ENGINEERING @ SP

ET0083 / ET0809

STRUCTURED PROGRAMMING

(Version 3.6.7)

School of Electrical & Electronic Engineering

Engineering @ SP

The Singapore Polytechnic's Mission

Life Ready. Work Ready. World Ready.

A future-ready institution that prepares our learners to be life ready, work ready and world ready.

The Singapore Polytechnic's Vision

Inspired Learner. Serve with Mastery. Caring Community.

A caring community of inspired learners committed to serve with mastery.

The SP CORE Values

- Self-Discipline
- Personal Integrity
- Care & Concern
- Openness
- Responsibility
- Excellence

For any queries on the notes, please contact:

Name: Lau Shook Fong
Room: T946
Email: lau_shook_fong@sp.edu.sg
Tel: 67721445

SINGAPORE POLYTECHNIC | SP

SP | SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

Industry-Collaborated TECH HUB

Diploma in Aerospace Engineering

Aerospace Engineering Hub

Themed Project : UAV-augmented Smart Environment
Facilities : A320 Flight Simulator
Industries : Aerospace, Air Transport

Industry Partners : AIRBUS, Rolls-Royce

Industry Co-location Lab : ST Engineering Aerospace

Diploma in Computer Engineering

AIoT Technology Hub

Themed Project : Smart Building & Facilities
Facilities : AIoT Nerve Centre, SP-TIBCO Adv Mfg Analytics Lab, SP Connected Solutions Lab, 5G Garage
Industries : ICT & Media, Smart City

Industry Partners : ADVANTECH, CONTEC, ENVISION, HUAWEI

Industry Co-location Lab : BOSCH Invented for life, ERICSSON, Singtel, TIBCO

Diploma in Engineering with Business

Technology to Business Hub

Themed Project : Technology To Business Venture
Facilities : IdeaBox, T2B Hub
Industries : Entrepreneurship & Start-up

Industry Co-location Lab : SPgroup, GEX VENTURES

APOL, BevEat, ItsGreat, Codesurence, MEMORY APE, OCTOPUS8, Pocketbot, REAL IMPACT, SMART X8, TOP TASTE, Vaster

T18A

T9 level 3 & T14 level 3

T14 level 3

T10 level 3

T8 level 2 & T12 level 1

T12/T12A level 2

T12 level 6

T8 level 2 & T12 level 1

Diploma in Electrical & Electronic Engineering

Robotics, Automation & Control Hub

Themed Project : Smart Workflow Solution for I4.0
Facilities : SP-SICK Smart Sensor Lab, SP-Rexroth I4.0 Innovation Lab
Industries : Logistics, Electronics (Manufacturing), Electronics & Chemicals (Manufacturing)

Industry Partners : ABB, BECKHOFF, SEW EURODRIVE, Weidmüller, ZEBRA

Industry Co-location Lab : rexroth A Bosch Company, SICK, ZEBRA

Power & Sustainable Energy Hub

Themed Project : Zero-Energy Space
Facilities : High Voltage Training, LE & LET Training, Energy Efficiency Lab
Industries : Energy & Power

Industry Partners : CHINT, ebmpapst, SUNLIGHT A CHINT COMPANY

Semicon Technology Hub

Themed Project : Semicon-Enabled AI Applications
Facilities : Nanofab & IC Design Labs, Infineon Innovation Lab
Industries : Semiconductor & Electronics

Industry Co-location Lab : Infineon, LG, SSIA

Biomedical Engineering Hub

Themed Project : Engaging Rehabilitation
Facilities : Biomedical Engineering Lab
Industries : Healthcare

Industry Partners : GE Healthcare, GOLD LITE, Khoo Teck Puat Hospital, NATIONAL INSTRUMENTS

Smart Urban Transportation Hub

Themed Project : Autonomous EV / Rapid Simulator System
Facilities : RSS Lab for RTT Training, Siemens Digital Experience Centre
Industries : Smart Transport, Land Transport

Industry Partners : ST Engineering Land Systems, SMRT, SIIX, SUTD, GreenPower

Industry Co-location Lab : Land Transport Authority, SIEMENS Ingenuity for life

CONTENTS

MODULE OVERVIEW

1 Introduction

This module teaches the basic concept and the fundamentals of computer programming.

2 Module Aims

The module introduces key concepts of structured programming techniques using C++ as the programming language. At the end of the module, students should gain the knowledge of the fundamentals of computer programming and enable them to proceed to other module(s) in programming in subsequent years.

