Mount Kenya University

\

BSC INFORMATION TECHNOLOGY

UNIT CODE: BIT1102

UNIT NAME: INTRODUCTION TO PROGRAMMING AND ALGORITHMS

BIT1102 : INTRODUCTION TO PROGRAMMING AND ALGORITHMS

Pre-requieste:

Purpose: To introduce learners to basics in programming and algorithms.

Objectives By the end of the course unit, a learner should:

i. To know types of programming languages

ii. To learn problem solving techniques

iii. To understand good programming practices

iv. To learn how to write algorithms and apply algorithm programming concepts

Course Content;

Introduction to programming concepts; program, errors e.g. syntax, semantics, and translators: compilers, interpreters and linkers, characteristics of programming languages.

Introduction to algorithmic problem solving;

definition of algorithm, types of algorithm: flow charts, pseudo code, Jackson structured diagram, Introduction to search and sort techniques; binary search, linear search, bubble sort Problem solving strategies; top down, bottom up decomposition Program development processes Control structures; sequence, selection, iteration Use of high level programming to demonstrate programming constructs

TABLE OF CONTENTS Page

Introduction

The basic aim of this module is to teach programming and algorithm using C language in a simpler way. As I write this module I have put in consideration the beginners and those who are new in programming world.

This module do not need prior knowledge in programming though basic computer skills will come in handy.

You will explore the world of C from the basic concepts to other facets of the language. Explanations, examples, exercises and tip notes are used. The guide uses an easy – to learn approach. You will find the guide comprehensive, concise and practical – oriented. To help grasp the concepts of the language, a number of real - life situation examples have been used.

You will find revision questions at the end of every chapter so that you can test your proficiency in what is covered.

Finally, the model examination papers at the end of the guide shall test your preparedness for the final examination.

I wish you all the best and I hope you will enjoy it!

CHAPTER ONE: PROGRAMMING BASIC IN C LANGUAGE.

Chapter Objectives

- 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.

1.1 Program

This is a set or series of instructed fed to computer system so as to perform a specific task. These instructions are like commands and they are written by a computer specialist called computer programmer.

1.2 Programmer

This is a qualified personnel or a specialist who writes computer programs, and mostly he/she does the work as a profession or an occupation.

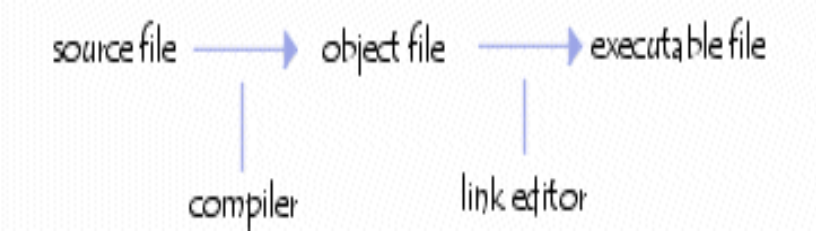
1.3 Computer programming.

Creating a sequence of instructions to enable the computer to do a specific task, The process of developing and implementing various sets of instructions to enable a computer to do a certain task.

These programs are then translated into machine code (in binary) by a compiler.

Generally, the program is a simple text file (written using a word processor or a text editor), this is called the **source file**.

The source file contains lines of program called **source code**. Once the source file has been completed it must be compiled. Compilation takes place in two stages:



- The compiler transforms the source code into object code, and saves it in an **object file**, i.e. it translates the source file into machine code (some compilers also create a file in assembler), a language similar to machine code as it possesses basic functions but is legible by humans.
- The compiler then makes a call to a **link editor** (or **linker** or **binder**) which enables it to embed all additional elements (functions or libraries) that are referenced in the program into the final file but which are not stored in the source file. Then an **executable file** is created which contains all items required for the program to run on its own (in Microsoft Windows or MS-DOS this file will have the extension *.exe)*.

1.4 Steps in Program Development

Design program objectives

In this level the all the ideas are formulated and the expected output drafted. At this stage, the programmer should think in general terms, not in terms of some specific computer language.

Design program

The programmer decides how the program will go about its implementation, what should the user interface be like, how should the program be organized, how to represent the data and what methods to use during processing. It may be possible to think along a certain general characteristics of the C language.

Write the Code

It is the design Implementation by writing the (C) code i.e. translating your program design into C language instructions. In this stage the programmer uses C's text editor or any other editor such as notepad to write the source code and the source file should have **.c** extension.

Compile the code

A compiler converts the source code into object code. Object code is instructions in machine language. Computers have different machine languages. C compilers also incorporate code for C libraries into the final program that 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.

Run the program

This is the actual execution of the final code, usually preceded by linking. Once the executable code is complete and working.

Test the program

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

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.

1.5 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 library routines and 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.
  - ✓ 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.

1.6 C Programs' Components

1.6.1Keywords

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

Examples of keywords

Auto	**Break**	**case**	**else**
Int	**Void**	**default**	**Do**
Double	**If**	**sizeof**	**Long**
Float	**For**	**goto**	**signed**
Unsigned	**Register**	**return**	**Short**
Union	**Continue**	**struct**	**switch**
Typedef	**const**	**extern**	**volatile**
While	**Char**	**enum**	**Static**

A typical C program is made of the following components:

- Preprocessor directives
- Functions

- Declaration statements
- Comments
- Expressions
- Input and output statements

Example of a program code.

This program will print out the message: C program is simple.

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

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. It is always found 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 mostly the body 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("C program is a simple language \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. C is case sensitive, that is, it recognizes a lower case letter and it's upper case equivalent as being different.

Example of basic program.

```
#include<stdio.h>
main()
{
        printf("C program  is a simple language");
        printf(" you will find it enjoyable. \n");
        printf ("This is another line to be displayed on the screen ");
        return 0;
}
```

The output of the program.

```
C program  is a simple language you will find it enjoyable
This is another line to be displayed on the screen.
```

1.6.2 Declaration of statement

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.

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.

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 the program is compiled, the code for library functions is automatically added to your program.

One of the most common library functions is called **printf()** which is found in the **"stdio.h"** header file in the c library. This is C's general purpose output function. Its simplest form is

printf("string – to – output");

The printf() prints out any character that are contained between the beginning and ending double quotes.

For example, **printf("C program is a simple language ");**

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( ).

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.

Then, the use of **return (0);** all the that have a return type must have a return statement, otherwise it should not contain it.

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.

1.6.3 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 programThere two types of comments i.e. single-line and multiple line comments.

Single –line: are basically for one line and we use // followed by the comment.

While multiple or block comments uses /*comments*/ and 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.

1.6.4 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;

1.6.5 Escape sequences

Escape sequences are character combinations that begin with a backslash symbol (\) used to format output.

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
```

This program shall produce a bell sound.

```
#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

1.6.6 Errors in Programming

Errors are codes or mistakes that make the program malfunction.

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.

Syntax errors include;

- Missing semi colon at the end of a statement e.g. area = radius*radius*3.14
- Use of an undeclared variable in an expression.
- Illegal declaration.
- Use of a keyword in uppercase e.g. FLOAT, WHILE
- Misspelling keywords e.g. init instead of int

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.

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.

1.6.7 Programming languages.

A **programming language** is a language designed to describe a set of consecutive actions to be executed by a computer. A programming language is therefore a practical way for us (humans) to give instructions to a computer.

On the other hand, **natural language** defines a means of communication shared by a group of individuals (for example: English or French)

Languages that computers use to communicate with each other, have nothing to do with programming languages, they are referred to as communication protocols, these are two very different concepts.

EACH instruction corresponds to ONE
processor action.

The language used by the processor is called **machine code**. The code that reaches the processor consists of a series of 0s and 1s known as (binary data).

Machine code is therefore difficult for humans to understand, that's why intermediary languages, which can be understood by humans, have been developed. The code

written in this type of language is transformed into machine code so that the processor can process it.

The assembler was the first programming language ever used. This is very similar to machine code but can be understood by developers. Nonetheless, such a language is so similar to machine code that it strictly depends on the type of processor used (each processor type may have its own machine code). Thus a program developed for one machine may not be *ported* to another type of machine.

A **programming language** is a code that can make programs which is use to make software. **Programming languages** have some built in function which is based on the protocols.

1.6.7.1 Classification of Programming Languages

Computer programming language can be classified into two major categories.

a) **Low Level**

b) **High Level**

Low Level Languages

The languages which use only primitive operations of the computer are known as low language. In these languages, programs are written by means of the memory and registers available on the computer. As to note, the architecture of computer differs from

one machine to another, so far each type of computer there is a separate low level programming language.

Programs written in one low level language of one, architectural can't be ported on any other machine dependent languages. Examples are Machine Language and Assembly Language.

Machine Language

In machine language program, the computation is based on binary numbers. All the instructions including operations, registers, data and memory locations are given in there binary equivalent.

