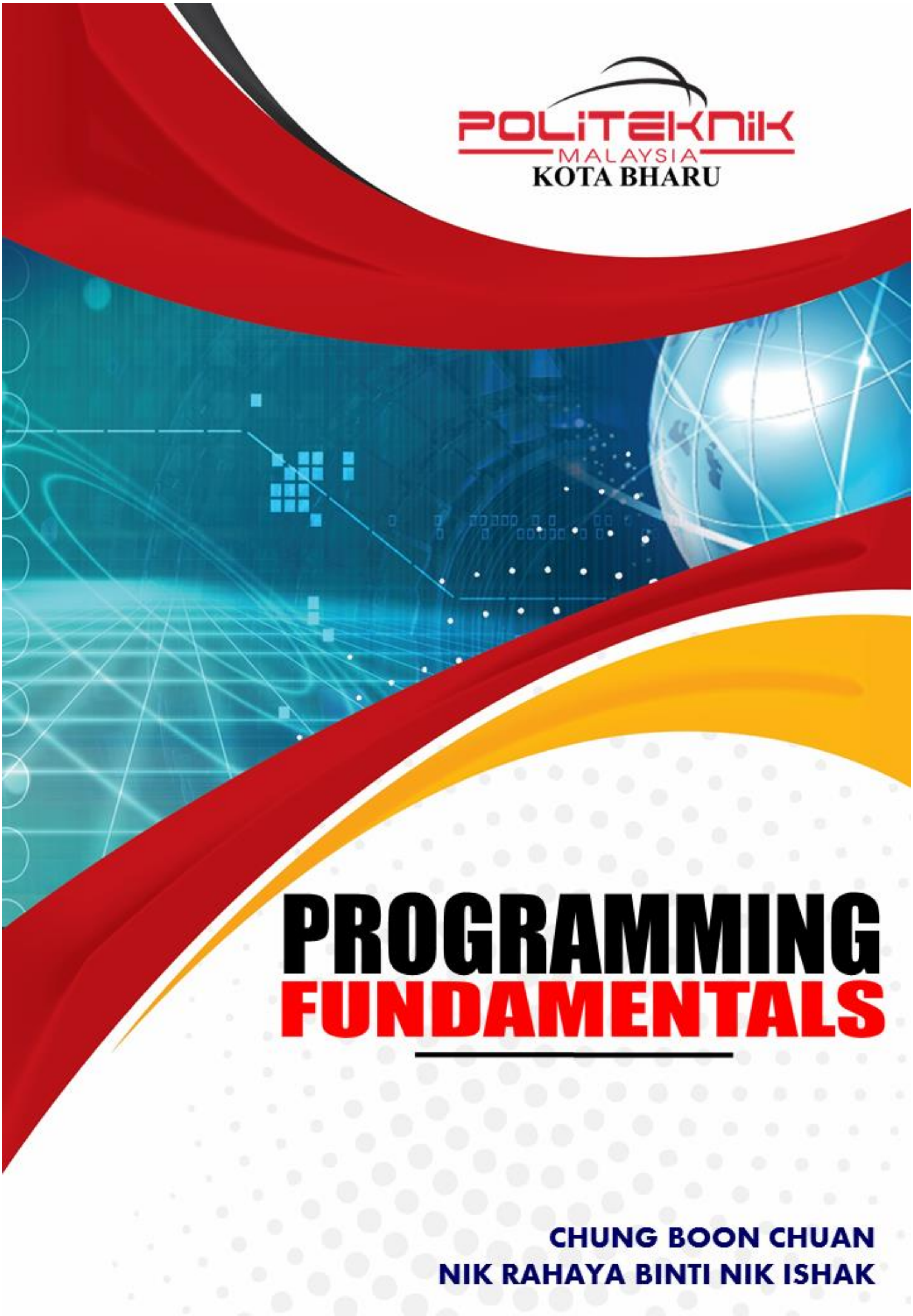


POLITEKNIK MALAYSIA
KOTA BHARU

# PROGRAMMING FUNDAMENTALS

CHUNG BOON CHUAN
NIK RAHAYA BINTI NIK ISHAK

1

# PROGRAMMING FUNDAMENTALS

**CHUNG BOON CHUAN**

**NIK RAHAYA BINTI NIK ISHAK**

**POLITEKNIK KOTA BHARU**

**ISBN -9789671663974**

9789671663974

Published and printed by:

Department of Electrical Engineering

Politeknik Kota Bharu

KM. 24, Kok Lanas, 16450 Ketereh, Kelantan.

Programming Fundamentals

First Edition 2021

B.C Chung

Programming Fundamentals/ B.C.Chung

# PREFACE

Apart from our efforts, the success of any book writing project depends largely on the encouragement and guidelines of many others. We would like to take this opportunity to express our sincere gratitude to the people who have been instrumental in the successful completion of this book. Finally Programming Fundamentals book was successfully published after working hard preparing this book since June 2020. This book is written and published as a guide or reference to all engineering students in polytechnic about the fundamental programming. This book will expose students to the application of C programming. C Programming is a general-purpose programming language that is extremely popular, simple and flexible. It is machine-independent, structured programming language which is used extensively in various applications.

I would like to express appreciation to committee member Nik Rahaya Binti Nik Ishak and linguist Melissa Khor Suan Chin for her effort in completing this book. Hopefully this book can preferably utilized by all lecturers and students. Thank you.

CHUNG BOON CHUAN
Electrical Engineering Department
Polytechnic Kota Bharu
KM24 Kok Lanas
16450 Ketereh
Kelantan

# SYNOPSIS

PROGRAMMING FUNDAMENTALS provides the skills necessary for the effective application of computer programming in engineering. Users will be able to develop their programming skills through various methods and practical by reviewing sample programmes. The learning outcome is for the users to acquire proficiency in writing small to medium programmes based on a procedural programming language.

Programming is an increasingly important skill, whether you aspire to a career in software development, or in other fields. This book is the first in the specialization Introduction to Programming in C, but its lessons extend to any language you might want to learn. This is because programming is fundamentally about figuring out how to solve a class of problems and writing the algorithm, a clear set of steps to solve any problem in its class. This course will introduce you to a problem solving process which can be used to solve any programming problem. In this book, you will learn how to develop an algorithm, then progress to reading code and understanding how programming concepts relate to algorithms.

This book contains a description of the basic concepts needed in learning fundamental programming. Each chapter has an example of a program. Apart from that, this book also has exercises that allow you to use it yourself with the topics described. This book contains several chapters namely the contents of the Introductory to Programming, Fundamentals of C Language, Selection Statements, Looping Statements, Function and Array.

# About the Authors

**Chung Boon Chuan**

The author was born in 1972 in Tanah Merah, Kelantan. He received his primary and secondary education in Tanah Merah before enrolling at Johor Bharu. Have academic qualification in Electrical Engineering First Degree (UTM, 97) and Bachelor of Education (UTM, 98). Began his career as a lecturer Polytechnic Seberang Prai (PSP) in August 1999 and is now being served at the Polytechnic Kota Bharu (PKB). Actively participate in the Innovative and Creative Convention (ICC) rankings Polytechnic and Ministry until his appointment as facilitator and invited as a jury. Experience in the field of computer software and frequently invited as a speaker courses involving computing. He is a Master Trainer MQA (Malaysian Qualification Accreditation) for the programs offered at the Department of Electrical Engineering, Polytechnic Kota Bharu. He is also one of the curriculum program offered Diploma in Electronic Engineering at the Department of Electrical Engineering, Polytechnic Malaysia.

**Nik Rahaya binti Nik Ishak**

The author was born in 1975 in Kota Bharu, Kelantan. She received her primary and secondary education in Kota Bharu before enrolling at Alor Setar, Kedah in 1995. She have an academic qualification in Information Technology First Degree (UUM, 98) and Masters of Education (UTM, 2000). Began her career as a lecturer at Polytechnic Kota Kuala Terengganu (PKT) in September 2020 and is now being served at the Polytechnic Kota Bharu (PKB). Actively participate in the Innovative and Creative Convention (ICC) rankings Polytechnic and Ministry. Experienced in teaching computing related subjects such as C and Visual Basic programming at current polytechnic.

# CONTENTS

# 1.0 INTRODUCTORY TO PROGRAMMING

**INTRODUCTION**

The word computer was derived from the word compute, which means to calculate. Computers were introduced to perform fast and accurate calculations. Today, computers are widely used in almost all fields. Computers, however, have to be instructed by humans to perform any task. The instructions are given in the form of programs. In this unit, you will learn the different types of programming languages and the various programming techniques involved in writing programs.

# Note

## 1.1 PROGRAMMING LANGUAGE

Computer needs human to provide it with instructions to make it perform tasks. These instructions must be provided in a language that the computer understands. This language through which humans communicate with computers is known as programming language.

### 1.1.1 C programming

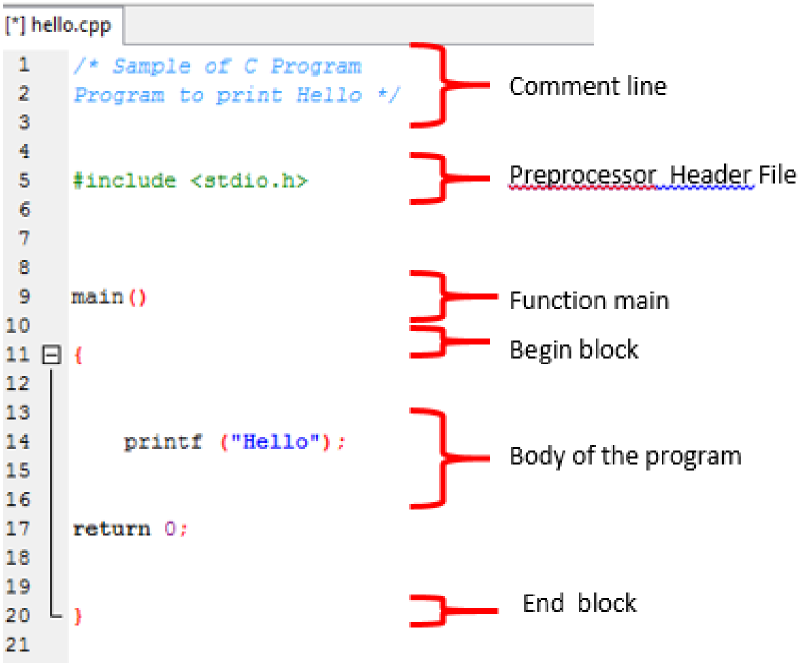
C is a computer programming language which can use to create lists of instructions for a computer to follow. C is a compiled language which means it must run through a C compiler to turn the program into an executable that the computer can run (execute). The C program is the human-readable form, while the executable that comes out of the compiler is the machine-readable and executable form.

### 1.1.2 Background of C programming

C is a programming language developed at AT&T Bell laboratories by Dennis Ritchie. This programming language was named C as it followed an earlier programming language named B.  C is portable, powerful and flexible. Because of these features, C is used for both system and application level programming. C is reliable, simple and easy to use. It is easier to learn newer languages once you are familiar with C language.

### 1.1.3 Sample of C program

A basic C program has the following form:



```
/* Sample of C Program
Program to print Hello */


#include <stdio.h>



main()

{


    printf ("Hello");


return 0;


}
```

### 1.1.4 Compile programs

Compilation is the process of conversion of high-level language into machine language. The process of conversion is called compilation.  During compilation, the entire program is compiled and all errors in the program are displayed. Once the errors are corrected, the program is compiled successfully. The source code gets converted into object code.

### 1.1.5 Execute programs

Execution is the process of running the program to get the desired output. Before a program is executed, it has to be loaded into the memory. A special program called the loader will take the executable code from the disk and place it in the memory. This process is called loading.

## 1.2 TYPES OF PROGRAMMING

### 1.2.1 Define the following terms

a. Programme: A program is a set of instructions given to the computer to perform any task.

b. Programmer: The person who writes the program is called a programmer.

c. Programming language: A programming language is a language that is used for writing program.

d. Programming: The process of writing these instructions is known as programming

### 1.2.2 Programming languages

Programming languages are broadly classified into low-Level languages and High-Level Languages.

**a. Low-Level Languages**

Low-level languages are programming languages in which programs are written based on the internal architecture (design) of machines. Low-level languages are further divided into two types:

**i. Machine language**

Machine language is the only language that computers can understand and all the instructions in this language are in the form of 1s and 0s.

**ii. Assembly language**

Assembly language is a programming language that use symbolic instructions called mnemonics or names.

**b. High-Level Language**

High-level languages are programming languages that use English words and mathematical symbols to represent instructions in programs.

Advantages:

i. Programs written in high-level languages are almost machine- independent.

ii. Programmer need not know about computer hardware to write programs in high-level languages.

iii. It is easier to learn and write programs in high-level languages than low-level languages.

## 1.3 TYPES OF PROGRAMMING LANGUAGES

### 1.3.1 Structured Programming

Structured programming is a programming methodology in which the instructions are written in a sequence. The structure of a program refers to the order in which the statements are executed in the program.

### 1.3.2 Modular Programming

Modular programming is a programming methodology in which the complex program is broken into number of simple modules. Complex program is broken down into *segments* of code called modules. A *module* is an independent segment of the program that performs a specific task.

### 1.3.3 Object-Oriented Programming

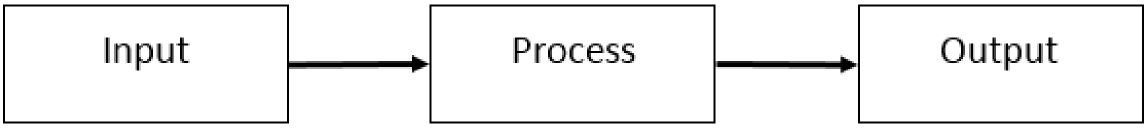
Object-oriented programming is a programming methodology in which the data and the code are treated as a single unit. In object-oriented programming the data and the code are treated as a single unit. Object-oriented programming gives more importance to the data. This programming approach aims to provide solutions for real-world related problems.

## 1.4 ALGORITHM, FLOWCHART AND PSEUDOCODE

Various programming tools, such as algorithm, flowchart and pseudocode, are used for solving problems. These tools are used to represent the instructions and designing the sequence of instructions in a program.

### 1.4.1 Algorithm in programming

An Algorithm is the sequence of steps required to provide a solution to a problem. Notice that the algorithm is a set of instructions written in simple English. Each step of the algorithm should be simple and it should clearly represent the input, process and output.



**Characteristics of Algorithms**

The characteristics of algorithms are:

a. Instructions must be simple.

b. Instructions must be accurate.

c. Instructions should be general statements. It should not be written with respect to any specific programming language.

d. Instructions should not be repeated infinitely.

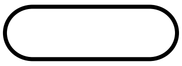
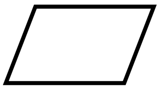
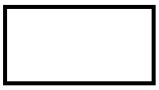
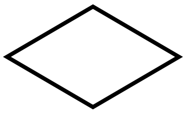
e. Algorithm should be stopped after performing the instructions.

### 1.4.2 Flowchart in programming

Flowchart is an algorithm represented graphically. Flowcharts are used to analyze problems. Then, it is used to develop the logic to solve problems. The process of drawing a flowchart is referred to as flowcharting. Instructions in a flowchart are represented using different symbols. These symbols are called as flowchart symbols.

#### a. Flowchart Symbols

Table 1.4.2 shows the basic flowchart symbols used for representing the different types of instructions.

| Symbol | Name | |
|---|---|---|
|  | Terminal | This symbol is used to represent the starting and ending point in the program logic flow. Every flowchart should have only one start and one end terminal symbol |
|  | Input/Output | This symbol is used to either accept the input or to display the output data. |
|  | Processing | This symbol is used to represent mathematical instructions |
|  | Decision | This symbol is used when there is a condition. The test condition is written within this symbol. Based on this condition, appropriate path is followed during program execution. |

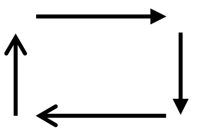
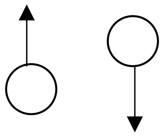
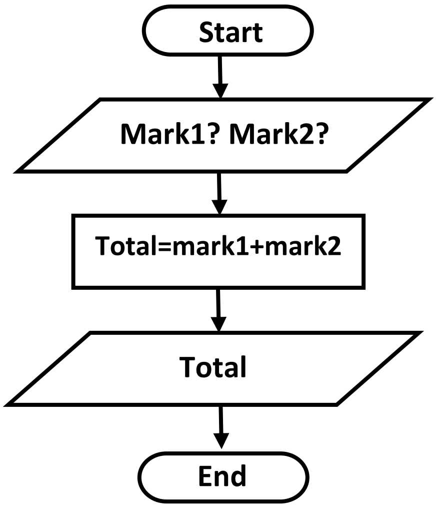
| | | |
|---|---|---|
|  | Flow Lines | This symbol is used to represent the flow of the algorithm. The exact sequence in which the instructions are executed is represented using this symbol. |
|  | Connectors | This symbol is used to join a flowchart that is drawn in many pages. A pair of identically labelled connector symbol is used to indicate a continued flow of the flowchart |

**Table 1.4.2: Flowchart Symbols**

**b. Flowchart Examples**



**c. Advantages**

- A flowchart is easy to understand since it is a graphical representation of the problem.
- If any errors occur in the program at a later stage, a programmer can easily identify the reason for the error by looking at the flowchart.
- A flowchart helps the programmer to avoid forgetting any steps while writing a program.
- It is easier to write programs after drawing the flowchart.

**d. Disadvantage**

- Drawing a flowchart is a time-consuming process.
- It is very difficult to draw the flowchart for programs that contain complex branches and loops.
- The amount of details that a flowchart should represent is not standardised.

### 1.4.3 Pseudocode in programming

Pseudocode is a detailed description of what a computer program or algorithm must perform. It is an outline of program that can be converted into programming instruction. Pseudocode use special reserved words for expressing the logic of programs.

#### a. Pseudocode keyword

Table 1.4.3 lists the keywords that are used in a pseudocode.

| Keyword | Description |
|---|---|
| **//** | Refers to a comment entry. It is used to give additional information about the program. It can be used anywhere inside the pseudocode. |
| **begin**<br>**---**<br>**end** | Specifies the block of statements that performs a specific task. The first statement is the **begin** statement and the last statement is the **end** statement. The other instructions are given between these two statements. |
| **Accept** | Accepts the input values. |
| **Display** | Displays the output. |
| **Set** | Initialises the value of variables. |
| **if --- else** | Specifies the condition and the block of statements that are executed based on the condition. |
| **for** | Specifies the conditions and number of times the block has to be executed based on the condition. |

**Table 1.4.3 : Keywords Used in Pseudocode**

#### b. Pseudocode Examples

```
begin
 accept the mark1 and mark2 obtained by the student
 add mark1 and mark2 into the running total
 display total
end
```

#### c. Advantages

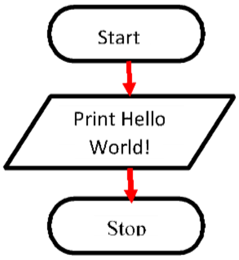
- It enables even a non-programmer to understand the working of programs.
- The time taken to write pseudocode is less compared to drawing a flowchart.
- It is easier to modify the pseudocode compared to modifying flowcharts.

- A pseudocode can easily be converted into a program by replacing the pseudocode instructions with equivalent program statements.

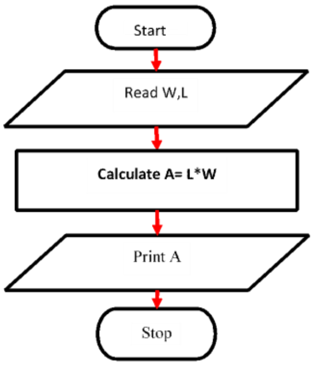
## 1.5 Apply Algorithm, flowchart, pseudocode and analyze problem

### 1.5.1 Construct flowcharts for the given problem

a. To print message Hello World!



b. To calculate area of rectangle



### 1.5.2 Apply flowchart for the following problems

The unit so far dealt with writing algorithms and pseudocode for simple problems. In these, all the instructions are executed in the same sequence as they appear in the algorithm or the pseudocode. However, certain instructions need to be executed only when a condition is satisfied. This is done by altering the sequence of execution. The order in which the instructions are executed will depend on the structure of the program. All simple or complex computer programs are written using one or a combination of the following basic structures:

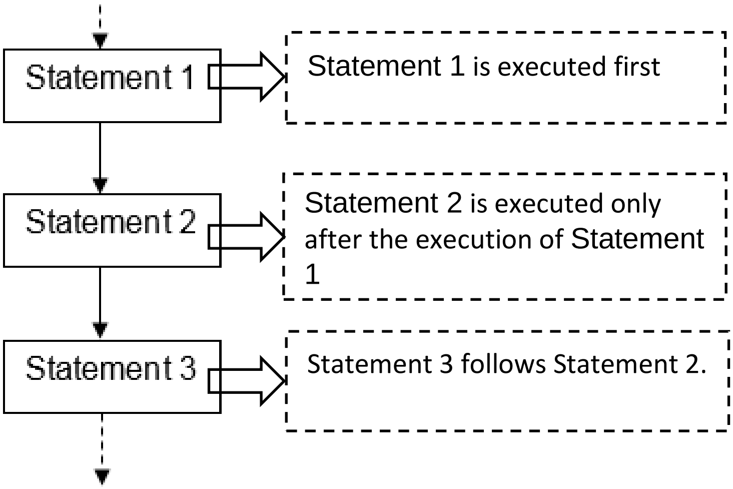
a. Sequence structure
b. Selection structure
c. Looping structure

### 1.5.3 Sequence structure

In a sequence structure, all the instructions in the program must run one after another. There is no skipping of any instruction; no task is skipped in the sequence. Here, the instructions are executed one by one, or in sequence.

a. **Sequential structure**

Figure 1.5.3 shows the flowchart for sequential execution of instructions.



Statement 1 is executed first

Statement 2 is executed only after the execution of Statement 1

Statement 3 follows Statement 2.

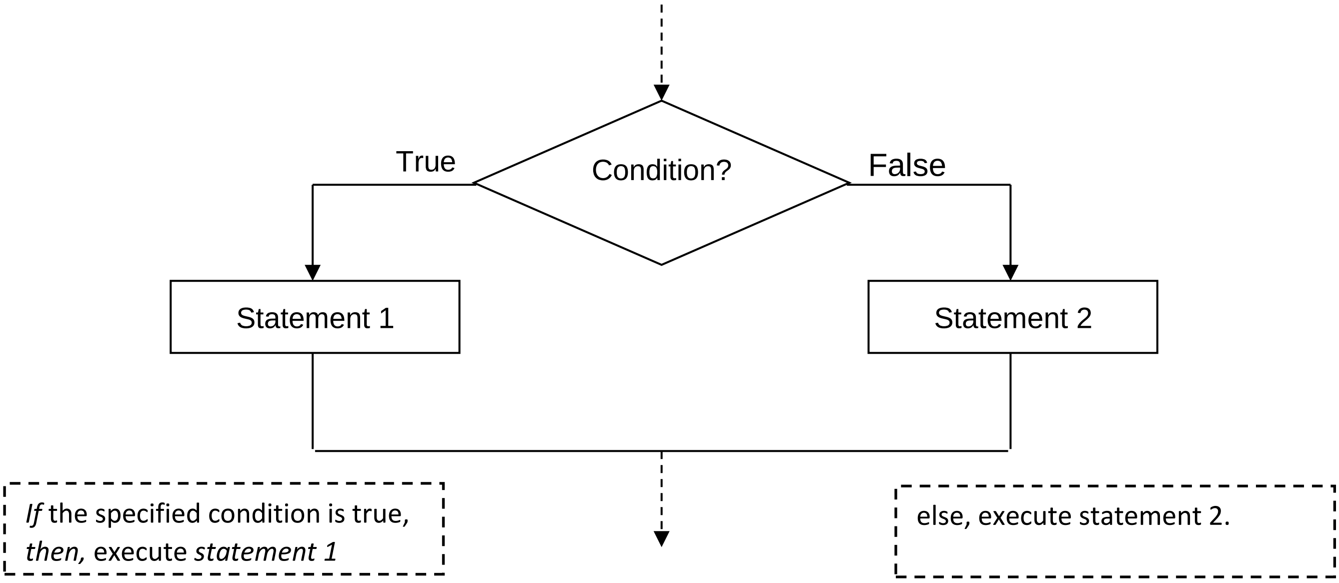
**Figure 1.5.3: Flowchart for Sequence Structure**

### 1.5.4 Selection structure

In a selection structure, the instructions are executed based on the answer for a stated condition. Based on the condition, only one of the two tasks will be executed.

a. **Selection structure.**

Figure 1.5.4 shows the flowchart for selection structure.