Chapter 1

Computer Basics

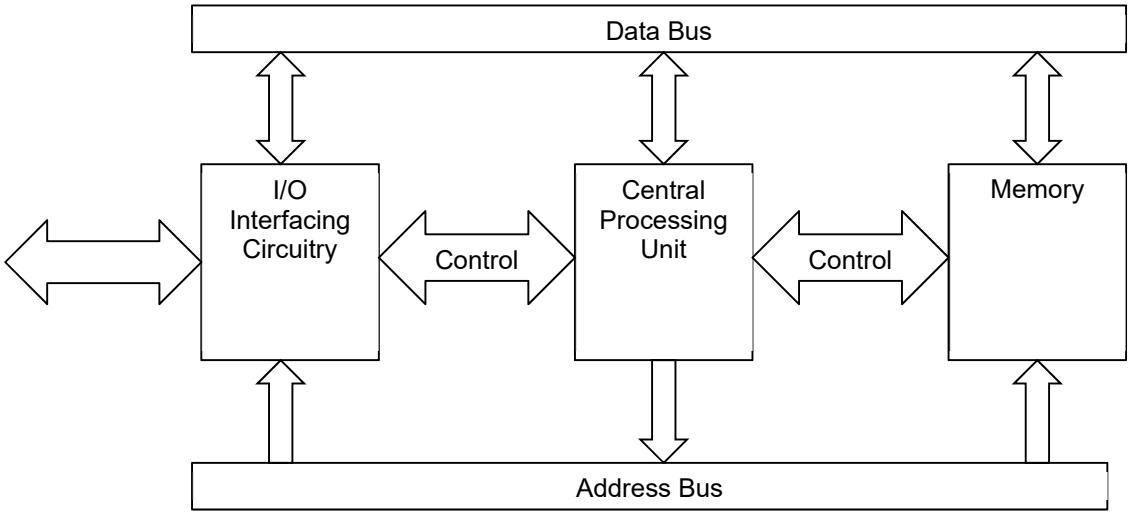
Introduction

- A computer system is made up of ***Hardware*** and ***Software***.

- The ***hardware*** is the actual physical devices of the computer system such as processing units, keyboard, screen, "mouse", memory, CD-ROM, etc. It can do nothing by itself.

Basic Computer Architecture

The basic computer architecture consists of four main parts:



Block diagram of a simple computer system

- The Central Processing Unit (CPU) – this is the "brain" of the whole system.

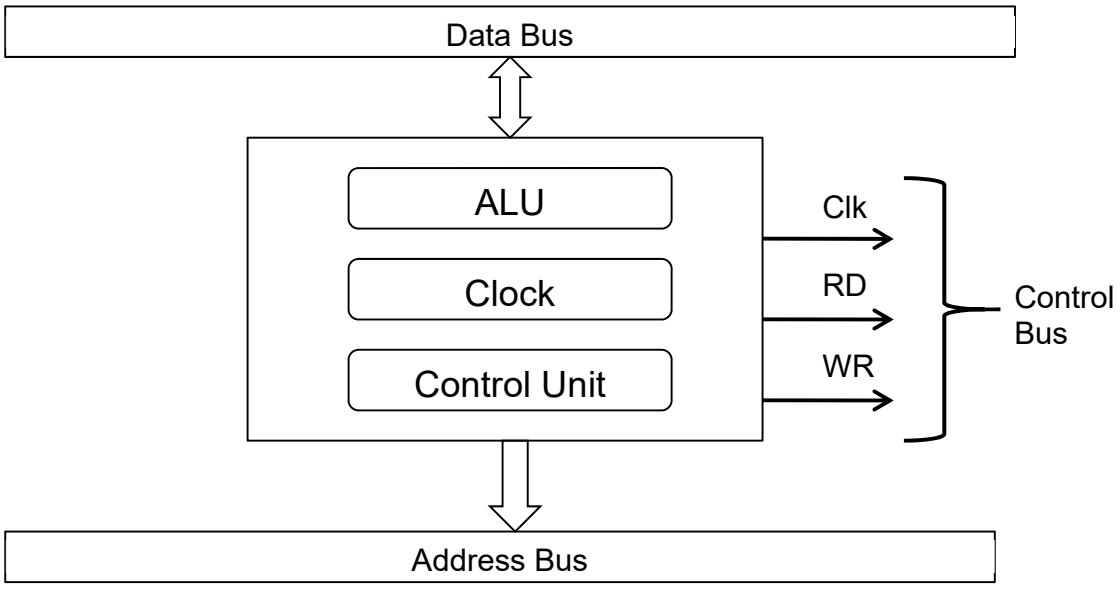
- The Memory – for storage of programs and data.

- The I/O Interfacing Circuitry – this is the interface between the computer system and the external devices in the outside world.

- The Three-Bus System: the ***address, data*** and ***control*** bus are used to transfer signal between the different parts of the computer system. (*A bus is a bundle or collective of hardware signal lines)*

The Central Processing Unit (CPU)

- It controls system operations.
- Usually consist of 3 subsystems: Arithmetic Logic Unit, Clock and Control Unit
- Arithmetic Logic Unit (ALU) performs all the arithmetic and logic operations specified in the program code.
- Clock synchronizes the timing between each subsystem, external memories and devices.
- Control Unit controls the operation of the subsystems and the external memories and devices.



