangle */
double rect_area(double side1, double side 2)
{
    return(side1*side2);
}
```

`function_name` represents any meaningful descriptive tag you will assign to the function for example `rect_area` above.

`parameter-list` represents the variables used in the function body (if any). If more than one, they are listed inside the parenthesis with each parameter (also known as an ***argument***) preceded by its data type as it is done in a declaration statement. Hence in the previous example, *side1* and *side2* are the parameters of the *rect\_area* function.

`statements` is the sequence of statements that make up the function body. If the function is expected to return some value, the sequence will have a *return* statement (discussed shortly), otherwise (*void* type functions), there wont be any need for a return statement.

The arguments are called **formal arguments** because they represent the names of data items that are transferred into the function from the calling portion of the program. They are also known as parameters or formal parameters.

The corresponding arguments in the function reference are called actual arguments since they define the data items that are actually transferred). Each formal argument must be of the same data type as the data item it receives from the calling portion of the program.

Information is returned from the function to the calling portion of the program via a **return** statement. The return statement also causes the program logic to return to the point from which the function was accessed.

In general terms the return statement is written as:

```
return (expression);
```

Only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion via return.

### ***Example : Factorial of an integer n***

The factorial of a positive integer quantity  $n$  is defined as  $n! = 1 * 2 * 3 * \dots * (n - 1) * n$ . Thus,  $2! = 1 * 2 = 2$ ;  $3! = 1 * 2 * 3 = 6$ ;  $4! = 1 * 2 * 3 * 4 = 24$ ; and so on.

The function shown below calculates the factorial of a given positive integer  $n$ . The factorial is returned as a long integer quantity, since factorials grow in magnitude very rapidly as  $n$  increases.

```
long int factorial (int n) /*Calculate the factorial of n */
{
    int i;
    long int prod = 1;
    if (n >1 )
        for(i =2; i <=n; i++)
            prod * = i;
    return(prod);
}
```

Notice the **long int** specification that is included in the first line of the function definition. The local variable **prod** is declared to be a long integer within the function. It is

assigned an initial value of 1 though its value is recalculated within a for loop. The final value of **prod** which is returned by the function represents the desired value of n factorial (n!).

If the data type specified in the first line is inconsistent with the expression appearing in the return statement, the compiler will attempt to convert the quantity represented by the expression to the data type specified in the first line. This could result in a compilation error or it may involve a partial loss in data (due to truncation). Inconsistency of this type should be avoided at all costs.

The keyword **void** can be used as a type specifier when defining a function that does not return anything or when the function definition does not include any arguments. The presence of this keyword is not mandatory but it is good programming practice to make use of this feature.

Consider a function that accepts two integer quantities, determines the larger of the two and displays it (the larger one). This function does not return anything to the calling portion. Therefore the function can be written as;

```
void maximum (int x, int y)
{
    int z;
    z = (x >= y)? x : y;
    printf("\n \n maximum value   = %d " , z),
}
```

## Accessing a function

A function can be accessed by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas. If the function call does not require any arguments an empty pair of parenthesis must follow the name of the function. The function call may be part of a simple expression (such as an assignment statement), or it may be one of the operands within an expression.

The arguments appearing in the function call are referred to as **actual arguments** in contrast to the formal arguments that appear in the first line of the function definition. (They are also known as actual parameters or arguments). In a normal function call, there will be one actual argument for each formal argument. Each actual argument must be of the same data type as its corresponding formal argument. Remember that it is the value of each actual argument that is transferred into the function and assigned into the corresponding formal argument.

There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. Within each function call however the actual arguments must correspond to the formal arguments in the functions definition; i.e. the number of actual arguments must be the same as the number of formal arguments and each actual argument must be of the same data type as its corresponding formal argument.

**Example: *Determining the maximum of two integers (Complete program)***

The following program determines the largest of three integers quantities. The program makes use of a function that determines the larger of two integer quantities. The overall strategy is to determine the larger of the first two quantities and then compare the value with the third quantity. The largest quantity is then displayed by the main part of the program.

```
/*Determine the largest of the three integer quantities*/
#include<stdio.h>
int maximum (int x, int y) /*Determine the larger of two quantities*/
{
    int z;
    z = (x >= y)? x : y;
    return(z);
}
main()
{
    int a , b , c ,d;
    /*read the integer quantities*/
    printf("\n a = ");
    scanf("%d", &a);
    printf("\n b = ");
    scanf("%d", &b);
    printf("\n c = ");
    scanf("%d", &c);
    /* Calculate and display the maximum value */
    d = maximum (a, b);
    printf ("\n \n maximum = %d ", maximum (c ,d));
    return 0;
}
```

The function **maximum** is accessed from two different places in **main**. In the first call to maximum, the actual arguments are the variables **a** and **b** whereas in the second call, the arguments are **c** and **d**. (d is a temporary variable representing the maximum value of a and b).

Note the two statements in main that access maximum, i.e.

```
d = maximum (a, b);
printf(" \n \n maximum = %d ", maximum (c, d));
```

A single statement can replace these two statements, for example:

```
printf (" \n\n maximum = %d " maximum(c, maximum (a, b)));
```

In this statement, we see that one of the calls to `maximum` is an argument for the other call. Thus the calls are embedded one within the other and the intermediary variable `d` is not required. Such embedded functions calls are permissible though their logic may be unclear. Hence they should generally be avoided by beginning programmers.

## Function prototypes

A prototype is a model (or representation) of the real thing.

All functions in a *main* program must be prototyped if their code definition appears after the *main* function. While a programmer is free to place function code definitions for functions used in the *main* program before the *main* program (see the previous example), programmers prefer a top down approach in which *main* appears ahead of the programmer-defined function definition.

In such a situation, the function call (within *main*) will precede the function definition. This can be confusing to the compiler unless the compiler is first alerted to the fact that the function being accessed will be defined later in the program. A function prototype is used for this purpose.

Function prototypes are usually written at the beginning of a program ahead of any programmer-defined function (including *main*) The general form of a function prototype is;

```
type  function-name (parameter-list);
```

Where `type` represents the type of the item that is returned by the function, `function-name` represents the name of the function, `parameter-list` represents the types and names of the arguments used in executing the function.

For example, the statement `void square(int number);` is a prototype for the function `square`

Note that a function prototype resembles the first line of a function definition (although a definition prototype ends with a semicolon).

The names of the argument(s) can be omitted (though it is not a good idea to do so). However the arguments data types are essential. Hence, the prototype for the function `square` may also be written as `void square(int);`

Although function prototypes are not mandatory in C, they are however desirable because they facilitate error checking between the calls to a function and the corresponding function definition. Since a prototype is a model of a function, the compiler uses it to check each of your calls to the function and determine if you have used the correct number of arguments in the function call and if they are of the correct type.

### ***Example: Factorial of an integer n***

Here is a complete program to calculate the factorial of a positive integer quantity. The program utilises the factorial function defined earlier in the chapter. Note that the function prototype precedes **main**.

```
/*Calculate the factorial of an integer quantity*/
#include<stdio.h>
long int factorial (int n);
main()
{
    int n;
    /* read in the integer quantity */
    printf ("\n n = ");
    scanf ("%d ", &n);
    /* Calculate and display the factorial*/
    printf ("\n n =%d", factorial (n));
    return 0;
}
/*Calculate the factorial of n*/
long int factorial (int n)
{
    int i;
    long int prod=1;
    if (n >1)
    for( i=2; i<=n; i ++)
        prod *= i;
    return (prod);
}
```



## Recursion

Recursion is the process by which a function calls itself repeatedly until a special condition is satisfied.

For a problem to be solved using recursion, two conditions must be satisfied. These are:

- (i) The solution must be written in a recursive form i.e. it should be possible to express the solution in form of itself.
- (ii) There must be a stopping case or a simple solution. This is the terminating condition.

We reuse the factorial function to illustrate this.

The factorial of any possible integer can be expressed as;

$$n! = n * (n-1) * (n-2) * \dots * 1.$$

e.g.  $5! = 5 * 4 * 3 * 2 * 1.$

However we can rewrite the expression as;  $5! = 5 * 4!$

Or generally,

$$n! = n * (n-1)! \quad (\text{Recursive statement})$$

This is to say that in the factorial example, the calculation of n is expressed in form of a previous result (condition (i) is satisfied).

Secondly  $1! = 1$  by definition, therefore condition (ii) is satisfied i.e. a stopping case.

**Example: Using the factorial function in recursive form.**

```
#include<stdio.h>
long int factorial (int n);    /*factorial function  prototype*/
main()
{
    int n;
    /*Read in the integer quantity*/
    printf ("n = " );
    scanf ("%d ", &n);
    /*Calculate and display the factorial*/
    printf ("n! =%d \n",  factorial (n));
    return 0;
}
/* Function definition */
long int factorial (int n)
{
    if (n <=1)                /*terminating condition*/
```

```
        return (1);
    else
        return (n * factorial (n-1));
}
```

The functional factorial calls itself recursively with an actual argument that decreases in magnitude for each successive call. The recursive call terminates when the value of the actual argument becomes equal to 1.

Consider the following two program examples:

```
#include<stdio.h>

/* define the prototype for the function used in this program. */
void count_dn(int count);
void main()
{
    int index;
    index = 8;
    count_dn(index);
}

/* -- Function definition ----- */
void count_dn(int count)
{
    count--;
    printf("The value of the count is %d\n",count);
    if(count > 0)
        count_dn(count);
    printf("Now the count is %d\n",count);
}
```

The output of the program is:

```
The value of the count is 7
The value of the count is 6
The value of the count is 5
The value of the count is 4
The value of the count is 3
The value of the count is 2
The value of the count is 1
The value of the count is 0
Now the count is 0
Now the count is 1
Now the count is 2
Now the count is 3
Now the count is 4
Now the count is 5
Now the count is 6
Now the count is 7
```

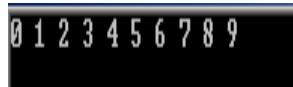
In this program the variable `index` is set to 8, and is used as the argument to the function `count_dn`. The function simply decrements the variable, prints it out in a message, and if the variable is not zero, it calls itself, where it decrements it again, prints it, etc. etc. etc. Finally, the variable will reach zero, and the function will not call itself again. Instead, it will return to the prior time it called itself, and return again, until finally it will return to the `main` program.

For purposes of understanding you can think of it as having 8 copies of the function `count_dn` available and it simply called all of them one at a time, keeping track of which copy it was in at any given time.

```
#include <stdio.h>
void recurse(int i);
void main()
{
    recurse(0);
}

/* -- Function definition ----- */
void recurse(int i)
{
    if(i<10)
    {
        printf("%d ", i);
        recurse(i+1);
    }
}
```

The output of the program is



```
0 1 2 3 4 5 6 7 8 9
```

The `recurse( )` function is first called with 0. This is `recurse's( )` first activation. Since 0 is less than 10, it is printed. After printing it, the function is called again with `(i+1)`, i.e. 1. The process repeats until `recurse( )` is called with the value 10. Therefore the numbers are printed in ascending order from 0 to 9.

## Revision Exercise

1. Explain the meaning of each of the following function prototypes

- (i) `int f(int a);`
- (ii) `void f(long a, short b, unsigned c);`

(iii)      `char f(void);`

2. Each of the following is the first line of a function definition. Explain the meaning of each.

(i)      `float f(float a, float b)`

(ii)      `long f(long a)`

3. Write appropriate function prototypes for each of the following skeletal outlines shown below.

(a)

```
main()
{
    int a, b, c;
    .....
    c =function1(a,b);
    .....
}

int fucntion1(int x, int y)
{
    int z;
    .....
    z = .....
    return(z);
}
```

(b)

```
main()
{
    int a;
    float b;
    long int c;
    .....
    c = funct1(a,b);
    .....
}

int func1(int x, float y)
{
    long int z;
    .....
    .....
    z = ...
    return (z);
}
```

4. Describe the output generated by the followed program.

