Mount Kenya University

SCHOOL OF PURE AND APPLIED SCIENCES

BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY

BIT 1208: STRUCTURED PROGRAMMING

BIT 1208: STRUCTURED PROGRAMMING

COURSE OUTLINE

Pre-requisite: Introduction to Programming and Algorithms

Purpose: To introduce structured computer programming methods of advanced algorithms for writing programmes to solve general problems.

Objective

By the end of the course unit, the learner will be able to:

(i) Describe the structured computer programming paradigm
(ii) Use data control structures and advanced computer algorithms
(iii) Apply a high-level programming language like C
(iv) Write computer programmes or commands to solve general problems

Course Content

modularity and function, Program structure; documentation, processor directives, global variables, functions prototypes, main method, function definition, Integrated development environment**;** editing, compiling linking and executing, Data types**;** variables, constants, typedef and enumerated
types, Operators and expression Control structures, Use of arrays and unions, Functions; prototypes, definition, inline, arguments and parameters, recursions, pass by value, pass by reference, pointers, Standard input and output; file input and output
Use of C language

Teaching Methodology

- Lectures
- Tutorials

Assessments

A learner is assessed through ;

- Continuous Assessment Tests (CATs) (30%)
- End of semester examination (70%)
- Total = (100%)

Required text books

Forouzan B, and Gilberg R(2000), Computer Science: A Structured Programming Approach Using C (2nd Edition)

Uckan Y (1998), Problem Solving Using C: Structured Programming Techniques

Text books for further reading

Barron J (1984), BASIC Programming Using Structured Modules

COURSE OUTLINE

CHAPTER ONE: BASIC CONCEPTS 1
  Chapter Objectives 1
  Structured Programming defined 1
  Steps In Program Development 1
  C Language Basic Features 2
  Advantages of C over Other Languages 2
  C Programs' Components 3
    Keywords 3
    Functions 4
    Preprocessor directives and header files 5
    Comments 6
    Declaration statements 6
    Assignment and Expression statements 6
    Escape sequences 6
  Types Of Errors 7
    Syntax errors 7
    Logic Errors 8
    Semantic errors 8
  Guidelines To Good C Programming 8
  Revision Exercises 8
CHAPTER TWO: DATA HANDLING 10
  Chapter Objectives 10
  Variables 10
    Variable Names 10
  Basic Data Types 11
    The 'int' specifier 11
    The 'char' specifier 11
    The 'float' specifier 11
    The 'double' specifier 11
  Using printf( ) To Output Values 12
  Inputting Numbers From The Keyboard Using scanf( ) 13
  Types of Variables 10
  Storage Classes 15
    auto 15
    extern 15
    register 17
    static 17
  Constants 18
    Types of Constants 18
  Type Conversions 19
  Revision Exercises 20
CHAPTER THREE: OPERATORS 21
  Operators And Operands 21
    Arithmetic Operators 21
    Operator Precedence 23
  The sizeof operator 24

The Assignment Operator 25
The Conditional Operator 25
Unary Operators 26
Relational Operators 26
Logical operators 27
Revision Exercises 27
CHAPTER FOUR: CONTROL STRUCTURES 28
Introduction 28
Selection Structure 28
The if statement 28
if - else statement 29
Multiple Choice: else if 30
The ‘switch’ and ‘break’ statements 31
The ‘continue’ statement 34
The ‘goto’ statement 34
Looping 35
The ‘while’ loop 35
The ‘do .. while’ loop 36
The ‘for’ loop 37
Nesting statements 40
Nested ‘if’ statement 40
Nested ‘for’ statement 40
Revision Exercises 41
CHAPTER FIVE: FUNCTIONS 42
Introduction 42
Why Use Functions? 42
Defining A Function 43
Accessing A Function 45
Function Prototypes 47
Recursion 48
Revision Exercises 49
CHAPTER SIX: ARRAYS 52
Introduction 52
What Is An Array? 52
Declaring Arrays 52
Array Dimensions 53
Two – dimensional array 53
Initialising Arrays 54
Processing an array 57
Strings 61
Revision Exercises 64
CHAPTER SEVEN: POINTERS 66
What Is A Pointer? 66
Pointer Declaration 66
Pointer Operators 66
Pointers To An Array 67
Operations Performed Using Pointers 68
Pointer Arrays 69
Revision Exercises 71

CHAPTER EIGHT: STRUCTURES 72
Introduction 72
What Is A Structure? 72
Difference Between Structures And Arrays 72
Setting Up A Structure 72
Defining A Structure Variable 73
Initializing A Structure 74
Accessing Structure Members 74
Arrays Of Structures 75
Use of Structures 76
Enumerated data types 76
Uses of Enumerated data type 77
User Defined Types (typedef) 79
Unions 80
Revision Exercises 82
CHAPTER NINE: FILE HANDLING 84
Introduction 84
Understanding the concepts of streams 84
Opening a file 84
File opening modes 85
Reading from a file 86
Closing The File 86
Understanding feof( ) and ferror( ) 90
String (line) I/O functions : fputs( ) and fgets( ) 91
Formatted disk I/O functions: fprintf( ) and fscanf( ) 93
Using fread( ) and fwrite( ) functions. 97
Revision Exercises 103
Model Examination Papers 105
References 106

CHAPTER ONE: BASIC CONCEPTS

Chapter Objectives

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

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

1. 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.

2. 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.

3. Write the Code

It is the design Implementation by writing the (C) code i.e. translating your program design into C language instructions. You will use C's text editor or any other editor such as notepad. Using another editor such as notepad requires you to save your program with the extension **.c**.