Block diagram of the Central Processing Unit

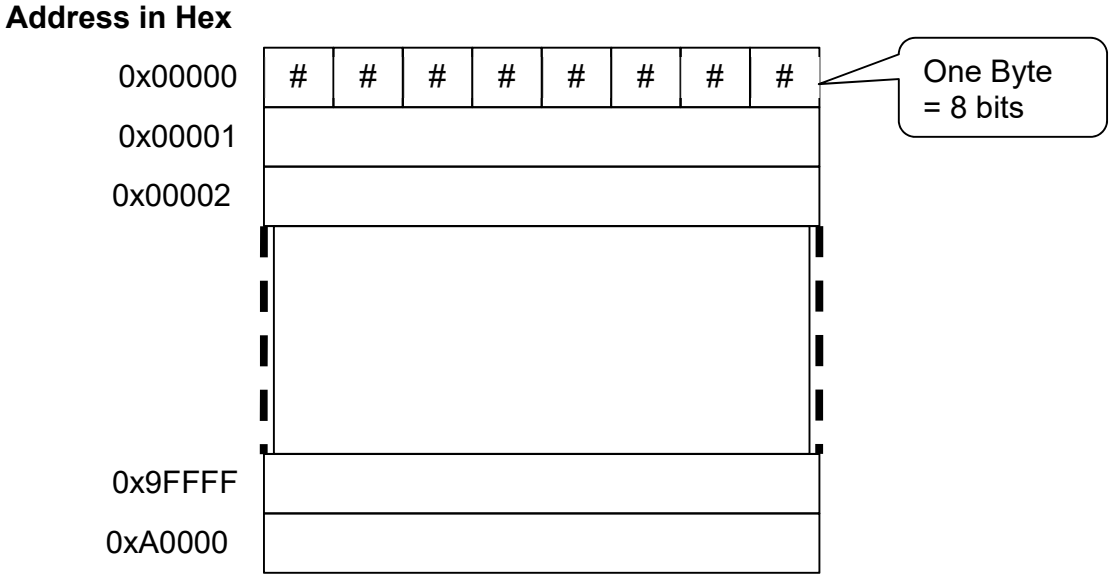
The Memory

- Memory is used to store program and data.

- There are two types of memory.

  - Primary Memory consist of
    - Random Access Memory (RAM): this location is volatile – information stored here will be lost when power to the system is removed.
    - Read Only Memory (ROM): this location is non-volatile – information stored here will not be lost even if power is removed.

  - Secondary Memory consist of
    - Hard disks
    - CD-ROMs
    - DVD-ROMs
    - Magnetic tapes
    - Thumb drives
    - Flash drives.. etc

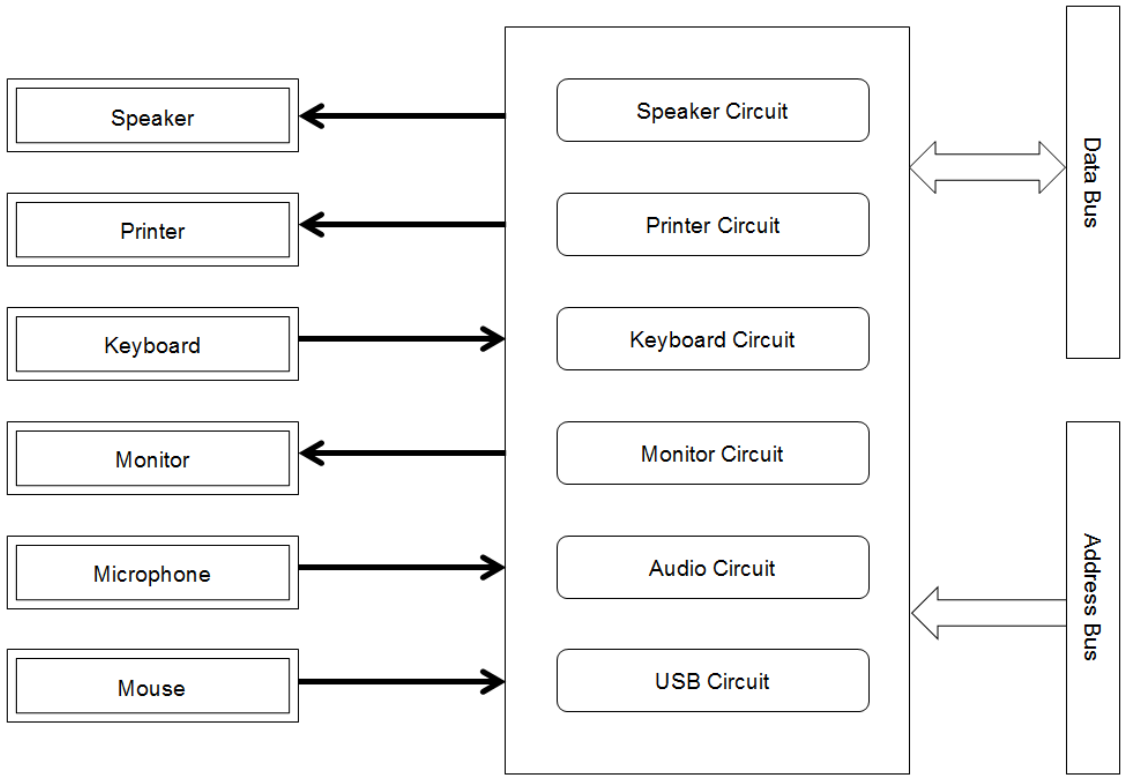


The main memory structure of a typical 640K system

Note that 1KB = 1024 Bytes

The Input / Output (I/O) Circuitry

- Subsystem of a computer system for connection to external devices.
- All signals from external devices passes through I/O circuitry before being transferred by data or control bus to the CPU or memory.