```
#include<stdio.h>
int func(int count);
main()
{
    int a,count;
    for (count=1; count<= 10; count++)
    {
        a = func(count);
        printf("%d",a);
    }
    return 0;
}
int func(int x)
{
    int y;
    y = x * x;
    return(y);
}
```

5. (a) What is a recursive function?
- (b) State two conditions that must be satisfied in order to solve a problem using recursion.
6. Write a program that creates a function, called **avg()** that reads ten floating-point numbers entered by the user and returns their average.
7. Write a program that uses a function called **hypot()** that returns the length of the hypotenuse of a right angled triangle when passed the length of the two opposing sides. Have the function return a **double** value. The type of the parameters must be **double** as well Demonstrate the function in a program. (The Pythagorean theorem states that the sum of the squares of the two opposing sides equals the square of the hypotenuse).
8. Write a program that uses function called **Lowest** that takes three float values and computes the lowest of the three values.
9. The Future Value (**fv**) earned from an investment amount (**p**), at a constant interest rate (**r**), over **n** years is calculated using the formula:

$$fv = p * \frac{1}{(1 + r)^n}$$

**Required:**

Write a program that reads the investment amount, rate and number of years, calculates and prints the future value.

**Hint:** Implement this program using a function **getfv( )**, which accepts three parameters (**p**, **r** and **n**), calculates the **fv** and returns the value of **fv**.

## ARRAYS

### Objectives

At the end of this chapter, the reader should be able to:

- Define an array
- Declare one-dimensional arrays
- Initialise arrays
- Write one-dimensional array programs
- Write two-dimensional arrays
- Use basic string functions.

### Introduction

It is often necessary to store data items that have common characteristics in a form that supports convenient access and processing e.g. a set of marks for a known number of students, a list of prices, stock quantities, etc. Arrays provide this facility.

### Array defined

An array is a homogeneous ordered set of elements or a series of data objects of the same type stored sequentially. That is to say that an array has the following characteristics;

- Items share a name
- Items can be of any simple data type e.g. char, float, int, double.
- Individual elements are accessed using an integer index whose value ranges from 0 to the value of the array size.

An array of 10 student ages (stored as integers)

22	19	20	21	21	22	23	10	19	20
----	----	----	----	----	----	----	----	----	----

An array of 5 characters in an employee's name

O	K	O	T	H
---	---	---	---	---

### Example

```
float debts [20];
```

This statement declares `debts` as an array of 20 elements. The first element is called `debts[0]`, the second `debts[1]`, - - - , `debts[19]`.

Because the array is declared as type float, each element can be assigned a float value such as `debts[5] = 32.54;`

Other examples;

```
int emp_no[15]; /*An array to hold 15 integer employee numbers*/
char alpha [26]; /*an array to hold 26 characters */
```

## Declaring arrays

An array definition comprises;

- (i) Data type
- (ii) Array name
- (iii) Arraysize expression (usually a positive integer or symbolic constant). This is enclosed in square brackets.

For a one-dimensional array (discussed shortly), we use the general form:

```
type array_name[size];
```

Where `type` is a valid C data type, `array_name` is the name of the array, and `size` specifies the number of elements in the array. For example, to declare an integer array with 10 elements called `mylist`, use the statement:

```
int mylist[10];
```

An array element is accessed by indexing the array using the number of the element. In C all arrays begin with 0. This means that if you want to access the first element in the array, use 0 for the index. To index an array, specify the index of the element you want inside square brackets. For example, the following accesses the third element in the array `mylist`.

```
mylist[2]
```

The last element in array is `array[size-1]`. The last element in `mylist` array is therefore `mylist[9]`.

To assign an array element a value, put the array reference on the left side of an assignment statement. For example, this gives the first element in `mylist` the value 200.

```
mylist[0] = 200;
```

## Array dimensions

An array's dimension is the number of indices required to manipulate the elements of the array.



## One-dimensional arrays

This is a single list row (or column) of values. For this reason, only a single index is required to access its values.

C stores one-dimensional arrays on one contiguous memory location with the first element at the lowest address. For example, after this fragment executes,

```
int i[5];
int j;

for(j=0;j<5;j++)
    i[j]= j+1;
```

array i will look like this:

	0	1	2	3	4
i	1	2	3	4	5

You may use the value of an array element anywhere you would use a simple variable or constant. For example, the following program loads the `sqr`s array with the squares of the numbers 1 through 10 and then displays them.

```
#include<stdio.h>
void main()
{
    int sqrs[10];
    int i;

    for(i=1;i<11;i++)
        sqrs[i-1]= i*i;

    for(i=0;i<10;i++)
        printf("%d ", sqrs[i]);
}
```

The program output will like this:



When you want to use `scanf( )` to input a numeric value into an array element, simply put the `&` in front of the array name. For example, this `scanf( )` reads an integer into `mylist[9]`.

```
scanf("%d", &mylist[9]);
```

When using arrays, you should note the following:

1. Single operations which involve entire arrays are not permitted in C. Operations such as assignment, comparison operators, sorting etc must be carried out on an element-by-element basis. This is usually accomplished within a loop where each pass through the loop is used to process one array element. The number of passes through the loops will therefore be equal to the number of array elements to be processed. The above program example demonstrates this.
2. C does not perform any bounds checking on array indexes. This means that it is possible to overrun the end of an array. For example, if an array called `a` is declared as having 5 elements, the compiler will still let you access the (nonexistent) tenth element with a statement like `a[9]`. Of course attempting to access nonexistent elements will generally have disastrous results, often causing the program-even the computer to crash. It is up to you, the programmer to ensure that that ends of arrays are never overrun.
3. In C, you must not assign one entire array to another. For example, this fragment is incorrect.

```
char a1[10],a2[10];  
.  
.  
.  
  
a2 = a1; /* this is wrong */
```

If you wish to copy the values of all the elements of one array to another, you must do so by copying each element separately. That is:

```
a2[0]=a1[0];  
a2[0]=a1[0];  
a2[0]=a1[0];  
.  
.  
.  
a2[9]=a1[9];
```

Arrays are very useful in programming when lists of information need to be managed. For example, the following program reads the noon day temperature for each day of a month and then reports the month's average temperature, as well as its hottest and coolest days.

```

#include<stdio.h>
void main()
{
    int temp[31],i,min,max,avg;
    int days;
    printf("How many days in the month      ? ");
    scanf("%d", &days);
    for(i=0;i<days;i++)
    {
        printf("Enter noonday temperature for day %d: "; i +1);
        scanf("%d", &temp[i]);
    }
    /* find average */
    avg=0;
    for(i=0;i<days;i++) avg=avg+temp[i];
    printf("Average temperature: %d", avg/days);

    /* find min and max */
    min = 200; /* Initialise min and max */
    max = 0;
    for(i=0;i<days;i++)
    {
        if (min>temp[i])
            min = temp[i];
        if (max<temp[i])
            max= temp[i];
    }
    printf("Minimum temperature: %d\n", min);
    printf("Maximum temperature: %d\n", max);
}

```

Below is a sample output:

```

How many days in the month      ? 10
Enter noonday temperature for day 1: 23
Enter noonday temperature for day 2: 24
Enter noonday temperature for day 3: 25
Enter noonday temperature for day 4: 22
Enter noonday temperature for day 5: 27
Enter noonday temperature for day 6: 26
Enter noonday temperature for day 7: 24
Enter noonday temperature for day 8: 25
Enter noonday temperature for day 9: 23
Enter noonday temperature for day 10: 25
Average temperature: 24 Minimum temperature: 22
Maximum temperature: 27

```

## Two – dimensional arrays

An m by n two-dimensional array can be thought of as a table of values having m rows and n columns. The number of elements can be known by the product of m (rows) and n(columns).

Its declaration takes the form

```
type array_name[m][n];
```

where `type` is the data type, `array_name` is the name of the array, `m` is the number of rows and `n` is the number of columns.

A conceptual view of a 4 x 5 two-dimensional array is as follows:

	0	1	2	3	4
0					
1					
2					
3					

### ***Examples of two-dimensional array declarations***

```
float table[50][50];  
char page[24][80];  
Static double records[100][60][255];
```

Two-dimensional arrays are used like one-dimensional ones. For example, the following program loads a 4 x 5 array with the products of the indices, then displays the array in row, column format.

```
#include<stdio.h>  
void main()  
{  
    int twod[4][5];  
    int i,j;  
  
    for(i=0;i<4;i++)  
        for(j=0;j<5;j++)  
            twod[i][j]=i*j;  
  
    for(i=0;i<4;i++)  
    {  
        for(j=0;j<5;j++)  
            printf("%d  ",twod[i][j]);  
        printf("\n");  
    }  
}
```

The program outputs something like this:

0	0	0	0	0
0	1	2	3	4
0	2	4	6	8
0	3	6	9	12

A good use of a two-dimensional array is to manage lists of numbers. For example, you could use this array to hold the noontime temperature for each day of the year, grouped by month.

```
float yeartemp[12][31];
```

Similarly, the following program can be used to keep track of the number of points scored per season by each member of a basketball team. There are four seasons and 5 members.

```
#include<stdio.h>
void main()
{
    int bball[4][5];
    int i,j;
    for(i=0;i<4;i++)
        for(j=0;j<5;j++)
        {
            printf("Season %d, Player %d, ",i+1,j+1);
            printf("Enter number of points: ");
            scanf("%d",&bball[i][j]);
        }
    /* display results */
    for(i=0;i<4;i++)
        for(j=0;j<5;j++)
        {
            printf("Season %d, Player %d, ",i+1,j+1);
            printf("%d\n",bball[i][j]);
        }
}
```

## Initialising arrays

Like other types of variables, you can give the elements of arrays initial values. This is accomplished by specifying a list of values the array elements will have. The general form of array initialisation for a one-dimensional array is shown below.

```
type array_name[size] = { value list };
```

The value list is a comma separated list of constants that are type compatible with the base type of the array. The first constant will be placed in the first position of the array, the second constant in the second position and so on. Note that a semi colon follows the }.

In the following example, a five – element integer array is initialised with the squares of the number 1 through 5.

```
int i[5] = {1, 4, 9, 16, 25};
```

This means that `i[0]` will have the value 1 and `i[4]` will have the value 25.

You can initialise character arrays in two ways. First, if the array is not holding a null -terminated string, you simply specify each character using a comma separated list. For example, this initialises `a` with the letters 'A', 'B', and 'C'.

```
char a[3] = { 'A', 'B', 'C' };
```

If the character array is going to hold a string, you can initialise the array using a quoted string, as shown here.