4. Compile the code

A compiler converts the source code into object code. Object code are instructions in machine language. Computers have different machine languages. C compilers also incorporate code for C libraries into the final program. The libraries contain standard routines. The end result is an executable file that the computer can understand.

The compiler also checks errors in your program and reports the error to you for correction. The object code can never be produced if the source code contains syntax errors.

5. Run the program

This is the actual execution of the final code, usually preceded by linking [#]. Once the executable code is complete and working, it can be invoked in future by typing its name in a 'run' command.

[#] Combining all object code segments from different modules and libraries functions.

6. 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 mistakes.

7. Maintain and modify 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 programming language, unlike other languages such as PASCAL and FORTRAN developed for some specific uses. C is designed to work with both software and hardware. C has in fact been used to develop a variety of software such as:

- ✓ Operating systems: Unix and Windows.
- ✓ Application packages: WordPerfect and Dbase.

- **Source Code files**

When you write a program in C language, your instructions form the source code (or simply source file). C filenames have an extension .c. The part of the name before the period is called the base name and the part after the period is called the extension.

- **Object code, Executable code and Libraries**

An executable file is a file containing ready to run machine code. C accomplishes this in two steps.

- ✓ Compiling – The compiler converts the source code to produce the intermediate object code.
- ✓ The linker combines the intermediate code with other code to produce the executable file. C does this in a modular manner.

You can compile individual modules, and then combine the compiled modules later. Therefore, if you need to alter one module, you don't have to recompile the others.

Linking is the process where the object code, the start up code[*], and the code for library routines used in the program (all in machine language) are combined into a single file - the executable file.

Advantages of C over Other Languages

- **C Supports structured programming design features.**
  It allows programmers to break down their programs into functions. Further it 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**
  - ✓ C has been used to write operating systems such as Unix, Windows.
  - ✓ 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.

[*] Code that acts as interface between the program and the operating system.

- ✓ 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 Programs' 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**Error! Bookmark not defined.**
registerreturn short union continue
struct switch typedefconst extern
volatilewhile char enum static

A typical C program is made of the following components:

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

Sample Program

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 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.

Example: Basic C program features

```
/* By  Gichuru */
#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.

To call a function, you specify its name followed by a parenthesized list of arguments that you will be passing to it. If the function does not require any arguments, no arguments will be specified, and the parenthesized list will be empty. If there is more than one argument, the arguments must be separated by commas.

In the above program, line 7 causes the message enclosed in speech marks " " to be printed on the screen. Line 8 does the same thing.

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

Line 9 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 10 has the same effect as line 8.

Line11 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 3 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.

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, 6, 7 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 6 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 6) 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.

For example, this program:

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

```
    }
```

displays the following output on the screen.

```
                    This is line one
                    This is line two
                    This is line three
```

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
\v	vertical tab
\t	horizontal tab
\\	back slash
\’	Single quote (‘)
\”	Double quote (“”)
\0	null

Types Of Errors

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 try to 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 since good logic will reduce coding time as well as result in programs that are easy to understand, error free and easily maintainable.
- Declare all variables before using them in any statements.
- Use sensible names for variables. Use of general names such as **n** instead of **net_sa**l 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 I/O statements with a semi colon.
- Restrict your variables names to eight characters. The compiler may truncate part of a variable name and possibly give it the meaning of another variable e.g. *Shakespeare* and *Shakespencil*.
- 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 Exercises

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. Provide the meaning of the following keywords, giving examples of their use in C programs.
   (i) void
   (ii) return
   (iii) extern
   (iv) struct
   (v) static
5. C is both ‘portable’ and ‘efficient’. Explain.
6. C is a ‘case sensitive’ language. Explain.
7. The use of comments in C programs is generally considered to be good programming practice. Why?

CHAPTER TWO: DATA HANDLING

Chapter Objectives

- 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 function
- Expand **printf()** capabilities.

Variables

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 type, followed by the name of one or more variables. For example,
int high, low, results[20];

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

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

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 variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name. Most modern ones don't apply this limit though. The rules governing variable names also apply to the names of functions.

Types of Variables

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

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.

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.

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 formal 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, they may be variables too.

Code	**Format**
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase ‘e’)
%E	Scientific notation (lowercase ‘E’)
%f	Decimal floating point
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	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;
}
```

Exercises

1. Enter, compile, and run the two programs above.
2. 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

Inputting Numbers 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. It allows the function to place a value into one of its arguments.

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;
}
```

Exercises

1. Enter, compile and run the example programs.
2. Write a program that inputs two floating-point numbers (use type **float)** and then displays their sum.
3. Write a program that computes the volume of a cube. Have the program prompt the user for each dimension.

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

It is used to declare *automatic variables.* Automatic variables are simply local variables, which are auto by default. You will never see auto used in any C program.

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 the second file, 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 */
}
```

*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.*

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 */
}
```

register

When you specify a register variable, you are telling the compiler that you want access to that variable as fast as possible. In other words, using register variables minimizes access time.

No matter what storage method used, only a given number of variables can be granted the fastest possible access time. For example, the CPU has a limited number of registers. When fast-access locations are exhausted, the compiler is free to convert register variables into regular variables. For this reason, you must choose carefully which variables you modify with register.

One good choice is to make a frequently used variable such as the variable that controls a loop, into a register variable. The more times a variable is accessed, the greater the increase in performance when its access time is decreased.

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'.

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

Note:
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, these two assignments are equivalent.