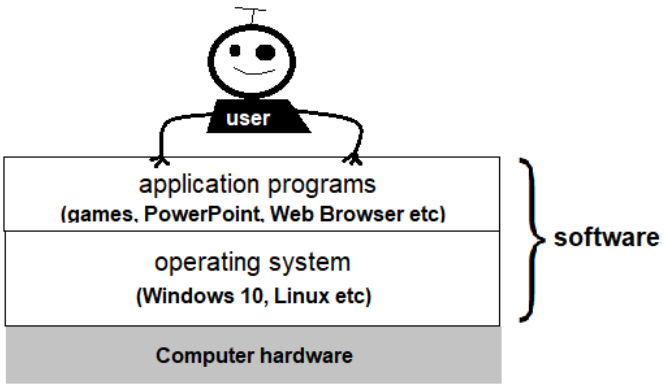


Various external devices connected to the I/O Circuitry

Requirement for Software

- The ***hardware*** requires to be given instructions that tell it what to do.

- A set of instructions that tell the computer system what to do is called a **Computer Program or Application Program.**

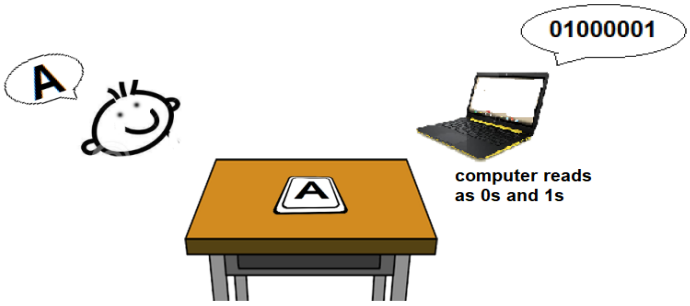


- Computer Programs are written in different languages.

- Computer Languages can be grouped into three categories :
  - Machine Language
  - Assembly Language
  - High-Level Language

Machine Language

- The hardware of a Computer can only understand binary data i.e. 0 and 1.
- A computer is designed to understand a given set of instructions, where each instruction has a Binary Code.
- The alphabets and numbers that we use every day are being interpreted as Binary Codes by computers.



- This Binary Code is called Machine Code.
- A set of machine-coded instructions is called Machine Language.

The table below shows some letters and their ***unique*** representations in ***binary codes*** (machine codes). *(The full set of representations is called the **ASCII code**, which can be found in Appendix.)*

Keyboard letter	Binary representation
A	**01000001**
B	**01000010**
C	**01000011**
D	**01000100**
E	**01000101**
F	**01000110**
G	**01000111**

Keyboard letter	Binary representation
a	**01100001**
b	**01100010**
c	**01100011**
d	**01100100**
e	**01100101**
f	**01100110**
g	**01100111**

- All programs eventually have to be mapped/converted into machine codes (or called binary/object codes) that make sense for the hardware to run.

Assembly Language

- Assembly language is one level above Machine Language.

- Instead of using binary data (0 and 1), it uses alphabetic abbreviations called ***Mnemonics*** that can be easily remembered.

Examples of mnemonics:

ADD for addition
SUB for subtraction
CMP to compare

```
10  main PROC
11      mov     eax, 3
12      mov     ebx, 8
13      add     eax, ebx
14      INVOKE ExitProcess, 0
15  main ENDP
```

assign value 3 to register eax

- The instructions manipulate the CPU registers (inside CPU hardware) directly.

- Although more easily understood by humans, they cannot be directly understood by the Computer.

- Therefore, an Assembly Language Program must be translated into Machine Code by an **ASSEMBLER.**

- Although Assembly Language is easier to use than Machine Language, it is still tedious and prone to error.

- The solution to these problems is found in High-Level Languages.

- You probably will not find yourself writing your next program in assembly language.

- Today, assembly language is used primarily for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

High-Level Language

- It consists of statements that are similar to English and uses everyday mathematical notations.

```
if ( temperature > 30 ) turn_on_aircon();
```

- A single High-Level Language statement is equivalent to many Machine Code instructions.

- Once you learn a High-Level Language, you can program any Computer that supports that language.

- Your program, written in high-level language, cannot be understood and run by the computer. **It must be translated to Machine Code.**

- Two types of System programs can be used to do this translation -
  - a **COMPILER**
  - an **INTERPRETER.**

- A **Compiler** is a program that accepts a High-Level Language program and translates the entire program into Machine Code before it is executed.

- An **Interpreter** translates and executes the High-Level Language statement one at a time.

- The program you write is often called the ***Source Code*** and usually in text format, i.e. can be open and read by text editors, such as ***notepad etc.***

- The Compiler also checks for errors in the Source Code.

- After all the errors have been corrected, the Compiler converts the **Source code** into an **Object code** and then links it with other Object codes to form an **Executable Code** which is executed by the computer.