```
char name[6] = "Peter";
```

Notice that no curly braces surround the string. They are not used in this form of initialisation. Because strings in C must end with a null, you must make sure that the array you declare is long enough to include the null. This is why `name` is 6 characters long, even though "Peter" is only 5 characters. When a string constant is used, the compiler automatically supplies the null terminator.

Multidimensional arrays are initialised the same way as one-dimensional ones.

For example, here the array `sqr` is initialised with the values 1 through 9, using row order.

```
int  sqr[3][3] = {
    1,  2,  3,
    4,  5,  6,
    7,  8,  9
};
```

This initialisation causes `sqr[0][0]` to have the value 1, `sqr[0][1]` to contain 2, `sqr[0][2]` to contain 3, and so forth.

If you are initialising a one-dimensional array, you need not specify the size of the array, simply put nothing inside the square brackets. If you don't specify the size, the compiler simply counts the number of initialisation constants and uses that value as the size of the array.

For example `int p[]={1,2,4,8,16,32,64,128};` causes the compiler to create an initialised array eight elements long.

Arrays that don't have their dimensions explicitly specified are called *unsized arrays*. An unsized array is useful because it is easier for you to change the size of the initialisation list without having to count it and then change the array dimension. This helps avoid counting errors on long lists, which is especially important when initialising strings.

Here an unsized array is used to hold a prompting message.

```
char prompt[ ] = "Enter your name:";
```

If at a later date, you wanted to change the prompt to "Enter your last name: ", you would not have to count the characters and then change the array size.

For multi dimensional arrays, you must specify all but the left dimension to allow C to index the array properly. In this way you may build tables of varying lengths with the compiler allocating enough storage for them automatically.

For example, the declaration of `sqr` as an unsized array is shown here.

```
int sqr[][3] = {
    1,  2,  3,
    4,  5,  6,
    7,  8,  9
};
```

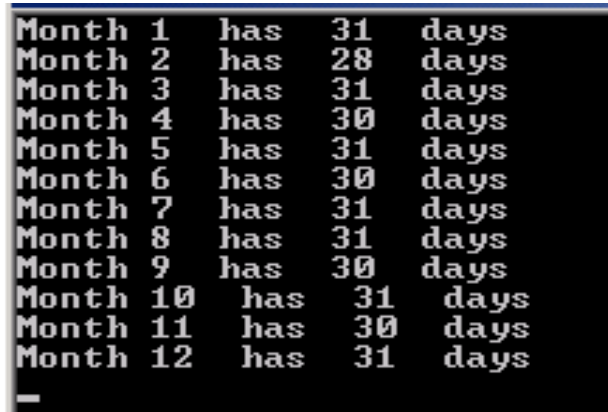
The advantage to this declaration over the sized version is that tables may be lengthened or shortened without changing the array dimensions.

To take your knowledge about arrays an extra step, go through the following examples and read the explanations below them.

### ***Example: Array that prints the number of days per month***

```
#include<stdio.h>
#define MONTHS 12
int days [MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
main()
{
    int index;
    extern int days[ ];
    for (index=0; index <MONTHS;  index + + )
        printf( "Month %d has %d days\n ", index+1,  days [index]);
    return 0;
}
```

The output will be as below:



```
Month 1 has 31 days
Month 2 has 28 days
Month 3 has 31 days
Month 4 has 30 days
Month 5 has 31 days
Month 6 has 30 days
Month 7 has 31 days
Month 8 has 31 days
Month 9 has 30 days
Month 10 has 31 days
Month 11 has 30 days
Month 12 has 31 days
_
```

### ***Explanation***

- By defining `days [ ]` outside the function, we make it external. We initialise it with a list enclosed in braces, commas are used to separate the members of the list.
- Inside the function the optional declaration `extern int days [ ]`; uses the keyword `extern` to remind us that `days` array is defined elsewhere in the program as an external array. Because it is defined elsewhere we need not give its size here. (ommitting it has no effect on how the program works)

### ***Example: Calculating the average of n numbers***

```
#include<stdio.h>
main()
{
    int n, count;
    float avg, d, sum =0.0;
    float list[100];

    /* Read in a value of n */
    printf(" \n How many numbers will be averaged ? ");
    scanf("%d", &n);
    printf("\n");
    /* Read in the numbers */
    for (count = 0; count < n; count++)
    {
        printf(" i = %d x = ", count+1);
        scanf("%f", &list[count]);
        sum+=list[count];
    }

    /* Calculate the average */
    avg = sum/n;
```



```
printf("\n The average is %5.2f \n\n", avg);

/* Calculate deviations from the average */
for (count =0; count < n; count ++){
    d = list[count] - avg;
    printf("i=%d x = %5.2f ,d=%5.2f",count+1,list[count],d);
}
return 0;
} /* End of program */
```

Here is sample output

```
How many numbers will be averaged ? 3

i = 1 x = 3
i = 2 x = -4
i = 3 x = 7

The average is 2.00

i=1 x = 3.00 ,d= 1.00 i=2 x = -4.00 ,d=-6.00 i=3 x = 7.00 ,d= 5.00 _
```

Is the output correct?

### Exercise

Assuming that the number of values in the list is already known to be 3, and that the list values are 5, 6, 8, rewrite the above program without having to request input from the keyboard.

### Example: Bubble sort

Arrays are especially useful when you want to sort information. For example, this program lets the user enter up to 100 numbers and then sorts them. The sorting algorithm is the **bubble sort**. The general concept of the bubble sort is the repeated comparisons and, if necessary exchanges of adjacent elements. This is a little, like bubbles in a tank of water with each bubble, in turn, seeking its own level.

The following code implements the bubble sort algorithm.

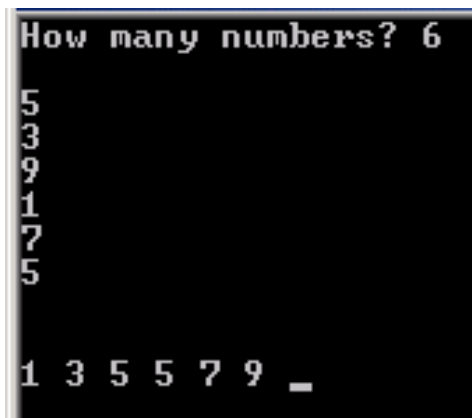
```
#include<stdio.h>
main()
```

```

{
    int item[100];
    int a,b,t;
    int count;
    /* Read in the numbers */
    printf("How many numbers? ");
    scanf("%d",&count);
    printf("\n");
    for(a=0;a<count;a++)
        scanf(" %d", &item[a]);
    /* Now sort them using a bubble sort */
    for(a=1;a<count;++a)
        for(b=count-1; b>=a;--b)
        {
            /* Compare adjacent items */
            if (item[b -1] > item[b])
            /* exchange the elements */
            {
                t = item[b - 1];
                item[b -1] = item[b];
                item[b] = t;
            }
        }
    /* Display sorted list */
    printf("\n\n");
    for(t=0;t<count;t++)
        printf("%d", item[t]);
    return 0;
} /* End of program */

```

Sample output:



```

How many numbers? 6
5
3
9
1
7
5

1 3 5 5 7 9 _

```

### Exercise

Run the program with a different set of random integers to sort.

### **Example: Two – dimensional array of scores**

```
#include<stdio.h>
#define STUDENT 3      /* Set maximum number of students */
#define CATS 4         /* Set maximum number of cats */
main()
{
    /* Declare and initialize required variables and array */
    int rows, cols, SCORES[STUDENT][CATS];
    float cats_sum , stud_average, total_sum=0.0, average;
    printf("Entering the marks ..... \n\n");

    /* Read in scores into the array */
    for(rows=0;rows<STUDENT; rows++) /* Outer student loop */
    {
        printf("\n Student % d\n", rows+1);
        cats_sum=0.0; /* Initializes sum of a student's marks */
        for(cols=0;cols<CATS;cols++) /* Inner loop for cats */
        {
            printf("CAT %d\n",cols+1);
            scanf(" %d", &SCORES[rows][cols]);
            cats_sum +=SCORES[rows][cols]; /* Adjust sum of marks */
        }
        stud_average=cats_sum/CATS;
        printf("\n Total marks for student %d is %3.2f" ,
            rows+1, cats_sum);

        printf("\n Average score for the student is %3.2f",
            stud_average);
        total_sum+=cats_sum; /* Adjust the class total marks */
    }

    average=total_sum/(STUDENT*CATS);
    printf("\n Total sum of marks for the class is %3.2f\n ", total_sum);
    printf("\n The class average is %3.2f\n ",average);

    /*Printing the array elements */

    printf("\nThe scores entered are: \n\n");
    for(rows=0;rows<STUDENT; rows++)
    {
        for(cols=0;cols<CATS;cols++)
        {
            printf("%d\t",SCORES[rows][cols]);
        }
        printf("\n"); /* print a new line after each row */
    }
    return 0;
}
```

```
}
```

Sample output

```
=== Entering the marks ===  
  
Student 1  
CAT 1:23  
CAT 2:25  
  
Total marks for student 1 is 48.00  
Average score for the student is 24.00  
  
Student 2  
CAT 1:17  
CAT 2:18  
  
Total marks for student 2 is 35.00  
Average score for the student is 17.50  
  
Student 3  
CAT 1:19  
CAT 2:20  
  
Total marks for student 3 is 39.00  
Average score for the student is 19.50  
  
Total sum of marks for the class is 122.00  
The class average is 20.33  
  
=== The scores entered are ===  
  
23      25  
17      18  
19      20
```

### Exercise

Change the number of students i.e rows to 5 and and Cats i.e columns to 3 then run program with cat scores of your choice. Confirm that the output is correct.

### Strings

In C, one or more characters enclosed between double quotes is called a *string*. C has no built-in string data type. Instead, C supports strings using one dimensional character arrays.

### ***String defined***

A string is defined as a *null terminated character array*. In C, a null is 0. This fact means that you must define the array is going to hold a string to be one byte larger than the largest string it is going to hold, to make room for the null.

### ***Reading a string***

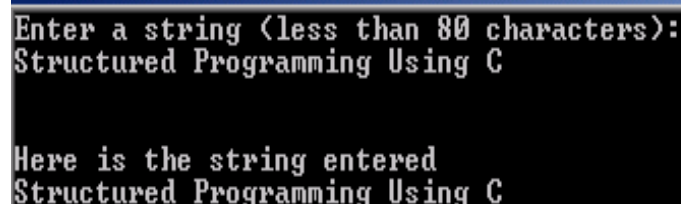
To read a string from the keyboard you must use another of C's standard library functions, **gets( )** which requires the **STDIO.H** header file. To use **gets( )**, call it using the name of a character array without any index. The **gets( )** function reads characters until you press **<ENTER>**. The carriage return is not stored, but it is replaced by a null, which terminates the string. For example, this program reads and writes a string entered at the keyboard.

```
#include<stdio.h>
main()
{
    char str[80];
    int i;
    printf("Enter a string (less than 80 characters): \n");
    gets(str); /* Read the string */

    printf("\n\n"); /* print 2 blank lines /

    /* print string */
    printf("Here is the string entered\n");
    for(i = 0;str[i]; i++)
        printf("%c", str[i]);
    return 0;
}
```

Compare the following sample output with the code above.