```
char ch;
ch = "A';
ch = 65;
```

Types of Constants

Constants can be used in C expressions in two ways:

- **Directly**
  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

- **Using a 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;
}
```

Note:
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.

Type Conversions

In an assignment statement in which the type of the right side differs from that on the left, the type of the right side is converted into that of the left. When the type of the left side is larger than the right side, this process causes no problems. However, when the type of the left side is smaller than the type of the right, data loss may occur.

When converting from a **long double** to a **double** or from a **double** to **float**, precision is lost. When converting from floating point value to an integer value, the fractional part is lost, and if the number is too large to fit in the target type, a garbage value will result.

As stated when converting from a floating-point value to an integer value, the fractional portion of the number is lost. The following program illustrates that fact.

```
#include<stdio.h>
main()
{
        int i;
        float f;
        f  = 1234.0098;
        i = f;
        printf(“ %f  %d “,f, i);
        return 0;
}
```

Revision Exercises

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 x 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.

CHAPTER THREE: OPERATORS

Operators and Operands

An **operator** is a component 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.

Note*: 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, 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 .

Exercise

Suppose a = 10, b = 3, v1 = 12.5, v2 = 2.0, c1 =’P’, c2 = ‘T’. Compute the result of the following expressions.

a + b	v1 * v2
a - b	v1 / v2
a * b	c1
a / b	c1 + c2 +5
a % b	c1 + c2 +’9’

Note:

(i) c1 and c2 are character constants

(ii) ASCII codes for 9 is 57, P = 80,T = 84.

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)

***Note*:** Whichever type of result an expression evaluates to, a programmer can convert the result to a different data type if desired. The general syntax of doing this is:

(data type) expression.
The data type must be enclosed in parenthesis (). For example the expression (i + f) above evaluates to 12.5. To convert this to an integer, it will be necessary to write
(int) (i + f).

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:
(25.0 + 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.

Exercise

The roots of a quadratic equation $\mathbf{ax^2 + bx + c = 0}$ can be evaluated as:

$$\mathbf{x1 = (-b + \sqrt{}(b^2 - 4ac))/2a}$$
$$\mathbf{x2 = (-b + \sqrt{}(b^2 - 4ac))/2a}$$

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.

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 minute and seconds \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(“% d seconds is % d minutes,% seconds\n “ ,sec,min,sec_left);
  return 0;
}
```

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;

}
```

The Assignment Operator

The Assignment operator (=) is a value assigning operator. There are several other assignment operators in C. All of them assign the value of an expression to an identifier.

Assignment expressions that make use of the assignment operator (=) 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 %

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 singe operand to produce a value. The operators may precede the operand or are 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)

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)

Revision Exercises

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

CHAPTER FOUR: CONTROL STRUCTURES

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**. For example, the following code prints digits from 1 to 5.

```
for(digit = 1;  digit < = 5; digit++)
     printf("\n %d", digit)
```

Selection Structure

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 that 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);
    }
```

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)
          statement;1
     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");
```

Multiple Choice: else if

Used when two or more choices have to be made.

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’ and ‘break’ statements

The *switch* - *break* 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 values */
   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
There is no match in your choice
```

The structure of a switch 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  */
```

The 'continue' statement

Like the break statement the continue statement is a jump that interrupts the flow of a program. It is used in loops to cause the rest of an iteration to be skipped and the next iteration to be started.

If a break is used in a loop it quits the entire loop.

The 'goto' 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 **Alternative 2**

```
if (a>14)            if (a>14)
```

```
  goto a;                     sheds=3;
 sheds=2;                   else
 goto b;                      sheds=2;
a: sheds=3;                 k=2*sheds;
b: k=2 * sheds;
```

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).
(v) Read in a number, say x.
(vi) Add the value of x to current value of sum.
(vii) Increase the value of count by 1.
(viii) Calculate the average: Divide the value of sum by n.
(ix) Write out the calculated value of average.

Solution

```
/* 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.

General 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.

General 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);
```

Output

```
0
1
2
3
4
5
```

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.

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 Exercises

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

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

2. 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.
3. In what circumstance is the *continue* statement used in a C program?
4. 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.
5. Write a *while* loop that will calculate th*e* 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.

CHAPTER FIVE: 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. A function will carry out its intended action whenever it is accessed (whenever the function is called) from some other portion of the program. The same function can be accessed from several different places within a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed.

Generally the function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via special identifiers called *arguments* (also called parameters), and returned via the return statement. Some functions however, accept information but do not return anything.

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; 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.***

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 easier to debug and their logical structure is more apparent 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.

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.

Defining A Function

A function definition has two principle components; the first line (including the argument declaration), and the body of a function. The first line of a function definition contains the value returned by the function, followed by the function name, and (optionally) a set of arguments separated by commas and enclosed in parenthesis. Each argument is preceded by its associated type declaration. Any empty pair of parenthesis must follow the function name if the function definition does not include any arguments.

In general terms, the first line can be written as:
data-type ***functionname*** *(type 1 argument 1 , type 2 argument 2 ,……, type n argument n)*

Where data-type represents the data type of the item that is returned by the function. *functionname* represents the function name , and type 1 , type 2 ,…….., type n represents the data types of the arguments, *argument 1, argument 2 ,……..argument n .*

The data types are assumed to be of type *int* if they are not shown explicitly. However, the omission of the data type is considered poor programming practice, even if the data items are integers.