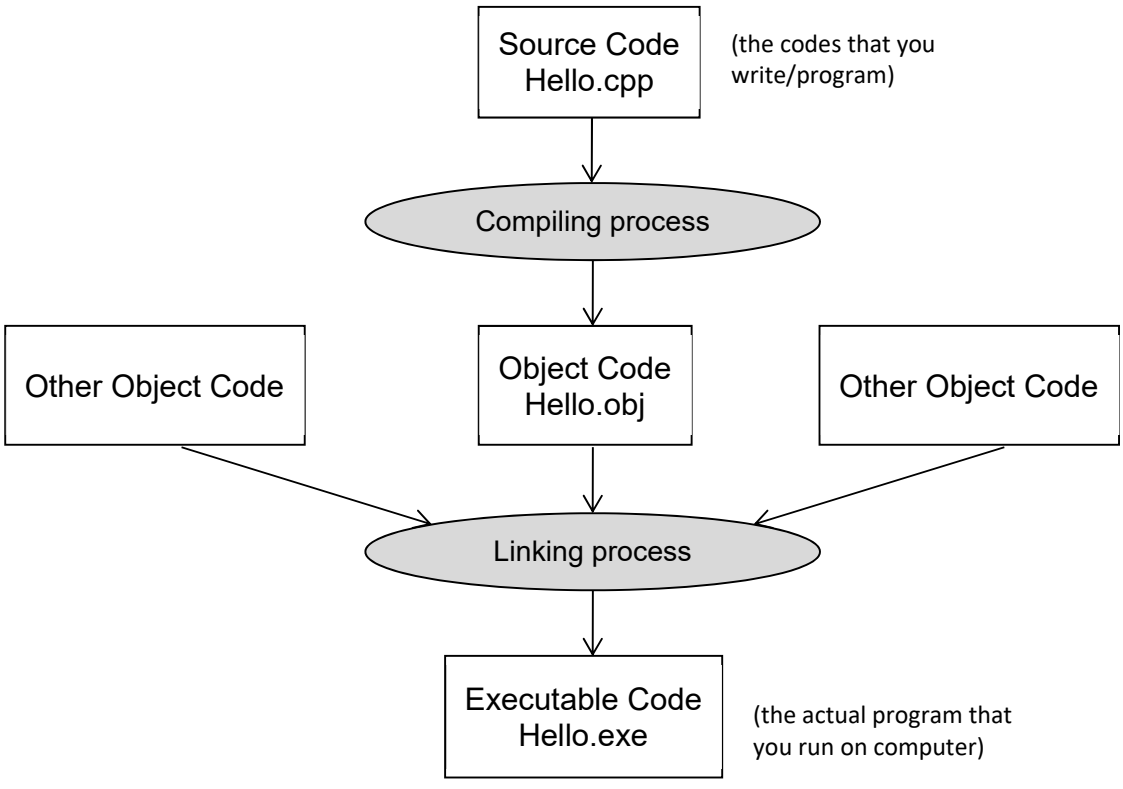
- You do not have to write all the codes yourself. You can make use of some of the existing codes and combine (or link) them with your own codes. That is the purpose of the '***Linking process'***.

- C++ is the High-Level Language used in this Module. This language is derived from C Programming Language.

- C was first developed by Dennis Ritchie at the AT & T Bell Laboratories for the implementation of the UNIX Operating System.

- C is a very popular language used in industries today, especially for engineering applications.

- Other popular High-Level Languages are: Java, Python, Java Script etc.



Designing Programs

- The design process of a Computer Program has six stages/steps. These are:
  1. Write the **problem statement**.
  2. Define the **Inputs, Computation (Processing) and Outputs**.
  3. Generate **test data**.
  4. Write the **Algorithm** using Pseudo-Code or Flow Chart.
  5. Translate the Algorithm into a Program (**Coding**).
  6. **Compile, debug and test** the program

- Of the above six steps, the three **major** steps are:
  - Define the **Inputs, Computation (Processing) & Outputs.**
  - Write the **Algorithm** using Pseudo-Code or Flow Chart
  - Translate the Algorithm into a Program (**Coding**).

- Computers are used to solve problems.

- It is important **to understand and analyze the problem and find out what the problem really is.**

- The best way to do this is to find out the ***Inputs*** required for the program, ***Computation*** to be done and the ***Outputs*** required from the program.

Algorithms

- An ***algorithm*** is the method to solve a problem.

- An **algorithm** can be expressed as a set of instructions/activities for accomplishing a task.

- We deal with **algorithms** for doing everyday things

  e.g.
  Preparing our favourite dish
  Building a DIY PC

- All these activities can be described by a set of instructions that specify how to accomplish a desired goal.

- When using computers to solve a problem, the set of instructions must be written in a very precise and logical way.

- The computer will do exactly what it is told to do - nothing more, nothing less.

- Algorithms can be expressed in two ways – **Flowchart** or **Pseudocode.**

- Flowchart is a graphical technique to represent the algorithm, while Pseudocode is a textual representation.

Flowcharts

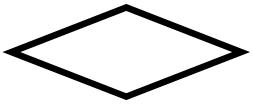
- Flowcharts use a set of graphical symbols to represent the flow of actions in a program.
- Flowcharts are more easily and quickly understood than pseudocode as they have more visual impact.
- However, flowchart is not so convenient when frequent modifications have to be made to an algorithm.
- In complex program designs, a combination of both is often used.