The machine directly understands this language by virtue of its circuitry design so these programs are directly executable on the computer without any translations. This makes the program execution very fast. Machine languages are also known as **first generation languages.**

A typical low level instruction consists essentially of two parts:

•An Operation Part :

Specifies operation to be performed by the computer, also known as Opcode.

•An Address Part :

Specifies location of the data on which operation is to be performed.

Advantages

a) Efficient use of computer system resources like storage, registers, etc. the instruction of a machine language program are directly executable so there is no need of translators.
b) Can be used to manipulate the individual bits in a computer system with high execution speed due to direct manipulation of memory and registers.

Drawbacks

a) Machine languages are machine dependent and, therefore, programs are not portable from one computer to other.
b) Programming in machine language usually results in poor programmer productivity.
c) Programmers need to control the use of each register in the computer's Arithmetic Logic Unit and computer storage locations must be addressed directly, not symbolically.
d) Requires a high level of programming skill which increases programmer training costs.
e) Programs written in machine language are more error prone and difficult to debug because it is very difficult to remember all binary equivalent of register, opcode, memory location, etc.

f) Program size is comparatively very big due to non-use of reusable codes and use of very basic operations to do a complex computation.

Machine Language: is expressed in binary using only 0 and 1.

Assembly Language

Assembly language are also known as **second generation languages**. These languages substitutes alphabetic or numeric symbols for the binary codes of machine language. There is the use of mnemonics for all opcodes, registers and for the memory locations which provide us with a facility to write reusable code in the form of macros. These language require a translator known as “Assembler” for translating the program code written in assembly language to machine language.

Advantages

a) Provide optimal use of computer resources like registers and memory because of direct use of these resources within the programs.
b) Assembly language is easier to use than machine language because there is no need to remember or calculate the binary equivalents for opcode and registers.
c) An assembler is useful for detecting programming errors.
d) Assembly language encourages modular programming which provides the facility of reusable code, using macro.

Drawbacks

a) Assembly language programs are not directly executable due to the need of translation.
b) Languages are machine dependent and, therefore, not portable from one machine to another.
c) Requires a high level of programming skills and knowledge of computer architecture of the particular machine.

High Level Languages (HLL)

All high level language are procedure-oriented language and are intended to be machine independent. Programs are written in statements akin(like) to English language, a great advantage over mnemonics of assembly languages require languages use mnemonics of assembly language.

That is, the high level languages use natural language like structures.

The early high level language come in **third generation** of languages, COBOL, BASIC, APL, etc.

These languages enable the programmer to write instruction using English words and familiar mathematical symbols which makes it easier than technical details of the computer. It makes the programs more readable too.

Procedures are the reusable code which can be called at any point of the program. Each procedure is defined by a name and set of instructions accomplishing a particular task. The procedure can be called by its name with the list of required parameters which should pass to tat procedure.

Advantages of High Level Languages

a) These are procedure-oriented languages and are machine independent.

b) These are easier to learn than assembly language.

c) Less time is required to write programs.

d) These provides better documentation.

e) These are easier to maintain.

f) These have an extensive vocabulary.

Limitation of HLL Programming language

a) Long sequence statements is to be written for every program.

b) Additional memory space is required for storing compiler or interpreter.

c) Execution time is very high as the HLL programs are not directly executable.

A programming language has therefore several advantages:

- it is much more understandable than machine code;
- it allows greater portability, i.e. can be easily adapted to run on different types of computers.

Imperative and functional programming languages

Programming languages are generally divided into two major groups according to how their commands are processed:

- imperative languages;
- Functional languages.

Imperative programming language

An imperative language programs uses a series of commands, grouped into blocks and comprising of conditional statements which allow the program to return to a block of commands if the condition is met. These were the first programming languages in use, even today many modern languages still use the principle.

Structured imperative languages suffer, however, from lack of flexibility due to the sequentiality of instructions.

Functional programming language

A **functional programming language** (often called *procedural language*) is a language which creates programs using functions, returning to a new output state and receiving an input result of other functions. When a function invokes or call itself, it is referred recursion.

Interpretation and compilation

Programming languages may be roughly divided into two categories:

- interpreted languages
- compiled languages

Interpreted language

A programming language is by definition different to machine code,This must therefore be translated so that the processor can understand the code. A program written in an interpreted language requires an extra program (the interpreter) which translates the programs commands as needed.

Compiled language

A program written in a **compiled** language is translated by an additional program called a **compiler** which in turn creates a new stand-alone file which does not require any other program to execute itself, such a file is called an **executable**.

A program written in a compiled language has the advantage of not requiring an additional program to run it once it has been compiled. Furthermore, as the translation only needs to be done once, at compilation it executes much faster. However, it is not as flexible as a program written in an interpreted language, as each

modification of the source file (the file understandable by humans: the file to be compiled) means that the program must be recompiled for the changes to take effect.

On the other hand, a compiled program has the advantage of guaranteeing the security of the source code. In effect, interpreted language, being a directly legible language, means that anyone can find out the secrets of a program and thus copy or even modified the program.

Intermediary languages

Some languages belong to both categories (LISP, Java, Python...) as the program written in these languages may in certain cases undergo an intermediary compilation phase, into a file written in a language different to the source file and non-executable (requiring an interpreter). Java applets, small programs, often loaded in web pages, are compiled files, which can only be executed from within a web browser (these are files with the .class extension).

The following are the characteristics of a programming language

a) *Readability:* A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. *Portability:* High-level languages, being essentially machine independent, should be able to develop portable software.

b) *Generality:* Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

c) *Error checking:* Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

d) *Familiar notation:* A language should have familiar notation, so it can be understood by most of the programmers.

e) *Efficiency:* It should permit the generation of efficient object code.

f) *Modularity:* It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

1.6.7.2 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

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?

8. Compare and contrast compile and interpreted languages.

9. Name and explain four characteristics of a programming language.

CHAPTER TWO: DATA STORAGE AND MANIPULATION.

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
- The use of **printf()** and **scanf()** functions.

2.1 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 sum=10;          /*assign the value 10 to sum. */
```

Variable Names

Every variable has a name and a value. The name identifies the variable and the value stores data. There are rules or conventions that needs to followed when writing a varaible.

These include:

- Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters.
- C language is case sensitive hence lower and upper case letters are treated differently i.e int sum ,int Sum ,int SUM are all different to the compiler,
- You cannot use any of C's keywords like main, while, switch etc as variable names.
- White spaces should not exist or be contained in the words or names that makeup the variable. Instead you should use underscore to join the words of just join the word.
- The variable name should not contain special characters like %,$,&,* ete, except the underscore(_)

Examples of legal variable names

t ,num3 ,tyyah1,sum,name,solution,match_score

2.2 Basic Data Types

C supports five basic data types.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

int

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

int num;

char

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

```
char name;
```

flloat

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 1632.90. To declare mynum be of type float, you would write:

```
float mynum;
```

Floating point variables typically occupy 4 bytes.

double

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 **s** to be of type double you would write:

```
double s;
```

Double-type variables typically occupy 8 bytes.

Using printf( ) To Output Values

Use printf ( ) to display values of characters, integers and floating - point values. For example:

```
printf("The number is %d ", 234);
```

displays **The number is 234** on the screen.,This call to the printf( ) function contains two arguments. The first one is the quoted string and the other is the constant 234. Note that the arguments are separated by a comma.

In general, when there is more than one argument to a function, the arguments are separated from each other by commas.

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, which can change depending on the datatype under use.

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.

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

This program shown below explain more about the concept. First, it declares a variable called **num**. Second, it assigns this variable the value 56. Finally, it uses **printf( )** to display **the value is 56** on the screen. Examine it closely.

```
#include<stdio.h>
main()
{
  int num;
  num = 56;
  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 name;
    float gyt;
```

```
        double und;
        name = 'student';
        gyt= 89.63;
        d  = 567.123;
        printf(" name is %c ", name);
        printf(" gyt is %f ", gyt);
        printf(" und  is %f ", und);
        return 0;
    }
```

Revision questions

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

2. Write a program that declares one integer variable called **num**. Give this variable the 908 and then, using one **printf ( )** statement, display the value on the screen like this:

```
908 is the value of num
```

Accepting 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, Example.

```
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 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 displays the value.

```
#include<stdio.h>
main()
{
  int num;
  printf(" \nEnter an integer: ");
  scanf( "%d ", &num);
  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.

2.3 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.

2.4 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.

Floating - point constants require the use of the decimal point followed by the number's fractional component. For example the body temperature can be set to be 37.1 and also for the constant PI can be set to be 3.1425

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 672.89 using scientific notation.

672.89E1

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

letter= ‘u’;

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;

2.4.1 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 Temp 37.1**

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 **Temp** are replaced with the replacement text 37.1.

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.

2.5 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 explains more.

```
#include<stdio.h>
main()
{
        int num;
        float decnum;
```

```
        decnum= 1234.0098;

        num = decnum;

        printf(" %f  %d ",decnum, num);

        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 3.0×10^{-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 in C Language

Chapter Objectives

- Describe Operators and operands.
- Use operators in a program
- Types of operators.
- Understand Operator Precedence.