*If* the specified condition is true, *then,* execute *statement 1*

else, execute statement 2.

**Figure 1.5.4: Flowchart for Selection Structure**

### 1.5.5 Looping structure

In a looping structure, a set of instructions is executed several times based on certain conditions.

**a. Looping structure**

The Figure 1.5.5 shows the flowchart for looping structure.

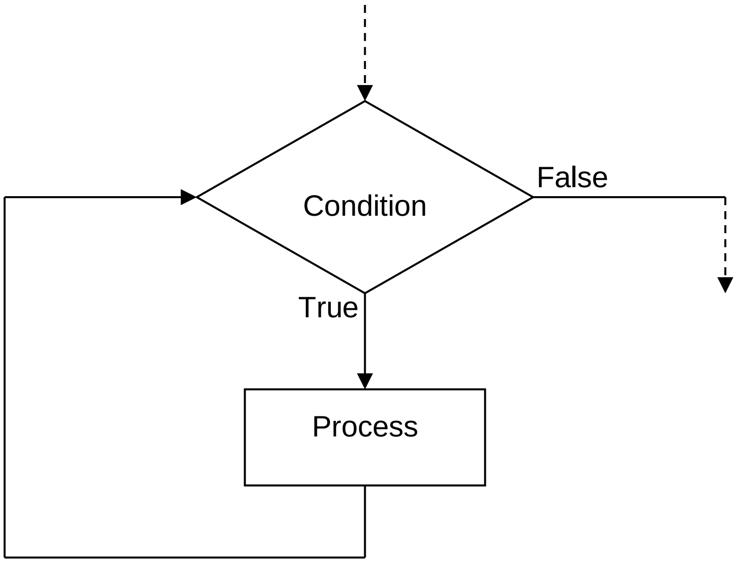


**Figure 1.5.5: Flowchart for Looping Structure**

# Tutorial

a) List THREE (3) types of programming

b) Define the following terms

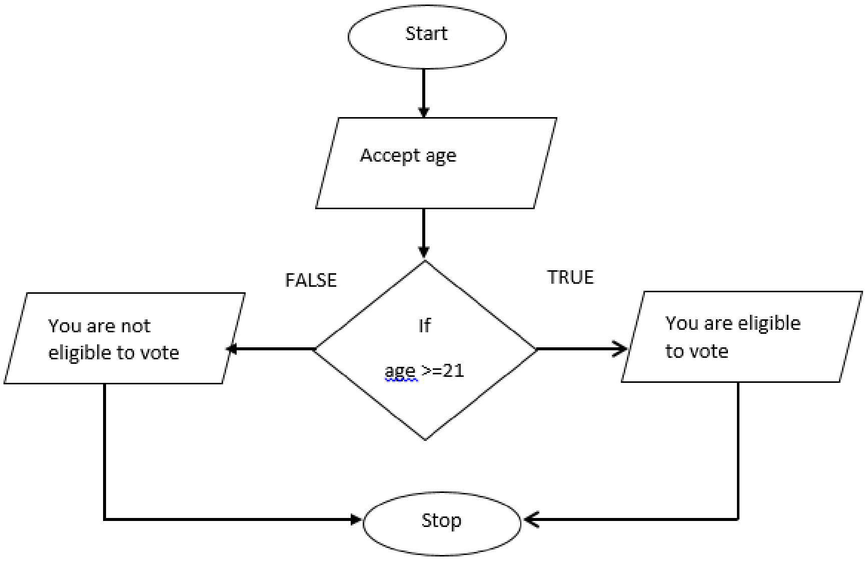
c) Draw the flowchart to calculate the average of three numbers

d) Define the programming language and list TWO (2) types of programming languages.

e) Sketch the flowchart by referring the pseudocode below :

```
Start
        Enter the current price and increase rate
        Calculate the increase amount by multiplying the current
        price by the increase rate
        Calculate the new price by adding the increase amount to the
        current price
        Display the increase amount and new price
End
```

f) Write a pseudocode based on flowchart below :



g) Define the high level language in programming.

h) Explain what C programming is and write TWO (2) lines of code to illustrate it.

i) Based on the pseudo code below, draw a flowchart

```
Start
Enter radius
Calculate area of the circle = pi X radius X radius
Display area
End
```

# 2.0 FUNDAMENTALS OF C LANGUAGE

## INTRODUCTION

Before you start writing programs, it is very important to understand the basic elements that are used in constructing simple C statements. These are C character set, variables, constants, data types and keywords as shown in Figure 2.0.

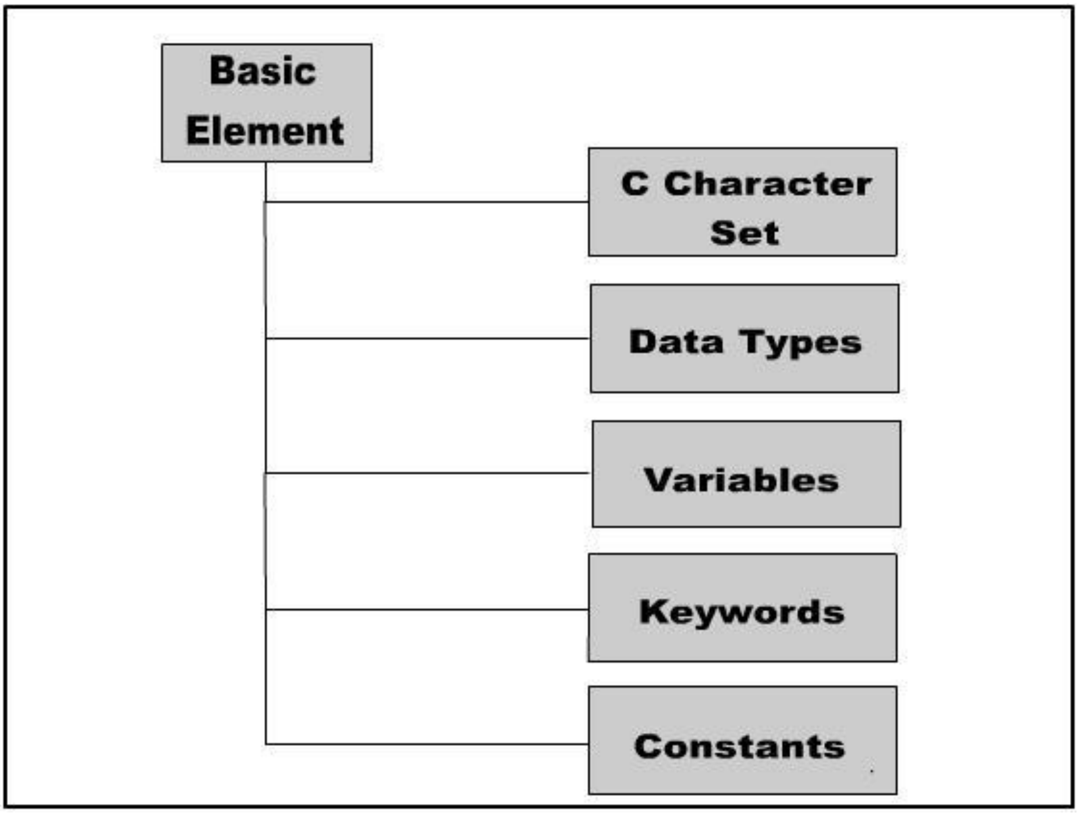


**Figure 2.0: C Basic Elements**

**Note**

## 2.1 VARIABLES, CONSTANTS AND DATA TYPES

The type of data for the variables and constants determines the type of information to be stored in the allocated memory space in C programming.

### 2.1.1 C Character set

A character or set of characters is used to represent any information. C uses alphabets, numbers and certain special characters for this purpose. Table 2.1.1 represents the alphabets, numbers and special characters that are valid in C.

| Type | Character Set |
|---|---|
| Alphabets | A, B, C … X, Y, Z<br>a, b, c … x, y, z |
| Digits | 0, 1, 2 … 9 |
| Special characters | + - * / % = & # < > ( ) : { } _ ; ? “ \| ‘ . , ~ \ ! ^ (blank space) |

**Table 2.1.1: C Character Set**

### 2.1.2 Data types available in C

**Definition*:*** *Data type* will define the type of the value to be stored in the memory.

Programming languages require the programmer to declare the data type of every data object used in the program.

C supports several different types of data. The following are the basic data types available in C:

| | |
|---|---|
| Integer | A whole number, a number that has no fractional part. |
| Character | A single character or a group of characters (string). |
| Floating-point | A number with a decimal point. |
| Byte | a group of binary digits or bits (usually eight) operated on as a unit. |
| Long | A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. |
| Double | A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. |
| String | A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. |
| Word | A word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. |

**Table 2.1.2: Data types available in C**

### 2.1.3 Keywords in C

**Definition***:* Keywords are reserved words for which the meaning is already defined to the compiler.

C has special reserved words that cannot be used as identifiers. These are keywords. There are 32 keywords in C language. They are listed in Table 2.1.3.

| Types | Keywords |
|---|---|
| Data types, modifiers and storage class specifiers | void, int, char, float, double, signed, unsigned, long, short, auto, const, extern, static, volatile, register and typedef |
| User defined data types and type related | struct, union, enum and sizeof |
| Conditional | if, else, switch, case and default |
| Flow control | for, while, do, break, continue, goto and return |

**Table 2.1.3: Keywords in C Language**

### 2.1.4 Variable name in C

**Definition:** Variable is the name given to the memory location where the data is stored. This value keeps changing during the program execution.

Computer stores data in the computer memory in a specific location. To use the data, you must know the address of the memory location, where the data is stored. The name given to this memory location is called a variable. The data of a variable changes during the execution of the program.

**Rules for Naming Identifiers**

The naming convention of the identifiers follows the following rules:

1. Can be a combination of alphabets and numbers but must start with an alphabet.
2. Comprise maximum of 40 characters.
3. No commas or blank space is allowed.
4. No special characters can be used except underscore (_).

### 2.1.5 Defining versus declaring variables

Definition of variable mean asking compiler to allocate memory to variable or define storage for that variable. Can define a variable only one time.

Declaration of variable mean to tell compiler that is a variable\function of particular data type.

Any variable used in a program must be declared in the beginning. This is to specify the variable data type to the compiler. Table 2.1.5 lists some of the declarations used in C language.

| Keywords Used for Data type | Description |
|---|---|
| **int** | Integer number: A whole number without fractional part. |
| **float** | Floating-point number (with fractional part). |
| **Char** | Character: A single character or string. |

**Table 2.1.5: Keywords for Declaration of Variables**

**a. Variable declaration : Example 1**

| | |
|---|---|
| char | student_name; |
| Data type is character | *student_name* is a variable |

- *student_name* is a variable, which can take character value.
- When you write **char name**, it is called *variable declaration*. Variable declaration means specifying the type of the variable.

**b. Variable declaration : Example 2**

| | |
|---|---|
| int | mark; |
| Data type is integer. | *mark* is a variable.. |

- *age* is a variable, which can take integer value.

**c. Variable declaration :Example 3**

| | |
|---|---|
| float | mark2; |
| Data type is float. | *Mark2* is a variable. |

- *mark* is a variable, which can take integer value.

## 2.2 FUNDAMENTALS OF C PROGRAMME

Fundamentals of C program focused on the basic elements used to construct a simple C program such as the C character set, identifiers and keywords, data types, constants, arrays, declarations , expressions and statements.

### 2.2.1 Explain structure of C programs

Languages like English have words, symbols and grammar rules. Similarly, programming languages too have words, symbols and rules. In a programming language, the rules are known as the *syntax*. If these rules are not followed, a program will not work. Each programming language has its own set of syntax and structures to be followed. A typical C program will appear as shown in Table 2.2.1.

| **Structure** | **Sample Program** |
|---|---|
| `< Comment entry >`<br>`< Preprocessor directives >`<br>`main()`<br>`{`<br>`  < Declarations >;`<br>`  < C statements >;`<br>`}` | `/* First C program */`<br>`include <stdio.h>`<br>`main()`<br>`{`<br>` int a;`<br>` printf("Welcome to C ");`<br>`}` |

**Table 2.2.1: Structure of a C Program**

### 2.2.2 Structure of a function in the main function

main() is the function where the program is written. Any C program will have one or more functions and the most important function, which must be present in all the programs, is the main() function. A program can have only one main() function.

### 2.2.3 C Preprocessor

**#include <stdio.h>** is a preprocessor directive statement. This statement directs the preprocessor to include the standard input and output header file.

The symbol hash (**#**) will invoke the preprocessor directives. *Preprocessor directives* are the instructions given to the compiler. The keyword **include** will direct the preprocessor to include the specified header file into the program. A *header file* contains definition for all the functions that could be shared by several other programs.

**Syntax**

```
#include   <filename>
```

**Example**

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

### 2.2.4 Valid identifiers

When you name a variable, choose a relevant identifier. For example, name the variable to accept the mark of students as *mark*, age as *age*, and so on. Table 2.2.4 gives a list of valid and invalid identifiers.

| Identifiers | Valid/Invalid | Reason |
|---|---|---|
| Grade | Valid | Identifier should start with an alphabet. |
| 2 | Invalid | Identifier starts with a number, which is not allowed. |
| item1 | Valid | Identifier can have combination of alphabets and numbers. |
| 1name | Invalid | Identifier starts with a number. |
| Amount 1 | Invalid | No blank space is allowed within an identifier. |
| code_1 | Valid | Special symbol underscore (_) can be used in identifier. |
| area* | Invalid | Special character asterix (*) cannot be used in identifier. |

**Table 2.2.4: List of Examples for Valid Identifiers**

### 2.2.5 Differentiate constants and variables in C

Difference between variables and constants.

| Variables | Constants |
|---|---|
| Variable is the name given to the memory location where the data is stored. This value keeps changing during the program execution. | Constant is a location in the memory that stores data that never changes during the execution of the program. |
| **Example:**<br>a, grade and mark | **Example:**<br>10, 'A' and 78.90 |

**Table 2.2.5: Difference Between Variables And Constants**

### 2.2.6 Explain input/output statement

An input/output statement or IO statement is a portion of a program that instructs a computer how to read and process information. It pertains to gather information from an input device, or sending information to an output device.

a. Input Statement

Definition: Input statements are used for accepting data from the user.

Input statements are used to make the program more interactive.

```
scanf( )
```

scanf( )

**Definition: scanf()**is a function, used as a statement, to get the value from the user.

**Example**

```
scanf("   %d",         &age);
          Format specifier   Identifier
```

The scanf()function has two parameters: the format specifier and the variable. These parameters have to be separated by the delimiter comma (,). In the example, scanf statement will request the user to enter the age. The variable age is of an integer data type.

The symbol ampersand (&) refers to the address of a variable in the memory. The input data has to be stored in the memory addressed by that variable.

**Syntax**

```
scanf(“<format specifier>",        & <variable>);
```

                    Format specifier                              Identifier

Format Specifiers

**Definition**: Format specifier defines the data type of the variable to the compiler.

To accept a value, you must specify the type of data expected. Consider the example given for scanf(). As the variable age is integer type, the format specifier, %d, is used. It indicates that the variable age is of integer type.

Some of the commonly used format specifiers are listed in Table 2.2.6.

| Format Specifier | Data Type |
|---|---|
| %d | Int |
| %f | Float |
| %c | Char |

**Table 2.2.6a: Format Specifiers**

b. Output Statement

**Definition**: Output statements are used for displaying the processed data on the screen.

**printf( )**

**Definition:** printf()is a function, used as a statement, to display the data on the screen.

**Example 1**

```
printf("Welcome");
```

**Output**

*Welcome*

```
Welcome
```

In example 1, the string *Welcome* will be displayed on the screen.

**Example 2**

| | | | |
|---|---|---|---|
| printf(" | My age is | %d", | age); |
| | String to be displayed on the screen | Format specifier. | Variable whose value will be displayed. |

**Output**

**My age is 16**

In example 2, age is a variable of integer data type. Assume that the user enters the value 21 for the variable age. To display the value of a variable on the screen, the relevant format specifier must be specified within double quotes.

**Syntax**

**printf("<string> <format specifier>",<variable>);**

**Definition:** Escape sequences are nonprinting characters that are used for formatting the output.

C provides various escape sequences to display the outputs on screen in the specified formats. For example, the escape sequence **\n** will display the text in a new line and **\t** will include horizontal tab spacing. Some commonly used escape sequences are listed in Table 2.2.6b.

| Escape Sequence | Description |
|---|---|
| **\n** | Newline |
| **\t** | Horizontal tab |
| **\’** | Apostrophe (‘) |
| **\”** | Quotation (“) |
| **%** | Percentage (%) |

**Table 2.2.6b: Escape Sequence**

### 2.2.7 Types of Operators

In some of the programs, you might need to perform some mathematical calculations. Various operations such as adding two numbers, comparing two numbers and so on need to be performed. Various operators and expressions are used for this purpose.

**Definition:** An operator is a symbol that instructs C to perform some operation, or action.

In a program, for adding two numbers you need to use the addition (+) operator. Similarly, for comparing two numbers you need to use a comparison operator. It is therefore necessary to use the appropriate operators to perform these calculations. For example, + is an operator that represents addition.

**Example**

**c = a + b**

In this code line, + is an operator and *a*, *b* and *c* are called operands. *Operands* are the variables/constants on which the operators operate.

In C, all operands are expressions. C operators can be classified as:

a. **Assignment operators**

**Definition:** Assignment operator (=) is used to assign a value to a variable.

**Example1**

**a = 10;** Value *10* is assigned to the variable *a*.

In example1, the value *10* is assigned to *a*.

In a C statement, the right side of the operator can be any expression, and the left side *must* be a variable name. Thus, the form is as follows:

**Variable = expression;**

When executed, expression is evaluated, and the resulting value is assigned to variable.

**Example2**

**x = y + z;**

In example2, the sum of y and z is calculated and the result is stored in x.

### b. Mathematical Operators

Definition: Mathematical operators are used for performing simple mathematical calculations such as addition, subtraction, multiplication and division.

The various Mathematical operators available in C are listed in Table 2.2.7a.

| Operator | Operation | Description | Example |
|---|---|---|---|
| + | Addition | Adds two operands. | z=x+y |
| - | Subtraction | Subtracts an operand from another. | z=a-b |
| * | Multiplication | Multiplies an operand with the other. | z=a*b |
| / | Division | Divides an operand by another. | z=a/b |
| % | Modulo | Calculates the remainder when an operand is divided by another. | z=a%b |

**Table 2.2.7a: Mathematical Operators**

You must know how to convert a general mathematical notation to equivalent C statement. Table 2.2.7b lists some of the examples of C expressions as shown:

| Mathematical Notation | C Expression |
|---|---|
| 5xy | 5*x * y |
| $\frac{a+b}{a-b}$ | (a+b)/(a-b) |
| (pq)-(rt) | (p*q)-(r*t) |

**Table 2.2.7b: Mathematical Notations and Their Equivalent in C**

### c. Relational Operators

**Definition:** Relational operators are used to compare two operands.

The output of the expression will be a Boolean value, which is either 0 (false) or 1 (true). The various relational operators are listed in Table 2.2.7c.

| Symbol | Description | Example |
|---|---|---|
| > | Greater than | a>b |
| < | Lesser than | a<b |
| >= | Greater than or equal to | a>=b |
| <= | Less than or equal to | a<=b |
| != | Not equal to | a!=b |
| == | Equal to | a==b |

**Table 2.2.7c: Relational Operators**

### d. Logical Operators

**Definition**: Logical operators are used to combine two simple statements into a compound statement.

Using logical operators, you can simulate Boolean algebra in C. The various logical operators are listed in Table 2.2.7d.

| Symbol | Operation | Description |
|---|---|---|
| && | AND | The AND (&&) operator will evaluate to true only if all the conditions in the expression returns a true value. |
| \|\| | OR | The OR (\|\|) operator will evaluate to true even if one of the conditions is true. |
| ! | NOT | The NOT (!) operator will evaluate to true if the condition fails and vice versa. |

**Table 2.2.7d: Logical Operators**

### e. Unary Operators

Definition: A unary operator which operates on one value or operand.

Minus sign (-) is used for substraction as a binary operator and for negotiation as an unary operator.

result = -x * y;

Result will contain a negative value of 40 which is -40.

| Symbol | Description | Example |
|---|---|---|
| + | Positive | a = +3 |
| - | Negative | b = -4 |

**Table 2.2.7e: Unary Operators**

### f. Increment Operators

**Definition:** Increment operators are used to increase the value of the variable by one in C programs.

**Syntax**

```
++var_name; (or) var_name++;
```

### g. Decrement Operators

**Definition:** Decrement operators are used to decrease the value of the variable by one in C programs.

**Syntax**

```
--var_name;

      (or)

var_name--;
```

## 2.3 Apply fundamentals of C programming

### 2.3.1 Construct a simple C program

Step 1: Open C editor and types :

```
#include <stdio.h>
main()
{
printf(" WELCOME ")
}
```

Step 2: Save the program.

### 2.3.2 Compile and execute programs

Step 3: Compile the program.

Step 3: Run the program and observe the output.

### 2.3.3 Use input statements in C Program

Step 1: Open C editor and types:

```
#include <stdio.h>
main()
{
int age;
scanf("%d",&age);
return 0;
}
```

Step 2: Save the program.

Step 3: Compile the program.

Step 3: Run the program and observe the output.

Step 3: output.

```
Enter the age : 18
 Age : 18
```

### 2.3.4 Apply output statements in simple C program

Step 1: Open C editor and types:

```
#include <stdio.h>
main()
{
int age;
char grade;
float mark;
printf("\n Enter the age : ");
scanf("%d",&age);
printf("\n Enter the grade : ");
scanf("%c",&age);
printf("\n Enter the mark : ");
scanf("%f",&mark);
return 0;
}
```

Step 2: Save the program.

Step 3: Compile the program.

Step 3: Run the program and observe the output.

### 2.3.5 Display the output in the specified format

**Step 1**: Open C editor and types:

```
#include <stdio.h>
main()
{
int age;
char grade;
float mark;
printf("\n Enter the age : ");
scanf("%d",&age);
printf("\n Enter the grade : ");
scanf("%c",&age);
printf("\n Enter the mark : ");
scanf("%f",&mark);
printf("\n The age :%d ",age);
printf("\n The grade :%c ",grade);
printf("\n The mark :%f ",mark);
return 0;
}
```

Step 2: Save the program.

Step 3: Compile the program.

Step 3: Run the program and observe the output.