```
Enter a string (less than 80 characters):
Structured Programming Using C

Here is the string entered
Structured Programming Using C
```

The `gets( )` function performs no bounds checking, so it is possible for the user to enter more characters that `gets( )` is called with can hold. Therefore be sure to call it with an array large enough to hold the expected input.

### ***Outputting a string***

In the previous program, the string that was entered by the user was output to the screen a character at a time. There is however a much easier way to display a string, using `printf( )`. Here is the previous program rewritten..

```
#include<stdio.h>
main()
{
    char str[80];
    printf( " Enter a string (less than 80 characters): \n");
    gets(str);
    printf(str);
    return 0;
}
```

If you wanted to output a new line, you could output `str` like this:

```
printf("%s \n", str);
```

This method uses the `%s` format specifier followed by the new line character and uses the array as a second argument to be matched by the `%s` specifier.

### ***String related functions***

The C standard library supplies many string-related functions. The four most important are `strcpy( )`, `strcat( )`, `strcmp( )` and `strlen( )`. These functions require the header file `STRING.H`.

#### **strcpy function**

The `strcpy( )` function has this general form.

```
strcpy(to,from);
```

It copies the contents of `from` to `to`. The contents of `from` are unchanged. For example, this fragment copies the string “**hello**” into `str` and displays it on the screen.

```
char str[80];
strcpy(str, "hello");
printf("%s", str);
```

The `strcpy( )` function performs no bounds checking, so you just make sure that the array on the receiving end is large enough to hold what is being copied, including the null terminator.

### `strcat()` function

The `strcat ( )` function adds the contents of one string to another. This is called **concatenation**. Its general form is

```
strcat (to, from);
```

It adds the contents of `from` to `to`. It performs no bounds checking, so you must make sure `to` is large enough to hold its current contents plus what it will be receiving. This fragment displays **hello there**.

```
char str[80];
strcpy (str, "hello");
strcat (str, "there");
printf(str);
```

### `strcmp()` function

The `strcmp ( )` function compares two strings. It takes this general form.

```
strcmp (s1,s2);
```

It returns 0 if the strings are the same. It returns less than 0 if `s1` is less than `s2` and greater than 0 if `s1` is greater than `s2`. The strings are compared lexicographically; that is in dictionary order. Therefore, a string is less than another when it would appear before the other in a dictionary. A string is greater than another when it would appear after the other. The comparison is not based upon the length of the string. Also, the comparison is case-sensitive, lowercase characters being greater than uppercase. This fragment prints 0, because the strings are the same.

```
printf( " %d ", strcmp (" one", "one"));
```

### `strlen()` function

The `strlen()` function returns the length , in characters, of a string. Its general form is

```
strlen(str);
```

The `strlen()` function does not count the null terminator.

### ***Example: Demonstrating string functions***

```
#include<string.h>
#include<stdio.h>
main()
{
    char str1[80], str2[80];
    int i;
    printf("Enter the first string and hit the <ENTER> key:");
    gets(str1);
    printf("\nEnter the second string and hit the <ENTER> key:");
    gets(str2);

    /* See how long the strings are */
    printf("\n%s is %d characters long\n",str1, strlen(str1));
    printf("\n%s is %d characters long\n", str2, strlen(str2));
    /* Compare the strings */
    i = strcmp(str1, str2);
    if (!i)
        printf("\nThe strings are equal\n");
    else if (i < 0)
        printf("\n%s is less than %s \n", str1,str2);
    else
        printf("\n%s  is greater than %s\n", str1,str2);

    /* Concatenate str2 to end of str1 if there is enough room */
    if(strlen(str1) + strlen(str2)<80)
    {
        strcat(str1, str2);
        printf("\n%s\n", str1);
    }
    /* copy str2 to str1 */
    strcpy(str1, str2);
    printf("\n%s  %s \n", str1, str2);
    return 0;
}
```

See below how this program would use the string functions as it runs.

```
Enter the first string and hit the <ENTER> key:STRUCTURED
Enter the second string and hit the <ENTER> key:PROGRAMMING
STRUCTURED  is 11 characters long
PROGRAMMING is 11 characters long
STRUCTURED  is greater than PROGRAMMING
STRUCTURED PROGRAMMING
PROGRAMMING PROGRAMMING
_
```



### Note

You can use `scanf( )` to read a string using the `%s` specifier, but you probably won't need to. Why? This is because when `scanf( )` inputs a string, it stops reading that string when the first white space character is encountered. A white space character is a space, a tab, or a new line. This means that you cannot use `scanf( )` to read input like the following string.

### This is one string

Because there is a space after the word *This*, `scanf( )` will stop inputting the string at that point. That is why `gets( )` is generally used to input strings.

### Revision Exercise

1. What is an array structure?
2. Distinguish between one-dimensional array and a two-dimensional array.
3. What is an array index and how are indexes assigned to array elements?
4. What is an unsized array? What is the advantage of using one?
5. What is wrong with this fragment?  

```
char name[6] = "Okoth Obua";
```
6. Give and explain the syntax of a two-dimensional array declaration.
7. In the course *Structured Programming using C*, the following percentage marks were recorded for six students in four continuous assessment tests.

	CAT 1	CAT 2	CAT 3	CAT 4
NANCY	90	34	76	45
JAMES	55	56	70	67
MARY	45	78	70	89
ALEX	89	65	56	90
MOSES	67	56	72	76
CAROL	70	90	68	56

- (a) (i) If you were to implement the above table in a C program:

- (ii) Write a statement that would create the above table and initialize it with the given scores.
- (b) Suppose the name of the above table was `SCORES`.
- (i) What is the value of `SCORES[2][3]`?
- (ii) What is the result of:  $(\text{SCORES}[3][3] \% 11) * 3$ ?
- (c) Write a complete program that initializes the above values in the table, computes and displays the total mark and average scored by *each student*.
8. Show how to initialise an integer array called **items** with the values 1 through 10.
9. (i) Write a program that defines a 3 by 3 by 3 three dimensional array, and load it with the numbers 1 to 27.
- (ii) Have the program in (i) display the sum of the elements.
10. Consider the following algorithm:
- (i) Declare an array 'mark' for storing the marks for three students in two subjects.
- (ii) Repeat the following for all students:
- Input the 6 marks into the array.
  - Compute the total and average mark for each student .
  - If average is at least 50 and not more than 100, then output "Student passed"; otherwise output "Wrong mark".
- Write a program to implement the above.
11. Write a program that uses a one –dimensional array to:
- (i) Enter scores of aptitude tests administered to 10 job applicants.
- (ii) Calculate the average of the ten scores
- (iii) Output the average and the ten scores on different lines.
12. Using an array, write a program that stores a 10 by 10 multiplication table displaying them on the screen. No input from the keyboard is required.

## POINTERS

### Objectives

At the end of this chapter, the reader should be able to:

- Define a pointer
- Declare pointer variables
- Use basic pointer operators \* and &
- Understand operations permitted with pointers
- Use pointers with arrays
- Use pointers with string variables

### Introduction

This chapter covers one of C's most important and sometimes most troublesome features: the pointer. A pointer is basically an address of an object. One reason that pointers are so important is that much of the power of the C language is derived from the unique way in which they are implemented. You will learn about special pointer operators, pointer arithmetic and how arrays and pointers are related.

### Pointer defined

A pointer is a variable that holds the memory address of another variable. For example, if a variable called **p** contains the address of another variable called **q**, then p is said to point to q.

Therefore if q were at location 100 in memory, then p would have the value 100.

### Pointer declaration

To declare a pointer variable, use this general form:

*type \*var\_name;*

Here, **type** is the base type of the pointer. The base type specifies the type of the object that the pointer can point to. Notice that an asterisk precedes the variable name. This tells the computer that a pointer variable is being created. For example, the following statement creates a pointer to an integer.

```
int *p;
```

## Pointer Operators

C contains two special pointer operators: **\*** and **&**. The **&** operator returns the address of the variable it precedes. The **\*** operator returns the value stored at the address that it precedes. The **\*** pointer operator has no relationship to the multiplication operator, which uses the same symbol). For example, examine this short program.

```
#include<stdio.h>
main()
{
    int *p, q;
    q = 100;          /* assign q 100 */
    p = &q;           /* assign p the address of q*/
    printf("%d", p); /* display q's value using pointer*/
    return 0;
}
```

This program prints **100** on the screen. Let us see why.

First, the line **int \*p, q;** defines two variables: **p**, which is declared as an integer pointer, and **q**, which is an integer. Next, **q** is assigned the value **100**.

In the next line, **p** is assigned the address of **q**. You can verbalize the **&** operator as “address of.” Therefore, this line can be read as: assign **p** the address of **q**. Finally, the value is displayed using the **\*** operator applied to **p**. The **\*** operator can be verbalized as “at address”.

Therefore the **printf( )** statement can be read as “print the value at address **q**,” which is 100. When a variable value is referenced through a pointer, the process is called indirection. It is possible to use the **\*** operator on the left side of an assignment statement in order to assign a variable a new value using a pointer to it. For example, this program assigns a value **q** indirectly using the pointer **p**.

```
#include<stdio.h>
main()
{
    int *p, q;
    p = &q;          /* get q's address */
    *p = 199;        /* assign q a value using a pointer */
    printf("q's value is %d", q);
    return 0;
}
```

**Note:** The type of the variable and type of pointer must match.

Take a look at two other examples below.

This program uses pointers to display the values of a loop counter.

```

#include<stdio.h>
main()
{
    int i,*p;
    p = &i;
    for (i =0; i <10; i++)
        printf ("%d\n", *p);
    return 0;
}

```

The following program performs some basic arithmetic using pointers.

```

#include<stdio.h>
main()
{
    int u1, u2;
    int v = 3;
    int *pv;
    u1 = 2 * ( v + 5 );
    pv = &v;
    u2 = 2 * (*pv + 5 );
    printf(" \n u1 = %d    u2  = %d ", u1, u2);
    return 0;
}

```

The output when you run the program will be:

```

u1 = 16  u2 = 16 _

```

Explain why.

### Pointers operations

It is possible to perform several operations with pointer variables as explained below.

- **Assignment**

One can assign an address to a pointer by:

- Using an array name or
- Using the address operator

From the previous example, p1 is assigned the address of the beginning of the array which is cell 234.

- ***Dereferencing (value – finding)***

The \* operator gives the value pointed to.

From the previous example, `p1 = 100` which is the value stored in location 234.

- ***Taking a pointer address***

Pointer variables have an address and a value. The & operator tells us where the pointer itself is stored.

From the previous example, `p1` is stored in address 3606 whose value is 234.

- ***Pointer arithmetic***

In general, pointers may be used like other variables. However, you need to understand a few rules and restrictions.

In addition to the & and \* operators, there are only four other operators that may be applied to pointer variables: the arithmetic operators +, ++, - and --. Further, you may add or subtract only integer quantities. You cannot, for example, add a floating point number to a pointer.

Pointer incrementation arithmetic differs from normal because it is performed relative to the base type of the pointer. Each time a pointer is incremented, it will point to the next item, as defined by the base type, beyond the one being currently pointed to

For example, assume an integer pointer called **p** contains the address **200**. After the statement `p++` executes, `p` will have the value **202**, assuming integers are 2 bytes long. If `p` had been a floating point value (4 bytes long), then the resultant value contained in `p` would be **204**.

Pointer arithmetic with character appears normal when character pointers are used. Because characters are 1 byte long, an increment increases the pointer value by one, decrement decreases it by one.

### *Addition and subtraction of pointers*

You may add or subtract any integer quantity you want, to or from a pointer. For example, the following is a valid fragment.