3.1 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.

3.1.1 Types of Operators

a) 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 the answer is 2.

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.

Examples

Suppose a =13, b = 4, valone = 9.5, valtwo = 5.0, letter =’n’.

Compute the result of the following expressions.

a + b valone * valtwo2

a - b valone / valtwo

a * b letter

a / b letterone + lettertwo +5

a % b letterone + lettertwo +'9'

Note:

(i) letterone and lettertwo are character constants

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.

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).

3.2 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)

Summary of Operator precedence.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication Division Modulus	Evaluated second. If there are several, they re evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Example: Use of operators and their precedence

```
/* Program to demonstrate use of operators and their precedence */

include<stdio.h >

main()

{
```

```
    int score,max;
    score = 90;
    max = score - (9*2) + 3 * (8+6) + (4+2);
    printf ("The maximum value is = %d \ n" , max);
    return 0;
}
```

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

Exercise

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

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

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

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

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

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

```
#include<stdio.h >
#define Sec_in_min 60
```

```
main()
{
  int seco, minute, seco_rem;
  printf(“ Converting  seconds to minute and seconds \n “) ;
  printf( “Enter number of seconds you wish to convert \n “) ;
  scanf(“% d” , & seco ) ;  /* Read in number of seconds */
  minute = seco / Sec_in_min;        / * Truncate number of seconds */
  sec_rem = seco % Sec_in_min;
  printf(“% d seconds is % d minutes,% seconds\n “ ,seco,minutes,sec_rem);
  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.

An example of ‘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: Introduction To Algorithmic Problem solving

Chapter Objectives

- Learn about problem solving skills
- Explore the algorithmic approach for problem solving
- Learn about algorithm development
- Become aware of problem solving process
- The relationship between data and algorithm.
- The characteristics of an algorithm.
- Using pseudo-codes and flowcharts to represent algorithms.

After understanding and analyzing a problem, one come up with a solution—an algorithm. That is a step-by-step procedure for solving a problem in a finite amount of time with a finite amount of data.

In the problem-solving phase of computer programming, one must design algorithms, This means you must be conscious of the strategies you use to solve problems in order to apply them to programming problems.

Note:

Programming is a process of problem solving

4.1 Problem solving techniques

- Analyze the problem
- Outline the problem requirements
- Design steps (algorithm) to solve the problem

Problem Solving Process

Step 1 - Analyze the problem

- Outline the problem and its requirements
- Design steps (algorithm) to solve the problem

Step 2 - Implement the algorithm

- Implement the algorithm in code
- Verify that the algorithm works

Step 3 - Maintenance

- Use and modify the program if the problem domain changes

Analyze the Problem

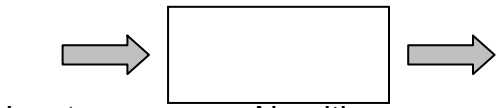
- Thoroughly understand the problem and problem requirements ,one would ask the following questions:
  - Does program require user interaction?
  - Does program manipulate data?
  - What is the output?
- If the problem is complex, divide it into subproblems
  - Analyze each subproblem as above.

4.2 What is an algorithm?

In computing, the focus is on the type of problems categorically known as **algorithmic problems**, where their solutions are expressible in the form

of algorithms.

An ***algorithm*** is a well-defined computational procedure consisting of *a* set of *instructions*, that takes some value or set of values, as *input*, and produces some value or set of values, as *output*. In other word, an algorithm is a procedure that accepts data, manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s).



People of different professions have their own form of procedure in their line of work, and they call it different names. A cook, for instance, follows a procedure commonly known as a recipe that converts the ingredients (input) into some culinary dish (output), after a certain number of steps.

An algorithm is a form that embeds the complete logic of the solution. Its formal written version is called a *program*, or *code*. Thus, algorithmic problem solving actually comes in two phases: **derivation of an algorithm that solves the problem**, and **conversion of the algorithm into code.** The

latter, usually known as coding, is comparatively easier; since the logic is already present hence it is easier so long as the syntax rules of the programming language are adhered to.

The idea behind the computer program

- **Stays the same independent of**
    - Which kind of hardware it is running on
    - Which programming language it is written in
- **Solves a well-specified problem in a general way**
- **Is specified by**
    - Describing the set of instances (input) it must work on
    - Describing the desired properties of the output

Before a computer can perform a task, it must have an algorithm that tells it what to do.

Informally: “An algorithm is a set of steps that define how a task is performed.”

Formally: “An algorithm is an ordered set of unambiguous executable steps, defining a terminating process.” It must have:

- Ordered set of steps: **structure.**
- Executable steps: **doable**.
- Unambiguous steps: **follow the directions**.
- Terminating: **must have an end**!

An example of an algorithm

The modern Euclidean algorithm to compute gcd (greatest common divisor) of two integers, is often presented as followed.

1. Let A and B be integers with $A > B \geq 0$.
2. If $B = 0$, then the gcd is A and the algorithm ends.
3. Otherwise, find q and r such that $A = qB + r$ where $0 \leq r < B$

   Note that we have $0 \leq r < B < A$ and $\gcd(A,B) = \gcd(B,r)$.

   Replace A by B, and B by r. Go to step 2.

4.2.1 Characteristics Of An Algorithm

There are four essential properties of an algorithm.

1. Each step of an algorithm must be **exact**.

An algorithm must be precisely and unambiguously described,

2. An algorithm must **terminate**.

The ultimate purpose of an algorithm is to solve a problem. If the program does not stop when executed, we will not be able to get any result from it.

3. A n algorithm must be **effective**.

Again, this goes without saying. An algorithm must provide the correct answer to the problem.

4. An algorithm must be **general**.

This means that it must solve every instance of the problem. For

example, a program that computes the area of a rectangle should work on all possible dimensions of the rectangle, within the limits of the programming language and the machine.

Representation of Algorithms

A single algorithm can be represented in many ways:

- Formulas: F = (9/5)C + 32
- Words: Multiply the Celsius by 9/5 and add 32.
- Flow Charts.
- Pseudo-code.

In each case, the algorithm stays the same; the implementation differs.

A program is a representation of an algorithm designed for computer applications.

Process: Activity of executing a program, or execute the algorithm represented by the program

Problem Solving: A creative process

- Problem solving techniques are not unique to Computer Science. The Computer Science field has joined with other fields to try to solve problems better.
- Ideally, there should be an algorithm to find/develop algorithms. However, this is not the case as some problems do not have algorithmic solutions.

4.3 Problem Solving Strategies

- **Working backwards**

- Reverse-engineer.
- Once you know it can be done, it is much easier to do
- What are some examples?

- **Look for a related problem that has been solved before**
  - Java design patterns
  - Sort a particular list such as: David, Alice, Carol and Bob to find a general sorting algorithm
- **Stepwise Refinement**
  - Break the problem into several sub-problems
  - Solve each subproblem separately
  - Produces a modular structure

4.3.1 Stepwise Refinement

Stepwise refinement is a top-down methodology in that it progresses from the general to the specific.

4.4 Top-Down Methodology.

In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate.

Top-down programming involves writing code that calls functions you haven't defined and working through the general algorithm before writing the functions that do the processing.

Advantage of Top-down

Top-down programming is, to a good degree, a very abstract way of writing code because it starts out by using functions you haven't designed, and that you perhaps do not know how to design.

Bottom-up methodologies progress from the specific to the general.

It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written.

The programmer writes the basic functions he/she realizes will be necessary at some point in the programming and then work up to the more complex parts of the program.

These approaches complement each other.

Solutions produced by stepwise refinement posses a natural modular structure - hence its popularity in algorithmic design.

Advantages of bottom-up programming

a) Testing is simplified since no stubs are needed.

b) Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down.

4.5 Pseudocode

Pseudocode is like a programming language but its rules are less stringent. It can also be said to be a written as a combination of English and programming constructs

Based on selection (if, switch) and iteration (while, repeat) constructs in high-level programming languages.

Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is a "text-based" detail (algorithmic) design tool.

The rules of Pseudocode are reasonably straightforward.

Design using these high level primitives and it is Independent of actual programming language.

***pseudo-code*,** which is normally a mixture of English statements, some mathematical n otations, and selected keywords from a programming language. There is no standard convention for writing pseudo-code; each author may have his own style, as long as clarity is ensured.

The problem concerned is to find the minimum, maximum, and average of a list of numbers. Make a comparison.

Solution

sum ß *count* ß*0* { *sum* = sum of numbers;

count = how many numbers are entered? }

min ß? { *min* to hold the smallest value eventually }

max ß? { *max* to hold the largest value eventually }

for each *num* entered,

increment *count*

sum ß *sum* + *num*

if *num* < *min* then *min* ß *num*

if *num* > *max* then *max* ß *num*

ave ß *sum*/*count*

Example 2.

This pseudo code is suppose to help capture the marks for the student and print relevant messages on the screen that's if the grade is equal to 60 or more then it should print passed other it should print failed.

If student's grade is greater than or equal to 60

Print "passed"

else

Print "failed"

Example 3.This is an example that helps to capture grade of ten student and then get the average of the class.

Set total to zero

Set grade counter to one

While grade counter is less than or equal to ten

Input the next grade

Add the grade into the total

Set the class average to the total divided by ten

Print the class average.

Exercise : explain what this pseudocode is a suppose to achieve.

Initialize total to zero

Initialize counter to zero

Input the first grade

while the user has not as yet entered the sentinel

```
        add this grade into the running total
        add one to the grade counter
        input the next grade (possibly the sentinel)
if the counter is not equal to zero
        set the average to the total divided by the counter
        print the average
else
        print 'no grades were entered' .
```

Exercise : explain what this pseudocode is a suppose to achieve.