### 2.3.6 Apply calculations by using operators and expressions

Find the answer, answer if m=5, n=15, p=12

a) answer = n % m + p – p / m

= 15 % 5 + 12 -12 / 5
= 0 + 12 -12 /5
= 0 + 12 – 2
=10

b) answer = (m + n) % n – p

= (5+15) % 15 - 12
= 20 % 15 -12
= 5 -12
= -7

c) answer = n / m + n % p * m

= 15/5 + 15 % 12 * 5
= 3 + 15 % 12 *5
= 3 +3*5
= 3 + 15
= 18

### 2.3.7 Implement mathematical calculations in simple C program

**Step 1**: Open C editor and types:

```
#include <stdio.h>
main()
{
int mark1, mark2;
float total;
printf("\n Enter the mark1 : ");
scanf("%d",&mark1);
printf("\n Enter the mark2 : ");
scanf("%d",&mark2);
total=mark1+mark2;
printf("\n %d  + %d = %.2f",mark1,mark2,total);
return 0;
}
```

Step 2: Save the program.

Step 3: Compile the program.

Step 3: Run the program and observe the output.

### 2.3.8 Implement mathematical calculations using the function in the main function

**Step 1**: Open C editor and types:

```
#include <stdio.h>
int addition(int var1, int var2)
{
int sum;
sum = var1+var2;
return sum;
}
int main()
{
int num1, num2;
printf("Enter number 1: ");
scanf("%d",&num1);
printf("Enter number 2: ");
scanf("%d",&num2);
int total = addition(num1, num2);
printf ("Output: %d", total);
return 0;
}
```

Step 2: Save the program.

Step 3: Compile the program.

Step 3: Run the program and observe the output.

# Tutorial

a) List THREE (3) types of operator used in C programming

b) Recognize the listed keyword in the table B1 as reserved word OR not reserves word C Programming, then complete below:

| Keyword | Invalid reserved word | valid reserved word |
|---|---|---|
| int | | / |
| name | / | |
| if | | / |
| structured | / | |
| looping | / | |

c) Observe every line and list the errors found on the following program below:

```
/?Compute volume of a Cone program?

#<include>stdio.h
#default PI 3.142
main()
{
double rad, Volume, height;

printf("~~~~~~~~CONE VOLUME CALCULATOR~~~~~~~~~~~~~~~~~~");
printf("Enter a radius value of cone");
scanf("%f", &radius);
printf("Enter a height  value of cone");
scanf("%f", %height);

          Volume = (height)* pi * radius ^2;

printf(" The volume of cone is %d", Volume);
       }
Return $;
}
```

d) Identify the data types for the following variables:

i. a = - 8.2

ii. b = 360

iii. c = ‘?’

e) Rewrite the following manes to be valid identifier

i. 4th value

ii. Order-no

iii. Error flag

iv. Break

v. @google.com

f) Solve the pseudo code below by writing C Language.

```
Start
Read user height as float
Read user weight as float
Calculate BMI = $\frac{\text{weight (kg)}}{\text{height}^2(\text{m}^2)}$
Display BMI
End
```

g) List THREE (3) data types used in C Programming.

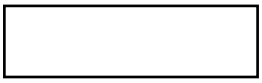
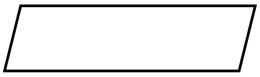
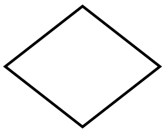
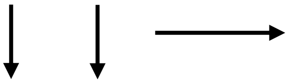
h) Identify the suitable variable declaration for each variable in table below based on the correct format in C programming.

| Variable | Variable Declaration |
|---|---|
| Alphabet = ‘A’ | |
| Num = 2 | |
| Price of book | |
| Number of student | |
| Average of three number | |

i) Complete the program below:

```
#________________<stdio.h>
int main()
___
char name[20];
int staff_no;
printf(“Enter your name:”);
scanf(“%s”, _______);
printf(“Enter your staff number:”);
____ (“____”, &staff_no);
printf(“Mr/Ms %c, your staff number is _______”, name, staff_no);
_____________;
}
```

j) Describe the function of each basic symbols in the table below that commonly used in flowchart:

| Symbol | Function |
|---|---|
| i.  | |
| ii.  | |
| iii.  | |
| iv. | |
| v.  | |

k) Based on program below, identify FIVE ( 5 ) errors.

```
#include<stdo.h>
main();
{
int x,y;
            x = 10;
            y = x++;
            printf("%f", x);
            printf("%d", &y);
}
```

l) Pak Ali, a durian seller's wants to use a computer in his selling. He is using the scale in pounds, but the selling is done in kg.

Use a C language to write a program to help him change each weight of durian from pound to kg.

(1kg = 2.205 pound @ 1 pound = 0.404kg)

# Practical Activities

**Program 1**

```
#include<stdio.h>
main()
{
char name[20];
int staff_no;
printf("Enter your name:");
scanf("%s",&name);
printf("Enter your Staff No:");
scanf("%d",&staff_no);
printf("Mr/Ms %s, your Staff No is %d",name,staff_no);
return 0;
}
```

**Program 2**

```
#include<stdio.h>

main()
{
int num1, num2;
float num3,num4,num5;
char cha, name[20];
printf("Input from user\n");
printf("Enter a Character\n");
scanf("%c",&cha);
printf("Enter a your name\n");
scanf("%s",&name);
printf("Enter 2 integer numbers number\n");
scanf("%d %d",&num1,&num2);
printf("\nEnter 3 floating point number\n");
scanf("%f %f %f",&num3,&num4,&num5);
printf("\n Output to user");
printf("\nThe two numbers You have entered are %d and %d", num1, num2);
printf("\nThe float or fraction that you have entered is %.1f, %.2f and %.3f.",
num3,num4,num5);
printf("\nThe character that you have entered is %c", cha);
printf("\nThe character that you have entered is %s", name);

return 0;
}
```

**Program 3**

```
#include <stdio.h>
main()
{
  int A,B,C,jumlah1,jumlah2,jumlah3;
  float purata;
  printf("input statement\n");
  printf ( "sila masukan nilai A=");
  scanf("%d",&A);
  printf ( "sila masukan nilai B=");
  scanf("%d",&B);
  printf ( "sila masukan nilai C=");
  scanf("%d",&C);
  printf("\noutput statement");
  jumlah1=A+B+C;
  printf("\n jumlah %d + %d +%d = %d ",A,B,C,jumlah1);
  jumlah2=A-B-C;
  printf("\n jumlah %d - %d -%d = %d ",A,B,C,jumlah2);
  jumlah3=A*B*C;
  printf("\n jumlah %d X %d X %d = %d ",A,B,C,jumlah3);
  purata=(A+B+C)/3;
  printf("\n purata (%d + %d + %d)/3 = %d ",A,B,C,purata);
  return 0;
}
```

**Program 4**

```
  #include <stdio.h>
   int main()
   {
      double firstNumber, secondNumber, product1,product2;
      printf("Enter two numbers: ");
      scanf("%lf %lf", &firstNumber, &secondNumber);
      product1 = firstNumber * secondNumber;
      printf(" \n Product = %.2lf", product1);

      product2 = firstNumber / secondNumber;
      printf("\n Product = %.2lf", product2);
      return 0;
   }
```

**Program 5**

```
#include <stdio.h>
int main()
{
   float r1,r2,r3,rt;
```

```
    printf("Enter the value of resistor 1:");
    scanf("%f",&r1);
    printf("Enter the value of resistor 2:");
    scanf("%f",&r2);
    printf("Enter the value of resistor 3:");
    scanf("%f",&r3);
    rt=r1+r2+r3;
    printf("Total resistance in series:%.2f\n",rt);
    return 0;
}
```

**Program 6**

```
#include <stdio.h>
int main()
{
    float v,i,r;
    printf("Enter the value of current flow through the circuit:");
    scanf("%f",&i);
    printf("Enter the value of total resistance in the circuit:");
    scanf("%f",&r);
    v=i*r;
    printf("The voltage value of the circuit:%.2f\n",v);
    return 0;
}
```

**Program 7**

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

        int p,t;
        float r,Simple_Interest,amount,Compound_Interest;
        printf("Please enter principal,time and rate of interest\n");
        scanf("%d%d%f",&p,&t,&r);
        Simple_Interest=p*t*r/100;
        printf("\nSimple interest = %.2f",Simple_Interest);
        amount=p*pow((1 +r/100),t);
        Compound_Interest=amount-p;
        printf("\nCompound interest = %.3f",Compound_Interest);

return 0;
}
```

# 3.0 Selection Statements

**INTRODUCTION**

Selection statements to make programs more user-interactive and flexible is to make the program able to handle more than one case. By using of **selection statements** allow a program to test several *conditions*, and execute instructions based on which condition is true. That is why selection statements are also referred to as **conditional** statements.

## Note

### 3.1 Remember selection statements

When it comes to computer programs, there is always a trade-off between the program's flexibility and the ease of using the program. Generally, the simpler a program is, the less flexible (and less powerful) the program will be. Programs that are very easy to use are usually not very flexible and often only work for a narrow range of problems. Having selection statements in a program make the program more flexible.

#### 3.1.1 Define control statements

Selection statements are used to evaluate expression and direct the execution of the program, depending on the result of the evaluation.

#### 3.1.2 List types of control statements

The selection statements available in C are:

a. **if**
b. **if-else**
c. **nested if else**
d. **switch**

#### 3.1.3 Define IF, IF-ELSE, NESTED IF ELSE and SWITCH statements

a. Simple ***if*** statement is used to execute a set of statements when the condition is satisfied.

b. The ***if-else*** statement is used to execute set of statements based on the condition.

c. The ***nested if else*** statement is present inside the body of another “if” or “else” statement.

d. The ***switch*** statement is used when you choose from a number of choices.

### 3.1.4 Describe selection statements

A selection statement selects among a set of statements depending on the value of a controlling expression. The selection statements are the if statement and the switch statement, which are discussed in the following sections.

### 3.1.5 Describe structure of simple IF, IF-ELSE, Nested IF-ELSE and SWITCH statements

a. Structure of simple **if:**

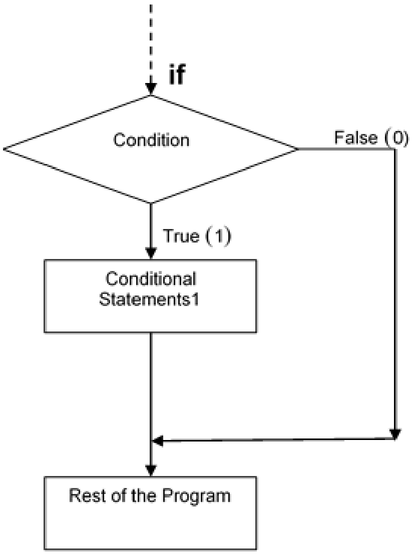


**Figure 3.1.5a: Flowchart for *if* Statement**

First, the if statement will evaluate the condition. If the condition evaluates to true, the conditional statements are executed. If the condition evaluates to false, the conditional statements are not executed. The execution of conditional statement depends on the result of the condition.

The syntax for the if statement is:

```
if (<Conditional expression>)
{
        <Conditional statements>;              Body
}
```

As shown in the syntax, the keyword if is followed by a conditional expression. The condition is enclosed within a pair of brackets ( ). A condition is an expression that will evaluate to a Boolean value. There are two Boolean values, True(1) or False(0). The if statement is followed by a pair of curly brackets { } within which the conditional statements are written.

b. structure of **if-else:**

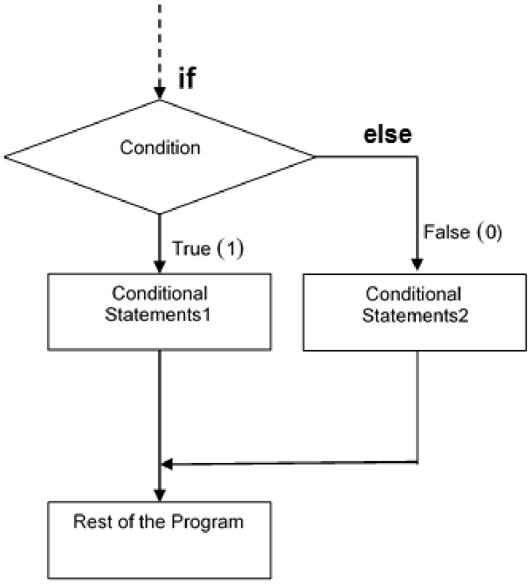


**Figure 3.1.5b: Flowchart for *if-else* Statement**

The **if-else** statement will first check the condition. When the condition is fulfilled, conditional statements1 will be executed. When the condition not fulfilled, conditional statements2 will be executed.

The syntax for the **if-else** statement is:

```
if (<Conditional expression>)
{
        <Conditional statements1>;
}
else
{
        <Conditional statements2>;
}
```

True- Condition Fulfilled

False - Condition Not Fulfilled

Observe the difference in syntax between a simple **if** statement and **if-else** statement. In the simple **if** statement, set of statements are executed when the condition is fulfilled. In **if-else** statement, a set of statements, which follows **if** is executed when the condition is fulfilled, and a set of statements which follows **else** is executed when the condition is not fulfilled.

c. structure of **nested if-else:**

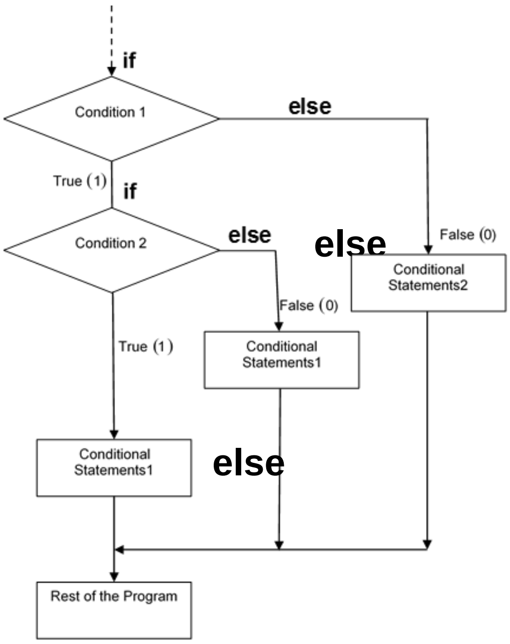


**Figure 3.1.5.c: Flowchart for nested *if-else* Statement**

The nested **if-else** statement will first check the condition 1. When the condition is fulfilled, the nested statement will check again the condition 2. When the condition 2 is fulfilled conditional statements1 will be executed. If not statement 2 will be executed. If condition 1 not fulfilled conditional statements 3 will be executed.

The syntax for the **if-else** statement is:

```
if (<Conditional expression 1>)
{
if (<Conditional expression 2>)

        <Conditional statements1>;
else
        <Conditional statements2>;
}
else
{
        <Conditional statements3>;
}
```

True- Condition 1 Fulfilled

True- Condition 2 Fulfilled

False - Condition Not Fulfilled

False - Condition Not Fulfilled

d. structure of **switch:**

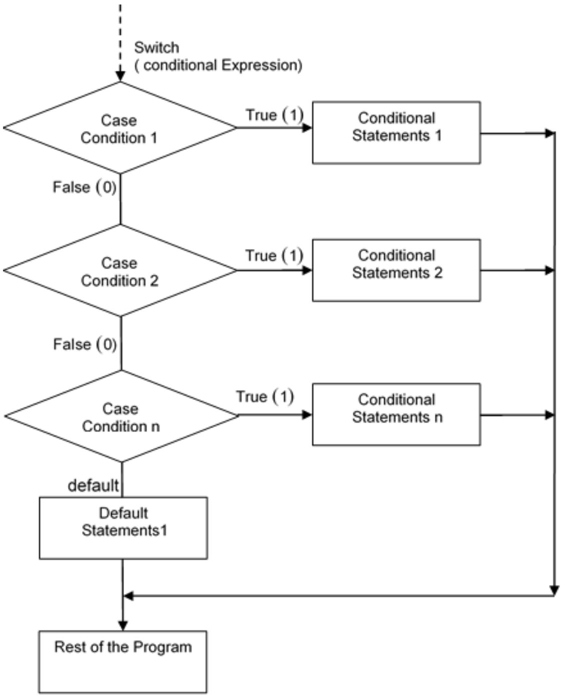


**Figure 3.1.5.d : Flowchart for switch Statement**

The syntax for switch statement is:

```
switch (<expression>)
{
        case <condition 1>:
        <Statements 1>;
        break;
        case <condition 2>:
        <Statements 2>;
        break;
        ...
        case <condition n>:
        <Statements n>;
        break;
        default:
        <default Statements >;
}
```

The expression following the keyword **switch** can either be an integer expression or a character expression. An integer or a character constant follows the keyword **case.** Based on the value returned by the expression, the respective case is executed. The expression is first evaluated and this value is then matched against the constants in each case. When a match is found, the statements following that particular case statement are executed. If no match is found, the statements following the default case are executed.

The keyword **break** must be included in each case. A **break** statement will enable you to skip all the cases following the current case and transfer the control outside the **switch** statement.

## 3.2 Understand and apply selection statements

### 3.2.1 Differentiate programs simple IF, IF-ELSE, Nested IF-ELSE and SWITCH statements

a. **Program IF**

```
#include <stdio.h>
void main()
{
  int day;
  printf("Number of days in December : ");
   scanf("%d",&day);   // Accepts the input from the user and stores it in the variable days.
   if (day == 31)   // This expression will check whether the value of days is equal to 31.
  {
    printf(" You are correct. "); // This statement will be executed when days is equal to 31.
   }
  printf(" \n You are out of if branch. ");
}
```

b. **Program IF-ELSE**

SAMPLE if-else statement. This program accepts two integers from the user and finds the greater of the two.

```
/* Program to find the greater of the two numbers */
#include <stdio.h>
void main()
{
      int x,y;
      printf("Enter X and Y values : ");
      scanf("%d %d",&x,&y);
      if (x > y) // This expression will check whether the value of x is greater than the value of y.
            printf(" X is greater than Y"); // This statement will be executed when x is greater than y.
      else
            printf(" Y is greater than X"); // This statement will be executed when x is lesser than y.

}
```

c. Program nested IF-ELSE

SAMPLE nested  if-else statement. This program accepts two integers from the user and finds the greater of the two.

```
#include <stdio.h>
main()
{
 int x,y;
printf("Enter X and Y values : ");
scanf("%d %d",&x,&y);
if (x > y) // This expression will check whether the value of x is greater than the value of y.
{
printf(" X is greater than Y"); // This statement will be executed when x is greater than y.
}
else    // If x not  greater than the value of y.
{
      if (x==y) // This expression will check whether the value of x is equal to value of y.
      {
      printf(" Y is equal to X"); // This statement will be executed when x is equal to y.
      }
      else
      {
      printf(" Y is greater than X"); // This statement will be executed when y is greater than x.
      }
}
}
```

d. Program Switch

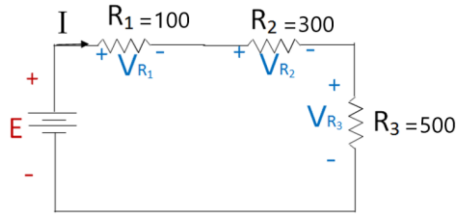
```
#include <stdio.h>
void main()
{
  int x;
  printf("\n Enter the value for X (1,2 or 3) : ");
  scanf("%d",&x);
  switch (x) // The value of x will be checked against each case and the corresponding statements will
                    be executed.
  {
     case 1: // This case is executed when the value of x is 1.
     printf(" You have entered One ");
     break;
     case 2: // This case is executed when the value of x is 2.
     printf(" You have entered Two ");
     break;
     case 3: // This case is executed when the value of x is 2
     printf(" You have entered Three ");
     break;
     default: // This case is executed when the value of x is not equal to 1, 2 or 3.
     printf(" Wrong Entry ");
  }
}
```

### 3.3 Apply selection statements

3.3.1 Construct programs simple IF, IF-ELSE, Nested IF-ELSE and SWITCH statements

a. Sample program IF

To calculate series circuits.



Series

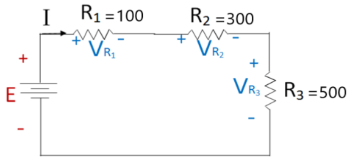
illustrate the working of IF statements to calculate the total resistance for series circuit and get the output

Program:

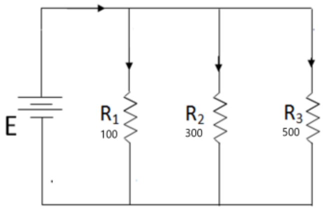
```
#include <stdio.h>
 main()
{
int selection;
float series, parallel;
float R1=100,R2=300, R3=500;
printf("1: Series Circuit\n");
printf("Please Select your choice \n ");
scanf("%d", &selection);
if (selection==1)
{
  series = R1+R2+R3;
  printf(" Total resistance = %.2f Ohm\n",series);
}
return 0;
}
```

b. Sample program IF-ELSE

To calculate series and parallel circuits.



Series



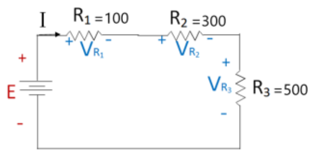
Parallel

Illustrate the working of IF-ELSE statements to calculate the total resistance for series and parallel circuit and get the output.

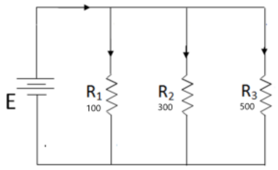
```
#include <stdio.h>
 main()
{
int selection;
float series, parallel;
float R1=100,R2=300, R3=500;
printf("1: Series Circuit\n");
printf("2. Parallel Circuit \n");
printf("Please Select your choice \n ");
scanf("%d", &selection);
if (selection==1)
{
  series = R1+R2+R3;
  printf(" Total resistance = %.2f Ohm\n",series);
}
else if (selection==2)
{
   parallel = 1/(1/R1 +1/R2+1/R3) ;
  printf(" Total resistance = %.6f Ohm\n",parallel);
   }
else
{
  printf(" wrong choice \n ");
          }
return 0;
}
```

c. Sample program Nested IF-ELSE

To calculate series and parallel circuits.



Series

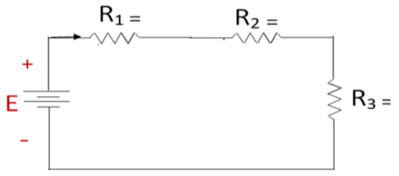


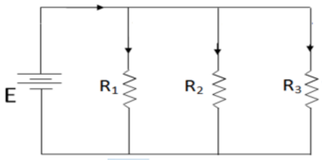
Parallel

Illustrate the working of Nested IF-ELSE statements to calculate the total resistance for series and parallel circuit and get the output.

```
#include <stdio.h>
 main()
{
int selection1,selection2;
float series, parallel;
float R1=100,R2=300, R3=500;
printf("1:You want to do the calculation\n");
scanf("%d", &selection1);
if (selection1==1)
{
     printf("1: Series Circuit\n");
     printf("2. Parallel Circuit \n");
     printf("Please Select your choice \n ");
     scanf("%d", &selection2);
     if (selection2==1)
     {
     series = R1+R2+R3;
     printf(" Total resistance = %.2f Ohm\n",series);
     }
     else
     {
     parallel = 1/(1/R1 +1/R2+1/R3) ;
     printf(" Total resistance = %.6f Ohm\n",parallel);
     }
}
else
{
  printf(" wrong choice \n ");
        }
return 0;
}
```

d. Sample program SWITCH statements





$R_T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$

$R_T = R_1 + R_2 + R_3$

Series

Parallel

Illustrate the working of SWITCH statements to calculate the total resistance for series and parallel circuit and get the output.