Common Flowchart Symbols

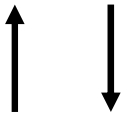
Start and End of Module or Function (Terminator)

Processing Block

Input and Output

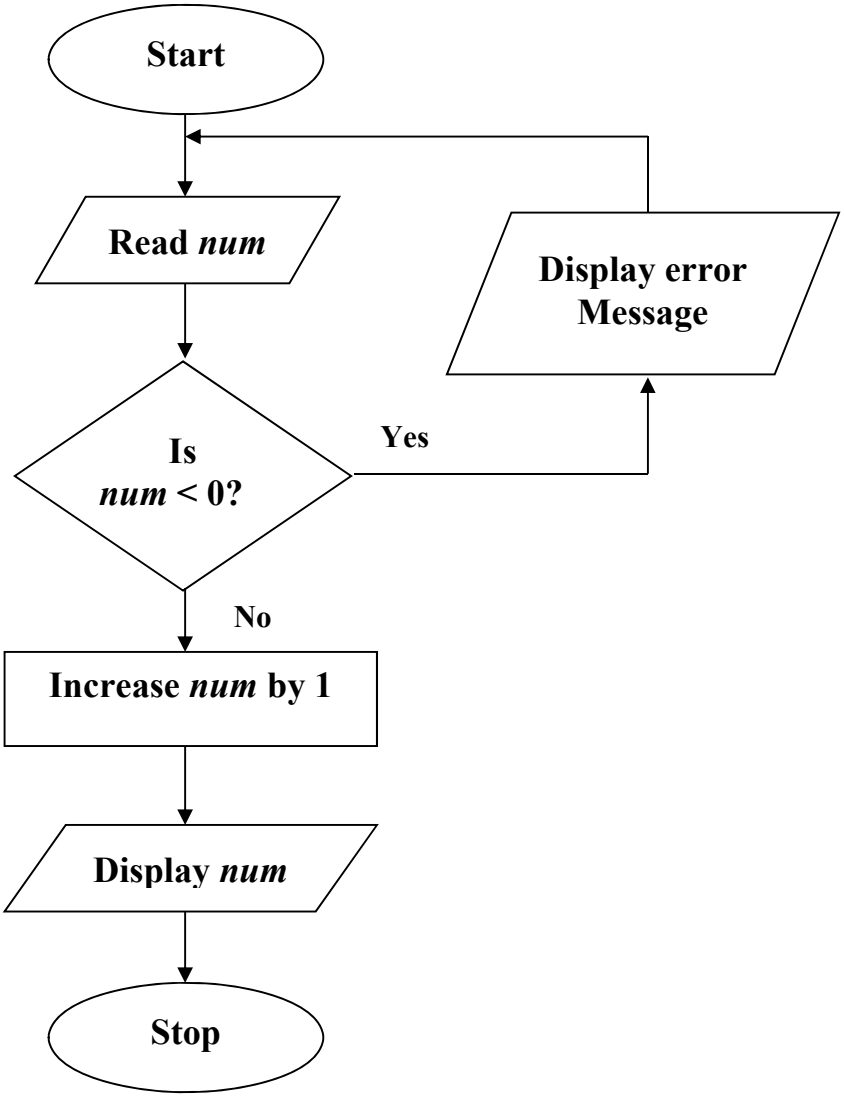


Decision



Flow Direction

Example:


Start
Read num
Display error Message
Is num < 0?
Yes
No
Increase num by 1
Display num
Stop

Writing algorithms using Pseudocode

Design a program to calculate a person's age in months, and then in days. You are given the age in years.

- What are the inputs needed by the program?
- What does the program have to calculate?
- What results must the program output?

These questions can be answered by the 2nd Design Step.

2nd Step : Define the Inputs, Computation and Outputs

Inputs	:	Name Age (in Years)
Computation (Processing)	:	Age in Months = 12 x Age Age in Days = 365.25 x Age
Outputs	:	Name Age (in years) Computed Values

- Test values, calculated by hand, may be included in the Computation Process whenever possible.

4th Step : Write the Algorithm using Pseudo-Code

- Pseudo-Code is just an informal way of writing an algorithm.

- We will be using the convention below for writing algorithms in Pseudo-Code.

1.0 Major Step 1
2.0 Major Step 2
3.0 Major Step 3

- As more detail is added to each step, we could indent and number as follows :

1.0 Major Step 1

1.1 Detail 1 of Step 1
1.2 Detail 2 of Step 1
:

2.0 Major Step 2

2.1 Detail 1 of Step 2
2.2 Detail 2 of Step 2
:

3.0 Major Step 3

3.1 Detail 1 of Step 3
:

- The first level of Pseudo-Code for the above problem would therefore be as follows :

1.0 Read Name and Age
2.0 Convert Age to months and to days
3.0 Write Name, Age and Computed Values.

- We could now add some details to the major steps listed above to obtain the second level pseudo-Code :

1.0 Read Name and Age

1.1 Read Name
1.2 Read Age

2.0 Convert Age to months and to days

2.1 AgeInMonths = 12 x Age
2.2 AgeInDays = 365.25 x Age

3.0 Write Name, Age and Computed Values

3.1 Write Name, Age
3.2 Write AgeInMonths, AgeInDays

- The above Algorithm may be refined further, adding still more details if necessary.