```
initialize passes to zero
initialize failures to zero
initialize student to one
while student counter is less than or equal to ten
        input the next exam result
        if the student passed
```

add one to passes

else

add one to failures

add one to student counter

print the number of passes

print the number of failures

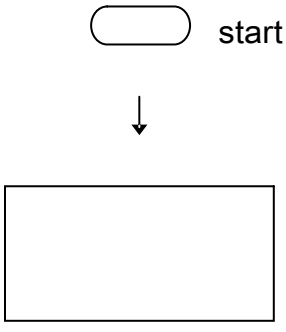
if eight or more students passed

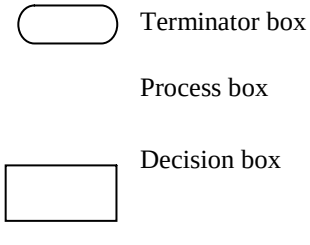
print "raise tuition

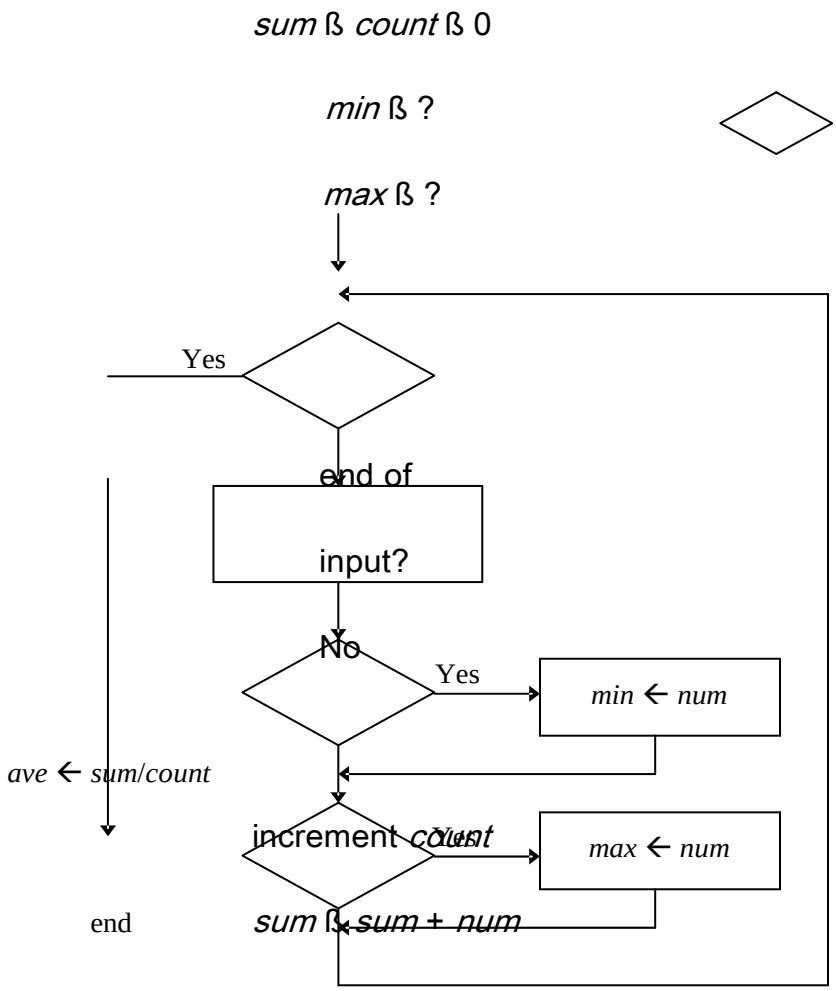
4.6 Flowchart

Algorithms may also be represented by diagrams. One popular diagrammatic method is the *flowchart*, which consists of terminator boxes, process boxes, and decision boxes, with flows of logic indicated by arrows.

The flowchart below depicts the same logic as the algorithms above in example 1.



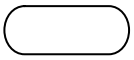




num<min?

No

num>max?



No

Whether you use the pseudo-code or the flowchart to represent your algorithm, remember to walk through it with some sets of data to check that the algorithm works.

Advantages and Disadvantages of flowchart and pseudo codes.

Pseudocode Advantages

i) Can be done easily on a word processor

ii) Easily modified

iii) Implements structured concepts well.

Pseudocode Disadvantages

i) It's not visual

ii) There is no accepted standard, so it varies widely from company to company

Flowchart Advantages

i) Standardized: all pretty much agree on the symbols and their meaning

ii) Visual (but this does bring some limitations)

Flowchart Disadvantages

i) Hard to modify

ii) Need special software (not so much now!)

Structured design elements not all implemented

4.7 Jackson Structured Programming (JSP)/Jackson System Development (JSD)

Takes the view of "paralleling the structure of input data and output (report) data will ensure a quality design"

More recent extensions to the methodology is the Jackson System Development (JSD)

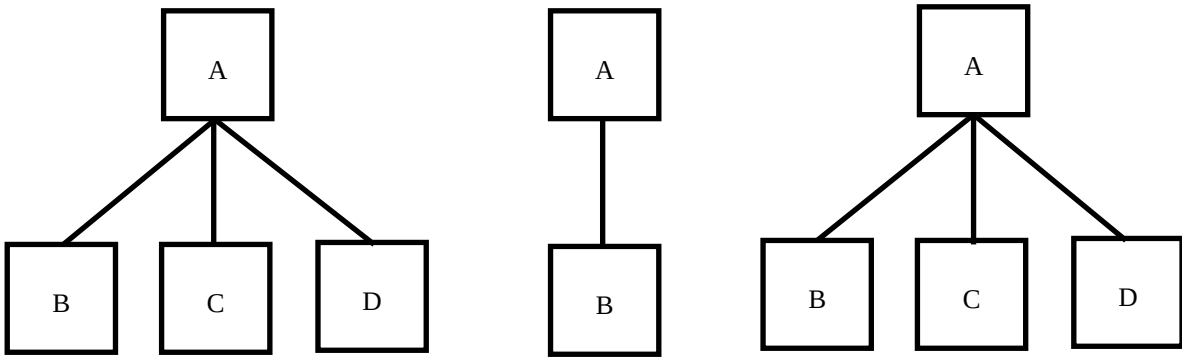
JSD focuses on the identification of information entities and the actions that applied to them,JSD emphasizes on developing techniques to transform data program structure

4.7.1 Jackson Structured Programming

Data driven program design method.

Produce data structure diagrams for input and output data streams.

Device independent and model as tree diagram using sequence, selection and iteration.



Sequence **Iteration** **Selection**

A is a sequence of B followed by C followed by D
A consists of zero or more occurences of B
A is a selection of either B or C or D

Characteristics of JSP

a) It is non-inspirational. This means that it depends little or not at all, on invention or insight on the part of the engineer.

b) It is rational, i.e. the design procedure is based on reasoned principles, and each step can be proven in the light of these principles.

c) It is teachable. People can be taught to practice the method and two or more programmers using the method to solve the same problem will arrive at substantially the same solution.

d) It is practical. The method itself is simple and easy to understand, and the designs produced can be implemented without difficulty in any ordinary programming environment.

Advantages of JSP

a) Enable correct programs to be produced.

b) Provide a method that is "workable" within the intellectual limitations of the average programmer.

c) Techniques that can be taught and do not rely on inspiration or perspiration

d) Facilitate the organized control of software projects.

Steps in JSP

i. Draw structure diagrams for each set of data such that the structure reflects the way in which the data is to be processed.

ii. Identify points of correspondence of a one-for-one nature between components of individual data structures.

iii. Produce a program structure diagram using the same notation as that used in data structures, and based on the data structures, combine them at the points of correspondence.

iv. For each iteration and selection appearing in the program structure diagram, construct appropriate conditions.

v. Examine the specification and the conditions and from these draw up a list of basic program operations in plain language.

vi. Allocate the conditions and operations to the appropriate components of the program structure.

vii. Produce schematic logic, also known as pseudo code from the program structure.

viii. Implement the pseudo-code in a target high-level programming language.

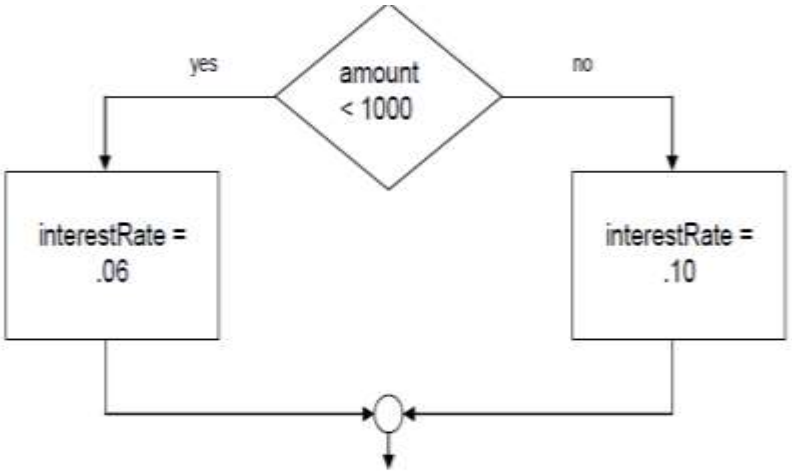
Examples for both flowchart and pseudo code.

Example 1

Consider certain amount and then award interest rate based on the whether the amount is less or more than 1,000.if the amount is less than 1000 award 6% otherwise award 10% interest rate.

Flowchart solution.

The use of selection structure.



The **pseudocode** for this would be:

IF amount < 1000

interestRate = .06 // the "yes" or "true" action

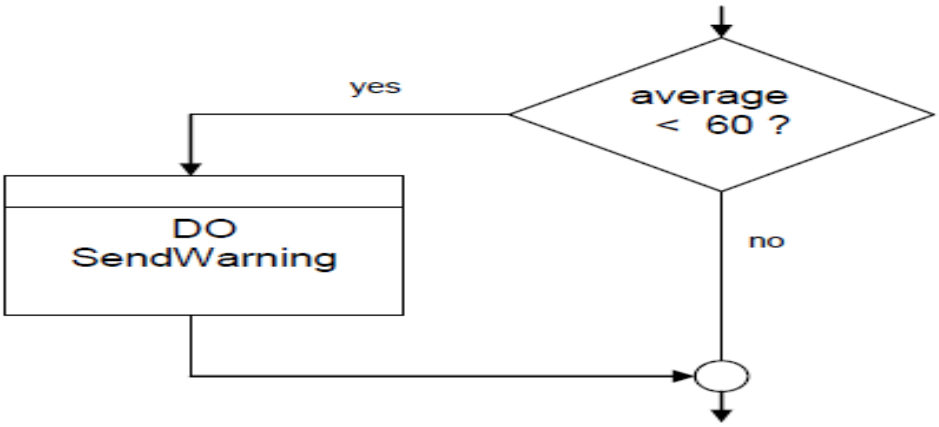
ELSE

interestRate = .10 // the "no" or "false" action

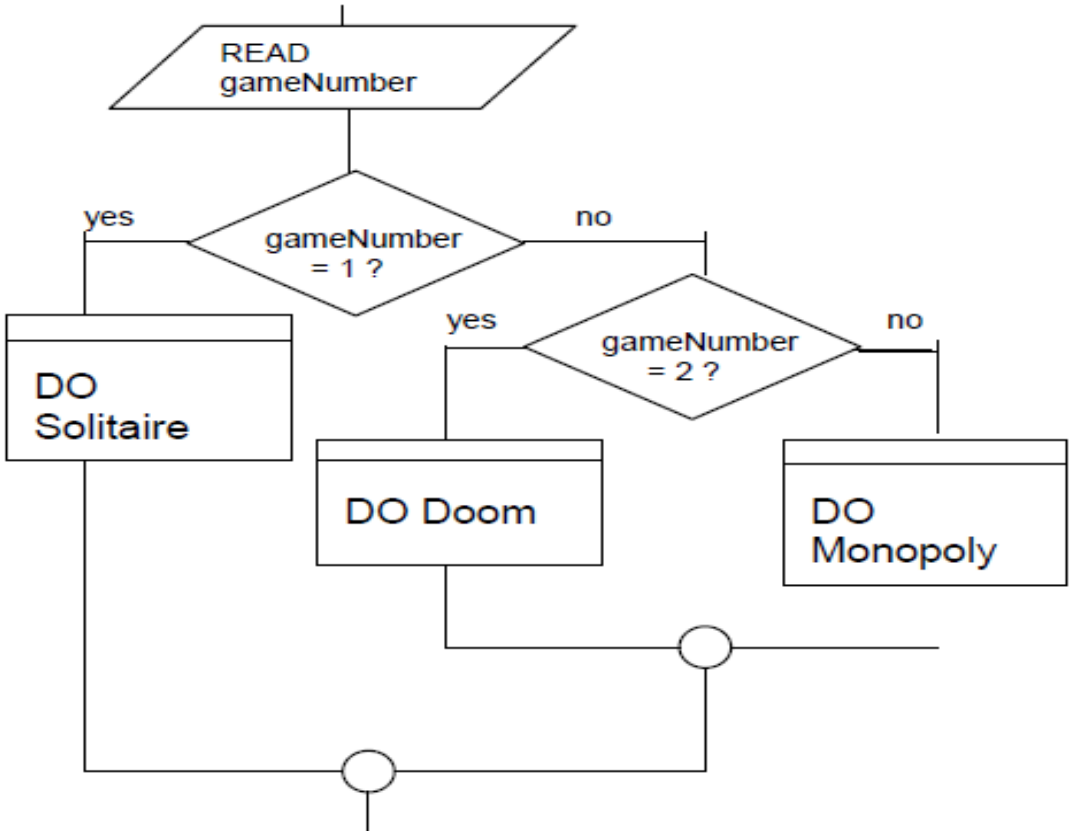
ENDIF

Revision questions

a) **Provide the pseudo code equivalent of the following flowchart.**



b) Study the following flowchart and then provide the pseudocode equivalent



c) Example the difference between flowchart and pseudo code.

d) List the advantages of the JSP.

e) Explain the following pseudo code and provide a flowchart version.

set average to zero

set count to zero

set total to zero

do

read a number

increment count by 1

total = total + number

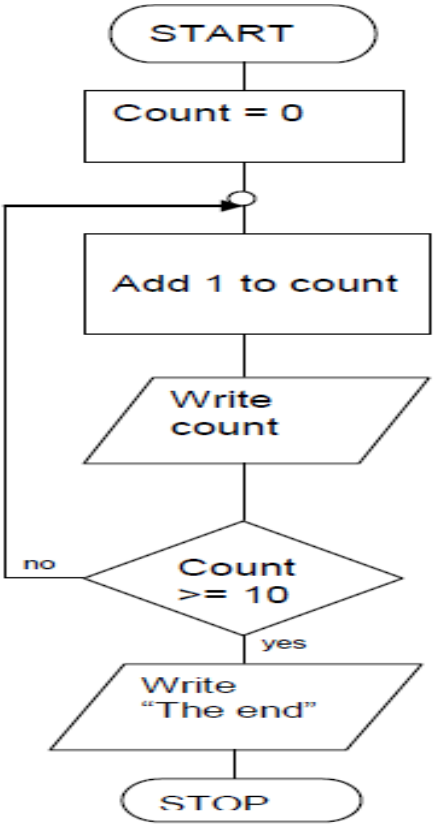
while (not end-of-data)

if (count > 0) then

average = total / count

display average

f) Study this and write a pseudo code for it.



CHAPTER FIVE: CONTROL STRUCTURES

Chapter objective.

- Understand control structures.
- Describe different Types of control structures.
- Be able to utilize the control structures in a program.

5.1 Introduction

Control structures these are programming constructs or elements that are used to alter the follow of the program execution.

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.

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)
```