```
#include <stdio.h>
 main()
{
int selection;
float series, parallel,C1,C2,C3;
printf("******************************************");
printf("\nEnter the value of C1= ");
scanf("%f",&C1);
printf("\nEnter the value of C2= ");
scanf("%f",&C2);
printf("\nEnter the value of C2= ");
scanf("%f",&C3);
printf("******************************************");
printf("\n1: Series Circuit");
printf("\n2. Parallel Circuit ");
printf("\nPlease Select your choice  ");
printf("******************************************");
scanf("%d",&selection);
switch(selection)
{
case 1:
     series = 1/(1/C1 +1/C2+1/C3) ;
     printf(" Total series capacitance = %.6f Farad\n",series);
     break;
case 2:
     parallel = C1+C2+C3;
     printf(" Total parallel capacitance = %.2f Farad\n",parallel);
     break;
default:
  printf(" wrong choice \n ");
}
return 0;
}
```

# Tutorial

a) Explain switch statements

b) Write a program that request two integer number as input from user to test both input is non zero or not. Use if-else statement and AND logical operators.

c) Draw flowchart for if-else statement and switch-case statement.

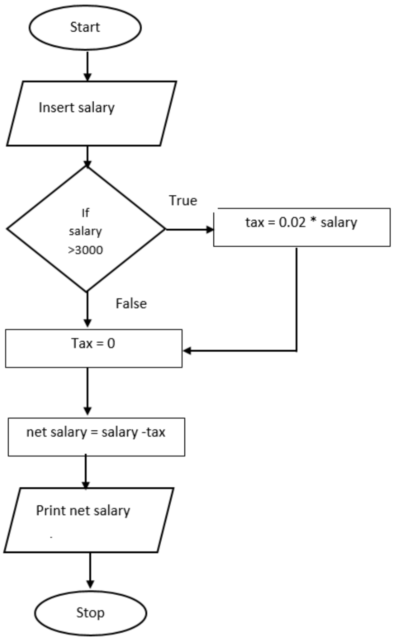
d) Carry out a program that ask user to enter the total of 100 marks and then display the letter grade and wish based on diagram below:

| Mark | Grade | Wish |
|---|---|---|
| 85 and above | A | Excellent |
| 75 - 84 | B | Best |
| 65 - 74 | C | Good |
| 55 - 64 | D | Satisfactory |
| 45 - 54 | E | Bad |
| Below 44 | F | Fail |

e) List THREE (3) types of selection statements

f) Discuss TWO(2) differences between if-else statement and switch statement.

g) By referring the flowchart, write a C program to calculate a simple net salary system.



h) Produce C language to display the examination result status and grade point when a mark is entered by user using if statement. The grading classification is shown in the table below:

| Mark | Grade | Status |
|---|---|---|
| 75-100 | A | PASS |
| 60-74 | B | PASS |
| 47-59 | C | PASS |
| 40-46 | D | FAIL |
| 20-39 | E | FAIL |
| 0-19 | F | FAIL |

i) Draw a flowchart to represent nested if-else statement.

j) Explain the meaning of the statement statement below :

i. if

ii. if-else

iii. nested if-else

iv. Switch

v. Break

k) Use a C language to write a program for the following statement :

> Request an input price for one item. Next, quantity of such item should also be entered. Calculate the total price of the items using the formula : (price * quantity = total). If the to tal price exceeds RM100, the rebate of 15% will be offered, but no deduction is granted if otherwise.

l) Use C language to write a program that accept values of 2 numbers from the user, compare their size and display the result on the screen. The output as follows:

| Option 1 | Option 2 |
|---|---|
| Enter value for first number : 4<br>Enter value for secondnumber : 3<br><br>First number is larger number than second number | Enter value for first number : 6<br>Enter value for secondnumber : 8<br><br>Second number is larger number than first number |

m) Use C language to write a program to calculate the total income of salesperson if the commission is given based on the table below:

| Total Sale | Commission |
|---|---|
| <RM500 | 5% |
| RM500 – RM999 | 7% |
| RM1000 – RM1499 | 9% |
| >=RM1500 | 12% |

n) Rewrite switch statement below using if-else-if statement.

```
switch (color) {
            case  ‘R’ : printf(\nRed”);break;
            case  ‘R’ : printf(“\nRed”);break;
            case  ‘R’ : printf(“\nRed”);break;
default : printf(“Wrong colou code”);
}
```

o) Use a C language to write a program for the statement below :

> An input numbere is required from the user to test either the number is multiple of three or not using if-else statement.

## Practical Activities

**Program 1:**

```
#include<stdio.h>
main()
{
   int number;
   printf("Please enter a number:\n");
   scanf("%d",&number);
   if(number < 500)
      printf("Number is less than 500!\n");
   return 0;
}
```

**Program 2:**

```
#include<stdio.h>
main()
{
   int number;
   printf("Please enter a number:\n");
   scanf("%d",&number);
   if(number < 500)
```

```
        printf("Number is less than 500!\n");
    else
        printf("Number is greater than 500!\n");
    return 0;
}
```

**Program 3:**

```
#include<stdio.h>
main()
{
    int number;
    printf("Please enter a number:\n");
    scanf("%d",&number);
    if(number < 500)
        printf("Number is less than 500!\n");
    else if(number == 500)
        printf("Number is 500!\n");
    else
        printf("Number is greater than 500!\n");
    return 0;
}
```

**Program 4:**

```
#include <stdio.h>
 main()
{
int selection;
float series, parallel,R1,R2,R3;
printf("**********************************************");
printf("\nEnter the value of R1= ");
scanf("%f",&R1);
printf("\nEnter the value of R1= ");
scanf("%f",&R2);
printf("\nEnter the value of R1= ");
scanf("%f",&R3);
printf("**********************************************\n");
printf("1: Series Circuit\n");
printf("2. Parallel Circuit \n");
printf("Please Select your choice \n ");
scanf("%d", &selection);
if (selection==1)
{
  series = R1+R2+R3;
  printf("series = R1+R2+R3;");
  printf("\n Total resistance = %.2f Ohm\n",series);
}
else if (selection==2)
{
   parallel = 1/(1/R1 +1/R2+1/R3) ;
   printf("parallel = 1/(1/R1 +1/R2+1/R3) ;");
  printf(" \nTotal resistance = %.6f Ohm\n",parallel);
   }
else
{
  printf(" wrong choice \n ");
        }
return 0;
}
```

**Program 5:**

```
#include <stdio.h>
 main()
{
int selection;
float series, parallel,R1,R2,R3;
printf("*******************************************");
printf("\nEnter the value of R1= ");
scanf("%f",&R1);
printf("\nEnter the value of R1= ");
scanf("%f",&R2);
printf("\nEnter the value of R1= ");
scanf("%f",&R3);
printf("*******************************************");
printf("\n1: Series Circuit");
printf("\n2. Parallel Circuit ");
printf("\nPlease Select your choice  ");
printf("\n*******************************************\n");
scanf("%d",&selection);
switch(selection)
{
case 1:
            series = R1+R2+R3;
            printf("series = R1+R2+R3;");
            printf("\nTotal resistance = %.2f Ohm\n",series);
            break;
case 2:
            parallel = 1/(1/R1 +1/R2+1/R3) ;
            printf("parallel = 1/(1/R1 +1/R2+1/R3) ;");
            printf("\nTotal resistance = %.6f Ohm\n",parallel);
            break;
default:
  printf(" wrong choice \n ");
}
return 0;
}
```

**Program 6:**

```
#include <stdio.h>
 main()
{
int selection;
float series, parallel,R1,R2,R3,E,Rtotal,I,VR3,IR1,VR1;
printf("******************************************");
printf("\nEnter the value of R1= ");
scanf("%f",&R1);
printf("\nEnter the value of R2= ");
scanf("%f",&R2);
printf("\nEnter the value of R2= ");
scanf("%f",&R3);
printf("\nEnter the value of E= ");
scanf("%f",&E);
printf("\n******************************************");
printf("\n1: Total Resistance (Rtotal)");
printf("\n2. Current I (I) ");
printf("\n3: Voltage drop across resistor R3 (VR3)");
printf("\n4. Current through esistor R1 (IR1)");
printf("\nPlease Select your choice  ");
printf("\n******************************************\n");
scanf("%d",&selection);


if (selection==1)
{

        Rtotal = 1/(1/R1 +1/R2+1/R3) ;
        printf(" Rtotal = 1/(1/R1 +1/R2+1/R3)  \n");
        printf(" Total Resistance (RTotal) = %.6f Ohm\n",Rtotal);
}
else if (selection==2)
{
        Rtotal = 1/(1/R1 +1/R2+1/R3) ;
        I=E/Rtotal;
        printf(" Rtotal = 1/(1/R1 +1/R2+1/R3) & I=E/Rtotal\n");
        printf(" Current I = %.4f Ampere\n",I);
}
else if (selection==3)
{
        VR3 = E ;
        printf(" VR3 = E ; \n");
        printf(" Voltage drop across Resistor VR3 = %.4f Volt\n",VR3);
}
else if (selection==4)
{
        VR1=E;
```

```
            IR1 = VR1/R1 ;
            printf("            VR1=E & IR1 = VR1/R1  \n  ");
            printf(" Current through Resistor R1 = %.4f Ampere\n",IR1);
}
else
{
  printf(" wrong choice \n ");
}
return 0;
}
```

# 4.0 LOOPING

## 4.0 INTRODUCTION

Selection statements are used when you need to perform an action once only, based on the condition specified. You may, however need to repeat a particular action in some situations again and again. Refer Figure 4.5.1. Consider the action of cycling to reach a destination. You will repeat the process of pedalling till you reach the destination. This can be compared to the concept of looping in programming languages. The mechanism of executing a set of statements repeatedly as long as the specified condition is satisfied is called *looping*. The statements that are executed repeatedly are termed as *looping statements*.

# Note

## 4.1 Looping statements

### 4.1.1 Define Looping statement

Looping statements are used to execute a set of statements repeatedly based on the condition.

### 4.1.2 Types of Looping Statement

There are three types of looping statements available in C language. They are:

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

### 4.1.3 Define FOR, WHILE, DO-WHILE loop statements

**FOR:** The for loop is a looping statement that enables you to repeat a set of instructions based on the condition. It is used when the number of loops is known before the first loop.

**WHILE:** The while loop is a looping statement that enables you to repeat a set of instructions based on a condition. It is used when the number of loops is not known before the first loop

**DO-WHILE:** The do-while loop is a looping statement that enables you to repeat a set of instructions based on the condition. In a do-while loop, the body of the loop of executed at least once before the condition is checked.

4.1.4 **BREAK, CONTINUE** and **GOTO** statements

**The *break* Statement**

**Definition:** The **break** statement will transfer the control to the statement outside the loop.

It is often necessary to transfer control out of a loop without getting back to the condition check. The **break** statement is used for this purpose. During execution, when the keyword **break** is encountered within a loop, the control is immediately transferred to the first statement outside the loop. A **break** statement is usually used along with an **if** statement.

**The *continue* Statement**

**Definition:** The ***continue*** statement will transfer the control to the beginning of the loop.

You can use the **continue** statement when you want the control to return to the beginning of the loop without executing the rest of the statements in the loop. When the keyword continue is encountered during the execution of the loop, the control is passed to the beginning of the loop bypassing the rest of the statements in the loop.

**The *goto* Statement**

**Definition:** The **goto** statement will transfer the control to the specified label.

During execution, **goto** statement is used to transfer the control to the specified label within the program. A *label* is a name that refers to a particular line in the program. A program can have any number of labels, but each label name must be unique. A label must be followed by a colon (:) before the C statement.

### 4.1.5 Describe structure of FOR, nested FOR, WHILE, DO-WHILE loop statements

**FOR Loop**

In a **for** loop, the first step is to assign the value of the initial counter. Next the condition is checked in the test counter. Based on the condition, body of the loop is executed. The value of the counter is then incremented as specified in the increment counter. The condition is checked again and the body of the loop is executed.

The body of the loop will be executed as long as the test condition is satisfied. When the condition fails, the control is passed to the first statement outside the loop.

The syntax for the **for** loop is:

```
for( <initial counter> ; <test counter> ; <increment counter> )
{
        C statement 1;
        C statement 2;
        ...
        ...
        ...
        C statement n;
}
```

Header Section

Body of the Loop

The initial, test and increment counters are specified in the header section of a **for** loop, each separated by a semicolon (;).

Consider the example given in Code Segment 4.1.5:

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

Initialise the counter value *i* to *1*.

Checks the condition whether the value of *i* is less than 5

Increment the value of *i* by *1*.

This statement will be executed when the value of *i* is less than or equal to *5*.

**Code Segment 4.1.5**

**Output**

```
1
2
3
4
5
```

Figure 4.5.4 shows the flowchart for Code Segment 4.5.2.

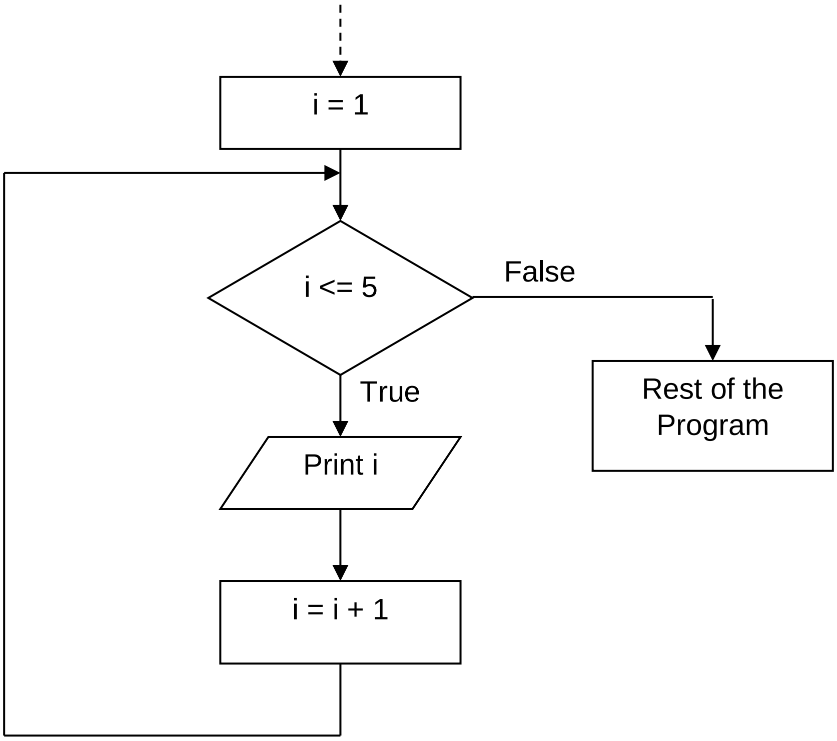


**Figure 4.1.5 : Flowchart**

In Code Segment 4.1.5, *i=1* is the initial counter, *i<=5* is the condition and *i=i+1* is the increment counter. The following steps explain the execution of Code Segment 4.1.5:

1. When the control enters the loop for the first time, the value of *i* is initialised to *1* by the initial counter.
2. Next, the condition is checked. If the value of *i* is less than or equal to *5*, the control is passed to the body of the loop.
3. The body of the loop is executed. Here, the value of *i* is displayed on the screen.
4. When the control reaches the closing brackets of the loop, the value of *i* is incremented by *1*.
5. Step 2, 3 and 4 are repeated as long as the test condition is satisfied.
6. When the test condition fails, the control comes out of the loop and the rest of the program statements are executed.

Table 4.5.1 gives the details of the dry run for Code Segment 4.1.5:

Dry run is the process of executing a program by hand. This is done by writing values of variables and other run-time data on paper, in order to check its operation.

| Value of i | Condition | Result | Output |
|---|---|---|---|
| 1 | 1 <= 5 | True | 1 |
| 2 | 2 <=5 | True | 2 |
| 3 | 3 <= 5 | True | 3 |
| 4 | 4 <= 5 | True | 4 |
| 5 | 5 <= 5 | True | 5 |
| 6 | 6 <= 5 | False | - |

**Table 4.1.5: Dry Run**

**WHILE loop**

The **while** loop is similar to the **for** loop and is used to perform certain tasks repeatedly. Use the **while** loop when the number of loops is not known before the first loop. This will be known only during the run time, based on the input given by the user. Figure 4.1.6 represents the flowchart for the **while** loop.

**Flowchart**

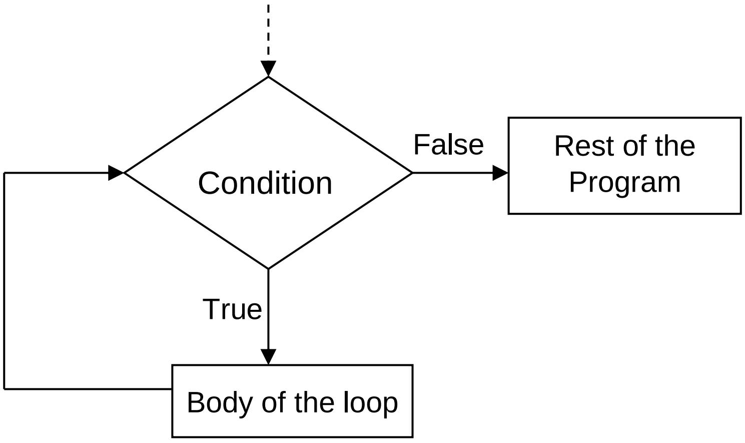


**Figure 4.1.6: Flowchart for *while* Loop**

In a **while** loop, the condition is initially checked. Based on this the statements below the **while** statement are executed. The general syntax for a **while** loop is:

Syntax

```
<Initialise loop counter>;
while( <conditional expression> )
{
        Statement1;
        Statement2;
        ...
        ...
        ...
        <Increment loop counter>;
}
```

Body of the Loop

As shown in the syntax, the keyword **while** is followed by a conditional expression. The condition is enclosed within a pair of brackets **( )**. The **while** statement is followed by a pair of curly brackets **{ }** within which the statements, that are to be executed, are written. You will notice that the counter value is initialised before the **while** statement. This counter value is incremented within the body of the loop. Do not terminate the **while** statement with a semicolon(;).

Code Segment 4.1.6 explains how a while loop can be used to print the numbers from 1 to 5:

```
i = 1;
while( i <= 5)
{
        printf("\n%d",i);
        i++;
}
```

The value of i s initialised to 1.

Condition is checked.

This statement is executed when the value of i is less than 6.

The value of i is incremented

**Code Segment 4.1.6**

**Output:**

```
1
2
3
4
5
```

The following steps explain the execution of Code Segment 4.5.5:

1. The value of the variable *i* is initialised to *1*.
2. The condition is checked. If the value of *i* is less than or equal to *5*, the control moves into the body of the **while** loop.
3. The value is printed on the screen.
4. The value of *i* is incremented.
5. Step 2, 3 and 4 are repeated as long as the condition is satisfied.
6. When the condition is not satisfied, the control moves out of the loop.

**DO-WHILE loop**

The **do-while** loop is a variation of the **while** loop, with the only difference being the location where the condition is checked, as shown in Figure 4.1.7.

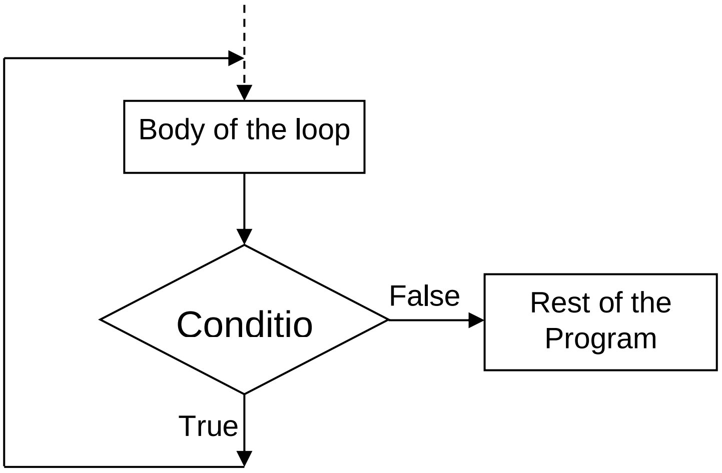
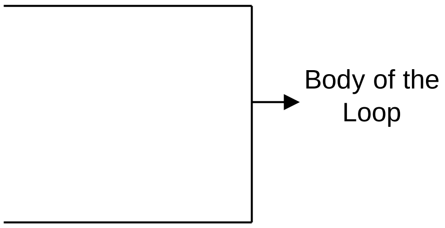


**Figure 4.1.7: Flowchart for do-while Loop**

In a **while** loop, first the condition is checked, followed by and then the body of the loop is executed. In a **do-while** loop, however, the body of the loop is first executed and the condition is checked subsequently. In a **do-while** loop, the loop is executed at least once even if the condition is not satisfied. In the flowchart, notice that the condition is checked after the execution of the statements within the body of the loop.

**Syntax**

```
do
{
    Statement1;
         Statement2;
    ...
              ...
              ...
}while( <conditional expression> );
```



### 4.2 Understand Looping statements.

### 4.2.1 Differentiate FOR, nested FOR, WHILE, DO-WHILE loop statements

**Difference between the looping structures in C**

Table 4.2.1 represents the difference between the various looping structures.

| The for Loop | The while Loop | The do-while Loop |
|---|---|---|
| Used when the number of loops is known before the first loop. | Used when the number of loops is not known before the first loop. | Used when the number of loops is not known before the first loop. |
| First the condition is checked and then the body of the loop is executed. | First the condition is checked and then the body of the loop is executed. | Body of the loop is first executed and then the condition is checked. |
| Loop may not be executed at all. | Loop may not be executed at all. | Loop will get executed at least once. |

**Table 4.2.1: Difference Between Various Looping Structure**

### 4.2.2 Explain working of BREAK statement

It is often necessary to transfer control out of a loop without getting back to the condition check. The **break** statement is used for this purpose. During execution, when the keyword **break** is encountered within a loop, the control is immediately transferred to the first statement outside the loop. A **break** statement is usually used along with **if** statement.

Program to illustrate the use of a break statement.