- This process of refinement by adding successive layers of details is known as **Top-Down Program Design.**

- Once the algorithm has been written as shown above, it is very easy to perform the fifth step and translate it into a program.

Design an algorithm to find the Area of a Right-Angled triangle.

- Remember the three **major** steps in designing programs?

  a. Define the **Inputs, Processing (Computation) and Outputs**
  b. Write the **Algorithm** using Pseudo-Code or Flowchart
  c. Translate the Algorithm to a Program (**coding**)

2nd Step : Define the Inputs, Computation and Outputs

Inputs :_____

Computation (Processing) :_____

Outputs :_____

4th Step : Write the Algorithm using Pseudo-Code

1.0 _____

1.1 _____

1.2 _____

1.0 _____

1.1 _____

1.2 _____

3.0 _____

1.1 _____

1.2 _____

5th Step : Translate Algorithm into a program (next chapter)

Chapter 2

Input / Output & Data Types

Introduction To C++

- ***What is a Computer Program?***

  A computer Program is a set of instruction telling the computer what to do.

- ***Why do we need to tell the computer what to do?***

  Because a computer is only a machine that is unable to think for itself. It does exactly what you tell it to do.

- ***Why do we need to learn a Programming Language?***

  Because computers do not understand English. You must therefore learn to give instructions in a language your computer recognizes.

- C++ is currently one of the most popular and widely used programming languages, especially for Engineering Applications. This is due to its **efficiency, flexibility and portability.**

Your First C++ Program

- A program needs to be able to send information to the screen **(output)** and take in information from the keyboard **(input)**.

- **Input** and **output** are important features of any program.

- The following program outputs the sentence

  Hello World!

  to the screen.

```
#include <iostream>
using namespace std;


 int main()
{
     cout << "Hello World!" ;
     return 0;
}
```

Output of the program:

```
Hello world!
```

- The ***#include <iostream>*** is a pre-processor directive which directs the compiler to include the **standard input/output header file** (iostream.h) into your program. This include is required because of the **cout** object. C++ uses **namespaces** to organize different names used in programs. Every name used in the iostream standard library file is part of a namespace called **std**.

- All C++ programs have a ***main()*** function which defines the entry point into the program. A C++ program starts executing from ***main().***

- The statement terminator (;) is used to end a line of code (or statement).

- The braces ({ }) show the beginning ({) and end (}) of a block of code.

- The above are just some of the many 'rules' that C++ enforces. In fact, all programming languages have their own 'rules'.

```
#include <iostream>
using namespace std;


 int main()
{
        cout << "Hello World!" ;
        return 0;
}
```

try removing any of these and see what would happen

Data Types

- All programs operate on Data (information). In fact, computers are used for the very purpose of processing data.
- Data can be in the form of **Variables** and **Constants**.
- If an item of data is declared as a variable, then it can be changed.
- If it is declared as a constant, then it cannot be changed.
- C++ can handle many types of data. However, we will begin by looking at the three fundamental data types, namely:
  1. The **integer** data type (keyword: **int**)
  2. The **floating point** data type (keyword: **double**)
  3. The **character** data type (keyword: **char**)

Integer

- Integers are whole numbers. An integer is therefore any number that does not contain a decimal point.

  Examples of integers are:

  45 -932 0 12 5421

- Integers typically range from INT_MIN to INT_MAX, which are defined by the compiler. (-2^{31} to $2^{31}-1$)
  which is -2,147,483,648 to +2,147,483,647.
- Whole numbers are for information where fractions do not apply. For example, how many students are there in your class?
  The answer could be 19 or 20, but not 19.35.
- For example,

```
 int main()
{
    int students;
    int car;
}
```

Floating Point

- A floating point number contains decimal places or is written using scientific notations.

  Examples of floating point numbers are:

  45.12 -2344.5432 0.00 0.04593

- A floating point number with double precision typically ranges from

  1.7E +/- 308 (15 digits)

  (1.7E+308 means $1.7 * 10^{+308}$)

- Double types are floating-point numbers where the value to the right of the decimal point is important. The term floating point comes from the fact that there is no fixed number of digits before or after the decimal point, that is the decimal point can float. Floating-point numbers are also called real numbers.

- Floating-point numbers can be typed as float or double. In this course, we will use double because it is the default data type.

- For example,

```
 int main()
{
    double weight;
    double height;
}
```

Character

- Characters are letters (a..z, A..Z), digits (0..9) or any of the other symbols (e.g. +, $, > , [etc) found on your keyboard. **A Space is also a character.**

- **A character variable can hold only a single character.**

- However, since computers can only deal with numbers, every character is stored as a number in the memory of the computer.

  For example:

  Decimal

  The character 'A' is stored as the number 65 (41 hex)
  The character 'B' is stored as the number 66 (42 hex)

- Every character is represented by a standard numeric code called the **"American Standard Code for Information Interchange"** or **ASCII.**
- The ASCII table contains the commonly used character set.