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). The names of the formal arguments need not be the same as the names of the **actual arguments** in the calling portion of the program. Each formal argument must be of the same data type, however, as the data item it receives from the calling portion of the program.

The remainder of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the body of the function. Like any other compound statement, this statement can contain expressions statements, other compound statements, control statements and so on. It should include one or more return statements in order to return a value to the calling portion of the program. A function can access other functions. In fact it can access itself (a process known as *recursion).*

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 1: Factorial of an integer n

The factorial of a positive integer quantity *n* is defined as n! = 1 * 2 * 3 *…….* (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 functional 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.

Example 2: Factorial of an integer n (return type not specified)

The following definition is identical to that in **Example 1** except that the first line does not include a type specification for the value that is returned by the function.

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

The function expects to return an ordinary integer quantity since there is no explicit type declaration in the first line of the function definition. However the quantity being returned is declared as a long integer within the function. This inconsistency can result in an error (some compilers will generate a diagnostic error and then stop without completing the compilation). The problem can be avoided however by adding a long int type declaration to the first line of the function definition as **Example 1** shows.

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),
}
```

The keyword **void** added to the first line indicates that the function does not return anything.

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 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 4: 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

In the previous function examples, the programmer -defined function has always preceded main. Thus when the programs are compiled, the programmer-defined function will have been defined before the first function access. However many programmers prefer a top drawn approach in which main appears ahead of the programmer-defined function definition. In such a situation, the function access (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;

data_type ***function_name*** *(type 1 argument 1, type 2 argument 2, ., type n argument n);*

Where *data_type* represents the type of the item that is returned by the function, *function_name* represents the name of the function, type 1, type 2, … …., type n represent the types of the arguments 1 to n.

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 within the function prototype need not be declared else where in the program since these are “dummy” argument names that are recognised only within the prototype. In fact, the argument names can be omitted (though it is not a good idea to do so). However the arguments data types are essential.

In practice, the argument names usually included are often the same as the names of the actual arguments appearing in one of the function calls. The data types of the actual arguments must conform to the data types of the arguments within the prototype.

Function prototypes are not mandatory in C. They are desirable however because they *further facilitate error checking between the calls to a function and the corresponding function definition.*

Example 6: Factorial of an integer n

Here is a complete program to calculate the factorial of a positive integer quantity. The program utilises the function factorial defined in example 1 and 2. Note that the function definition 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);
}
```

The programmer-defined function makes use of an integer argument (n) and two local variables (an ordinary integer and a long integer). Since the function returns a long integer, the type declaration long int appears in the first line of the function definition.

Recursion

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

To use recursion, two conditions must be satisfied:
(i) The problem must be written in a recursive form.
(ii) There must be a stopping case (terminating condition).

Example

The factorial of any possible integer can be expressed as;
*n ! =n * (n-1) * (n-2) *………* 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)!   (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.

Example: factorial 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.

Revision Exercises

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.

CHAPTER SIX: ARRAYS

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.

What Is An Array?

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

Examples:

(i) int c[100];

int is the data type of the array (type of elements held), **c** is the array name
100 is the maximum number of elements (array size)

(ii) static char message[20]; A 20 character-type array called **message**. The individual array values persist within function calls (static).

Array Dimensions

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

(i) A **one-dimensional** array requires a single index e.g. int numbers [10];
Resembles a single list of values and therefore requires a single index to vary between 0 to (array size -1).

(ii) **Multi dimensional arrays**
They are defined the same way as a one-dimensional array except that a separate pair of square brackets is required for each subscript. Thus a two-dimensional array will require two pairs of brackets, a three dimensional array will require three pairs of square brackets, etc.

Two – dimensional array

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).

Examples of two-dimensional array declarations

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

Example

Two-dimensional array representing sales ('000 tonnes for a product in four months for five years).

	Yr1	**Yr2**	**Yr3**	**Yr4**	**Yr5**
Month 1	23	21	27	23	22
Month 2	24	20	19	18	20
Month 3	26	23	26	29	24
Month 4	27	25	24	23	25

Arrays, like simple variables can be automatic, external or static.

An **automatic array** is one defined inside a function including formal arguments. C allows us to initialise automatic array variables as follows.

```
main()
{

        int marks[5] = {30, 40, 50, 90, 60];
        ---------
        ---------
}
```

Because the array is defined inside main, its an automatic array. The first element marks[0] is assigned the value of 30 , marks[1] as 40 and so on.

An **external array** is one defined outside a function.

They
(i) are known to all functions following them in a file e.g. from above, both main ( ) and feed ( ) can use and modify the array SOWS.

(ii) Persist (retain values) as long as the program runs. Because they are not defined in any particular function, they don't expire when a particular function terminates.

Have a look at the following example.

```
int SOWS [5] =  {12, 100, 8, 9 ,6};
main()
{

   ----------
   ----------

}

int feed(int n)
{

   ---------
   ---------

}
```

(iii) A **static array** is local to the function in which it is declared but like an external array, it retains its values between function calls and is inititialised to zero by default.

Example

```
int account(int n, int m)
{
        static int k[2] = {343, 332};
        ---------
        ---------
}
```

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 though 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 though 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 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 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.

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;
}
```

Here is the output

```
                Month 1 has 31 days.
                Month 2 has 28 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)

Note:
The number of items in the list should match the size of the array.

Processing an array

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.