```
int    *p;  
.  
.  
p = p + 200;
```

causes p to point to the 200<sup>th</sup> integer past the one to which p was currently pointing to.

### Note

You may not perform any other type of arithmetic operations. You may not divide, multiply or take modulus of a pointer. However, you may subtract a pointer from another to find the number of elements separating them.

### Incrementation and decrementation of pointers

You can apply the increment and decrement operations to either the pointer itself or the object to which it points. However you must be careful when attempting to increment the object pointed to by a pointer.

For example, assume **p** points to an integer that contains the value 1. Now consider the statements:

```
*p++; and
*(p)++;
```

\*p++ first increments p and then obtains the value at the new location. To increment what is pointed to by a pointer, you must use the second statement.

### Pointer precautions

- Never use a pointer of one type to point to an object of a different type.

For example:

```
int q;
float *fp;
fp = &q; /* pointer fp assigned address of an integer */
fp = 100.23; /* address used for assignment */
```

- Do not use a pointer before it has been assigned the address of a variable. May cause program to crash.

For example:

```
main()
{
    int *p;
    *p = 10; /*Incorrect since p is not pointing to anything */
    ...
}
```

```
}
```

The above is meaningless and dangerous.

## Pointers and arrays

Pointers can be extended for use in arrays. Array elements can be accessed using pointers. It is possible to create an array of pointers.

### *Accessing arrays using pointers*

We can declare an array of characters and a pointer to a character

For example

```
char line[100], *p;
```

We may refer to the first two elements of the array line using

```
line[0] = 'a';  
line[1] = 'b';
```

 and for each assignment, the compiler calculates the address.

Another way to perform the assignments is to use a pointer. First, we must initialize the pointer **p** to point to the beginning of the array.

i.e. `p = &line [0];`

Since an array's name is a synonym to the array's starting address, we can use:

```
p = line;
```

We can now perform the assignments:

```
*p = 'a'; and *(p+1) = 'b';
```

### *Example: Array manipulation using pointers*

```
#include<stdio.h>  
main()  
{  
    static int x[3] = {10, 20, 30};  
    int *p1, *p2;  
    p1 = x; /* assign address to a pointer */  
    p2 = & x [2]; /* assign p2 to address of x[2] */  
    printf (" p1 = %u, *p1 = %d, &p1 = %u \n", p1, *p1, &p1);  
    return 0;  
}
```



How close is your output to the one below which was produced when the program was run? Can you explain any differences.

```
p1 = 4223092, *p1 = 10, &p1 = 1245064
```

### ***Creating pointer arrays***

Pointers may be arrayed like any other data type. For example the following statement declares an integer pointer array that has 20 elements.

```
int *pa[20];
```

The address of an integer variable `myvar` is assigned to the ninth element of the array as follows;

```
pa[8]= &myvar;
```

Because `pa` is an array of pointers, the only value that the array elements may hold are addresses of integer variables. To assign the integer pointed to by the third element of `pa` the value 50, use the statement;

```
*pa[2] = 50;
```

### **String variables as pointers**

In C a string variable is defined to be simply a pointer to the beginning of a string. Take a look this example to understand the relationship between pointers and strings.

```
#include<stdio.h>
void main( )
{
    char strg[40],*there,one,two;
    strcpy(strg,"This is a character string.");
    one = strg[0]; /* one and two are identical */
    two = *strg;
    printf("The first output is %c %c\n", one, two);
    one = strg[8]; /* one and two are identical */
    two = *(strg+8);
    printf("the second output is %c %c\n", one, two);
    there = strg+10; /* strg+10 is identical to strg[10] */
    printf("The third output is %c\n", strg[10]);
    printf("The fourth output is %c\n", *there);
}
```

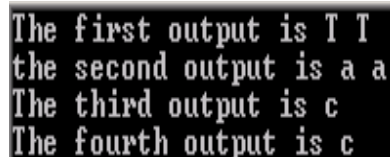
You will notice that first we assign a string constant to the string variable named `strg` so we will have some data to work with. Next, we assign the value of the first element to the variable `one`, a simple char variable.

Next, since the string name is a pointer by definition, we can assign the same value to `two` by using the asterisk and the string name. The result of the two assignments are such that `one` now has the same value as `two`, and both contain the character 'T', the first character in the string. Note that it would be incorrect to write the ninth line as `two = *strg[0];` because the asterisk takes the place of the square brackets.

For all practical purposes, `strg` is a pointer. It does, however, have one restriction that a true pointer does not have. It cannot be changed like a variable, but must always contain the initial value and therefore always points to its string. It could be thought of as a pointer constant, and in some applications you may desire a pointer that cannot be corrupted in any way. Even though it cannot be changed, it can be used to refer to other values than the one it is defined to point to, as we see in the next section of the program.

Moving ahead to line 9, the variable `one` is assigned the value of the ninth variable (since the indexing starts at zero) and `two` is assigned the same value because we are allowed to index a pointer to get to values farther ahead in the string. Both variables now contain the character 'a'.

Hence the following output.



```
The first output is T T
the second output is a a
The third output is c
The fourth output is c
```

## Revision Exercise

1. What is a pointer?
2. How do arrays and pointers relate to each other?
3. How does pointer arithmetic differ from ordinary arithmetic?
4. State two rules one should observe when using pointers.
5. Using examples, give the meaning of the pointer operators `&` and `*`.
6. Write a program that uses a *for* loop to print digits from 1 to 5 using pointers

7. Write an appropriate declaration for each of the following situations;

- (a) Declare two pointers whose objects are the integer variables *i* and *j*.
- (b) Declare a function that accepts two integer arguments and returns a long integer. Each argument will be a pointer to an integer quantity.
- (c) Declare a one-dimensional floating point array using pointer notation.
- (d) Declare a function that accepts another function and returns a pointer to a character. The function passed as an argument will accept an integer argument and return an integer quantity.

8. What does this fragment display?

```
int temp[5] = {10, 19, 23, 8, 9};  
int *p;  
p = temp;  
printf( "%d ", *(p+3));
```

9. Write a program that assigns a value to a variable indirectly by using a pointer to that variable.
10. Assume **p** is declared as a pointer to a **double** and contains the address 100. Further, assume that **doubles** are 8 bytes long. After **p** is incremented, what will its value be?

## STRUCTURES

### Objectives

At the end of this chapter, the reader should be able to:

- Explain what a structure is
- Create structures
- Write programs that use structure variables
- Create arrays of structures
- Appreciate uses of structures in advanced programming.
- Use typedef to create structures.
- Distinguish between structures and unions

### Introduction

Suppose you want to write a program that keeps tracks of students [Name, Marks] i.e. a variety of information to be stored about each student. This requires;

- An array of strings (for the Names).
- Marks in an array of integers.

Keeping track of many related arrays can be problematic, for example in operations such as sorting all of the arrays using a certain format.

A data form or type containing both the strings and integers and somehow keep the information separate is required. Such a data type is called a **structure**.

### Structure defined

A structure is an aggregate data type that is composed of two or more related elements.

Unlike arrays, each element of a structure can have its own type, which may differ from the types of any other elements.

### Declaring a structure

Declaring a structure is a two-stage process. The first stage defines a new data type that has the required structure which can then be used to declare as many variables with the same structure as required.

### ***Defining a new data type***

The keyword **struct** is used to create a new structure data type.

For example, suppose we need to store a name, age and salary as a single structure. You would first define the new data type using.

```
struct emprec
{
    char name[25];
    int age;
    int pay;
};
```

### ***Declaring structure variables***

This is the second stage. Once you have defined a new struct data type, you can declare a new variable. For example

```
struct emprec employee;
```

Notice that the new variable is called **employee** and it is of type **emprec** which has been defined earlier. This means that **employee** is a particular example of the general type **emprec**. Notice too that the statement `struct emprec employee;` has name **emprec** duplicated.

It might help to compare the above situation with that of a general **int** type and a particular **int** variable such as **count** - **emprec** is a type like **int** and **employee** is a variable like **count**.

Here is a bank account structure :

#### **Bank account details:**

- Account number (integer)
- Account type (character)
- Account holder name [30 characters]
- Account balance (float)

```
struct Bank_ account
{
    int acc_number;
    char acc_type;
```

```

        char holder_name [30];
        float balance;
    };

```

Following is student structure definition followed by variable declarations. The structure is made up of a character array **name** and int variable **marks**.

```

struct student
{
    char name[SIZE];
    int marks;
};
struct student mystudent;

```

You can see that in general you can define a structure using:

```

struct name
{
    list of component variables
};

```

and you can have as long a list of component variables as you need. Once defined you can declare as many examples of the new type as you like using:

```

struct name list of variables;

```

For example:

```

struct emprec employee, oldemploy, newemploy;

```

If you want to you can also declare a structure variable within the type definition by writing its name before the final semi-colon. For example:

```

struct emprec
{
    char name[25];
    int age;
    int pay;
} employee;

```

## Initializing a structure

A structure can be initialized like any other variable - external, static or automatic. This will depend on where the structure is defined. For example mystudent variable created above can be initialized as follows:

```
struct student mystudent =  
{  
    "Fred Otieno",25;  
};
```

Each member is given its own line of initialization and a comma separator, one member initialization from the next.

## Accessing structure members

When you first start working with arrays it seems obvious that you access the individual elements of the array using an index as in **a[i]** for the ith element of the array, but how to get at the individual components of a structure?

The answer is that you have to use qualified names. You first give the name of the structure variable and then the name of the component separated by a dot. For example, given:

```
struct emprec employee;
```

then:

```
employee.age
```

is an **int** and:

```
employee.name
```

is a **char** array. Once you have used a qualified name to get down to the level of a component then it behaves like a normal variable of the type. For example:

```
employee.age=32;
```

is a valid assignment to an **int** and:

```
employee.name[2] = 'X';
```

is a valid assignment to an element of the **char** array.

Notice that the qualified name uses the structure variable name and not the structure type name.

You can use `employee.age` or `employee.name` exactly the way you use other variables.

For example, you can use `scanf("%d",&employee.age);`

Or

```
gets(employee.name);
```

### **Note**

Although **employee** is a structure, **employee.age** is an **int** type and is like any other integer variable. Therefore;

The use of `scanf("%d",.....)` requires the address of an integer and that is what `&employee.age` does.

You can use this method even to read values into another variable.

If **employee1** is another employee structure declared as follows;

**struct employee employee1;** then it is possible to read the age and name into the variable using the statements;

```
gets(employee1.name);  
scanf("%d",&employee1.age);
```

You can also define a structure that includes another structure as a component and of course that structure can contain another structure and so on. In this case you simply use the name of each structure in turn, separated by dots, until you reach a final component that isn't a structure. For example, if you declare a `struct firm` which includes a component `employee` which is an `emprec` then:

```
firm.employee.age
```

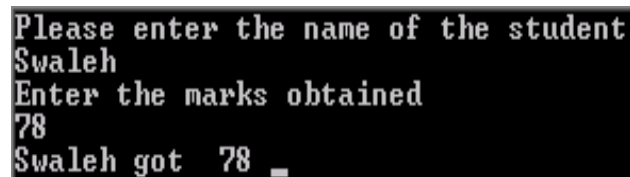
is an **int**.



### **Example: A student structure program**

```
#include<stdio.h>
#define SIZE 40
struct student
{
    char name[SIZE];
    int marks;
};
main()
{
    struct student mystudent; /* declare mystudent as a student type */
    printf("Please enter the name of the student \n");
    scanf("%s", &mystudent.name);
    printf("Enter the marks obtained \n");
    scanf("%d", &mystudent.marks);
    printf("%s got  %d ", mystudent.name, mystudent.marks);
    return 0;
}
```

Run the program and test it with your choice studentr name and marks as has been done below.