5.2 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 (age>18)

```
Printf("You are and adult")
```

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(age>18)
{
  printf("You are an adult");
        printf("You should have the national Identity Card")
}
```

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 mks;
char grd [10];
printf (“ Enter the students marks  \n”);
scanf( “%d “,&mks ) ;
if ( mks > =75  &&  mks <=100)
{
   strcpy(grd, “Distinction”);          /* Copy the string to the grade */
   printf(“The grade is %s” , grd);
}
else if( mks > = 60 &&  mks < 75 )
{
        strcpy(grd, “Credit”);
   printf(“The grade is %  s” , grd );
}
else if(mks>=50  &&  mks <60)
{
  strcpy( mks, “Pass”);
  printf(“The grade is %  s” , grd );
}
else if (mks >=0 && mks <50)
```

```
 {
        strcpy(grd, “Fail”);
    printf (“The grade is %  s” , grd) ;
  }
  else
    printf(“The mark is in valid!” );
   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.

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  */
```

5.2 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 numbers 1 through 9 */
main()
```

```
{
   int num=0;                /* Initialisation */
        while (num<10)
        {
          printf("%d \n", digit);
        num++;
        }
        return 0;
}
```

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

Solution

```
/* To add numbers and compute the average */
#include<stdio.h>
main()
{
        int n, count = 1;
        float fnum, average, sum=0.0;
        /* initialise and read in a value of n */
        printf("How many numbers do you want to work with? ");
        scanf("%d", &n);
```

```
        /*Read in the number */
        while (count<=n)
        {
          printf("fnum = ");
          scanf("%f", &fnum);
          sum+=fnum;
          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).