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 */
```

Output

```
How many numbers will be averaged ?    3
i = 1               x = 3
i = 2               x = - 4
i = 3               x = 7

The average is 2

i = 1               x = 3           d = 1
i = 2               x = - 4                 d = - 6
i = 3               x =  7          d =  5
```

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);
        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  */
        for(t = 0; t < count; t++)
          printf(" %d ", item[t]);
        return 0;
}
```

Example: Two – Dimensional array of scores

```
#include<stdio.h>
#define STUDENT 5      /* 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; /*Calculate the average of each student */
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);              /* Compute the class average */
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 */
for(rows=0;rows<STUDENT; rows++)
for(cols=0;cols<CATS;cols++)
{
            printf("\n Student %d, Cat %d ",rows+1, cols+1);
            printf("\n\t %d \n", SCORES[rows][cols]);
}
               return 0;
}
```

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. 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 then the largest string it is going to hold, to make room for the null.

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);
        for( i = 0 ; str[i]; i++)
        printf(" %c", str[i]);
        return 0;
}
```

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.

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.

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.

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.

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);
```

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"));
```

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: ");
   gets(str1);
   printf(" Enter the second string: ");
   gets(str2);
   /* See how long the strings are */
   printf( " %s is %d characters long \n ", str1, strlen(str1));
   printf( " %s is %d characters long \n ", str2, strlen(str2));
   /* Compare the strings */
   i = strcmp(str1, str2);
   if ( ! i)
             printf("The strings are equal. \n");
   else if (i < 0)
             printf("%s is less than %s  \n", str1,str2);
   else
             printf("%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( "%s \n", str1);
   }
```

```
        /* copy str2 to str1 */
        strcpy(str1, str2);
        printf( "%s  %s \n", str1, str2);
        return 0;
}
```

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 Exercises

1. What is an array structure?
2. Give and explain the syntax of a two-dimensional array declaration.
3. 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

If you were to implement the above table in a C program:
   (a) 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: (SCORES[3][3] % 11) *3?
      (iii) Write a complete program that initializes the above values in the table, computes and displays the total mark and average scored by *each student*.
4. Show how to initialise an integer array called **items** with the values 1 through 10.
5. (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.

CHAPTER SEVEN: POINTERS

What Is A Pointer?

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. Lets 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.

Pointers To An Array

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

For example
e.g. 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 compliler calculates the address.

Another way to perform the assignments is to use a printer. 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 synonynm to the array’s starting addres, we can use:
p = line;
We can now perform the assignments:
***p = ‘a’; and *(p+1) = ‘b’;**

Example: Array maniplulation 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;
}
```

Output

```
        p1 =  234, *p1=10, &p1 =3606.
```

Operations Performed Using Pointers

(a) **Assignment**

One can assign an address to a pointer by:

(i) Using an array name or

(ii) Using the address operator

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

(b) **Dereferencing (value – finding)**

The * operator gives the value pointed to.

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

(c) **Take a pointer address**

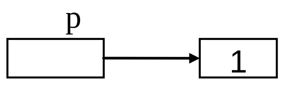
Pointer variables have an address and a value. The & operator tells us where the pointer itself is stored.

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

(d) **Incrementing or decrementing a pointer**

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.



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 (p)+ +;

Incrementing a pointer to an array involves moving to the next element of an array.

Thus, from the above array example, p1+ + makes the pointer point to the next element, x[1]. Therefore p1++ becomes 20.

Pointer incrementation arthmetic 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.

(c) **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 200th integer past the one to which p was currently pointing to.

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

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 folows;

```
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;
```

Example: Demonstrating pointers

Program that uses a for loop that counts from 0 to 9. It puts in the numbers using a pointer.

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

Example : Further demonstration of 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;
}
```

Output

```
        u1 = 16,  u2 = 16.
```

Explain why.

Note

- 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 the 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.

Revision Exercises

1. 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 rotation.
   (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.
2. What does this fragment display?

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

3. Write a program that assigns a value to a variable indirectly by using a pointer to that variable.
4. 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?
5. Write a program that passes a pointer to an integer variable to a function. Inside that function, assign the variable the value –1. After the function has returned, demonstrate that the variable does, indeed contain –1 by printing its value.

CHAPTER EIGHT: STRUCTURES

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***.

What Is A Structure?

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

Difference Between Structures And Arrays

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

Setting Up A Structure

A template is the master plan describing how a structure is put together.

In our example, we can have the following template;

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

The above describes a structure made up of a character array **name**, and int variable **marks**.

Explanation

- The keyword *struct* announces to the computer that what follows is a structure data type template.
- The *tag* follows: which is a shorthand label that can be used later to refer to this structure. In the above example, the tag name is student.
- A list of structure members enclosed in a pair of braces. Each member is described by its own declaration. For example; the name portion is a character array of SIZE elements.

Note: Members of a structure can be any data types including other structures.

Example: Bank Account

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

```
 struct Bank_ account
 {
          int acc_number;
          char acc_type;
          char holder_name [30];
         float balance;
  };
```

Defining A Structure Variable

The structure template doesn't tell the computer how to represent the data. Having set up the template we can declare a structure variable as follows;

```
        struct student  mystudent;
```

The computer creates a variable mystudent following the structure template. It allocates space for a character array of SIZE elements and for an integer variable.

One can declare as many variables of type **student** as possible. For example,

struct student mystudent, student1, x, y, z; *etc.*

***Note**:* One can combine the process of defining the structure template and the process of defining a structure variable as follows;

```
        struct student
        {
                    char name;
                    int marks;
          }mystudent;
```

A general form of a structure can be given as follows;

```
        struct tagname{
                type1 element1;
                type2 element2;
                        .
                        .
                typeN elementN;
                }variable list;
```

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.