```
Please enter the name of the student
Swaleh
Enter the marks obtained
78
Swaleh got 78 _
```

### **Arrays of structures**

Let us extend our student program to handle a greater number of students.

One student can be described by one structure variable of type student, 2 students by two variables, 3 students by three variables, n students by n such structure variables, etc.

To have  $n$  students ( $n$  being any number), we use an array structure of  $n$  elements.

### **Example: Entering a student's details using an array structure**

```
#include<stdio.h>
#define SIZE 40
#define MAXSTU 4
struct student{
    char name[SIZE];
    int marks;
};
```

```

main()
{
    struct student mystudent[MAXSTU];
    int count;
    for(count=0; count<MAXSTU; count++){
        printf("Enter the name and marks of student %d\n", count+1);
        scanf("%s", &mystudent[count].name);
        scanf("%d", &mystudent[count].marks);
    }

    printf("\n\n ===Student details as entered===\n\n");

    for(count=0; count<MAXSTU; count++){
        printf("%s got %d\n", mystudent[count].name, mystudent[count].marks);
    }

    return 0;
}

```

Run the program with a different array size and sets of your choice names and marks (watch out that you do not use a very large size unless you have a lot of patience to run the program for a long time!). Compare your results with the output of the above program shown below.

```

Enter the name and marks of student 1
Maneno
50
Enter the name and marks of student 2
Swaleh
78
Enter the name and marks of student 3
Atieno
69
Enter the name and marks of student 4
Kamande
64

===Student details as entered===

Maneno got 50
Swaleh got 78
Atieno got 69
Kamande got 64

```

## Uses of structures

Where are structures useful?

The immediate application that comes to mind is database management. For example, to maintain data about employees in an organization, books in a library, items in a store, financial transactions in a company, etc.

Their use however, stretches beyond database management. They can be used for a variety of applications like:

- Checking the memory size of the computer.
- Hiding a file from a directory
- Displaying the directory of a disk.
- Interacting with the mouse
- Formatting a floppy
- Drawing any graphics shape on the screen.
- Changing the size of the cursor.

To program the above applications, you need thorough knowledge of internal details of the operating system.

### **User defined types (typedef)**

The typedef feature allows users to define new data types that are equivalent to existing data types. Once a user defined type has been established, then new variables, arrays, structures, etc. can be declared in terms of this new data type.

For example,

```
typedef int age;
```

In this declaration, age is a user defined data type which is equivalent to type int. Hence the declaration:

```
age male, female; - is equivalent to writing int male,male;
```

In other words, male and female are regarded as variables of type age, though they are actually integer type variables.

Similarly, the declarations;

```
typedef float height[100];  
  
height men , women;
```

define height as a 100 - element floating type array, hence men and women are 100 element floating point arrays.

typedef is particularly convenient when defining structures, since it eliminates the need to write the struct tag whenever a structure is referenced.

### ***Example: Demonstrating typedef***

```
typedef struct
{
    char name[SIZE];
    int marks;
}student;    /*Student is a user defined data type*/

student student1,student2;
```

### ***Example: typedef further demonstrated***

```
typedef struct
{
    int month;
    int day;
    int year;
}date; /*date is a user defined data type*/

typedef struct
{
    int acc_number;
    char acc_type;
    char name[30];
    float balance;
    date lastpayment;
}BankAcct; /*BankAcct is a user defined type */

BankAcct customer[100];
```

## **Unions**

A union is a single memory location that stores two or more variables. Members within a union all share the same storage area, whereas each member within a structure is assigned its own unique storage area.

Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all members at a time.

The similarity between structures and unions is that both contain members whose individual data types may differ from one another.

A union takes the following form

```
union tag
{
    member 1;
    member 2;
    ....
    ....
    member n;
};

Or

union tag
{
    member 1;
    member 2;
    ....
    ....
    member n;
}variable list;
```

Consider that a C program contains the following union declaration:

```
union id{
    char color[12];
    int size;
}shirt, blouse;
```

### ***Explanation***

- (i) There are two union variables **shirt** and **blouse**. Each variable can represent either a 2 character string (colour) or a integer quantity (size) at any one time. The 12-character string will require more storage area within the computer's memory than the integer quantity.

Therefore a block of memory large enough for the 12-character string will be allocated to each union variable.

- (ii) A union may be a member of a structure and a structure may be a member of a union and may be freely mixed with arrays.

Also consider the following example.

```
union id{
    char color[12];
    int size;
};

struct clothes
{
    char manufact[20];
    float cost;
    union id descr;
}shirt,blouse;
```

### ***Explanation***

**shirt** and **blouse** are structure variables of type **clothes**.

Each variable will contain the following members;

- A string manufact
- A floating point quantity cost
- A union descr. The union may represent either a string (color) or an integer quantity (size).

An alternative declaration of the variables **shirt** and **blouse** is:

```
struct clothes
{
    char manufact[20];
    float cost;
    union
    {
        char color[12];
        int size;
    }descr;
```

```
};shirt, blouse;
```

### Revision Exercise

1. How is a structure different from:
  - (a) a union?
  - (b) an array?
2. Show how to create a structure called *stype* that contains these five elements:

ch - character  
d - floating point;  
i - integer;  
str - 80 character string;  
balance - double floating pint;

Also define one variable called *s\_var* using this structure.

3. What is wrong with this fragment?

```
struct s_type
{
    int a;
    char b;
    float bal;
}myvar,*p;
p = &myvar;
p.a = 10;
```

4. Set up a suitable structure for an invoice that should hold the following details:

Element	Type
Invoice number	integer

Customer number	integer
Invoice date	structure (with three integer elements; day, month, year)
Customer address	string (20 characters)
Item	structure [with product code (integer), unit price (float) quantity (float) , amount (double)]
Invoice Total	double

5. (a) Write a program that sets up a structure of a student record comprising the students name, age and fee balance, then reads in the three items into a structure variable and outputs to the screen.

(b) Rewrite the program in part (a) above such that you enter the student data into an array of 5 structure variables and then print it.

6 (a) What is a user defined data type?

(b) Using a user defined type named invoices set up the structure template in part (a) above, show how you can set two simple structure variables **invoice1**, **invoice2** using the new type.



## FILE HANDLING

### Objectives

At the end of this chapter, the reader should be able to:

- Understand streams
- Master file system basics
- Understand random access
- Apply various file-system functions
- Develop a simple file-based system

### Introduction

Disk input/output (I/O) operations are performed on entities called files.

Although C does not have any built – in method of performing file I/O, the standard C library contains a very rich set of I/O functions.

High-level Disk I/O functions are more commonly used in C programs, since they are easier to use than low-level disk I/O functions.

The low-level disk I/O functions are more closely related to the computer's operating system than the high-level disk I/O functions.

Low-level disk I/O is harder to program than high-level disk I/O. However, low-level disk I/O is more efficient both in terms of operation and the amount of memory used by the program.

This chapter is concerned with the high level functions.

### The streams concept

In C, the stream is a common, logical interface to the various devices that comprise the computer. In its most common form, it is a logical interface to a file.

A stream is linked to a file using an **open** operation. A stream is dissociated from a file using a **close** operation.

There are two types of streams; text and binary. A text stream is used with ASCII characters. When a text stream is being used, some character translations take place.

For example, when the new line character is output, it is converted into a carriage return/line feed sequence. For this reason, there may not be a one-to-one correspondence between what is sent to the stream and what is written to the file.

A binary stream is used with any type of data. No character translations will occur, and there is a one-to-one correspondence between what is sent to the stream and what is actually contained in the file.

One final concept you need to understand is that of the *current location*. The current location also referred to as the *current position* is the location in a file where the next file access will begin. For example, if a file is 100 bytes long and half the file has been read, the next read operation will occur at location 50, which is the current location.

## Opening a file

Before we can write information to a file or disk or read it we must open the file. Opening a file establishes a link between the program and the operating system. We provide the operating system with the name of the file and whether we plan to read or write to it.

To open a file and associate it with a stream, use **fopen()** function.

The link between our program and the operating system is a structure called **FILE** (which has been defined in the header file **stdio.h**).

If the open operation is successful, what we get back is a pointer to the structure **FILE**. That is why we make the following declaration before opening the file,

```
FILE *fp;
```

Each file we open will have its own file structure. The file structure contains information about the file being used, such as its current size, memory location, access modes, etc.

Consider the following statements,

```
FILE *fp;  
fp = fopen ("myfile.txt", "r");
```

**Meaning?** **fp** is a pointer variable, which contains address of the structure **FILE** which has been defined in the "stdio.h" header file.

**fopen ( )** will open a file "**myfile.txt**" in 'read' mode, which tells the C compiler that we would be reading the contents of the file.

Legal values for modes that can be used with **fopen ( )** are summarised below.

## ***File opening modes***

“r” Searches file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist it returns NULL.

Operation possible - reading from the file.

“w” Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created; Returns NULL if unable to open file.

Operations possible – Writing to the file.

“a” Searches file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist, a new file is created. Returns NULL if unable to open file.

Operations possible – Appending new contents at end of file.

“r+” Searches file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. Returns NULL if unable to open file.  
Operations possible – Reading existing contents, writing new contents, modifying contents of the file.

“w+” Searches file. If the file exists, its contents are destroyed. If the file doesn't exist, a new file is created. Returns NULL if unable to open file.

Operations possible – Writing new contents, reading them back and modifying existing contents of the file.

“a+” Searches a file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. Returns NULL if unable to open file.

Operations possible: Reading existing contents, appending new contents, appending new contents to end of file. Cannot modify existing contents.

## ***Note***

The header file “stdio.h” defines the macro NULL which is defined to be a null pointer.

It is very important to ensure that a valid file pointer has been returned. You must check the value returned by **fopen ( )** to be sure that it is not **NULL**.

For example, the proper way to open a file called **myfile.txt** for text input is shown below.

```
FILE *fp;
```

```

fp = fopen ("myfile.txt", "r");
if (fp == NULL)
{
    printf("Error opening file \n");
    exit();
}

```

## Closing a file

When we have finished reading from a file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose (fp);
```

## Formatted disk I/O functions: fprintf( ) and fscanf( )

The functions **fprintf ( )** and **fscanf ( )** operate exactly like **printf( )** and **scanf( )** except that they work with files.

Here is a program which illustrates the use of these functions.