```
/* Program to illustrate the use of break statement */
#include <stdio.h>
void main()
{
 int i;
 for( i=1; i<=10; i++)
 {
 scanf("\n %d",&i);
if (i == 0)
  break; // This statement will exit from the for loop when the value of i is equal to 0.
 }
   printf("\n The control is out of the loop");
}
```

**Code Sample 4.2.2**

**Output**

```
4
3
6
8
0
The control is out of the loop
```

Flowchart for the execution of Code Sample 4.2.2 is given in the Figure 4.2.2.

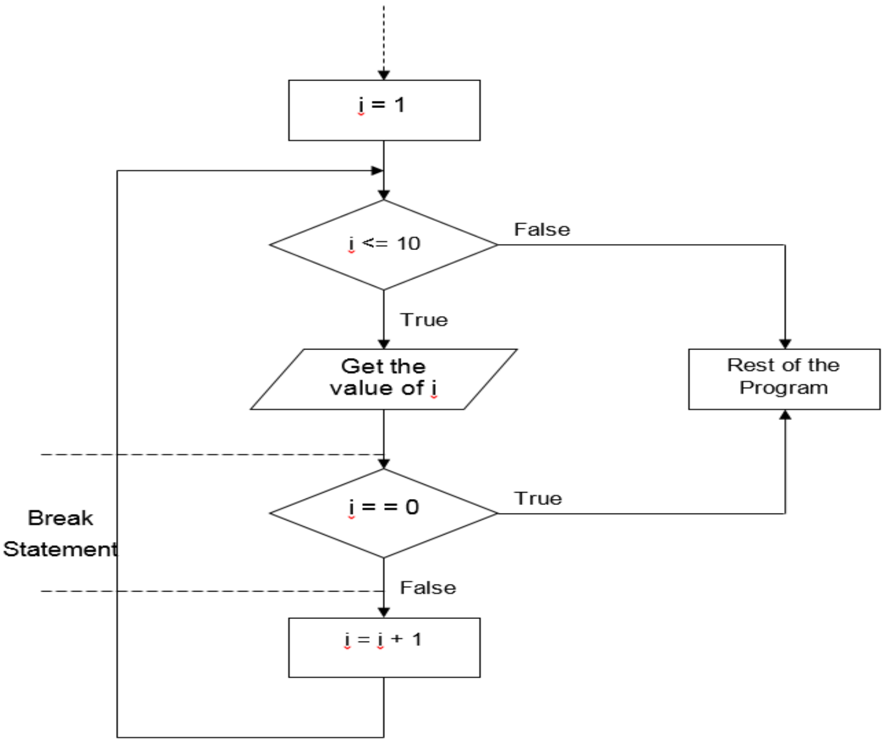


**Figure 4.2.2: Flowchart**

Consider the program given in Code Sample 4.5.6. In this example, when the user enters *0*, the control is transferred to the statement outside the **for** loop.

## 4.3 Apply Looping statements

### 4.3.1 Construct program(s) that use FOR, nested FOR, WHILE, DO-WHILE loop statements

a. Program use FOR : to print numbers from 1 to 5

```
#include <stdio.h>
main()
{
    int i;
    for( i=1; i<=5; i=i+1)
    {
        printf("\n  %d",i);
    }
    printf("\n\n You are out of the for loop");
}
```

```
OUTPUT:
1
2
3
4
5
You are out of the for loop
```

b. Program use FOR : to print numbers from 10 to 1

```
#include <stdio.h>
main()
{
    int i;
    for( i=10; i>=1; i=i-2)
    {
        printf("\n  %d",i);
    }
    printf("\n\n You are out of the for loop");
}
```

```
OUTPUT:
10
8
6
4
2
You are out of the for loop
```

c. Program use nested FOR

```
#include <stdio.h>
main()
{
    int i,j;
    for( i=1; i<=5; i=i+1)
    {
                        for( j=10; j>=1; j=j-2)
    {
                          printf("\n  i=%d  j=%d",i ,j);
    }
                          printf("\n End of loop %d",i);
    }

    printf("\n\n You are out of the for loop");
}
```

**OUTPUT:**

```
i=1    j=10
i=1    j=8
i=1    j=6
i=1    j=4
i=1    j=2
End of loop 1
i=2    j=10
i=2   j=8
i=2    j=6
i=2    j=4
i=2    j=2
End of loop 2
i=3   j=10
i=3   j=8
i=3    j=6
i=3    j=4
i=3    j=2
End of loop 3
i=4    j=10
i=4    j=8
i=4    j=6
i=4    j=4
i=4    j=2
End of loop 4
i=5   j=10
i=5    j=8
i=5    j=6
i=5    j=4
i=5    j=2
End of loop 5
You are out of the for loop
```

d. Program use WHILE: to print numbers from 1 to 5

```
#include <stdio.h>
main()
{
    int i;
    i=1;
    while (i <= 5)
    {
        printf("\n %d",i);
        i++;
    }
    printf("\n\n You are out of while loop");
}
```

**OUTPUT:**

```
1
2
3
4
5

You are out of the for while loop
```

e. Program use WHILE: to print numbers from 10 to1

```
#include <stdio.h>
main()
{
    int i;
    i=10;
    while (i >= 1)
    {
        printf("\n %d",i);
        i=i-2;
    }
    printf("\n\n You are out of while loop");
}
```

**OUTPUT:**

```
10
8
6
4
2

You are out of the while loop
```

f. Program use DO-WHILE: to print numbers from 1 to 5

```
#include <stdio.h>
main()
{
      int i=1;
      do
      {
            printf("Value of variable i is: %d\n", i);
            i=i+1;
      }while (i<=5);
      printf("\n You are out of do-while loop");
      return 0;
}
```

**OUTPUT:**

```
Value of variable i is:1
Value of variable i is:2
Value of variable i is:3
Value of variable i is:4
Value of variable i is:5
You are out of the do-while loop
```

g. Program use DO-WHILE: to print numbers from 10 to 1

```
#include <stdio.h>
main()
{
        int i=10;
        do
        {
                printf("Value of variable i is: %d\n", i);
                i=i-2;
        }while (i>=1);
        printf("\n You are out of do-while loop");
        return 0;
}
```

**OUTPUT:**

```
Value of variable i is:10
Value of variable i is:8
Value of variable i is:6
Value of variable i is:4
Value of variable i is:2
You are out of the do-while loop
```

### 4.3.2 Construct program(s) that use before BREAK statements

```
#include <stdio.h>
main()
{
 int i;
 for( i=1; i<=6; i++)
 {
 printf("\n Value of variable i is: %d",i);
   }
 printf("\n The control is out of the for loop");
}
```

**OUTPUT:**

```
Value of variable i is:1
Value of variable i is:2
Value of variable i is:3
Value of variable i is:4
Value of variable i is:5
Value of variable i is:6

The control is out of the for loop
```

### 4.3.3 Construct program(s) that use BREAK statements

```
#include <stdio.h>
main()
{
  int i;
  for( i=1; i<=6; i++)
  {
  printf("\n Value of variable i is: %d",i);
  if (i == 3)
   break;
  }
  printf("\n The control is out of the for loop");
}
```

**OUTPUT:**

```
Value of variable i is:1
Value of variable i is:2
Value of variable i is:3
The control is out of the for loop
```

### 4.3.4 Construct program(s) that use GOTO statements

```
#include <stdio.h>
int main()
{
  int sum=0;
  for(int i = 0; i<=10; i++)
  {
          printf("\n Value of variable i is: %d",i);
          sum = sum+i;
          if(i==5)
          {
            goto addition;
          }
  }
  addition:
  printf("\n The sum of i=%d", sum);
  return 0;
}
```

**OUTPUT:**

```
Value of variable i is:0
Value of variable i is:1
Value of variable i is:2
Value of variable i is:3
Value of variable i is:4
Value of variable i is:5
The sum of i=15
```

## Tutorial

a) List THREE(3) types of looping statements

b) The program below contains FIVE (5) errors. Find the errors and rewrite this program with the correct code.

```
#include <stdio.x>
int main{}
{
int i=0, sum=0;
while (i <=5)
{
        sum +=i;
        printf('sum [%d]=%d\n",1, sum);
        i++;)
return 0;
}
```

c) Write a program which displays the following output by using do-while statements.

Output:

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
```

d) The loop statements consist of for, while and do-while. Program below uses for loop statement to calculate the average of 5 numbers that entered by user. However the program had some errors. Solve the errors and write the correct program. Then change the program to do-while loop statement.

```
#include <studio.h>
void main()
{
Int no=1, total=0, Avg=0, num;

for (no=1;no<=5;no++)
{
        scanf("%c", &num);
        total=total+ Num;
}
                avg=total/5;
        printf(" Average=%d",avg);
return 0;
}
```

e) List the correct syntax of for loop.

f) Rewrite the above program segment to do-while statement.

```
#include <stdio.h>
main()
{
int a, jum;
for (a=1;a<10;a+=2)
{
        jum=jum+a;

printf("%d\n",a);
}
        printf(" Jumlah ialah %d\n",jum);

return 0;

}
```

g) Apply C language to write a program to print 10 times of the below statemen using for.

```
I want to get EXCELLENT in C Programming
```

h) Produce C program to display multiplication table using while looping statement and draw the flowchart. The example of a output as follows :

```
Please enter number of multiplication table : 3

                1 X 3 = 3

                2 X 3 = 6

                3 X 3 = 9

                    :

                    :

               12 X 3 = 36
```

i) Define looping statement with an example.

j) Choose for loop statement  to write a program code to display he output as below. The symbol * must be stored in the variable.

k) Use the C language to write program using for loop statement which displays the output below.

```
Value of a : 10
Value of a : 11
Value of a : 12
Value of a : 13
Value of a : 14
Value of a : 15
Value of a : 16
Value of a : 17
Value of a : 18
Value of a : 19
```

# Practical Activities

**Program 1**

```
#include <stdio.h>
main()
{
            int j;
            for ( j=5; j<=50; j=j+3)
            {
            printf( "Value of variable j is: %d\n",j );
            }
  return 0;
}
```

**Program 2**

```
#include <stdio.h>
main()
{
            int j;
            for ( j=50; j>=5; j=j-3)
            {
            printf( "Value of variable j is: %d\n",j );
            }
  return 0;
}
```

**Program 3**

```
#include <stdio.h>
main()
{
       int I,V;
       I=1;
       while (I <= 9)
       {
              V= I* 150;
                       printf("\n Current at V%d = %d",I,V);
              I++;
       }
       printf("\n\n You are out of while loop");
    return 0;
}
```

**Program 4**

```
#include <stdio.h>
main()
{
    int I,V;
    I=9;
    while (I >= 1)
    {
        V= I* 150;
        printf("\n Current at V%d = %d",I,V);
        I--;
    }
    printf("\n\n You are out of while loop");
  return 0;
}
```

**Program 5**

```
#include <stdio.h>
int main()
{
        int I,V;
        I=1;
        do
        {
        V= I* 150;
        printf("\n Current at V%d = %d",I,V);
        I++;
        }while (I<=10);
        printf("\n\n You are out of while loop");
        return 0;
}
```

**Program 5**

```
#include <stdio.h>
int main()
{
        int I,V;
        I=10;
        do
        {
        V= I* 150;
        printf("\n Current at V%d = %d",I,V);
        I--;
        }while (I>=1);
        printf("\n\n You are out of while loop");
        return 0;
}
```

# 5.0 FUNCTION

## 5.1 INTRODUCTION

In your daily life, you depend on many appliances to complete your work. For example, you might be using a washing machine to wash clothes, an oven to cook food and a toaster to toast bread. These electronic appliances are used to accomplish specific tasks quickly and easily. Programs too depend on elements such as functions to perform specific tasks. C language offers various built-in functions. In this unit, you will learn to use these built-in functions in the programs. You will also learn to define your own functions according to the need.

## Note

## 5.2 Remember FUNCTION statements

### 5.2.1 Define FUNCTION statement

**Definition:** A function is a block of statements that perform a specific task.

Functions divide the code and modularize the program for better and effective results. A larger program is divided into various subprograms which are called as functions

### 5.2.2 Identify the need for FUNCTION statements

Advantages of Using Functions

Functions offer a number of advantages such as:

1. **Reusability -** A function once written can be used anywhere in a program. You need not write the same set of statements repeatedly in a program.
2. **Function Portability -** You can use functions across many programs.
3. **Modularity -** You can break down large programs into simple functions. This will make the program readable.

4. **Easy to Debug -** Since functions make programs more readable, you can easily debug programs.
5. **Easy to Modify and Extend:** As you can easily add or remove functions in programs, the capability of programs can be easily extended.

A function will have the following three components:

- **Function Header**- Specifies the function name along with parameters.
- **Function Body** - Contains the actual code of the function.
- **Return Statement** - Returns a value to the main program.

### 5.2.3 Describe the structure of FUNCTION statements

**Function Declaration**

Specify the name of a function that we are going to use in our program like a variable declaration. We cannot use a function unless it is declared in a program. A function declaration is also called function prototype.

```
return_data_type function_name (data_type arguments);
```

- **return_data_type**: is the data type of the value function returned back to the calling statement.
- **function_name**: is followed by parentheses
- **Arguments** names with their data type declarations optionally are placed inside the parentheses

```
#include <stdio.h>
/*Function declaration*/
int add  (int x,y);
/*End of Function declaration*/
main() {
```

**Function Definition**

Function definition means just writing the body of a function. A body of a function consists of statements which are going to perform a specific task. A function body consists of a single or a block of statements. It is also a mandatory part of a function.

```
int add (int x, int y)  //function body
{
int z;
z=x+y;
return  0;
}
```

Function definition syntax:

```
<return type> < function name> ( < arguments>)
{
< local variable declarations>;
< function statemnets>;
}
```

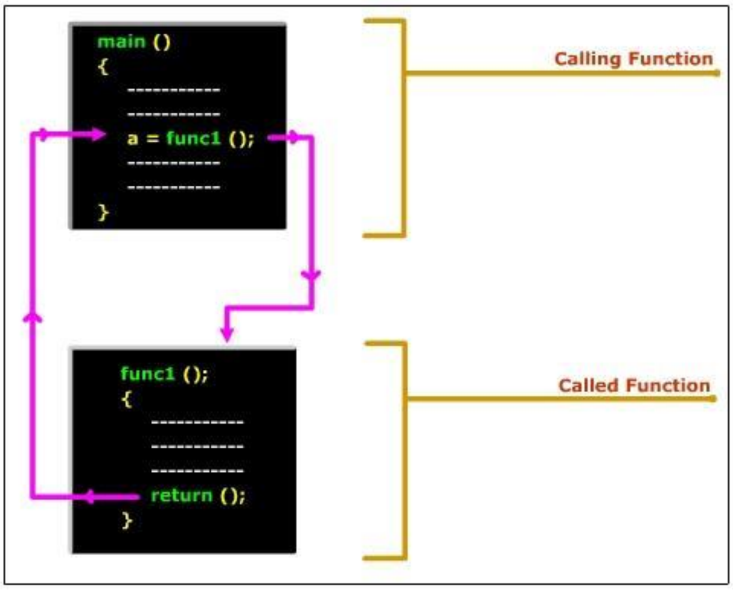
- **Return Type -** The type of the value that the function will return. The return type can be of any data type. The default return type is integer.
- **Function Name -** Any valid name can be given as the function name. Rules for naming a function name are same as that of a variable name.
- **Arguments -** The values or variables that the function requires along with their data types. Arguments can also be referred to as parameters.
- **Local Variable Declaration -** The variables that are declared within the functions.
- **Function Statements -** C statements within the function.

**Example**

```
int mul (int a,int b)   // Function Name & Arguments
{
int prod;      //  Local Variable Declaration
prod = a * b;  // Function Statements
return(prod);  // Function Statements
}
```

**Return Statement**

The main function will call the user-defined function to perform the specified action. Thus, main() is the calling function and the user-defined function is the called function. The calling function is the function that calls the user-defined function. The called function is the function that is being called by the calling function.



```
main ()
{
    -----------
    -----------
    a = func1 ();
    -----------
    -----------
}
```

```
func1 ();
{
    -----------
    -----------
    -----------
    return ();
}
```

When the main function calls another function, the control passes from main() to the called function. After execution of the statements in the called function, the control returns to main(). The called function may or may not return a value to main(). The function will return a value to the calling function with the use of the return statement. When the return statement is executed, the function execution will be terminated and the control is passed back to the calling function.

## 5.3 Understand FUNCTION statements

### 5.3.1 Declaring a Function

A function can be declared either before defining the main function or after the main function. When you define a function before the main function, it is not necessary to declare the function. On the other hand, if you define a function after main(), you must declare the function. This declaration, also known as function prototype, notifies the compiler that the definition for this function will appear later in the program. Function prototype is the function header with a semicolon and is used for declaring the function before defining the function.

**Example 1**

```
void main()
{
int func(int,int);   // Function Prototype


        …
        …
        …
}
```

**Example 2**

```
int func(int,int); // Function Prototype
void main()
{
    …
    …
    …
}
```

Remember the following points when you declare a function:

1. Mention the return type of the function. If the function is not returning any value, the return type must be declared as **void.**
2. Specify the arguments along with their data type.

### 5.3.2 Passing Arguments to Functions

Functions are written to perform specific tasks. Functions might need values to perform these tasks. These values can be accepted from the user either in the called function or in the calling function. If the values are accepted in the calling function, these values have to be passed to the called function. The values are passed through the arguments during the function call. Function calls can be classified into two types.

They are:

- Call by value
- Call by reference

Call by value refers to passing values as arguments to the function. Call by reference refers to passing the address of the variables as argument to the function. In this unit you will learn about call by value.

**Call by Value – Direct Value**

It is possible to pass values from one function to another function for performing calculations. This type of a function call is referred to as call by value.

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **poli()** definition as follows.

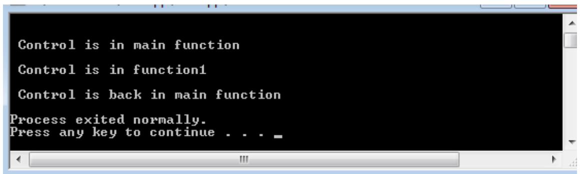
```
# /* function definition to swap the values */
void poli(int x, int y) {
  int temp;
  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put temp into y */
    return;
}
```

## 5.4 Apply FUNCTION statements

### 5.4.1 Construct program(s) that use FUNCTION statement

```
#include <stdio.h>
main()
{
      void func1();
      printf("\n\n Control is in main function ");
      func1();
      printf("\n\n Control is back in main function ");
}
void func1()
{
      printf("\n\n Control is in function1 ");
}
```

**OUTPUT:**

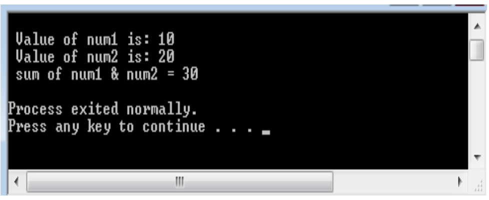


```
Control is in main function
Control is in function1
Control is back in main function
Process exited normally.
Press any key to continue . . .
```

## 5.4.2 Construct program(s) that use Call by Value statements (Example 1)

```
#include <stdio.h>
int sum(int a, int b)
{
     int c=a+b;
     return c;
}
main()
{
    int num1,num2,num3;
    num1 =10;
    num2 = 20;
    printf(" \n Value of num1 is: %d", num1);
    printf(" \n Value of num2 is: %d", num2);
    num3 = sum(num1, num2);
    printf("\n sum of num1 & num2 = %d", num3);
    return 0;
}
```

**OUTPUT:**

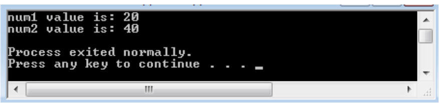


```
Value of num1 is: 10
Value of num2 is: 20
sum of num1 & num2 = 30

Process exited normally.
Press any key to continue . . .
```

## 5.4.3 Construct program(s) that use Call by Value statements (example 1)

```
#include <stdio.h>
int increment(int var)
{
    var = var+20;
    return var;
}

main()
{
   int num1,num2;
   num1=20;
   num2 = increment(num1);
   printf("num1 value is: %d", num1);
   printf("\nnum2 value is: %d", num2);
   return 0;
}
```

**OUTPUT:**

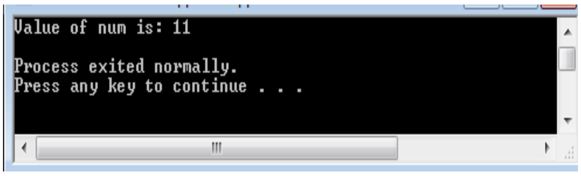


```
num1 value is: 20
num2 value is: 40

Process exited normally.
Press any key to continue . . .
```

### 5.4.4 Construct program(s) that use Call by Reference statements (example 1)

```
#include <stdio.h>
void increment(int  *var)
{

   *var = *var+10;
}
main()
{
    int num=1;
    increment(&num);
    printf("Value of num is: %d", num);
   return 0;
}
```

**OUTPUT:**

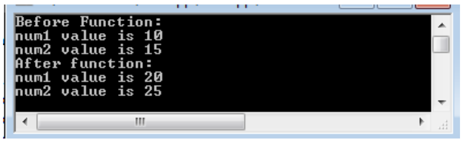


```
Value of num is: 11

Process exited normally.
Press any key to continue . . .
```

### 5.4.5 Construct program(s) that use Call by Reference statements(example 2)

```
#include <stdio.h>
void func_num ( int *var1, int *var2 )
{
  *var1 = *var1 +10 ;
  *var2 = *var2 +10 ;
}
main( )
{
  int num1 = 10, num2 = 15 ;
  printf("Before Function:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);

  /*calling  function*/
  func_num( &num1, &num2 );
  printf("\nAfter function:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);
  return 0;
```

**OUTPUT:**



```
Before Function:
num1 value is 10
num2 value is 15
After function:
num1 value is 20
num2 value is 25
```

# Tutorial

a) Explain the following function prototypes:

i. int func(void);

ii. char func(char a, char b);

iii. void func(void);

iv. void func(int a, char b);

v. float func(int a);

b) Declare the following functions:

vi. A function called change that will accept an integer value and return a float value.

vii. A function called calc that will accept two float values but does not return any value.

viii. A function called sample that will accept three integer values and return an integer value.

ix. A function called convert that will accept an integer value and two float values and return a float value.

x. A function called getdata that will not accept or return any value.

c) Write a program to find the sum of two numbers using functions. The function should have no arguments and no return value.

d) Write a program to accept the number of books from the user in the main function. Write a function to accept the book's name and their price. Write another function to display these details.

e) Write a program to find the sum of 12+22+32+…+n2 using function.

f) Explain the structure of a function with an example.

g) Write a program to find the factorial value of a number. The function should have argument but no return value.

h) Differentiate between local and global variables with an example each.

i) Write a function named square to calculate the area and perimeter of the square. Write another function named rectangle to calculate the area and perimeter of the rectangle. The main() should prompt the user to choose whether the calculation is for  the rectangle or square and the suitable function call must be made. Finally the area and perimeter must be displayed on the screen.

## Practical Activities

**Program 1:**

```
#include <stdio.h>
int addition(int Rr1, int Rr2)
{
    int TotalR;
    TotalR = Rr1+Rr2;
    return TotalR;
}

int main()
{
    int R1, R2;
    printf("Enter  value of R1: ");
    scanf("%d",&R1);
    printf("Enter value of  R2: ");
    scanf("%d",&R2);
    int Total = addition(R1, R2);
    printf ("Output: %d", Total);
    return 0;
}
```

**Program 2:**

```
#include <stdio.h>
int kira()
{
    int R1, R2, Total;
    printf("Enter  value of R1: ");
```

```
     scanf("%d",&R1);
     printf("Enter value of  R2: ");
     scanf("%d",&R2);
     Total = R1+R2;
     printf ("Output: %d", Total);
}
int main()
{
  kira();
  return 0;
}
```

**Program 3:**

```
#include <stdio.h>
void increment(int  *Total)
{

     int R1, R2;
     printf("Enter  value of R1: ");
     scanf("%d",&R1);
     printf("Enter value of  R2: ");
     scanf("%d",&R2);
    *Total = R1+R2;

}
int main()
{
     int num=20;
     increment(&num);
     printf("Value of num is: %d", num);
   return 0;
}
```

**Program 4:**

```
#include <stdio.h>
int addNumbers(int a, int b);        int main()
{
    int n1,n2,sum;
    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);
    printf("sum = %d",sum);
    return 0;
}
```

```
int addNumbers(int a, int b)
{
   int result;
   result = a+b;
   return result;
}
```

# 6.0 ARRAY

**INTRODUCTION**

An array is a collection of data items, all of the same type, accessed using a common name. A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.

# Note

## 6.1 Remember ARRAYS

### 6.1.1 Define ARRAY statement

**Definition:** Array is a collection of values having similar data type and are stored in consecutive memory locations.

### 6.1.2 Identify the need for ARRAY in programming

An array is a data structure, which can store a fixed-size collection of elements of the same data type. An array is used to store a collection of the same type data.

### 6.1.3 Describe the structure of ARRAY

**Definition:** A single-dimensional array, also referred as 1D array, will have a single row and can have any number of columns.

Single-dimensional arrays will have only one subscript. Subscript refers to the dimension of the array.

When you declare a single-dimensional array as x[7], it means that the array has one row and seven columns, as shown in Figure 6.1.3.a. This array can thus hold up to seven values.