Example

```
static 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

Individual structure members are accessed using a period (.) - structure member operator.

For example **mystudent.marks** which means the marks portion of struct mystudent.

Note: You can use mystudent.marks exactly the way you use other integer variables.
For example, you can use **gets(mystudent.name);**

or

```
scanf(“%d”, &mystudent.marks);
```

Note*:* Although **mystudent** is a structure, **mystudent.marks** 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 *&mystudent.marks does.*

If **stud1** is another student structure declared as follows;

> **struct student stud1;**, then it is possible to read the name and marks into the variable using the statements;

```
gets(stud1.name)
scanf(“%d”, &stud1.marks);
```

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;
}
```

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 30
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 ", count+1);
             scanf("%s", mystudent[count].name);
                scanf("%d", &mystudent[count].marks);
            }
            return 0;
}
```

Use 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

Enumerated data types
This is a data type that gives you an opportunity to invent your own data type and define what values the variable of this data type can take. This can help in making the program listings more readable, which can be an advantage when a program gets complicated.

Using enumerated data type can also help reduce programming errors.

Example
We could invent a data type called **mar_status** (for marital status) which can have four possible values - single, married, widowed or divorced. *(Do not confuse these names with variable names; they are only*

possible values for marital status).

The format of the **enum** definition is similar to that of a structure. The example can be implemented as follows:

```
enum mar_status
{
   single,
   married,
   divorced,
   widowed
};

enum mar_status person1, person2;
```

Like structures, this declaration has two parts:

(a) The first part declares the data type and specifies its possible values. These values are called 'enumerators'.
(b) The second part declares variables of this type.

Now we can give values to these variables:

```
person1 = married;
person2 = divorced;
```

We cannot use the values that are not in the original declaration. Thus the expression,

```
        person1 = unknown;
```

would cause an error.

Internally, the compiler treats the enumerators as integers. Each value on the list of permissible values corresponds to an integer, starting with 0. Thus in our example, single is stored as 0, married is stored as 1, divorced as 2 and widowed as 3.

This way of assigning numbers can be overridden by the programmer by initializing the enumerators to different integer values as shown below.

```
enum mar_status
{
   single = 100,
   married = 200,
   divorced = 300,
   widowed = 400
};
enum mar_status person1, person2;
```

Uses of Enumerated data type

Enumerated variables are usually used to clarify the operation of a program. For example, if we need to use employee departments in a payroll, it makes the listing easier to read if we use values like Assembly, Manufacturing, Accounts, rather than the integer values 0, 1, 2, etc.

Example: Illustrating the use of enumerated data type

```
#include<string.h>
#include<stdio.h>
main()
{
        enum emp_dept
        {
           assembly,
           manufacturing,
           accounts,
           stores
        };
        struct employee
        {
           char name[30];
           int age;
           float bs;
           enum emp_dept department;
        };
        struct employee e;
        strcpy(e.name, “Rodgers Okumu”);
        e.age = 30;
        e.bs = 20000.50;
        e.department = manufacturing;

        printf(“ Name is %s \n”, e.name);
        printf(“ Age = %d\n”,e.age);
        printf(“Basic Salary = %f \n”, e.bs);
        printf(“Department = %d \n”, e.department);

        if(e.department == accounts)
        printf(“Rodgers is an accountant”);
           else
        printf(“Rodgers is not an accountant”);
        return 0;
}
```

Explanation

We first defined the data type **enum emp_dept** and specified the four possible values, namely **assembly, manufacturing, accounts and stores**. Then we defined a variable **department** of the type **enum emp_dept** in a structure. The structure **employee** has three other elements containing employee information.

The program first assigns values to the variables in the structure. The statement,

e.department = manufacturing;

assigns the value **manufacturing** to **e.department** variable. This is much more informative to anyone reading the program than a statement like,

e.department = 1;

A weakness of using enum variables is shown in the next part of the program. *There is no way to use enumerated variables in input/output functions like*
printf( ) *and* ***scanf( ).***

The **printf( )** function is not smart enough to perform the translation and hence the department is printed out as **1** and not **manufacturing**.

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 actualy 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 2: 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.

Format of a union

```
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 Exercises

1. How is a structure different from:
   (a) a union?
   (b) an array?
2. Give a suitable structure declaration for a bank account that should hold the details; Account number (int), Account type (character), account holder name (string), account balance (float)
3. Show how to create a structure called s_type that contains these five elements:

```
                char ch;
                float d;
                int i;
                char str[80];
                double balance;
```

   Also define one variable called s_var using this structure.
4. What is wrong with this fragment?

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

5. 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

6. How is a structure different from a union?

7. (a) What is a user defined data type?

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

8. (a) Declare a structure **card** which stores the following library information.

   Student Number (int)
   Student Name (25 characters)
   Course Code (int)
   Telephone (12 characters)
   Address (14 Characters)

   (b) Set up a structure called **card1** using typedef. It should store similar data items to **card**.

CHAPTER NINE: FILE HANDLING

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.

Understanding the concepts of streams
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 is 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();
}
```

Reading from a file
Once the file has been opened for reading using **fopen(),** the file's contents are brought into memory (partly or wholly) and a pointer points to the very first character. To read the file's contents from the memory there exists a standard library function called **fgetc( ).**

For example,
A character declared as **char ch;** may be used to read the contents of a file as below:
ch = fgetc(fp);

fgetc( ) function reads the character from current pointer position , advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable **ch.** Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer **fp.**

Closing The File
When we have finished reading from the file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose(fp);
```

This deactivates the file and it can no longer be accessed using **fgetc( )**,

Writing to a file
The function **fputc( ),** also from the C standard library writes to the file signified by the file pointer such

as **fp**.