### **Example: fprintf( ) demonstrated**

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main( )
{
    FILE *fp;
    char another = 'Y';
    char name[40];
    int age;
    float bs;

    fp=fopen ("EMPLOYEE.DAT","w");
    if(fp==NULL)
    {
        printf("Cannot open file");
        exit(1);
    }

    while(another=='Y')
    {
        printf("\n Enter name, age and basic salary \n");
        scanf("%s%d%f", name,&age,&bs);
        fprintf (fp, "%s%d%f\n", name,&age,&bs);
        printf("\n Another employee (Y/N)? ");
        fflush(stdin);
    }
}

```

```

    another=getche ( );
}

fclose (fp);
return 0;
}

```

Run the program using the following test input:

```

Enter name, age and basic salary
Ongiri 34 1550
Another employee (Y/N)? Y
Enter name, age and basic salary
Mutua 24 1200
Another employee (Y/N)? Y
Enter name, age and basic salary
Halima 26 2000
Another employee (Y/N)? N

```

The key to this program is the function **fprintf ( )**, which writes the values of the three variables to the file. This function is similar to **printf ( )**, except that a **FILE** pointer is included as the first argument.

As in **printf ( )**, we can format the data in a variety of ways, by using **fprintf ( )**. In fact all the format conventions of **printf()** function work with **fprintf()** as well.

Why use the **fflush( )** function? The reason is to get rid of peculiarity of **scanf()**. After supplying data for one employee, we would hit the **<ENTER>** key. What **scanf()** does is it assigns name, age and salary to appropriate variables and keeps the enter key unread in the keyboard buffer. So when it is time to supply Y or N for the question **'Another employee (Y/N)?'**, **getch( )** will read the enter key from the buffer thinking the user has entered the **<ENTER>** key. To avoid this problem we use the function **fflush()**. It is designed to remove or 'flush out' any data remaining in the buffer. The argument to **fflush()** must be the buffer which we want to flush out here we use 'stdin', which means buffer related with standard input device, the keyboard.

Suppose we want to read back the names, ages and basic salaries of different employees which we stored through the earlier program into the **EMPLOYMENT.DAT** file. The following program does just this:

**Example: fscanf ( ) demonstrated**

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *fp;
    char name[40];

```

```

int age;
float bs;

fp=fopen ("EMPLOYEE.DAT","r");
if (fp==NULL)
{
    printf("cannot open file");
    exit(1);
}

while(fscanf (fp,"%s%d%f", name,&age,&bs)!=EOF)
    printf("%s %d %f\n", name, &age, &bs);

fclose (fp);
return 0;
}

```

You should get this output .

```

Ongiri 34 1550.000000
Mutua 24 1200.000000
Halima 26 2000.000000

```

This program uses the **fscanf ( )**, function to read the data from disk. This function is similar to **scanf ( )**, except that as with **fprintf ( )**, a pointer to **FILE** is included as the first argument.

So far we have seen programs which write characters, strings or number to a file. if we desire to write a combination of these, that is a combination of dissimilar data types, what should we do? Use structures.

### ***Example: Use of structures for writing records of employees***

```

/*writing records to an employee file using a structure*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main()
{
    FILE *fp;
    char another = 'Y';
    struct emp
    {
        char name[40];
        int age;
        float bs;
    };
    struct emp e;
    fp = fopen ("EMPLOYEE.DATA", "w" );

```

```

    if(fp==NULL)
    {
        printf("cannot open file");
        exit(1);
    }

    while(another=='Y')
    {
        printf("\n Enter name, age and basic salary:  ");
        scanf("%s%d%f", e.name,&e.age,&e.bs);
        fprintf (fp, "%s %d %f\n",e.name,e.age,e.bs);
        printf("\n Add another record (Y/N)? ");
        fflush(stdin),
        another=getche ();
    }
    fclose (fp);
    return 0;
}

```

Test input as below when you run the program:

```

Enter name, age and basic salary: Kanja 34 1250
Add another record (Y/N)? Y
Enter name, age and basic salary: Wanja 21 1300
Add another record (Y/N)? Y
Enter name, age and basic salary: Mwashe 34 1400
Add another record (Y/N)? N

```

In the above program, we are just reading data into a structure using **scanf( )** and dumping it into disk file using **fprintf ( )**. The user can input as many records as he desires. The procedure ends when the user supplies 'N' for the question '**Add another record (Y/N)?**'.

The above program has two disadvantages:

- (a) The numbers (basic salary) would occupy more number of bytes, since the file has been occupied in text mode. This is because when the file is opened in text mode, each number is stored as a character string.
- (b) If the number of fields in the structure increase (say, by adding address, house rent allowance etc), in that case writing structures using **fprintf ( )**, or reading them using **fscanf ( )** becomes quite clumsy.

Before we can eliminate these disadvantages, let us first complete the program that reads employee records created by the above program.

```

/*Read records from a file using structure*/

```

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    FILE *fp;
    struct emp
    {
        char name[40];
        int age;
        float bs;
    };

    struct emp e;
    fp = fopen ("EMPLOYEE.DATA", "w");

    if(fp==NULL)
    {
        printf("cannot open file");
        exit(1);
    }

    while(fscanf(fp, "%s%d%f", e.name, &e.age, &e.bs) != EOF)
        printf("\n %s %d %f", e.name, e.age, e.bs);
    fclose(fp);
    return 0;
}

```

Expected output:

```

Kanja    34 1250.000000
Wanja    21 1300.000000
Mwashe   34 1400.000000

```

### Using fread( ) and fwrite( ) functions

Let us now see a more efficient way of reading/writing records (structures). This makes use of two functions **fread ( )** and **fwrite ( )**.

#### ***Example: Program to write records to a file using fwrite ( )***

```

/*Receiving records from the key board & writing them to a file in a
binary mode*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

main( )
{

```



```

FILE *fp;
char another = 'Y';
struct emp
{
    char name[40];
    int age;
    float bs;
};

struct emp e;

fp=fopen ("EMP.DAT","wb");
if(fp==NULL)
{
    printf("cannot open file");
    exit(1);
}

while (another == 'Y')
{
    printf("\n Enter name, age and basic salary: ");
    scanf("%s%d%f", e.name,&e.age,&e.bs);
    fwrite(&e,sizeof(e), 1, fp);
    printf(" \n Add another record (Y/N)? ");
    fflush(stdin);
    another=getche ( );
}

fclose (fp);
return 0;
}

```

**Again, test run this program with the following input**

```

Enter name, age and basic salary: Lelei  24 1250
Add another record (Y/N)? Y
Enter name, age and basic salary: Koech 21 1300
Add another record (Y/N)? Y
Enter name, age and basic salary: Hadija 28 1400
Add another record (Y/N)? N

```

Most of this program is in similar to the one that we wrote earlier, which used **fprintf ()** instead of **fwrite ( )**. Note, however, that the file "EMP.DAT" has now been opened in binary mode.

The information obtained about the employee from the key board is placed in the structure variable **e**. then, the following statement writes to the structure to the file:

```
fwrite (&e,sizeof(e),1,fp);
```

Here, the first argument is the address of the structure to be written to the disk.

The second argument is the size of the structure in bytes. Instead of counting the bytes occupied by the structure ourselves, we let the program do it for us by using the **sizeof()** operator which gives the size of variable in bytes. This keeps the program unchanged in event of change in the elements of the structure. The third argument is the number of such structures that we want to write at one time. In this case, we want to write only one structure at a time. Had we had an array of structures, for example, we might want have wanted to write the entire array at once.

The last argument is the pointer to the file we want to write to.

***Example: Program to read records from a file using fread ( )***

```
/*Read the records from the binary file and display them on VDU*/
#include <stdio.h>
#include <stdlib.h>
main( )
{
    FILE *fp;
    struct emp
    {
        char name[40];
        int age;
        float bs;
    };

    struct emp e;

    fp=fopen ("EMP.DAT", "rb");
    if(fp==NULL)
    {
        printf("cannot open file");
        exit(1);
    }

    while (fread (&e, sizeof(e),1,fp)==1)
        printf("%s %d % f \n",e.name,e.age,e.bs);
    fclose (fp);
    return 0;
}
```

What results do you get when you run this program?

Here the **fread ()** function causes the data read from the disk to be placed in the structure variable **e**. The format of **fread()** is same as that of **fwrite ()**.

The function **fread** () returns the number of records read. Ordinarily, this should correspond to the third argument, the number of records we asked for - 1 in this case. If we have reached the end of file, since **fread**() cannot read anything, it returns a 0.

### **Note**

You can now appreciate that any database management application in C must make use of **fread** () and **fwrite** () functions, since they store numbers more efficiently, and make writing/reading of structures quite easy. Note that even if the number belonging to the structures increases, the format of **fread**() and **fwrite**() remains the same.

### **Simple database management application**

The following application uses a menu driven program. It has a provision to **Add, Modify, List and Delete** records, the operations that are common in any database management.

The following comments would help you in understanding the program easily.

- Addition of records must always take place at the end of existing records in the file, much in the same way you would add new records in a register manually.
- Listing records means displaying the existing records on the screen. Naturally, records should be listed from the first record to last record.
- In modifying records, first we must ask the user which record he intends to modify. Instead of asking the record number to be modified, it would be more meaningful to ask for the name of the employee whose record is to be modified. On modifying the record, the existing record gets overwritten by the new record.
- In deleting records, except for the record to be deleted, rest of the records must first be written to a temporary file, then the original file must be deleted, and the temporary file must be renamed back to original.
- Observe carefully the way the file has been opened, first for reading and writing, and if this fails(the first time you run this program it would certainly fail, because that time there is no data file existing), for writing and reading. It is imperative that file should be opened in binary mode.
- Note that the file is being opened only once and being closing only once, which is quite logical.