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 num=0;      /* Initialisation */
        do
        {
          printf("%d \n", num);
          num++;
```

```
    } while (num<10);
    return 0;
}
```

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

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 (initial).

expression2 is a test expression,or just the condition.

expression3 is used to alter the value of the initial parameter either increasing or decreasing.

Example

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

Output

0 1 2 3 4

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.

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. Using a nested if statement, write a program that prompts the user for a number and then reports if the number is positive, zero or negative.

4. Write a *while* loop that will calculate the sum of every fourth integer, beginning with the integer 3 (that is calculate the sum 3 + 7 +11 + 15 + ...) for all integers that are less than 30.

Chapter Six: Introduction to Array Programming.

Chapter Objectives

- Understand array concepts
- How to use arrays in a program
- Describe and use array types in C language.

6.1 Introduction

It is often necessary to store data items that have common characteristics in a single 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 data structure that store values of the same type under one unit name.

An example of array of 10 students.

An array of 10 student ages (stored as integers)

22	19	20	21	21	22	23	10	19	20

Declaring Arrays

An array declaration comprises;

(i) Storage class (optional)

(ii) Data type

(iii) Array name

(iv) Arraysize expression (usually a positive integer). This is enclosed in square brackets.

Syntax.

Data type array_name[size];

Examples:

(i) int num[10];

int is the data type of the array (type of elements held), **num** is the array name

10 is the maximum number of elements (array size)

(ii) static char names[30]; A 30 character-type array called **names**. The individual array values persist within function calls (static).

Array Dimensions or types of an array

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. but mostly 2D will be used in accordance to the scope of this course.

6.2 Two – dimensional array

Two-dimensional array can be thought of as a table of values or matrix 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 marks for 5 students in 5 units.

	BIT1102	BST 1102	BST 1103	BST 1104	BST 1105
Laura	33	41	57	83	32
Lilian	64	50	89	98	60
Rawlynes	56	73	86	69	74
Houghton	72	53	74	63	75
John	45	60	78	70	86

6.3 Initialising Arrays

How to initial an array with values at declaration.

```
main()
{
        int marks[5] = {80, 50, 20, 90, 70];
        ---------
        ---------
}
```

Because the array is defined inside main, its an automatic array. The first element marks[0] is assigned the value of 80 , marks[1] as 50 and so on. 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.

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] = “Laura”;
```

To work with multiple dimensional arrays is the same way as shown.

For example, here the array **multval** is initialised with the values 1 though 6, using row order.

```
int  multval[2][3] = {
                        1,  2,  3,
                        4,  5,  6,

                    };
```

This initialisation causes multval **[0][0]** to have the value 1, multval **[0][1]** to contain 2, multval **[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, 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 intnum[] = {6,9,5,8,16,32,64,128};** causes the compiler to create an initialised array eight elements long.

Example:An array to print the numbers on the screen

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

```
int intnum[] = {6,9,5,8,16,32,64,128};


        int i;

        for (i=0; i<8;i++ )

        printf( “the numbers are  %d  \n ”,intnum[i]);

        return 0;

}
```

Here is the output

the numbers are 6,9,5,8,16,32,64,128

Note:

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

6.4 Processing an array

Example calculating the a average of n numbers

```
#include<stdio.h>

main()

{

        int n,i;

        float avg, d, sum =0.0;

        float num[150];

          /* Read in a value of n */
```

```
    printf(" \n How many numbers will be averaged ? ");
  scanf(" %d ", &n);
    /* Read in the numbers */
  for (i = 0; i < n; i++)
  {
    printf(" i  = %d x = ", count+1);
    scanf(" %f ", &list[i]);
    sum+=list[i];
  }
  /* Calculate the average */
  avg = sum/n;
  printf("\n  The  average is %5.2f  \n\n ", avg);
  /* Calculate deviations from the average */
  for (i =0; i < n; i++)
  {
    d = list[i] – avg;
    printf(" I = %d   x = %5.2f , d = %5.2f ", i+ 1, list[i], d);
  }
  return 0;
}   /* End of program */
```

Exercise

Write a program to store even numbers between 10 and 50, The program should store the numbers on an array named even,the program should also compute the average of the numbers.

Example Two-Dimensional Arrays 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;
}
```

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: FUNCTIONS

Chapter Objectives

a) Describe functions in C language.

b) Type of functions.

c) Importance of functions in a program.

d) Understand recursive functions

7.1 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.

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.

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.

7.1.1 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 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. 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.

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.*

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.

7.1.2 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 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 o

r arguments).

Example Determining the largest value among two integer numbers

The following program determines the largest of three integers quantities.

```
/*Determine the largest of the three integer quantities*/
#include<stdio.h>
int largest (int i ,int j ) /*Determine the larger of two quantities*/
{
```

```
    int m;
     m= (i > = j )? i : j;
    return(m);
}
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 = largest (a, b);
          printf ("\n \n  maximum = % d ", largest (c ,d));
          return 0;
}
```

The function **largest** 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 = largest(a, b);

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

7.1.3 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).

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 : A program to find the factorial of a number using a function.

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);
}
```

7.1.4 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).

Like the factorial of a number example uses the recursion concepts.

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)

Revision Exercises

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.

c) Differentiate between user-defined and inbuilt functions in C language.

CHAPTER EIGHT: INTRODUCTION TO SEARCH AND SORT TECHNIQUES

Chapter Objectives

a) Describe searching and sorting in C language.

b) Understand how search and sorting algorithms work.

c) Describe the types of searching and sorting.

d) Importance of using searching and sorting techniques.

8.1 Introduction to Searching and sorting

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. In searching and sorting will use mostly the array data structures so as to enhance better understanding though even linked list can also be used. A search algorithm is a method of locating a specific item of information in a larger collection of data.

8.2 Linear Search

This is a very simple algorithm,It uses a loop to sequentially step through an array, starting with the first element.It compares each element with the value being searched for and stops when that value is found or the end of the array is reached.The most

obvious algorithm is to start at the beginning and walk to the end, testing for a match at each item:

Example of a pseudo code for the same.

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Basic sequential search
  bool found = false;
  int i;
  for ( i = 0; i < size; i++ ) {
    if ( key == list[i] )
      break;
  }
  if ( i < size ) {
    found = true;
    rec = &list[i];
  }
  return found;
}
```

Example of a working program code to demonstrate the same.

```
// This program demonstrates the searchList function, which
```

```
// performs a linear search on an integer array.
#include <iostream.h>
// Function prototype
int searchList(int [], int, int);
const int arrSize = 5;
void main(void)
{       int tests[arrSize] = {87, 75, 98, 100, 82};
        int results;
        results = searchList(tests, arrSize, 100);
        if (results == -1)
        cout << "You did not earn 100 points on any test\n";
        else
        {
                cout << "You earned 100 points on test ";
                cout << (results + 1) << endl;
        }
}
// The searchList function performs a linear search on an
// integer array. The array list, which has a maximum of numElems
// elements, is searched for the number stored in value. If the
```

```
// number is found, its array subscript is returned. Otherwise,
// -1 is returned indicating the value was not in the array.
int searchList(int list[], int numElems, int value)
{
  int index = 0;      // Used as a subscript to search array
  int position = -1;  // To record position of search value
  bool found = false; // Flag to indicate if the value was found
  while (index < numElelments && !found)
  {
                        if (list[count] == value)
                        {
          found = true;
          position = index;
        }
        index++;
            }
            return position;
}
```

The Output.

You earned 100 points on test 4

This algorithm has the benefit of simplicity; it is difficult to get wrong, unlike other more sophisticated solutions.

The above code follows the conventions, they are as follows:

1. All search routines return a true/false Boolean value for success or failure.
2. The list will be either an array of integers or a linked list of integers with a key.
3. The found item will be saved in a reference to a pointer for use in client code.

The algorithm itself is simple. A familiar (0 to n-1) loop to walk over every item in the array, with a test to see if the current item in the list matches the search key.

The loop can terminate in one of two ways. If it reaches the end of the list, the loop condition fails. If the current item in the list matches the key, the loop is terminated early with a break statement. Then the algorithm tests the index variable to see if it is less that size (thus the loop was terminated early and the item was found), or not (and the item was not found).

The search technique is much more useful for small and unsorted arrays

Efficiency of the Linear Search

The advantage is its simplicity.

- It is easy to understand

- Easy to implement
- Does not require the array to be in order

The disadvantage is its inefficiency

If there are 20,000 items in the array and what you are looking for is in the 19,999th element, you need to search through the entire list and this is time consuming.

8.3 Binary Search

The binary search is much more efficient than the linear search.It requires the list to be in order i.e. the list must be sorted.

The algorithm starts searching with the middle element.

- If the item is less than the middle element, it starts over searching the first half of the list.
- If the item is greater than the middle element, the search starts over starting with the middle element in the second half of the list.
- It then continues halving the list until the item is found.

Binary search : simple pseudocode to explain the technique.

- For sorted arrays
- Compares middle element with key

- If equal, match found
- If key < middle, looks in first half of array
- If key > middle, looks in last half
- Repeat