Note: The **fgetc( )** function reads the next byte from the file described by **fp** and returns an integer. The reason that it returns an integer is that if an error occurs, **fgetc( )** returns **EOF (**A macro has been defined in the file **stdio.h**), which is an integer value. The **fgetc( )** also returns **EOF** when the end of file is reached.

The **fputc( )** returns the character written if successful or **EOF** if an error occurs.

Historical note: The traditional names for **fgetc( )** and **fputc( )** are **getc( )** and **putc( ).** The ANSI standard still defines these names, and they are essentially interchangeable with fgetc( ) and fputc( ). One reason the names were added was for consistency. All other ANSI file system functions names begin with 'f' and so the 'f' was added to the getc( ) and the putc( ). The traditional names are still supported by the ANSI standard. If you see programs that use getc( ) and putc( ), do not worry, they are essentially different names for fgetc( ) and fputc( ).

Example: Basic file - system functions

1.The program demonstrates the four file – system functions you have learned so far. First, it opens a file called MYFILE for output. Next, it writes the string "This is a file system test" to the file. Then it closes the file and reopens it for read operations. Finally, it displays the contents of the file on the screen and closes the file.

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

char str80] = "This is a file system test";
main()
{
        FILE *fp;
        char *p;
        int i;

        /* Open myfile for output */
        if((fp = fopen(("myfile", "w")) = = NULL)
        {
                printf("Cannot open file \n");
                exit(1);
        }
        /* Write str to disk */
        p = str;

        while *p)
        {
                if(fputc(*p, fp)= = EOF)
                {
                        printf("Error writing file \n");
                        exit(1);
                }
                p++;
        }
        fclose(fp);
```

```
                /* Open file for input */
                if((fp = fopen((“myfile”, “r”)) = = NULL)
                {
                        printf(“Cannot open file \n”);
                        exit(1);
                }
                /* read back the file */
                for(; ;)
                {
                        i= fgetc(fp);
                        if(i = =  EOF)
                                break;
                                putchar(i);
                }
                fclose(fp);
                return 0;
}
```

Note:
The **exit( )** function is used to terminate execution of a program.

putchar( ) outputs a character to the screen (Its input opposite is **getchar( )**. If an output error occurs, **EOF** is returned. If output to the screen fails, the computer has probably crashed anyway, so most programmers do not bother checking the return value of putchar( ) for errors.

The reason that you might want to use putchar( ) rather than printf( ) with the %c specifier to output a character is that putchar( ) is faster and more efficient.

In this version, the return value of **fgetc( )** is assigned to **int**. The value of this integer is then checked to see if the end of the file has been reached. For most compiler, however, you can simply assign the value returned by **fgetc( )** to a **char** and still check for **EOF** as is shown in the following version.

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

      char str80] = “This is a file system test”;
      main()
      {
                FILE *fp;
                char ch, *p;

                /* Open myfile for output */
                if((fp = fopen((“myfile”, “w”)) = = NULL)
                {
                        printf(“Cannot open file \n”);
                        exit(1);
                }
                /* Write str to disk */
                p = str;

                while *p)
                {
                        if(fputc(*p, fp)= = EOF)
                        {
                                printf(“Error writing file \n”);
                                exit(1);
                        }
                        p++;
                }
                fclose(fp);

                /* Open file for input */
                if((fp = fopen((“myfile”, “r”)) = = NULL)
                {
                        printf(“Cannot open file \n”);
                        exit(1);
                }
                /* read back the file */
                for(; ;)
                {
                        c = fgetc(fp);
                        if(ch = = EOF)
                                break;
                                putchar(ch);
                }
                fclose(fp);
                return 0;
}
```

The reason this approach works with most C compilers is that when a **char** is being compared with an **int**, the char value is automatically elevated to an equivalent **int** value.

There is however, even a better way to code this program. For example, there is no need for a separate comparison step because the assignment and the comparison can be performed at the same time within the **if**, as shown here.

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

        char str80] = “This is a file system test”;
        main()
        {
                FILE *fp;
                char ch, *p;

                /* Open myfile for output */
                if((fp = fopen((“myfile”, “w”)) = = NULL)
                {
                        printf(“Cannot open file \n”);
                        exit(1);
                }
                /* Write str to disk */
                p = str;
```

```
          while *p)
          {
                    if(fputc(*p, fp)= = EOF)
                    {
                              printf("Error writing file \n");
                              exit(1);
                    }
                    p++;
          }
          fclose(fp);

          /* Open file for input */
          if((fp = fopen(("myfile", "r")) = = NULL)
          {
                    printf("Cannot open file \n");
                    exit(1);
          }
          /* read back the file */
          for(; ;)
          {
                    if(ch = c = fgetc(fp)) = = EOF) /* The change is
                                                                      here */
                              break;
                              putchar(ch);
          }
          fclose(fp);
          return 0;
}
```

A professional programmer would write the last **if** structure as follows:

```
                    if(ch = c = fgetc(fp)) != EOF) /* The change is
                                                            here */
                              putchar(ch);
```

Notice that now each character is read, assigned to **ch** and tested against **EOF**, all within the expression of the **while** loop that controls the input process. This is much more efficient than the original version.

Understanding feof( ) and ferror( )

As learnt, when fgetc( ) returns EOF, either an error has occurred or the end of the file has been reached, but how do you know which event has occurred?

The **feof( )** function returns non - 0 if the file associated with fp has reached the end of file. Otherwise, it returns 0. This method works for both text and binary files.

The **ferror( )** function returns non -0 if the file associated with fp has experienced an error, otherwise, it returns 0.

Using the feof( ) function, this code fragment shows how to read the end of a file.