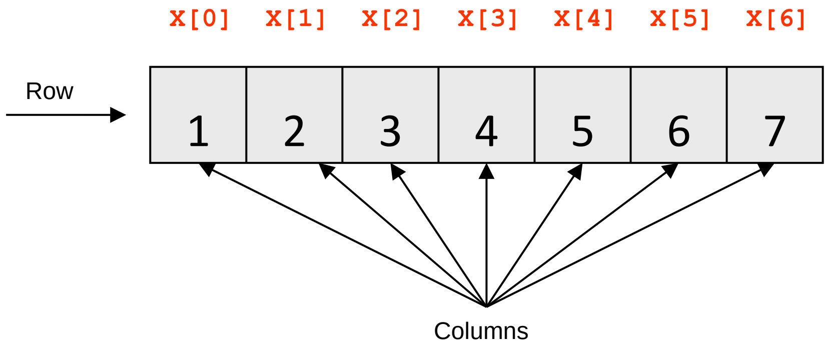


**Figure 6.1.3.a: Representation of a Single-Dimensional Array**

**Declaring an Array**

**Syntax**

**<Data type> <Variable name>[Size of the array];**

**Example**

```
int x[3];
```

| Data type | variable | Saiz of the array |
|---|---|---|
| **int** | **x** | **[3]** |

**Figure 6.1.3.b: Declaring an Array**

Here x is an array variable that can hold three values. Each value is referred as an element. This array, therefore, will have three elements of integer data type. Notice that the array size is specified within the square brackets [ ] as shown in Figure 6.1.3.b.

### 6.1.4 Initialising an Array

**Syntax**

**<Variable name> [Array index] = <Value>;**

**Example**

```
x[0] = 15;
x[1] = 26;
x[2] = 37;
```

Refer to Figure 6.1.4.a, x[0], x[1] and x[2] refers to the first, second and third elements, respectively. The first element will always have the array index as 0 as shown in Figure 6.1.4.b. Array index refers to the location of the values in an array.

| X[0] | X[1] | X[2] |
|---|---|---|
| 1st Element | 2nd Element | 3rd Element |

**Figure 6.1.4.a : Array Elements**

| X[0] | X[1] | X[2] |
|---|---|---|
| 15 | 26 | 37 |
| 1st Element | 2nd Element | 3rd Element |

**Figure 6.1.4.b : Representation of Index in Array Elements**

In the Figure 6.1.4.c, you will notice that the values in an array are stored in consecutive memory locations.

| x[0] | x[1] | x[2] | |
|---|---|---|---|
| 15 | **26** | **37** | |
| | | | |

**Figure 6.1.4.c : Array Elements Stored In Consecutive Memory Locations**

You can initialise the array at the time of declaration in the following manner:

**int x[3] = {15,26,37};**

When initialising an array, the values are enclosed within the curly brackets { }.

You will be able to access the array elements through the array index. For example, if you need to print the second element of the array, the code will be:

**printf(" %d",x[1]);**

**Program to declare and initialise an array**

```
#include <stdio.h>
main()
{
  int x[3]={15,26,37};
  printf("\n x[0] (1st element) : %d",x[0]); // Displays the 1st element.
  printf("\n x[1] (2nd element) : %d",x[1]); // Displays the 2nd element.
  printf("\n x[2] (3rd element) : %d",x[2]);// Displays the 3rd element.
}
```

**Output**

```
 x[0] (1st element) : 15
 x[1] (2nd element) : 26
 x[2] (3rd element) : 37
```

### 6.1.5 Entering Data into an Array

You know that an array can have any number of elements in it. If an array has 100 elements, it is not possible to accept all 100 values through 100 **scanf** statements. Therefore, it is better to use a **for** loop along with arrays, to accept and display the values.

```
for( i=0; i<=9; i++)
scanf("%d",&x[i]);
```

**Code Segment 6.1.5**

The following steps explain the execution of Code Segment 6.1.5:

1. Initially the value of *i* is initialised to *0*. Now, *x[i]* becomes *x[0]*, which is the first element in the array.
2. The condition *(i<=9)* is now checked and if the condition is true, the next step is executed.
3. Now, the **scanf** statement will accept the value from the user.
4. The value of the variable *i* is incremented by *1*.

5. Step 2, 3 and 4 is repeated till the condition fails. Once the condition fails, the control comes out of the **for** loop.

As the array has to accept ten values, the array index is specified only till *9*. *x[9]* is the 10th element in the array, as the index starts from zero .

Table 6.1.5 gives the details for the dry run for Code Segment 6.1.5.

**Table 6.1.5: Dry run Code Segment 6.1.5.**

| Value of i<br>[ Array index] | Condition | x[ i ] | Element |
|---|---|---|---|
| 0 | True | x [ 0 ] | 1st |
| 1 | True | x [ 1 ] | 2nd |
| 2 | True | x [ 2 ] | 3rd |
| 3 | True | x [ 3 ] | 4th |
| 4 | True | x [ 4 ] | 5th |
| 5 | True | x [ 5 ] | 6th |
| 6 | True | x [ 6 ] | 7th |
| 7 | True | x [ 7 ] | 8th |
| 8 | True | x [ 8 ] | 9th |
| 9 | True | x [ 9 ] | 10th |
| 10 | False | - | - |

### 6.1.6 Reading Data from an Array

You should be familiar with how to enter a data into an array. Now, you will learn how to read the data stored in an array. You can use a **for** loop with single **printf** statement to print the values from an array.

```
for( i=0; i<=9; i++)
printf("%d",x[i]);
```

**Code Segment 6.1.6**

The following steps explain the execution of Code Segment 6.1.6 :

1. Initially the value of *i* is initialised to *0*. Now, *x[i]* becomes *x[0]*, which is the first element in the array.

2. The condition (*i<=9*) is now checked and if the condition is true, the next step is executed.
3. Now the **printf** statement will print the value on the screen.
4. The value of the variable *i* is incremented by *1*.
5. Step 2, 3 and 4 is repeated till the condition fails. Once the condition fails, the control comes out of the **for** loop.

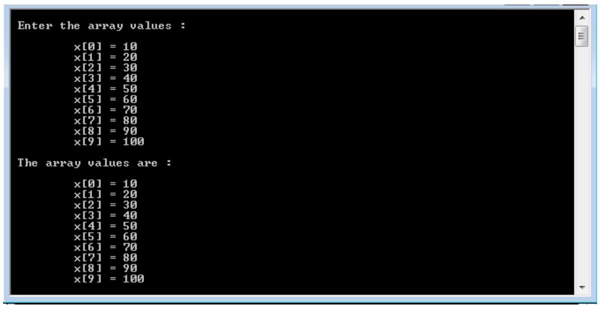
Program to accept 10 values from the user and to print the values on the screen.

**/* Program to accept 10 values from the user and to print them on the screen */**

```
#include <stdio.h>
void main()
{
  int x[10],i;
  printf("\n Enter the array values : \n\n");
  for( i=0; i<10; i++) // Array index starts with 0 - 9
  {
    printf("\t x[%d] = ",i);
    scanf("%d",&x[i]);
  }
  printf("\n The array values are : \n");
  for( i=0; i<10; i++)
  printf("\n\t x[%d] = %d ",i,x[i]);
}
```

**Code Sample 6.1.6**

**Output**

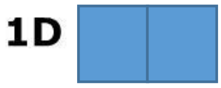


```
Enter the array values :

        x[0] = 10
        x[1] = 20
        x[2] = 30
        x[3] = 40
        x[4] = 50
        x[5] = 60
        x[6] = 70
        x[7] = 80
        x[8] = 90
        x[9] = 100

The array values are :

        x[0] = 10
        x[1] = 20
        x[2] = 30
        x[3] = 40
        x[4] = 50
        x[5] = 60
        x[6] = 70
        x[7] = 80
        x[8] = 90
        x[9] = 100
```

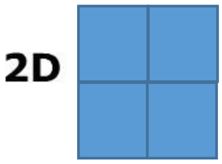
## 6.2 Understand ARRAY

### 6.2.1 Differentiate the one, two and three Dimensional ARRAY in C language

Multidimensional arrays are arrays with more than one dimension. Multidimensional arrays are used to represent data in terms of rows and columns. A multidimensional array that has two subscripts is called two-dimensional array (also known as 2-D arrays). A two-dimensional array is an array of single-dimensional arrays. Similarly, a three-dimensional array (also called a 3-D array) is an array of two-dimensional arrays. In other words, an n-dimensional array is an array of (n-1) dimensional arrays.

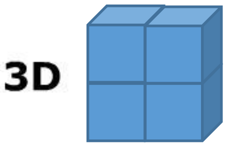


syntax
<Data type> <Variable name> [Array index] ;



syntax

<Data type> <Variable name>[Number of rows][Number of columns];



syntax

<Data type> <Variable name>[N1][N2][N3];

### 6.2.2 Two-Dimensional Array

Single-dimensional array stores the simple list of values in a linear fashion. A two-dimensional array will have n number of rows and m number of columns.

A two-dimensional array is also referred as *matrix*, which is the combination of rows and columns. Figure 6.2.3b illustrates the arrangement of rows and columns in a two dimensional array. The array always starts with 1st row and 1st column and the array index for which will be *0* (represents first row), *0* (represents first column).

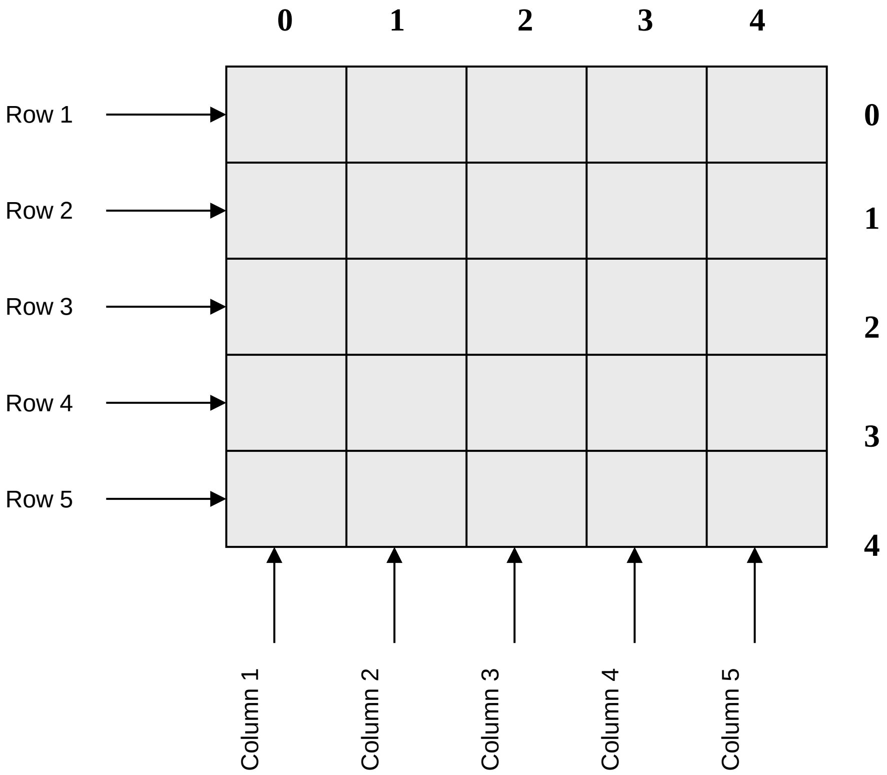


**Figure 6.2.2 : Representation of 2-D Array**

### 6.2.3 Declaring a Two-Dimensional Array

When declaring a two-dimensional array, you need to mention the data type of the variable and the number of rows and columns.

**Syntax**

**<Data type> <Variable name>[Number of rows][Number of columns];**

**Example**

**int x[5][5];**

Here, *x* is an array variable, which has five rows and five columns, as shown in Figure 6.2.3. This array can hold twenty five values (5*5).

| Data type | variable | Number of Rows | Number of Columns |
|---|---|---|---|
| int | X | [4] | [4] |

**Figure 6.2.3 : Declaring a 2-D Array**

**<Data type> <Variable name>[N1][N2][N3]...[Nn];**

### 6.2.4 Initialising a Two-Dimensional Array

A two-dimensional array can be initialised in the same way as how a single-dimensional array is initialised, as shown here:

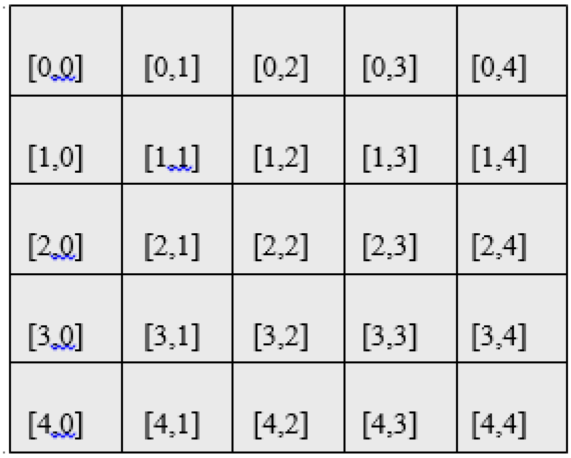
**Syntax**

```
<Variable name> [Row][Column] = <Value>;
```

**Example**

```
int x[5][5]={ {40,50,87,66,45},
              {54,72,24,36,77},
              {88,66,76,76,77},
              {22,56,89,33,44},
              {78,65,56,87,95} };
```

Figure 6.2.4.a clearly shows the array index of the various elements of the array. Notice that the first element in the array has the index *0,0*. Figure 6.2.4.b and 6.2.4.c represents the elements of the array *x* arranged in rows and columns.



| | | | | |
|---|---|---|---|---|
| [0,0] | [0,1] | [0,2] | [0,3] | [0,4] |
| [1,0] | [1,1] | [1,2] | [1,3] | [1,4] |
| [2,0] | [2,1] | [2,2] | [2,3] | [2,4] |
| [3,0] | [3,1] | [3,2] | [3,3] | [3,4] |
| [4,0] | [4,1] | [4,2] | [4,3] | [4,4] |

**Figure 6.2.4.a: Array Index of the Elements of a Two-Dimensional Array**

| | | | | |
|---|---|---|---|---|
| 1st | 2nd | 3rd | 4th | 5th |
| 5th | 6th | 7th | 8th | 9th |
| 10th | 11th | 12th | 13th | 14th |
| 15th | 16th | 17th | 18th | 19th |
| 20th | 21st | 22nd | 23rd | 24th |

**Figure 6.2.4.b: Elements of a Two-Dimensional Array**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 40 | 50 | 87 | 66 | 45 |
| 1 | 54 | 72 | 24 | 36 | 77 |
| 2 | 88 | 66 | 76 | 76 | 77 |
| 3 | 22 | 56 | 89 | 33 | 44 |
| 4 | 78 | 65 | 56 | 87 | 95 |

**Figure 6.2.4c: Two-Dimensional Array Representation**

Refer to Figure 6.2.4.d and 6.2.4.e to understand how array elements and array indexes are represented.

| | | |
|---|---|---|
| x | [0] | [0] |

1st Element

| | | |
|---|---|---|
| x | [0] | [1] |

2nd Element

**Figure 6.2.4.d : Array Elements**

| | | | |
|---|---|---|---|
| x | [0] | [0] | =40 |
| | 1st Row | 1st Column | |

| | | | |
|---|---|---|---|
| x | [0] | [1] | =50 |
| | 1st Row | 2nd Column | |

**Figure 6.2.4.e : Array Index**

### 6.2.5 Accessing Array Elements

You can access the array elements by using the array index. For example, if you want to print the element in the third column of the second row of an array, use the following code:

```
printf("\n x[2][3] %d",x[2][3]);
```

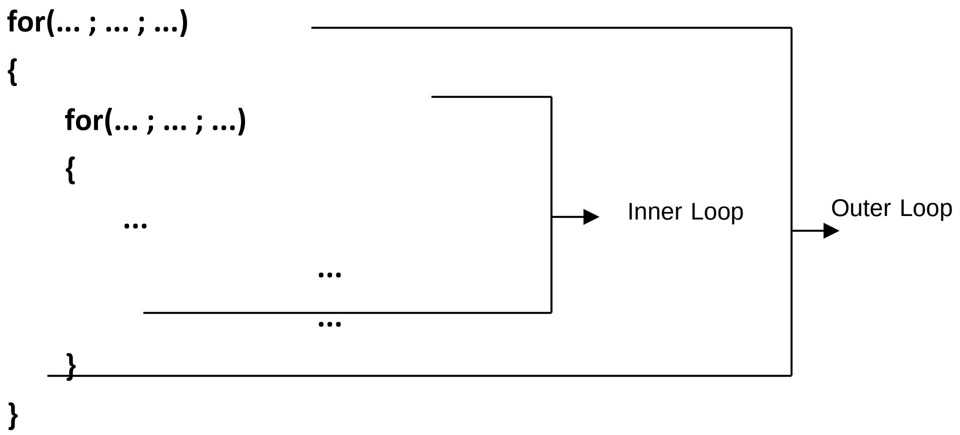
### 6.2.6 Entering Data in a Two-Dimensional Array

To enter data into a two-dimensional array, you need to specify the row and the column number. In a single-dimensional array, you used a single **for** loop to enter data into the columns of the single row. In a two-dimensional array, you will use two **for** loops to enter data into the array. One **for** loop will be used for specifying the row number and the other **for** loop will be used for specifying column number.

#### a. Nested *for* loop

You can nest the **for** loops in the same way as you nest the **if** statements. Nesting for loops refers to using a **for** loop within a **for** loop. You can have any number loops within a loop. The general syntax of nested **for** loop is as follows:

**Syntax**



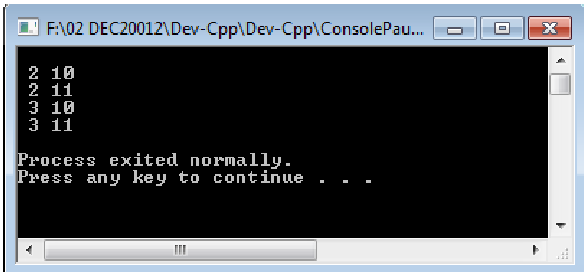
```
for(... ; ... ; ...)
{
    for(... ; ... ; ...)
    {
        ...
                  ...
                  ...
    }
}
```

In a nested **for** loop, the control begins from the outer loop. The control then passes to the inner loop. The inner loop will be executed until the condition specified in the inner loop fails. After executing the inner loop completely, the control passes to the outer loop to execute the next iteration of the outer loop. To understand the working of a nested **for** loop clearly, examine Code program 6.2.6:

```
#include<stdio.h>
main ()
{
      int i,j;
       for(i=2;i<=3;i++)
      {
         for(j=10;j<=11;j++)
         {
          printf("\n %d %d",i,j);
         }
      }
}
```

**Code program 6.2.6**

**Output**



```
 2 10
 2 11
 3 10
 3 11

Process exited normally.
Press any key to continue . . .
```

The flowchart for the execution of Code Segment 6.2.6 is shown in the Figure 6.2.6.

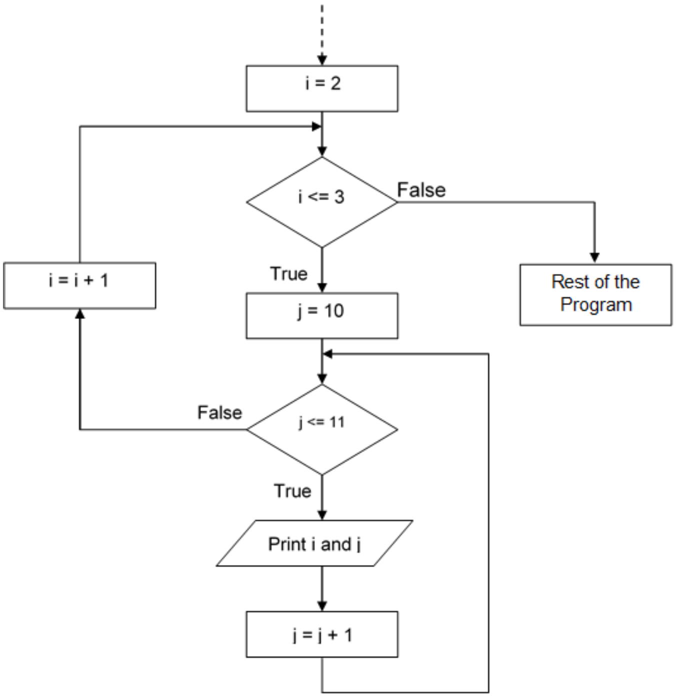


**Figure 6.2.6: Flowchart**

The following steps explain the execution of Code Segment 6.2.6:

1. In the outer loop, the value of *i* is initialised to *2*.
2. The test condition is executed and if the condition is true, that is if the value is less than or equal to *3*, the control moves to the inner loop.
3. Now the value of *j* is initialised to *10*.
4. Test condition for the inner loop is checked. If the condition is true, which happens when the value of *j* is less than or equal to 11, the control moves to the body of the inner loop.
5. The value of the variables *i* and *j* are printed.
6. The control again moves to the inner **for** loop statement and the value of *j* is incremented.
7. Steps 4, 5 and 6 are executed until the test condition in the inner loop is true.
8. When the test condition in the inner loop is false, the control passes to the outer loop.
9. Now the value of *i* is incremented.
10. Steps 2 to 9 are executed as long as the test condition in the outer loop is true.

11. When the test condition is not satisfied, the control comes out of the loop and the rest of the program statements are executed.

Table 6.2.6  gives the details of the dry run for Code Segment 6.2.6.

| Value of i | Condition for i | Value of j | Condition for j | Result | Output | |
|---|---|---|---|---|---|---|
| | | | | | i | j |
| 1 | 2 <= 3 | - | - | True | - | - |
| 1 | - | 10 | 10 <= 11 | True | 2 | 10 |
| 1 | - | 11 | 11 <= 11 | True | 2 | 11 |
| 1 | - | 12 | 12 <= 11 | False | - | - |
| 2 | 3 <= 3 | - | - | True | - | - |
| 2 | - | 10 | 10 <= 11 | True | 3 | 10 |
| 2 | - | 11 | 11 <= 11 | True | 3 | 11 |
| 2 | - | 12 | 12 <= 11 | False | - | - |
| 3 | 4 <= 3 | - | - | False | - | - |

**Table 6.2.6: Dry Run**

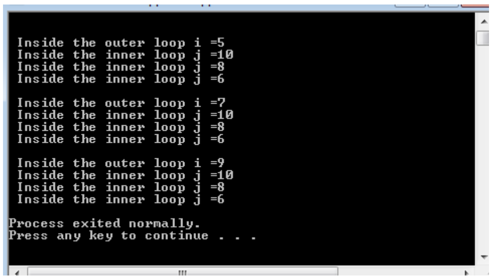
## 6.3  Apply ARRAYS

### 6.3.1  Construct program(s) that use ARRAY in C Language

```
#include <stdio.h>
main()
{
  int i,j;

  for( i=5; i<=10; i=i+2)  // working of a nested for loop
  {
    printf("\n\n Inside the outer loop i =%d",i);
    for( j=10; j>=5; j=j-2)
      printf("\n Inside the inner loop j =%d",j);
  }
}
```

**Code Sample 6.3.1a**

**Output**



```
 Inside the outer loop i =5
 Inside the inner loop j =10
 Inside the inner loop j =8
 Inside the inner loop j =6

 Inside the outer loop i =7
 Inside the inner loop j =10
 Inside the inner loop j =8
 Inside the inner loop j =6

 Inside the outer loop i =9
 Inside the inner loop j =10
 Inside the inner loop j =8
 Inside the inner loop j =6

Process exited normally.
Press any key to continue . . .
```

### 6.3.2 Construct program(s) that use Two Dimensional ARRAY in C Language

You can read the data entered into an array. You can use a nested **for** loop with a single **printf** statement to display all the values of a two-dimensional array on the screen.