```
/*A menu driven program for elementary database management*/  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<conio.h>  
#include<string.h>  
main( )  
{
```

```

FILE *fp,*ft;
char another, choice;
struct emp
{
    char name[40];
    int age;
    float bs;
};

struct emp e;
char empname[40];

fp=fopen ("EMP.DAT", "rb+");
if (fp==NULL)
{
    fp=fopen ("EMP.DAT", "wb+");
    if(fp==NULL)
    {
        printf("cannot open file");
        exit(1);
    }
}

while(1)
{
    printf("\n1. Add Records");
    printf("\n2. List Records");
    printf("\n3. Modify Records");
    printf("\n4. Delete Records");
    printf("\n0.Exit");
    printf("\n\n Your choice  ");

    choice=getche ();
    switch(choice)
    {
        case '1':
            fseek(fp,0,SEEK_END);
            another = 'Y';
            while(another=='Y')
            {
                printf("\n Enter name,age and basic sal.  ");
                scanf("%s %d %f",e.name, &e.age, &e.bs);
                fwrite (&e,sizeof(e), 1, fp);
                printf("\n Add another Record(Y/N) ");
                fflush(stdin);
                another = getche ( );
            }
            break;

        case '2':
            rewind(fp);

```

```

        while (fread (&e,sizeof(e),1,fp)==1)
        printf("%s %d %f\n",e.name,e.age, e.bs);
        break;

case '3':
    another = 'Y';
    while (another=='Y')
    {
        printf("\n Enter name of employees to modify ");
        scanf("%s",empname);
        rewind(fp);
        while(fread (&e, sizeof(e), 1,fp)==1)

        if(strcmp(e.name,empname)==0){
            printf("\nEnter new name, age & bs  ");
            scanf("%s %d %f", e.name, &e.age, &e.bs);
            fseek(fp, -sizeof(e), SEEK_CUR);
            fwrite (&e, sizeof(e), 1, fp);
            break;
        }
    }

    printf("\n Modify another record(Y/N)  ");
    fflush(stdin);
    another = getche ();
}
break;

case '4':
    another = 'Y';
    while(another=='Y')
    {
        printf("\n Enter name of employee to delete  ");
        scanf("%s", empname);

        ft = fopen ("TEMP.DAT", "wb");
        rewind(fp);
        while(fread (&e, sizeof(e),1,fp)==1)
        {
            if (strcmp(e.name,empname)!=0)
                fwrite (&e, sizeof(e), 1, ft);
        }
        fclose (fp);
        fclose (ft);

        remove("EMP.DAT");
        rename("TEMP.DAT", "EMP.DAT");

        fp = fopen ("EMP.DAT", "rb+");

        printf("Delete another Record (Y/N)  ");

```

```

        fflush(stdin);
        another = getche ();
    }
    break;

    case '0':
        fclose (fp);
        exit(1);
    }
}
}

```

To understand how this program works, you need to be familiar with the concept of pointers.

A pointer is initiated whenever we open a file. On opening a file, a pointer is setup which points to the first record in the file. On using the functions **fread ()** or **fwrite ()**, the pointer moves to the beginning of next record. On closing a file the pointer is deactivated.

The **rewind ()** function places the pointer to the beginning of the file, irrespective of where it is present right now.

Note that pointer movement is of utmost importance since **fread** always reads that record where the file pointer is currently placed. Similarly, **fwrite ()** always writes the record where the file pointer is currently placed.

The **fseek()** function lets us move the file pointer from one place to another. In the program above, to move the pointer to the previous record from its current position, we used the function,

```
fseek(fp, -sizeof(e), SEEK_CUR);
```

Here, **-sizeof (e)** moves the pointer back by **sizeof(e)** bytes from the current position. **SEEK\_CUR** is a macro defined in "stdio.h"

Similarly, the following **fseek()** would place the pointer beyond the last record in the file.

in fact **-sizeof (e)** or **0** are just the offsets which tell the compiler by how many bytes should the pointer be moved from a particular position. The third argument could be either **SEEK\_END**, **SEEK\_CUR** or **SEEK\_SET** all these act as reference from which the pointer should be offset. **SEEK\_END** means move the pointer from the end of the file, **SEEK\_CUR** means move the file in reference to its current position and **SEEK\_SET** means move the pointer with reference to the beginning of the file.

if we wish to know where the pointer is positioned right now, we can use the function **ftell()**. It returns this position as a **long int** which is an offset from the beginning of the file. the value returned by **ftell()** can be used in subsequent calls to **fseek()**. A sample call to **ftell( )** is shown below:

```
position=ftell(fp);
```

where **position** is a **long int**,

### Revision Exercise

1. Define an input file handle called *input\_file*, which is a pointer to a type FILE.
2. Using *input\_file*, open the file *results.dat* for read mode as a text file.
3. Write C statements which tests to see if *input\_file* has opened the data file successfully. If not, print an error message and exit the program.
4. What will be the output of the following programs?

(a)

```
#include<stdio.h>
main( )
{
    char str[20];
    FILE *fp;

    fp = fopen (strcpy(str,"ENGINE.C"), "W");
    fclose (fp);
    return 0;
}
```

(b)

```
#include<stdlib.h>
#include<stdio.h>
main( )
{
    FILE *fp;
    char c;

    fp = fopen (strcpy(str,"ENGINE.C"), "W");
    fclose (fp);

    fp = fopen ("TRY.C","r");
    if(fp=NULL)
    {
        printf("Cannot open file");
        exit(1);
    }
}
```

```

    }
    while((c = getch(fp)) != EOF)
        putchar(c);
    fclose (fp);
    return 0;
}

```

(c)

```

#include<stdio.h>
main( )
{
    FILE fp,*fs,*ft;
    char str[80];

    /*TRIAL-Contains only one line It's a round, round, round world */

    fp = fopen ("TRIAL.C", "r");
    while(fgets (str,80,fp) != EOF)
        puts(str);

    return 0;
}

```

6. Create a small database program to keep track of books in a library. Your program should be menu driven and should allow a user to:

- a. Add a newly acquired book to the books file
- b. Search for a book.
- c. Modify a book's record
- d. Delete a book record
- e. Print a list of available books



## MODEL EXAMINATION PAPER

### INSTRUCTIONS:

- This examination paper has eight (8) questions.
- The candidate is required to attempt any five (5) questions.
- **Time:** 3 Hours

### Question One

- (a) What do you understand by 'structured programming'? (3 marks)
- (b) C language is said to be both portable and efficient. Explain. (4 marks)
- (c) Give the meaning of the following components of a C program.
- (i) Preprocessor directive
  - (ii) Declaration
  - (iii) Functions
  - (iv) Expression
  - (v) Comment
- (10 marks)
- (d) (i) What is a 'keyword'? (2 marks)
- (ii) What situation will make a keyword not to be recognised during the compilation of a C program. (1 mark)

### Question Two

- (a) Distinguish between a simple variable and an array variable. (2 marks)
- (b) Discuss four fundamental data types in C, giving examples and stating their conventional storage requirements. (12 marks)
- (c) (i) What is a symbolic constant? (2 marks)
- (ii) What is the advantage of using symbolic constants over direct constants? (1 mark)
- (d) What is a storage class? Explain how any two of storage classes are used in C. (3 marks)

### Question Three

(a) What is a structure?

(3 marks)

(b) (i) Set up a suitable structure for an invoice that should hold the following details:

Element	Type
Invoice number	integer
Customer number	integer
Invoice date	structure (with three integer elements; day, month, year)
Customer address	string (20 characters)
Item	structure [with product code (integer), unit price (float) quantity (float) , amount (double)]
Invoice Total	double

(7 marks)

(ii) Write a declaration statement that would create a 5-element array variable named **invoice\_file** from the structure type in b(i).

(2 marks)

(c) How is a structure different from a union?

(2 marks)

(d) (i) What is a user defined data type?

(2 marks)

(ii) Using a user defined type named invoices set up the structure template in b(i) above. Show how you can set two simple structure variables **invoice1**, **invoice2** using the new type.

(4 marks)

### Question Four

(a) Outline the stages of developing a working program in C.

(14 marks)

(b) (i) Why is linking necessary in a program?

(2 marks)

(ii) A program may compile successfully but fail to generate desired results. Why?

(2 marks)

(c) Kelly encountered the following error messages on compiling a program:

(i) Misplaced else

(ii) Statement missing

What advice would you offer him to debug the above errors?

(2 marks)

### Question Five

- (a) (i) C programs are basically made up of functions, one of which is called **main**. State three advantages offered by functions in C programs. (3 marks)
- (ii) What is a function prototype? (2 marks)
- (b) Write a program that requests two integers values and outputs the larger value on the screen. Use a function to perform the comparison of the two integers to determine the larger one. The main function should pass the entered values to this function. (10 marks)
- (c) Suggest the output of the following program. (5 marks)
- ```
#include <stdio.h>
int mult(int);
main()
{
    int a, count;
    for(count=1; count <=5; count++)
    {
        a = mult(count);
        printf("\n %d", a);
    }
}

int mult(int in_value)
{
    int prod;
    prod = in_value * in_value;
    return(prod);
}
```

### Question Six

- (a) Muajiri Company Ltd uses the following PAYE (Pay As You Earn) percentage tax rates for all its employees salary categories.

| Gross Salary (Ksh.) | PAYE rate (%) |
|---------------------|---------------|
| 1. 50,000 and above | 14            |
| 2. 40,000 - 50,000  | 12            |
| 3. 35,000 - 40,000  | 11            |
| 4. 25,000 - 35,000  | 8             |
| 5. 16,000 - 25,000  | 5             |
| 6. 9,500 - 16,000   | 3             |
| 7. Below 9500       | 0             |

(The rates are exclusive of the upper boundary salary figures for categories 2,3,4,5,6)

The following standard deductions apply to all employees.

N.S.S.F = Ksh. 80.00

N.H.I.F = Ksh. 200.00

Service charge = Ksh. 100.00

The overtime rate is Ksh. 300 for the first 50 hours an employee has worked overtime. Any extra overtime hour is paid at Ksh. 350.

At the end of the month, the payroll clerk runs a payroll program through which he enters each employee's basic salary into the computer and overtime hours worked as recorded in a claims form filled by the employee. The computer in turn adds up the basic salary and the overtime pay (if any) to get the gross pay.

The computer then determines the PAYE amount payable from the gross pay. Finally the employee's net pay is calculated using the formula:

Net pay = Gross pay - [PAYE + (N.S.S.F + N.H.I.F + Service charge)]

**Required:**

Write a program that performs the above mentioned payroll activities for a single employee and outputs the following on the screen:

- Gross pay
- PAYE amount
- Net pay

(15 marks)

(b) Suggest the output of the following program.

```
#include <stdio.h>
main()
{
    int pica = 0, delta = 0;
    while (pica <= 20)
    {
        if (pica % 5 == 0)
        {
            delta + = pica;
            printf(" \n %d  ", delta);
        }
        pica++;
    }
}
```

(5 marks)

### Question Seven

- (a) Give the meaning of each of the following terms, giving examples.
- (i) Escape sequence
  - (ii) Recursion
  - (iii) Binary operator
  - (iv) Conditional expression
- (8 marks)
- (b) Interpret the following statements:
- (i) `float sigma(int p, float s);`
  - (ii) `double *meta(float n, float m);`
  - (iii) `char char_value, *pc; pc = &char_value;`
  - (iv) `int nums[] = {34, 45, 67, 90, 57};`
- (4 marks)
- (c) s and t are integers with values 500 and 800 and stored at memory locations 1200 and 1205 respectively. ps and pt are integer pointers to s and t respectively. Give the results of:
- (i) `*ps + *pt`
  - (ii) `*ps++`
  - (iii) `(*pt)++`
- (3 marks)
- (d) Write a program using a *for* loop that counts down from 10 down to 0, displaying only the even numbers in this range. The numbers should be displayed using a pointer.
- (5 marks)

### Question Eight

- (a) State with examples, four types of operators used in C.
- (4 marks)
- (b) List three types of unary operators. Give an expression in each case to show how they are used.
- (6 marks)
- (c) The following table shows Kenya's average exports (in tonnes) of five commodities over four years.

|        | 1974  | 1975  | 1976  | 1977  |
|--------|-------|-------|-------|-------|
| TEA    | 18000 | 19450 | 23890 | 28820 |
| COFFEE | 20000 | 27000 | 29000 | 33452 |
| SISAL  | 3400  | 4501  | 3890  | 3973  |
| SUGAR  | 6500  | 7200  | 8100  | 8805  |
| FRUITS | 12780 | 13210 | 14300 | 15302 |

Given that the name of the above table is **EXPORTS**;

- (i) What are the values of `EXPORTS[5][2]`, `EXPORTS[3][1]/3 * 2` ?
  - (ii) Write a program that initializes the above export values in the table, computes and displays the total and average sales for *each year*.
- (4 marks)  
(6 marks)