```
FILE *fp;
.
```

```
.
.
while(!feof(fp))
        fgetc(fp);
```

This code works for any kind of file and is better in general than checking for **EOF**. However, it does not provide for error checking. Error checking is added here:

```
FILE *fp;
.
.
.
while(!feof(fp))
{
        fgetc(fp);
        if ferror(fp))
        {
                printf(" file error \n");
                break;
        }
}
```

Instead of printing the error message using **printf( )**, we can use the standard library function **perror( )** which prints the error message specified by the compiler.

For example, the above fragment, instead of writing ,
printf(" file error \n"); we could write **perror("TRIAL");**
When the error occurs, the error message that is displayed is:

```
        TRIAL: Permission denied
```

String (line) I/O functions : fputs( ) and fgets( )
When working with text files, C provides four functions which make file operations easier. These are **fputs( ), fgets( ), fprintf( )** and **fscanf( )**.

Similar to fgetc( ) and fputc( ), fgets( ) and fputs( ) read strings from and writes stings to a file respectively.

Example: Using fputs( )

```
/* Receives string from keyboard and writes them to file */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main()
{
        FILE *fp;
        char str[80];
        fp = fopen("POEM.TXT", "w");
        if(fp==NULL)
        {
                printf(“Cannot open file \n“);
                exit(1);
        }
        printf(“\n Enter a few lines of text: \n“);
        while(strlen(gets(str)) > 0)
        {
                fputs(str, fp);
                fputs(“\n”);
        }
                fclose(fp);
                return 0;
}
```

And here is a sample run of the above poem program..

```
Enter a few lines of text:
        The earth and the underworld
        Lie side by side
        They face each other
        Like antagonistic pairs
```

Notice that each string is terminated by hitting the <ENTER> key. To terminate the execution of the program, hit <ENTER> at the beginning of a line. This creates a string of zero length., which the program recognises as the signal to close the file and exit.

We have set up a character array (str) to receive the string; the fputs( ) function then writes the contents of the array to the disk. Since **fputs( )** does not automatically add a new line character to the end of the string, we must do this explicitly to make it easier to read the string back from the file.

Example: Using fgets( )

```
/* Reads strings from the file and displays them on the screen */
#include <stdio.h>
#include <stdlib.h>
main()
{
        FILE *fp;
        char str[80];

        fp = fopen(“POEM.TXT”, “r”);
        if(fp = = NULL)
        {
                printf(“Cannot open file \n“);
                exit(1);
        }
        while(fgets(str,79, fp)!= NULL)
                printf(“%s”, str);
                fclose(fp);
                return 0;
}
```

And here is the output:

The earth and the underworld
Lie side by side
They face each other
Like antagonistic pairs

The function **fgets( )** takes three arguments. The first is the address where the string is stored and the second is the maximum length of the string. The argument prevents fgets( ) from reading in too long a string and overflowing the array. The third pointer as usual, is the pointer to the structure **FILE**. When all the lines from the file have been read, we attempt to read one more line, in which case fgets( )returns a NULL.

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;
}
```

And here is the output of the program.

```
Enter name, age and basic salary
James 34 1550
Another employee (Y/N)? Y
Enter name, age and basic salary
Mary 24 1200
Another employee (Y/N)? Y
Enter name, age and basic salary
Peter 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 tom **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 key board.

Suppose we want to read back the names, ages and basic salaries of different employees which we stored through the earlier program into the EMPOYMENT.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;
}
```

And here is the out put .

```
James 34 1550.000000
Mary 24 1200.000000
Peter 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 a file using 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;
}
```

And here is the output of the program.

```
        Enter name,  age and basic salary:  John  34  1250
        Add another record (Y/N)? Y
        Enter name,  age and basic salary:  Anne  21 1300
        Add another record (Y/N)? Y
        Enter name,  age and basic salary:  Rodgers 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;
}
```

And here is the output of the program.

```
John      34  1250.000000
Anne      21  1300.000000
Rodgers   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;
}
```

And here is the out put.

```
Enter name, age and basic salary:  Sarah  24  1250
Add another record (Y/N)?  Y
Enter name, age and basic salary:  Richard 21 1300
Add another record (Y/N)?Y
Enter name, age and basic salary:  Harriet  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,sizeof9e),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;
}
```

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 same.

Example: 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 Exercises

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)
                        putch(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;
        }
```

Model Examination Papers

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.

```
#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);
}
```

(5 marks)

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 though 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 and1205 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 ?

(4 marks)

   (ii) Write a program that initializes the above export values in the table, computes and displays the total and average sales for *each year*. (6 marks)

References

1. Gottfried, Byron S. *Schaum's Outlines programming in C*, 2nd Ed. Tata McGraw Hill 1996.
2. Kanetkar, Yashavant. *Let us C*, 3rd Ed. New Delhi: BPB 1999.
3. Kanetkar, Yashavant. *Working with C.*1st Ed. New Delhi: BPB 1994.
4. Schild, Herbert. *Teach Yourself C* 2nd Ed. Osborne McGraw-Hill 1994.
5. Waite, Mitchel. Prata, Stephen. *C Step by* Step 1st Ed. The Waite Group 1989.