Consider the Code Segment 6.3.2:

```
for( i=0; i<=2; i++)
      for( j=0; j<=2; j++)
            printf("\n%d",x[i][j]);
```

**Code Segment 6.3.2**

The following steps explain the execution of Code Segment 6.3.2 :

1. The value of *i* is initialised to *0*. This value is used to refer to the 1st row.
2. The condition for *i* is then checked. If the condition is true, that is if the value of *i* is less than or equal to *2* the control is passed to the inner loop.
3. The value of *j* is initialised to *0*. This value is used to refer to the 1st column.
4. The condition for *j* is checked. If the condition is true, that is if the value of *j* is less than or equal to *2*, the control is passed to the body of the loop.
5. The **printf** statement will print the value on the screen.
6. The control is again passed to the inner **for** loop and the value of *j* is incremented by *1*.
7. Steps 4,5 and 6 are executed till the condition for *j* is faise.
8. When the condition for *j* fails, the control is passed to the outer loop. Here, the value of *i* is incremented by *1*.

9. Steps 2 to 9 are executed till the condition for *i* is false.

10. When the condition fails, the control comes out of the **for** loop.

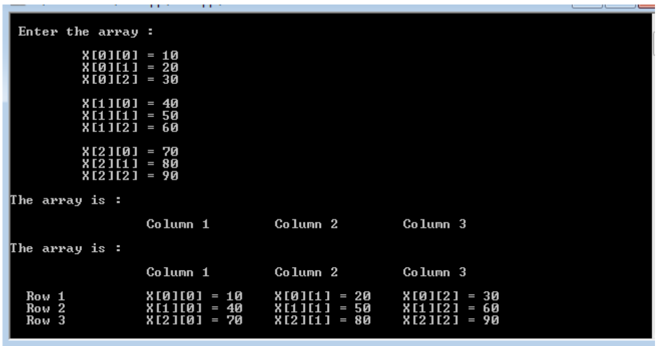
### 6.3.3 Hands-On!

The following program will accept integer values from the user; store it in a two-dimensional array and display the values on the screen.

```
#include <stdio.h>
main()
{
  int x[3][3],i,j;
  printf("\n Enter the array : \n");
  for ( i=0; i<=2; i++)
  {
    printf("\n");
          for( j=0; j<=2; j++)
          {
            printf("\t X[%d][%d] = ",i,j);
            scanf("%d",&x[i][j]);
          }
  }
  printf("\nThe array is : \n");
  printf("\nThe array is : \n");
  printf("\n\t\t Column 1 \t Column 2 \t Column 3\n");
  for( i=0; i<=2; i++)
  {
    printf("\n");
    printf("  Row %d  ",i+1);
          for( j=0; j<=2; j++)
          {
            printf("\t X[%d][%d] = %d ",i,j,x[i][j]);
          }
  }
}
```

**Code Sample 6.3.3**

**Output**



```
 Enter the array :

         X[0][0] = 10
         X[0][1] = 20
         X[0][2] = 30

         X[1][0] = 40
         X[1][1] = 50
         X[1][2] = 60

         X[2][0] = 70
         X[2][1] = 80
         X[2][2] = 90

The array is :

                 Column 1        Column 2        Column 3

The array is :

                 Column 1        Column 2        Column 3

  Row 1          X[0][0] = 10    X[0][1] = 20    X[0][2] = 30
  Row 2          X[1][0] = 40    X[1][1] = 50    X[1][2] = 60
  Row 3          X[2][0] = 70    X[2][1] = 80    X[2][2] = 90
```

# Tutorial

a. What is an array? List the types of array in C language.

b. Define a single-dimensional integer array called *arr1* with 5 elements. Assign the values 15,24,33,55 and 100 to the array elements.

c. Declare and initialise an array that can hold the values 3, 5 and 7.

d. Explain the need for arrays with an example.

e. Write a program to accept ten numbers and find their average.

f. Write short notes on single-dimensional array. How will you declare and initialise a single-dimensional array?

g. Write a program to accept ten numbers and arrange the elements in the ascending order.

h. Write a program to accept two arrays (4 elements each). Find the sum of two arrays.

i. Write a program to accept five elements of an array. Decrement the value of each element by two and display the array.

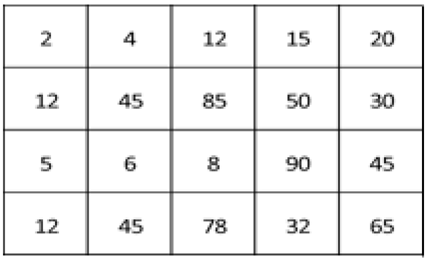
j. Write a program to accept the array size from the user. Accept the array elements from the user and display the same on the screen.

k. Write a program to accept 15 elements of an array. Find the sum of the elements.

l. Write a program to accept 10 numbers from the user. Find the number of positive and negative numbers.

m. Write a program to accept 6 elements of an array and display the array in reverse order.

n. The following figure represents the data stored in a two-dimensional array.



| | | | | |
|---|---|---|---|---|
| 2 | 4 | 12 | 15 | 20 |
| 12 | 45 | 85 | 50 | 30 |
| 5 | 6 | 8 | 90 | 45 |
| 12 | 45 | 78 | 32 | 65 |

o. Based on this figure, answer the following questions:

i. How many rows are present in the array?

ii. How many columns are present in the array?

iii. How many elements are present in the array?

iv. What is the value of the 3rd element in the array?

v. What is the array index of the 11th element?

vi. In which row is the value 90 present?

vii. In which column is the value 6 present?

viii. What is the value of the element x[2][2]?

ix. How will you access the value 65 in the array?

p. Consider a 3 X 3 integer array *arr*.

i. Write the declaration for *arr*.

ii. How many columns does the array have?

iii. How many rows does the array have?

iv. How many elements are present in the array?

v. What are the elements present in the 1st row.

vi. What are the element present in the 3rd column.

vii. Initialise the value of each element to ten (without using **for** loop).

viii. Initialise the value of each element to ten (using **for** loop).

q. Write a program to accept English and Mathematics marks of 6 students. Display the same.

i. In the same program, calculate the total marks scored by each student.

ii. Calculate the average marks in English and Mathematics in the same program.

r. Write a program to accept the values for a 2X3 dimension array. Display the values in the matrix form.

i. In the same program, write code to count the number of zeros stored in the array.

ii. Calculate the sum of all the values stored in the array.

s. Write a program to accept the values for a 2X4 dimension array. Display the values in the matrix form.

i. Increment the value of each element by one and display the matrix.

ii. Calculate the sum of the diagonal elements.

# Practical Activities

**Program 1:**

```
#include <stdio.h>
main()
{
  int array[10] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20};
  printf(" array[10] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ");
  printf("\n array [0]=%d ", array[0]);
  printf("\n array [3]=%d ", array[3]);
  printf("\n array [5]=%d ", array[5]);
  printf("\n array [6]=%d ", array[6]);
  printf("\n array [2]=%d ", array[2]);
  printf("\n array [9]=%d ", array[9]);
  return 0;
}
```

**Program 2:**

```
#include <stdio.h>
main()
{
  int x, array[10] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20};
  printf(" array[10] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}");
  for(x = 0; x < 10; x++)
    printf("\n array [%d] = %d ", x,array[x]);
  return 0;
}
```

**Program 3:**

```
#include <stdio.h>
main()
{
  int x, array[10];
  printf(" array[10] \n");
 for(x = 0; x < 10; x++)
 {
    printf("array [%d] =  ", x);
          scanf ("%d",&array[x]);
        }
  for(x = 0; x < 10; x++)
    printf("\n array [%d] = %d ", x,array[x]);
  return 0;
}
```

**Program 4:**

```
#include <stdio.h>
main()
{
  int array[2][5] = {{1,2,3,4,5},{6,7,8,9,10}};
  printf(" array[2][5] = {1,2,3,4,5},{6,7,8,9,10} ");
  printf("\n array [0][0]=%d ", array[0][0]);
  printf("\n array [1][3]=%d ", array[1][3]);
  printf("\n array [0][4]=%d ", array[0][4]);
  printf("\n array [0][2]=%d ", array[0][2]);
  printf("\n array [1][4]=%d ", array[1][4]);
  printf("\n array [1][1]=%d ", array[1][1]);
  return 0;
}
```

**Program 5:**

```
#include <stdio.h>
main()
{
  int array[4][4] ={{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}};
  printf(" array[4][4] ={{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15} ");
  printf("\n array [0][0]=%d ", array[0][0]);
  printf("\n array [1][1]=%d ", array[1][1]);
  printf("\n array [2][2]=%d ", array[2][2]);
  printf("\n array [3][3]=%d ", array[3][3]);
  printf("\n array [0][3]=%d ", array[0][3]);
  printf("\n array [3][0]=%d ", array[3][0]);
  return 0;
}
```

**Program 6:**

```
#include <stdio.h>
main()
{
  int x,y, array[4][4] ={{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}};
  printf(" array[4][4] ={{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15} ");
  for(x = 0; x < 4; x++)
  {
    for(y = 0; y < 4; y++)
          printf("\n array [%d][%d] = %d ", x,y,array[x][y]);
        }
  return 0;
}
```

**Program 7:**

```
#include <stdio.h>
main()
{
  int x,y, array[2][2];
  printf(" for array[2][2]  \n ");
  for(x = 0; x < 2; x++)
 {
         for(y = 0; y < 2; y++)
         {
         printf("array [%d][%d] =  ", x,y);
           scanf ("%d",&array[x][y]);
         }
 }
  for(x = 0; x < 2; x++)
  {
    for(y = 0; y < 2; y++)
    printf("\n array [%d][%d] = %d ", x,y,array[x][y]);
  }
  return 0;
}
```

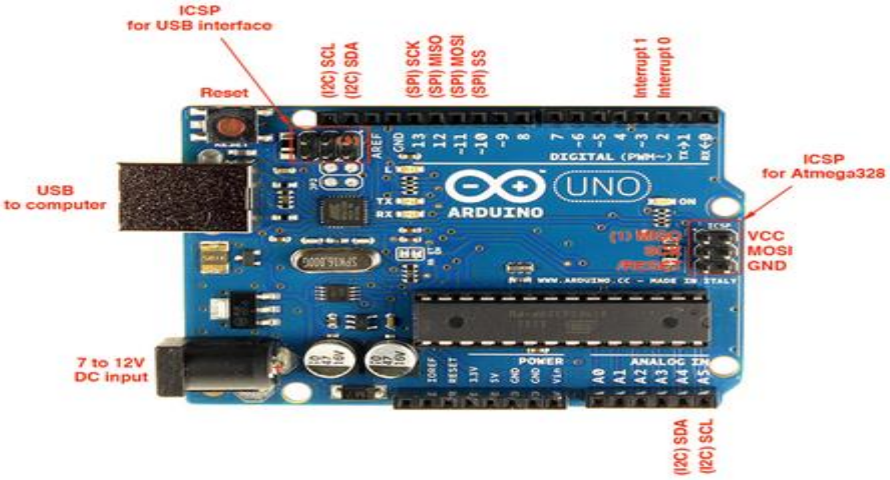
**Program 8:**

```
#include <stdio.h>
main()
{
  int x,y, array[5][5];
  printf(" for array[5][5]  \n ");

   for(x = 0; x < 5; x++)
                  {
                  for(y = 0; y < 5; y++)
                  {
         printf("array [%d][%d] =  ", x,y);
                  scanf ("%d",&array[x][y]);
         }
                  }
  for(x = 0; x < 5; x++)
  {
    for(y = 0; y < 5; y++)
           printf("\n array [%d][%d] = %d ", x,y,array[x][y]);
         }
  return 0;
}
```

# 7.0 DEMONSTRATE THE I/O OPERATION

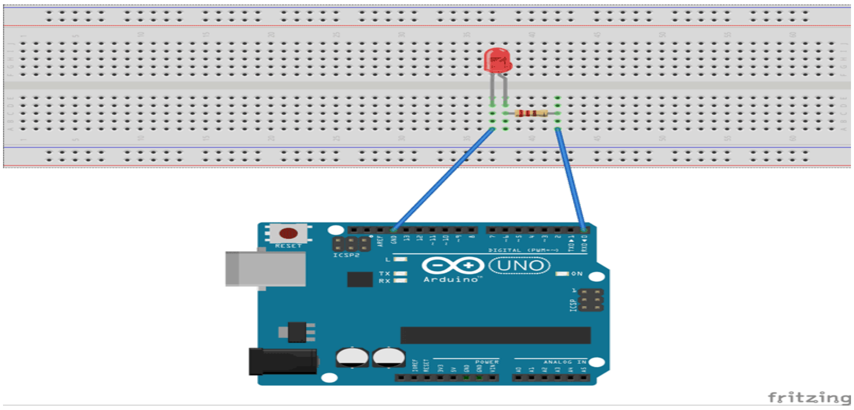
## ARDUINO

1. Introduction of Auduino
   a. Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board.
   a. Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs.
   a. Arduino projects can be stand-alone, or they can be communicate with software running on your computer (e.g. Flash, Processing, MaxMSP.) The boards can be assembled by hand or purchased pre-assembled; the open-source IDE can be downloaded for free.
   a. The Arduino programming language is an implementation of Wiring, a similar physical computing platform, which is based on the Processing multimedia programming environment.



## 1. LED Blinking with Arduino Uno

LED Blinking with Arduino Uno



```
// The setup function runs when you press reset or power the board
void setup()
{
 // initialize digital pin 0 as an output.
pinMode(0, OUTPUT);
 }
// the loop function runs over and over again forever
void loop()
{
digitalWrite(0, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(500);              // wait for a second
digitalWrite(0, LOW);    // turn the LED off by making the voltage LOW
delay(500);              // wait for a second
}
```

## 2. LED Blinking with Arduino Uno

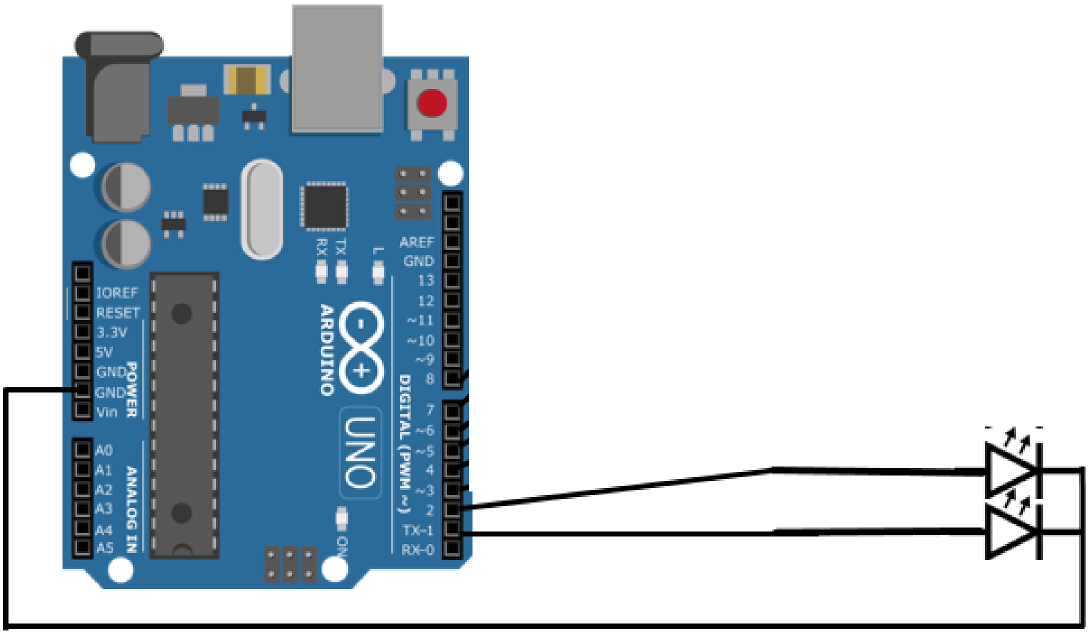
**Step 1: Introduction**

In this lesson , we build the first external circuit and control it from Arduino. We use digitalWrite command to turn LED on and off.

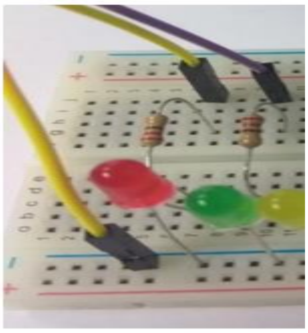
**Step 2 : Materials**

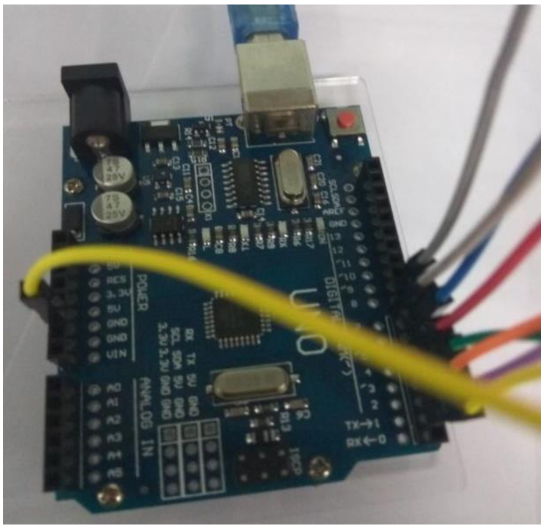
2 x LED
Arduino
1 x USB cable
Breadboard
Jumper Wires

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**





**Step 5 : The Code**

```
int led1=1;
int led2=2;
void setup() {
pinMode(led1,OUTPUT);
pinMode(led2,OUTPUT);
}
void loop() {
digitalWrite ( led1, HIGH);
delay (500);
digitalWrite ( led1, LOW);
delay (500);
digitalWrite ( led2, HIGH);
delay (500);
digitalWrite ( led2, LOW);
delay (500);
}
```

### 3. 4 LED Blinking with Arduino Uno

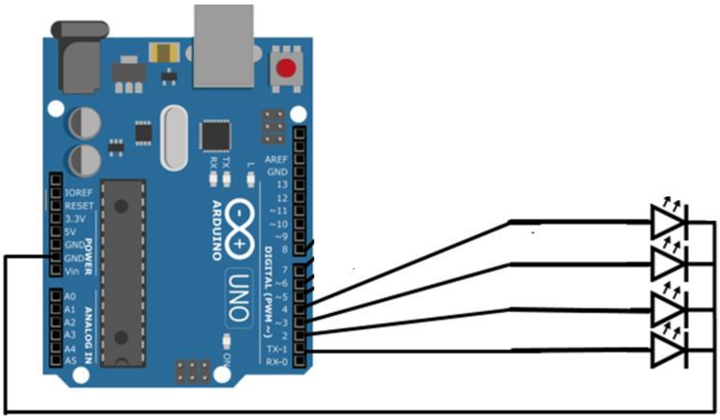
**Step 1 : Introduction**

In this lesson, we build the first external circuit and control it from Arduino. We use digitalWrite command to turn LED on and off.

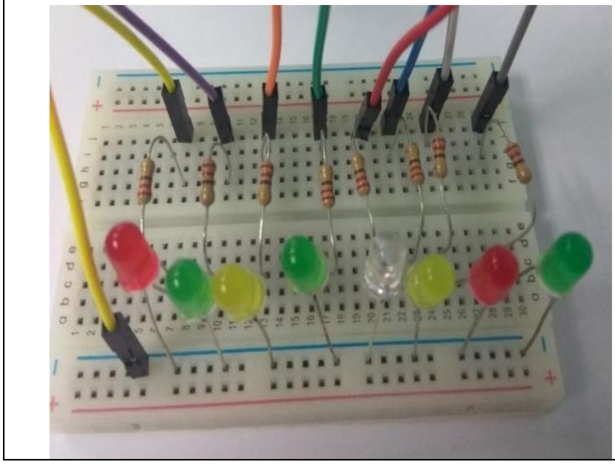
**Step 2 : Materials**

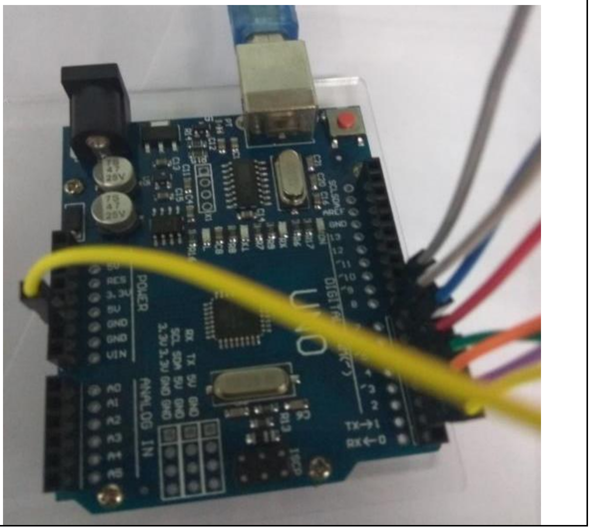
4 x LED
Arduino
1 x USB cable
Breadboard
Jumper Wires

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**





**Step 5 : The Code**

```
int led1=1;
int led2=2;
int led3=3;
int led4=4;
void setup() {
pinMode(led1,OUTPUT);
pinMode(led2,OUTPUT);
pinMode(led3,OUTPUT);
pinMode(led4,OUTPUT);
}
void loop() {
digitalWrite ( led1, HIGH);
delay (500);
digitalWrite ( led1, LOW);
delay (500);
digitalWrite ( led2, HIGH);
delay (500);
digitalWrite ( led2, LOW);
delay (500);
digitalWrite ( led3, HIGH);
delay (500);
digitalWrite ( led3, LOW);
delay (500);
digitalWrite ( led4, HIGH);
delay (500);
digitalWrite ( led4, LOW);
delay (500);
}
```

### 4. Multiple Blink LED

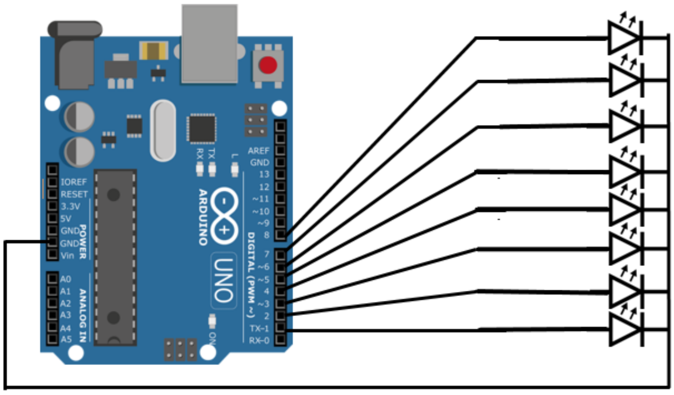
**Step 1 : Introduction**

In this lesson , we build the first external circuit and control it from Arduino. We use digitalWrite command to turn LED on and off.