```
// This program demonstrates the binarySearch function, which
// performs a binary search on an integer array.
#include <iostream.h>
// Function prototype
int binarySearch(int [], int, int);
const int arrSize = 20;
void main(void)
{
int tests[arrSize] = {101, 142, 147, 189, 199, 207, 222,234, 289, 296, 310, 319, 388,
394,417, 429, 447, 521, 536, 600};
int results, empID;
cout << "Enter the Employee ID you wish to search for: ";
```

```
cin >> empID;
results = binarySearch(tests, arrSize, empID);
        if (results == -1)
                cout << "That number does not exist in the array.\n";
        else
        {
                cout << "That ID is found at element " << results;
                cout << " in the array\n";
        }
}
// The binarySearch function performs a binary search on an integer array. Array,
// which has a maximum of numElems elements, is searched for the number
// stored in value. If the number is found, its array subscript is returned.
// Otherwise, -1 is returned indicating the value was not in the array.
int binarySearch(int array[], int numelems, int value)
{
```

```
int first = 0, last = numelems - 1, middle, position = -1;
   bool found = false;
         while (!found && first <= last)
         {         middle = (first + last) / 2;  // Calculate mid point
                   if (array[middle] == value)   // If value is found at mid
        {   found = true;
                      position = middle;
        }
                   else if (array[middle] > value) // If value is in lower half
           last = middle - 1;
                   else
                      first = middle + 1;              // If value is in upper half
          }
         return position;
}
```

Program Output with Example Input.

Enter the Employee ID you wish to search for: 199

That ID is found at element 4 in the array.

Efficiency of the Binary Search

- Much more efficient than the linear search.
- Works best with sorted items or values.

Exercises

1. Write a test program to verify the correct operation of the functions given.
2. Can you think of a more efficient way to perform sequential search? What about a non-sequential search?

8.4 Sorting Arrays

Introduction to Sorting Algorithms.

Sorting is very important in every computer application. Sorting refers to arranging of data elements in some given order. Many Sorting algorithms are available to sort the given set of elements. We will discuss two sorting techniques. The two techniques are:

1. Internal Sorting
2. External Sorting

Internal Sorting

Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.

We will examine the following methods of internal sorting

Sorting algorithms are used to arrange random data into some order,it is Important computing application,Virtually every organization must sort some data mostly Massive amounts must be sorted.

a) Insertion sort

b) Selection sort

c) Merge Sort

d) Radix Sort

e) Quick Sort

f) Heap Sort

g) Bubble Sort

8.4.1 Insertion Sort

In this sorting we can read the given elements from 1 to n, inserting each element into its proper position. For example, the card player arranging the cards dealt to him. The player picks up the card and inserts them into the proper position. At every step, we insert the item into its proper place.

This sorting algorithm is frequently used when n is small. The insertion sort algorithm scans A from A[l] to A[N], inserting each element A[K] into its proper position in the previously sorted

sub array A[l], A[2], . . . , A[K-1]. That is:

Pass 1. A[l] by itself is trivially sorted.

Pass 2. A[2] is inserted either before or after A[l] so that: A[l], A[2] is sorted.

Pass 3. A[3] is inserted into its proper place in A[l], A[2], that is, before A[l],between A[l] and A[2], or after A[2], so that: A[l], A[2], A[3] is sorted.

Pass 4. A[4] is inserted into its proper place in A[l], A[2], A[3] so that: A[l], A[2], A[3], A[4] is sorted.……

Pass N. A[N] is inserted into its proper place in A[l], A[2], . . . , A[N - 1] so that: A[l], A[2], . . . ,A[N] is sorted.

INSERTION (A , N)

This algorithm sorts the array A with N elements

1. Set A[0] :- [initializes the element]
2. Repeat Steps 3 to 5 for K= 2,3, ……,N
3. Set TEMP := A[K] and PTR:= K-1
4. Repeat while TEMP < A[PTR]

   (a) Set A[PTR +1]:=A[PTR] [Moves element forward]

   (b) Set PTR := PTR-1

[End of loop].

5. Set A[PTR+1] := TEMP [inserts element in proper place]

   [End of Step 2 loop]

6. Return

8.4.2 Merge Sort

Combining the two lists is called as merging. For example A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with **n = r + s** elements is called merging. After combining the two lists the elements are sorted by using the following merging algorithm. Suppose one is given two sorted decks of cards.

That is, at each step, the two front cards are compared and the smaller one is placed in the combined deck. When one of the decks is empty, all of the remaining cards in the other deck are put at the end of the combined deck. Similarly, suppose we have two lines of students sorted by increasing heights, and suppose we want to merge them into a single sorted line. The new line is formed by choosing, at each step, the shorter of the two students who are at the head of their respective lines. When one of the lines has no more students, the remaining students line up at the end of the combined line.

Merging pseudo code.

MERGING (A, R, B, S, C)

Let A and B be sorted arrays with R and S elements. This algorithm merges A and B into an array C with N = R + S elements.

1. [Initialize] Set NA : = 1 , NB := 1 AND PTR : = 1
2. [Compare] Repeat while NA <= R and NB <= S

   If A[NA] < B[NB] , then

   (a)[Assign element from A to C] set C[PTR] := A[NA]

   (b)[Update pointers] Set PTR := PTR +1 and NA := NA +1

   Else

   (a) [Assign element from B to C] Set C[PTR] := B[NB]

   (b) [Update Pointers] Set PTR := PTR +1 and NB := NB +1

   [End of loop]
3. [Assign remaining elements to C]

   If NA > R , then

   Repeat for K = 0 ,1,2,……..,S- NB

   Set C[PTR+K] := B[NB+K]

   [End of loop]

   Else

   Repeat for K = 0,1,2,……,R-NA

   Set C[PTR+K] := A[NA+K]

   [End of loop]

4. Exit

8.4.3 Bubble Sort.

An easy way to arrange data in ascending or descending order.,Several passes through the array are made.Successive pairs of elements are compared

- If increasing order (or identical), no change
- If decreasing order, elements exchanged

Then Repeat the process again.

- Example:

original: 3 4 2 6 7

pass 1: 3 2 4 6 7

pass 2: 2 3 4 6 7

Small elements "bubble" to the top

Pseudocode for a Bubble sort.

Do

Set count variable to 0

For count is set to each subscript in Array from 0 to the next-to-last subscript

If array[count] is greater than array[count+1]

```
   swap them
   set swap flag to true
   end if
End for
```

While any elements have been swapped.

Example of a program to demonstrate Bubble sort.

```
// This program uses the bubble sort algorithm to sort an
// array in ascending order.
#include <iostream.h>
// Function prototypes
void sortArray(int [], int);
void showArray(int [], int);
void main(void)
{
        int values[6] = {7, 2, 3, 8, 9, 1};
        cout << "The unsorted values are:\n";
```

```
        showArray(values, 6);

        sortArray(values, 6);

        cout << "The sorted values are:\n";

        showArray(values, 6);

}

Definition of function sortArray. This function performs an ascending

// order bubble sort on Array. elements is the number of elements in the array.

void sortArray(int array[], int elems)

{

        int swap, temp;

        do

        {

                swap = 0;

                for (int count = 0; count < (elems - 1); count++)

                {

                        if (array[count] > array[count + 1])
```

```
                {
                        temp = array[count];
                        array[count] = array[count + 1];
                        array[count + 1] = temp;
                        swap = 1;
                }
        }
    } while (swap != 0);
}
// Definition of function showArray.
// This function displays the contents of array. elems is the
// number of elements.
void showArray(int array[], int elems)
{
    for (int count = 0; count < elems; count++)
        cout << array[count] << " ";
```

```
        cout << endl;
}
```

Program Output.

The unsorted values are:

7 2 3 8 9 1

The sorted values are:

1 2 3 7 8 9

8.4.4 Selection Sort.

The bubble sort is inefficient for large arrays because items only move by one element at a time. The selection sort moves items immediately to their final position in the array so it makes fewer exchanges.

Selection Sort Pseudocode:

For Start is set to each subscript in Array from 0 through the next-to-last subscript

  Set Index variable to Start

  Set minIndex variable to Start

  Set minValue variable to array[Start]

```
   For Index is set to each subscript in Array from Start+1 through the next-to-last
subscript
      If array[Index] is less than minValue
         Set minValue to array[Index]
         Set minIndex to Index
      End if
      Increment Index
   End For
   Set array[minIndex] to array[Start]
   Set array[Start] to minValue
End For
```

Program example to demonstrate Selection Sort.