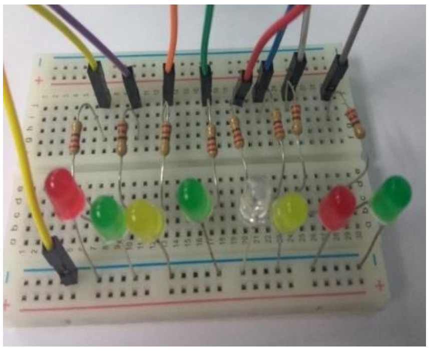
**Step 2 : Materials**

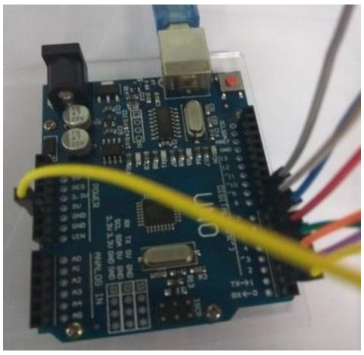
8 x LED
Arduino
1 x USB cable
Breadboard
Jumper Wires

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**





**Step 5 : The Code**

```
int led1=1;
int led2=2;
int led3=3;
int led4=4;
int led5=5;
int led6=6;
int led7=7;
int led8=8;
void setup() {
pinMode(led1,OUTPUT);
pinMode(led2,OUTPUT);
pinMode(led3,OUTPUT);
pinMode(led4,OUTPUT);
pinMode(led5,OUTPUT);
pinMode(led6,OUTPUT);
pinMode(led7,OUTPUT);
pinMode(led8,OUTPUT);
}
void loop() {
digitalWrite ( led1, HIGH);
delay (500);
digitalWrite ( led1, LOW);
delay (500);
digitalWrite ( led2, HIGH);
delay (500);
digitalWrite ( led2, LOW);
delay (500);
digitalWrite ( led3, HIGH);
delay (500);
digitalWrite ( led3, LOW);
delay (500);
digitalWrite ( led4, HIGH);
delay (500);
digitalWrite ( led4, LOW);
delay (500);
digitalWrite ( led5, HIGH);
delay (500);
digitalWrite ( led5, LOW);
delay (500);
digitalWrite ( led6, HIGH);
delay (500);
digitalWrite ( led6, LOW);
delay (500);
digitalWrite ( led7, HIGH);
delay (500);
digitalWrite ( led7, LOW);
delay (500);
digitalWrite ( led8, HIGH);
delay (500);
digitalWrite ( led8, LOW);
delay (500);
}
```

## 5. Model Traffic Signal

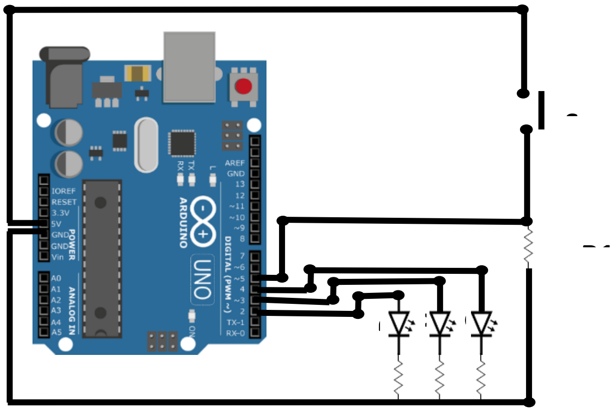
Step 1 : Introduction

So now we know how to set a digital pin to be an input, we can build a project for model traffic signals using red, yellow, and green LEDs. Every time we press the button, the traffic signal will go to the next step in the sequence. In the UK, the sequence of such traffic signals is red, red and amber together, green, amber, and then back to red. As a bonus, if we hold the button down, the lights will change in sequence by themselves with a delay between each step.

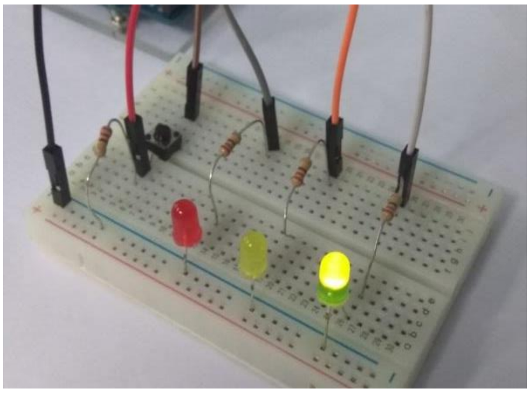
Step 2 : Components And Equipment

Arduino UNO
4x 220 ohm resistor
hook-up wires
breadboard
red LED
yellow LED
green LED
1x push switch

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**



**Step 5 : The Code**

```
int redPin = 2;
int yellowPin = 3;
int greenPin = 4;
int buttonPin = 5;
int state = 0;
void setup()
{
pinMode(redPin, OUTPUT);
pinMode(yellowPin, OUTPUT);
pinMode(greenPin, OUTPUT);
pinMode(buttonPin, INPUT);
}
void loop()
{
if (digitalRead(buttonPin))
{
if (state == 0)
{
setLights(HIGH, LOW, LOW);
state = 1;
}
else if (state == 1)
{
setLights(LOW, HIGH, LOW);
state = 2;
}
else if (state == 2)
{
setLights(LOW, LOW, HIGH);
state = 3;
}
else if (state == 3)
{
setLights(LOW, HIGH, LOW);
state = 0;
}
delay(1000);
}
}
void setLights(int red, int yellow,
int green)
{
digitalWrite(redPin, red);
digitalWrite(yellowPin, yellow);
digitalWrite(greenPin, green);
}
```

## 6. Model Traffic Signal (auto)

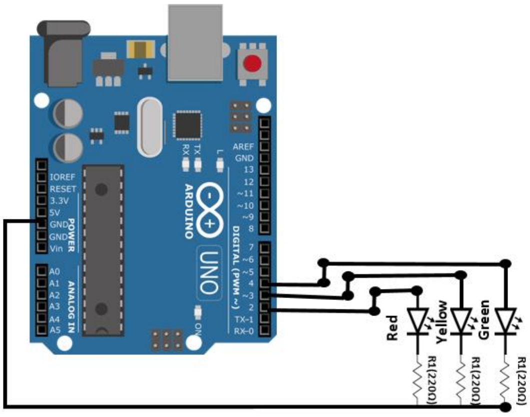
### Step 1 : Introduction

So now we know how to set a digital pin to be an input, we can build a project for model traffic signals using red, yellow, and green LEDs. Every time we press the button, the traffic signal will go to the next step in the sequence. In the UK, the sequence of such traffic signals is red, red and amber together, green, amber, and then back to red. As a bonus, if we hold the button down, the lights will change in sequence by themselves with a delay between each step.

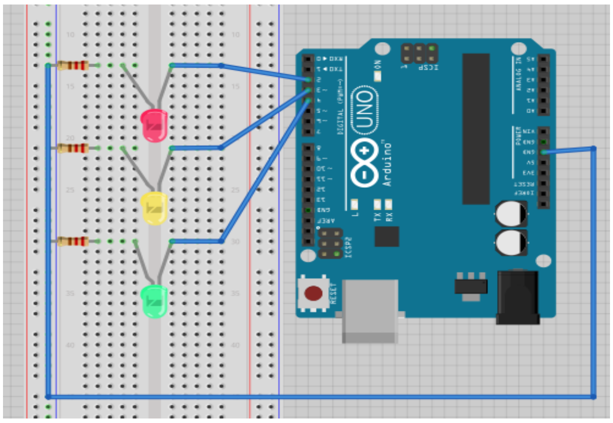
### Step 2 : Components And Equipment

Arduino UNO
3x 220 ohm resistor
hook-up wires
breadboard
red LED
yellow LED
green LED

### Step 3 : Circuit Diagram



### Step 4 : Diagram



**Step 5 : The Code**

```
int red = 2;
int yellow = 3;
int green = 4;
int x;
void setup()
{
pinMode(red, OUTPUT);
pinMode(yellow, OUTPUT);
pinMode(green, OUTPUT);
}
void loop()
{
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
delay(500);
digitalWrite (red, LOW);

digitalWrite (yellow, LOW);
digitalWrite (green, HIGH);
delay(500);
digitalWrite (green, LOW);

for(x=1; x<=4;x++)
{
digitalWrite (yellow, LOW);
delay(500);
digitalWrite (yellow, HIGH);
delay(500);
}
}
```

## 7. Model Traffic Signal (auto 2 traffic light)

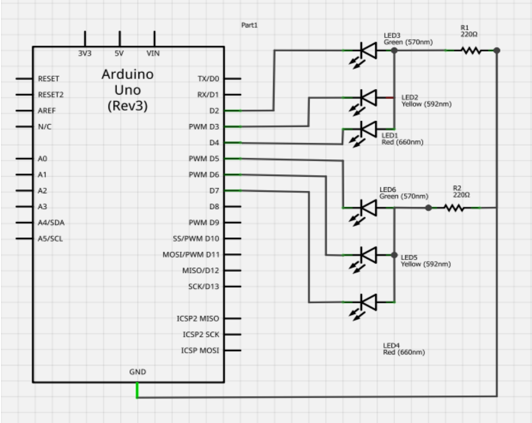
**Step 1 : Introduction**

So now we know how to set a digital pin to be an input, we can build a project for model traffic signals using red, yellow, and green LEDs. Every time we press the button, the traffic signal will go to the next step in the sequence. In the UK, the sequence of such traffic signals is red, red and amber together, green, amber, and then back to red. As a bonus, if we hold the button down, the lights will change in sequence by themselves with a delay between each step.

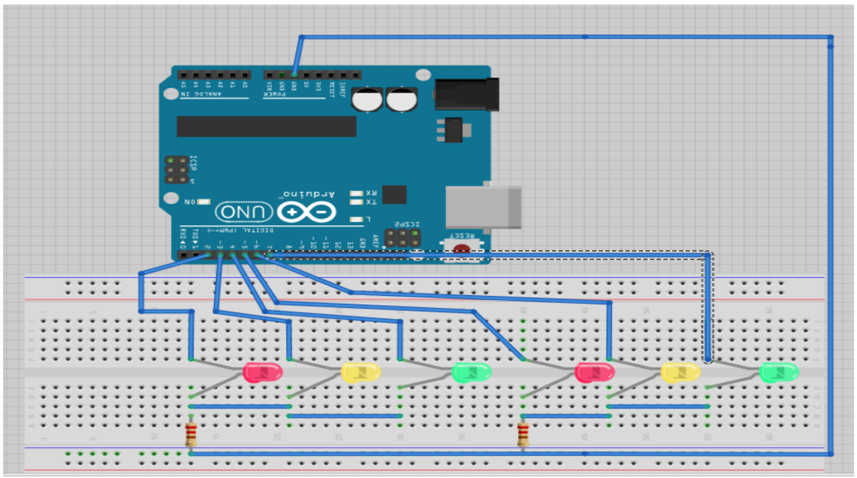
**Step 2 : Components And Equipment**

Arduino UNO
2x 220 ohm resistor
hook-up wires
breadboard
2x red LED
2x yellow LED
2x green LED

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**



**Step 5 : The Code**

```
int red = 2;
int yellow = 3;
int green = 4;
int red2 = 5;
int yellow2 = 6;
int green2 = 7;
int x;
void setup()
{
pinMode(red, OUTPUT);
pinMode(yellow, OUTPUT);
pinMode(green, OUTPUT);
pinMode(red2, OUTPUT);
pinMode(yellow2, OUTPUT);
pinMode(green2, OUTPUT);

}
void loop()
{
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
delay(1000);
digitalWrite (red2, LOW);
digitalWrite (green2, HIGH);
delay(5000);
digitalWrite (green2, LOW);
for(x=1; x<=5;x++)
{
digitalWrite (yellow2, LOW);
delay(500);
digitalWrite (yellow2, HIGH);
delay(500);
}
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
digitalWrite (green2, LOW);
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
delay(1000);
digitalWrite (red, LOW);
digitalWrite (green,HIGH);
delay(5000);
digitalWrite (green,LOW);

for(x=1; x<=5;x++)
{
digitalWrite (yellow, LOW);
delay(500);
digitalWrite (yellow, HIGH);
delay(500);
}
}
```

## 8. Model Traffic Signal (auto 3 traffic light)

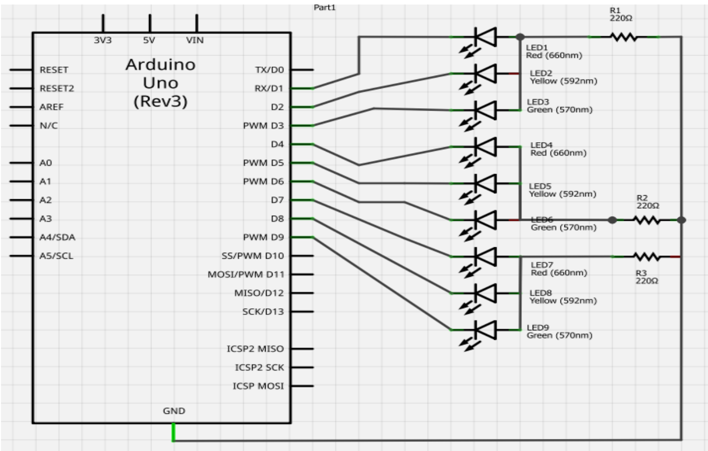
**Step 1 : Introduction**

So now we know how to set a digital pin to be an input, we can build a project for model traffic signals using red, yellow, and green LEDs. Every time we press the button, the traffic signal will go to the next step in the sequence. In the UK, the sequence of such traffic signals is red, red and amber together, green, amber, and then back to red. As a bonus, if we hold the button down, the lights will change in sequence by themselves with a delay between each step.

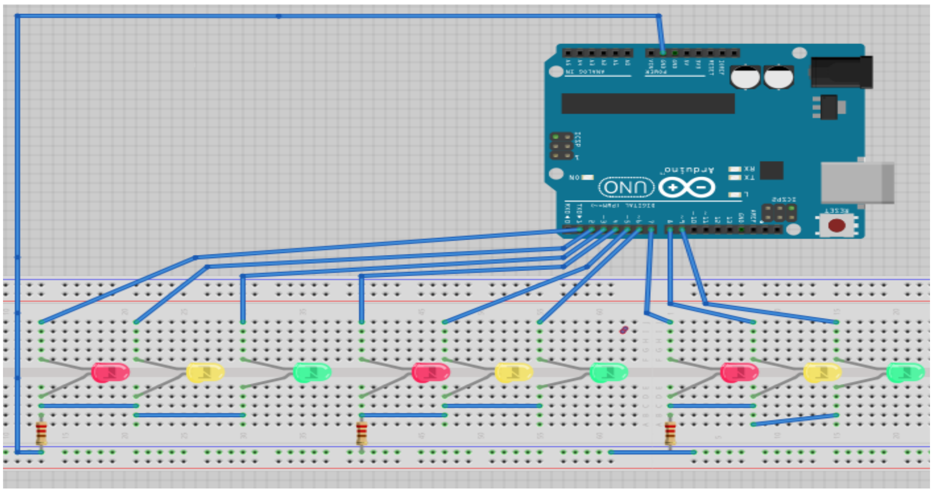
**Step 2 : Components And Equipment**

Arduino UNO
3x 220 ohm resistor
hook-up wires
breadboard
3x red LED
3x yellow LED
3x green LED

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**



**Step 5 : The Code**

```
int red = 1;
int yellow = 2;
int green = 3;
int red2 = 4;
int yellow2 = 5;
int green2 = 6;
int red3 = 7;
int yellow3 = 8;
int green3 = 9;
int x;
void setup()
{
pinMode(red, OUTPUT);
pinMode(yellow, OUTPUT);
pinMode(green, OUTPUT);
pinMode(red2, OUTPUT);
pinMode(yellow2, OUTPUT);
pinMode(green2, OUTPUT);
pinMode(red3, OUTPUT);
pinMode(yellow3, OUTPUT);
pinMode(green3, OUTPUT);
}
void loop()
{
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
digitalWrite (green3, LOW);
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
```

```
delay(1000);
digitalWrite (red2, LOW);
digitalWrite (green2, HIGH);
delay(5000);
digitalWrite (green2, LOW);

//side ke 2
for(x=1; x<=5;x++)
{
digitalWrite (yellow2, LOW);
delay(500);
digitalWrite (yellow2, HIGH);
delay(500);
}
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
digitalWrite (green2, LOW);
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
delay(1000);
digitalWrite (red3, LOW);
digitalWrite (green3,HIGH);
delay(5000);
digitalWrite (green3,LOW);

//side ke 3

for(x=1; x<=5;x++)
{
digitalWrite (yellow3, LOW);
delay(500);
digitalWrite (yellow3, HIGH);
delay(500);
}

digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
digitalWrite (green3, LOW);
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
digitalWrite (green2, LOW);
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
delay(1000);
digitalWrite (red, LOW);
digitalWrite (green,HIGH);
```

```
delay(5000);
digitalWrite (green,LOW);
for(x=1; x<=5;x++)
{
digitalWrite (yellow, LOW);
delay(500);
digitalWrite (yellow, HIGH);
delay(500);
}
}
```

## 9. Model Traffic Signal (auto 4sides)

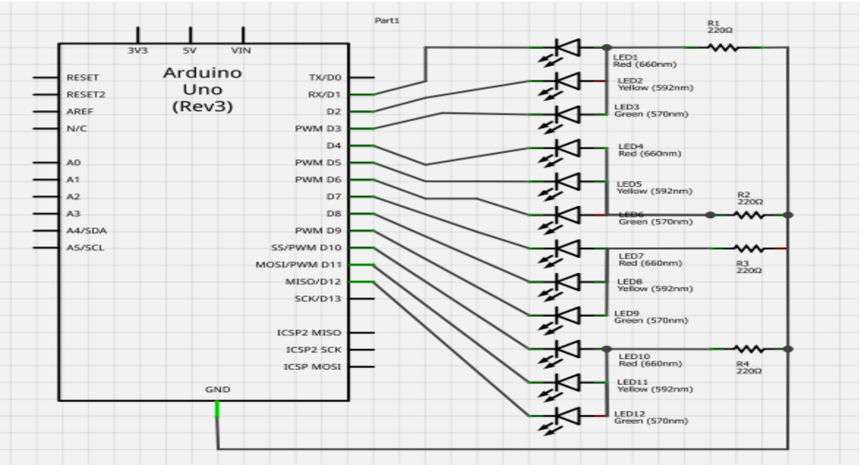
**Step 1 : Introduction**

So now we know how to set a digital pin to be an input, we can build a project for model traffic signals using red, yellow, and green LEDs. Every time we press the button, the traffic signal will go to the next step in the sequence. In the UK, the sequence of such traffic signals is red, red and amber together, green, amber, and then back to red. As a bonus, if we hold the button down, the lights will change in sequence by themselves with a delay between each step.

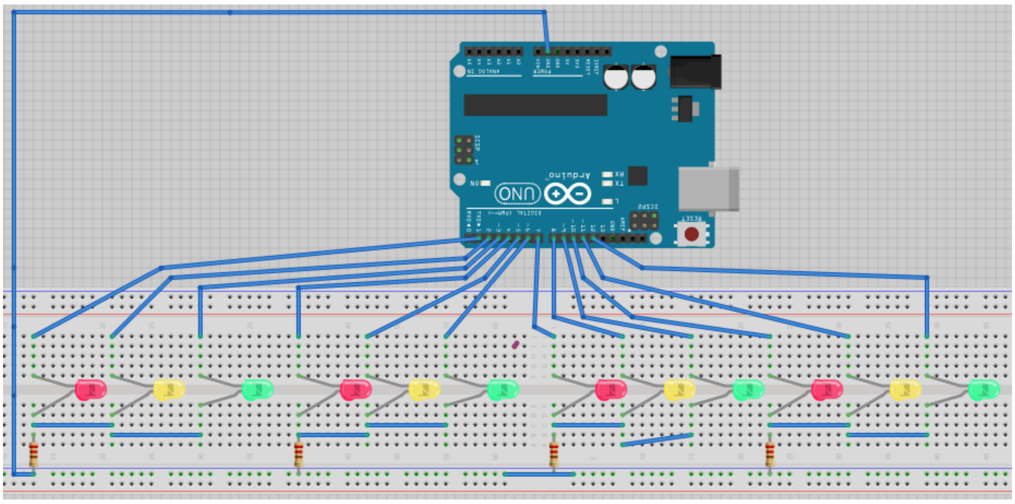
**Step 2 : Components And Equipment**

Arduino UNO
4x 220 ohm resistor
hook-up wires
breadboard
4x red LED
4x yellow LED
4x green LED

**Step 3 : Circuit Diagram**



**Step 4 : Diagram**



**Step 5 : The Code**

```
int red = 1;
int yellow = 2;
int green = 3;

int red2 = 4;
int yellow2 = 5;
int green2 = 6;

int red3 = 7;
int yellow3 = 8;
int green3 = 9;

int red4 = 10;
int yellow4 = 11;
int green4 = 12;

int x;
void setup()
{
pinMode(red, OUTPUT);
pinMode(yellow, OUTPUT);
pinMode(green, OUTPUT);
pinMode(red2, OUTPUT);
pinMode(yellow2, OUTPUT);
pinMode(green2, OUTPUT);
pinMode(red3, OUTPUT);
pinMode(yellow3, OUTPUT);
pinMode(green3, OUTPUT);
pinMode(red4, OUTPUT);
pinMode(yellow4, OUTPUT);
pinMode(green4, OUTPUT);
```

```
}
void loop()
{
 // side yg pertama
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
digitalWrite (green3, LOW);
digitalWrite (red4, HIGH);
digitalWrite (yellow4, LOW);
digitalWrite (green4, LOW);
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
delay(1000);
digitalWrite (red2, LOW);
digitalWrite (green2, HIGH);
delay(5000);
digitalWrite (green2, LOW);
//side ke 2
for(x=1; x<=5;x++)
{
digitalWrite (yellow2, LOW);
delay(500);
digitalWrite (yellow2, HIGH);
delay(500);
}
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
digitalWrite (green2, LOW);
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
digitalWrite (green, LOW);
digitalWrite (red4, HIGH);
digitalWrite (yellow4, LOW);
digitalWrite (green4, LOW);
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
delay(1000);
digitalWrite (red3, LOW);
digitalWrite (green3,HIGH);
delay(5000);
digitalWrite (green3,LOW);
//side ke 3
for(x=1; x<=5;x++)
{
digitalWrite (yellow3, LOW);
delay(500);
digitalWrite (yellow3, HIGH);
delay(500);
}
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
digitalWrite (red4, HIGH);
delay(1000);
digitalWrite (red4, LOW);
digitalWrite (green4,HIGH);
delay(5000);
digitalWrite (green4,LOW);
//side ke 3
for(x=1; x<=5;x++)
{
digitalWrite (yellow4, LOW);
delay(500);
digitalWrite (yellow4, HIGH);
delay(500);
}
digitalWrite (red4, HIGH);
digitalWrite (yellow4, LOW);
digitalWrite (green4, LOW);
digitalWrite (red3, HIGH);
digitalWrite (yellow3, LOW);
digitalWrite (green3, LOW);
digitalWrite (red2, HIGH);
digitalWrite (yellow2, LOW);
digitalWrite (green2, LOW);
digitalWrite (red, HIGH);
digitalWrite (yellow, LOW);
delay(1000);
digitalWrite (red, LOW);
digitalWrite (green,HIGH);
delay(5000);
digitalWrite (green,LOW);
for(x=1; x<=5;x++)
{
digitalWrite (yellow, LOW);
delay(500);
digitalWrite (yellow, HIGH);
delay(500);
}
}
```

## References

David, G & Dawn, G. (2012). Head First C: A Brain-Friendly Guide 1st Edition. O'Reilly Media.

Kernighan, Brian W.; Ritchie, Dennis M. (February 1978). The C Programming Language (1st ed.). Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3.

Knuth, D. (1998). The art of computer programming Volume 2 2nd edition. In Reading MA.

McGrath,M. (2018). C Programming in easy steps, 5th Edition. In Easy Steps Limited.

Perry,G.& Miller,D. (2015). C Programming Absolute Beginner's Guide 3rd Edition. Pearson Education.

Ritchie, Dennis M. (1993). "The Development of the C Language". History of Programming Languages, 2nd Edition. Retrieved 2018-11-11.

Schildt, H. ( 2018).The Complete Reference, 4th Edition. McGraw-Hill.

POLITEKNIK
MALAYSIA
KOTA BHARU

9789671663974