```
// This program uses the selection sort algorithm to sort an
// array in ascending order.
#include <iostream.h>
```

```
// Function prototypes

void selectionSort(int [], int);

void showArray(int [], int);

void main(void)

{

        int values[6] = {5, 7, 2, 8, 9, 1};

        cout << "The unsorted values are\n";

        showArray(values, 6);

        selectionSort(values, 6);

        cout << "The sorted values are\n";

        showArray(values, 6);

}

// Definition of function selectionSort. This function performs an

// ascending order selection sort on Array. elems is the number of

// elements in the array.

void selectionSort(int array[], int elems)
```

```
{
        int startScan, minIndex, minValue;
        for (startScan = 0; startScan < (elems - 1); startScan++)
        {       minIndex = startScan;
                minValue = array[startScan];
                for(int index = startScan + 1; index < elems; index++)
                {       if (array[index] < minValue)
                        {
                                minValue = array[index];
                                minIndex = index;
                        }
                }
                array[minIndex] = array[startScan];
        array[startScan] = minValue;
}
}
```

```
// Definition of function showArray.

// This function displays the contents of Array. elems is the

// number of elements.

void showArray(int array[], int elems)

{

        for (int count = 0; count < elems; count++)

                cout << array[count] << " ";

        cout << endl;

}
```

Program Output

```
The unsorted values are

5 7 2 8 9 1

The sorted values are

1 2 5 7 8 9
```

8.4.5 External Sort

The External sorting methods are applied only when the number of data elements to be

sorted is too large. These methods involve as much external processing as processing in the CPU. To study the external sorting, we need to study the various external devices used for storage in addition to sorting algorithms. This sorting requires auxiliary storage.

The following are the examples of external sorting

1. Sorting with Disk
2. Sorting with Tapes

Revision questions

1. What is searching and sorting?
2. Distinguish between sequential search and binary search.
3. Define binary search tree.
4. Which of the following traversal techniques list the nodes of a binary search tree in ascending order?
5. write a program to store the following values in an named Mynnumber that contains 91,89.2.34,2,8,10,5. The program should then sort the values in both ascending and descending of using the selection, insertion and quick sort.
6. Explain the difference between external and internal sorting techniques.

Further reading and other resources

1. Fundamentals of Data Structures in C by Horowitz & Sahni.
2. Fundamentals of Computer Algorithms by Horowitz & Sahni.
3. Data structures and algorithms by Alfred V.Aho, John E.Hopcroft & Jeffrey D.Ullman.

Model examination Papers

Paper One

QUESTION ONE

a. What do you understand by the term structured programming? [3marks]
b. C language is said to be both portable and efficient. Explain. [4 marks]
c. Given the meaning of the following component of a c program.
   i. preprocessor directive
   ii. declaration
   iii. functions
   iv. expression
   v. comment. [10marks]
d. (i) what is a keyword [2 marks]
   (ii) What situation will make a keyword not to be recognized during the compilation of a c programming?

[2marks]

e. Identify the errors in the following program and write a correct program:

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

[4 marks]

f. Write a program that ask the user for an integer and then tells the user if that number is even or odd. [5m marks]

QUESTION TWO

a. Outline the stages of developing a working program in C. [7 marks]
   i. Why is linking necessary in a program? [2marks]
   ii. A program may compile successfully but fail to generate desired results, what would be the problem and how do we solve it?. [2marrks]

b. Alvin 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?. [2marks]

c. Write a program that prompts the user to enter the radius of a circle and computer the area and circumference of the circle the program should display the computed value on the screen declare PI as a constant . [5marks]

d. Explain briefly the meaning of the following escape sequence characters
   i. \a
   ii. \r [2 marks]

QUESTION THREE

a. A program is required to accept two numbers A and B, if the value of A is greater Than that of B then they are added together otherwise the value of A is subtracted from B required:
   i. Write the pseudo code for you solution [4 marks]
   ii. Write the program code to accomplish this [6 marks]

b. Explain any two type of tools of algorithm that are using during software development. [2 marks]

c. Differentiate between the following pair of terms
   i. source code and object code
   ii. Variable and structure
   [4 marks]

d. Explain the following statement as used in C language.
   i. int*x;
   ii. return (y); [2marks]

e. What will be the output for the following program code. [2 marks]

```
int a,b;
    a=2;
    b=0
   a++;
```

--b

Printf(a+b);

QUESTION FOUR

a. Define the following terms as used in programming:
   i. Function definition.
   ii. Function prototype. [4 marks]
b. Differentiate between logical and syntax error, giving an example. [4 marks]
c. Using a **For** loop structure; write a program to get the sum of the first 10 numbers in the following series.
   2, 4, 8, 16…… [6 marks]

d. Describe briefly any two categories of operators that are used in C programming language. [2 marks]

e. Outline any two qualities of a good program. [2 marks]

f. Explain the term an array as used in C programming language. [2 marks]

g. List and explain the two main types of function found in C program [4 marks]

QUESTION FIVE

a. Briefly explain the two advantages of using functions in a C program. [2marks]
b. Write a program that utilizes two function RECT_AREA and RECT_PERIMETER to computer the area and perimeter of the rectangle respectively.The input and outputs are done in the main function.
   [6marks]

c. Write a program that inputs numbers and stores in an **even** and **odd** number array structure.
   Display the content of each array assume that the two arrays can stores up to100 elements each.

   The program should ask the user how many numbers he/she would want to work with. [6 marks]

d. Explain any two categories of control structures in C language. [2 marks]

Model examination Paper.

SECTION A (30 MARKS) – THE SETION IS COMPULSORY.

Question 1

a) Write a program that calculates the product of Two numbers by allowing the user to enter them on the keyboard, then calculate the result and output it to the user. (6 marks)

b) Mention and briefly discuss the basic **C** data types (8 marks)

c) Define the following terms. (6 marks)

- Variable
- Arrays
- Functions

d) A program error can either be syntax, logic or semantic. Classify the errors below

i) An indefinite loop
ii) Misspelling keywords
iii) Division by zero error (3 marks)

e) Using 3.14 as **pie** value demonstrate how you create symbolic constants (2 marks)

f) Provide a **C** statements to declare an **int** array and initialize its' elements to 10 , 20 , 30, 40 and 50 respectively (4 marks)

g) Provide function prototype for a function to calculate and return the product of three integer values (3 marks)

h) Create a simple c program to that allows user to enter name and display the name on the screen. (3 marks)

SECTION B – EACH QUESTION CARRIES 20 MARKS

ATTEMPT ANY TWO QUESTIONS

Question 2

a) Give reason(s) why the following identifier names are invalid
i) 98mark
ii) First Name
iii) #total
i) char (4 marks)

b) Write program to read the values a, b and c and display the value of x, when x = a / b – c (6 marks)

c) Find errors in the following function prototypes

i) float average(x,y); -no error
ii) int mul(int a,b); -b is not declared. (2 marks)

a) What do these loops print?

i) for(i = 0; i < 10; i = i + 2)

```
    printf("%d\n", i);
ii) for(i = 100; i >= 0; i = i - 7)
    printf("%d\n", i);
iii)for(i = 1; i <= 10; i = i + 1)
    printf("%d\n", i);
iv)for(i = 2; i < 100; i = i * 2)
printf("%d\n", i);
```
(8 marks)

Question 3

a) Write a program that uses a function to return the sum of two numbers. (6 marks)
b) Write a function celsius() to convert degrees Fahrenheit to degrees Celsius. (The conversion formula is °C = 5/9 * (°F - 32).). (6 marks)
c) Explain how while is different from do...while (2 marks)

d)
i) What is a pointer variable (2 marks)

ii) Provide program statements to demonstrate how to create a pointer variable , assign address to a pointer variable and output the value at the memory location pointed to by the pointer variable (4 marks)

Question 4

a) Write a c program that asks the user to enter the total marks of the student. The program then prints 'proceed' if the marks are greater than 40 otherwise it prints 'fail'. (6 marks)

b) Write a program that prompts for circle diameter, then calculates and outputs both circle radius and area (6 marks)
c) Why is the function main() special in a c program? (2 marks)
d) Discuss the two methods of including comments in your **C** program (2 marks)
e) Write a program that outputs the string "*C is a structured programming language"* on the screen. (4 marks)

Question 5

a) Define a function that returns the factorial of a number (you have to use a loop). The argument to the function is an integer whose factorial is to be found. Use the function in a main program where you ask the user to enter a number then your program outputs the factorial. (10 marks)

b) Write a program that uses an array to read marks scored by a diploma student in 4 subjects, and then calculates the mean score. The program should finally output the scores in the four subjects and the mean (10 marks)