DEC	CHR	DEC	CHR	DEC	CHR	DEC	CHR	DEC	CHR
48	0	64	@	80	P	96	`	112	p
49	1	65	A	81	Q	97	a	113	q
50	2	66	B	82	R	98	b	114	r
51	3	67	C	83	S	99	c	115	s
52	4	68	D	84	T	100	d	116	t
53	5	69	E	85	U	101	e	117	u
54	6	70	F	86	V	102	f	118	v
55	7	71	G	87	W	103	g	119	w
56	8	72	H	88	X	104	h	120	x
57	9	73	I	89	Y	105	i	121	y
58	:	74	J	90	Z	106	j	122	z
59	;	75	K	91	[	107	k	123	{
60	<	76	L	92	\	108	l	124	|
61	=	77	M	93	]	109	m	125	}
62	>	78	N	94	^	110	n	126	~
63	?	79	O	95	_	111	o	127	DEL

You can refer to the standard ASCII Table in Appendix B.

- For example,

```
 int main()
{
    char grade;
    char gender;
}
```

Declaring Variables

- ***Why do we need to declare variables?***

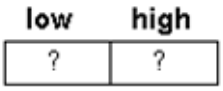
  When we declare variables, we are reserving space to store data in the memory of the computer.

- Different types of variables occupy different amount of memory.

char	:	1 byte
int	:	4 bytes
double	:	8 bytes

- Suppose we want to declare two integer variables *low* and *high*, we would declare them as follows :

```
#include <iostream>
using namespace std;

int main()
{
     int low;
     int high;
     return 0;
}
```



- The above declaration reserves space to hold two integer variables in the memory of the computer.

- The above variable declarations can also be done in one line:

```
int low; int high;
     or
int low, high;
```

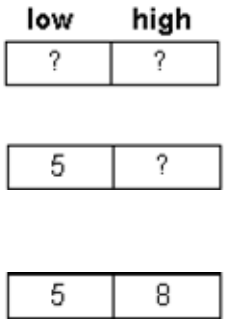
- However, since we have not given any values to the two variables, we have no idea what values they contain.

- Un-initialized variables contain **unknown (garbage)** values.

- Once a variable has been declared, we may assign values to these variables.

Assigning Values to Variables

```
#include <iostream>
using namespace std;

int main()
{
     int low, high;

     low = 5;


     high = 8;
     return 0;
}
```



- As seen above, the equal sign (**=**) is used to **assign** values to the variables. It is called the **assignment operator**.

- The value on the right hand side of the assignment operator is put into (assigned to) the variable on the left hand side of the operator.

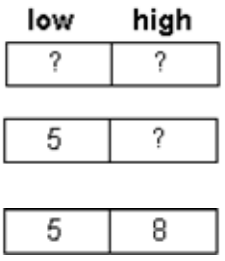
  As seen above, 5 is put into the variable ***low***
  8 is put into the variable ***high***

- Assigning a value to a variable actually puts the value in the memory location occupied by the variable.

Displaying (outputting) Variables

- To output the values to the screen, we use *cout* object, the output stream is as follows:

```
#include <iostream>
using namespace std;

int main()
{
  int low, high;

  low = 5;

  high = 8;

  cout << "low equals " <<  low;
  cout << " high equals " << high;
  return 0;
}
```



Output : low equals 5 high equals 8

cout is pronounced “see-out”. The cout is predefined in C++ systems to denote an output stream. The “out” refers to the direction in which cout sends a stream of data.

The standard output device is the monitor, though it can be something else, such as a printer or a file on the hard-disk.

The << following the cout is an operator known as the stream insertion operator. It inserts the information immediately to its right which is the data stream. The cout object then sends the information to the output device – the monitor.

Output Statement

cout << expression << expression . . . ;

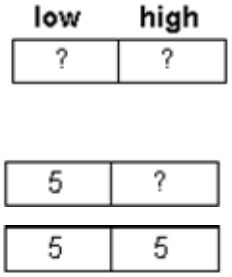
The insertion operator converts its right-hand expressions into a sequence of characters and inserts them into the output stream.
You can use << operator several times in a single output statement.

Assigning one variable to another variable

- A variable can also be assigned to another variable, provided both are of the **same data type**.

- The program below shows how the value in ***low*** is assigned to (put into) ***high***.

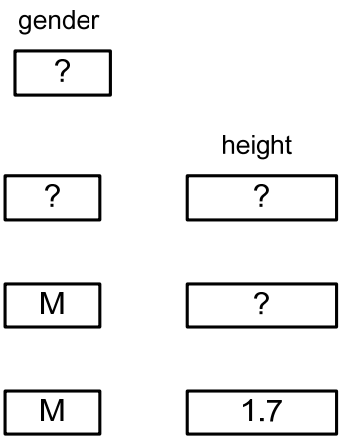
```
#include <iostream>
using namespace std;

int main()
{
  int low, high;

  low = 5;

  high = low;

  cout << "low equals " <<  low;
  cout << " high equals " << high;
  return 0;
}
```



Output : **low equals 5 high equals 5**

- Character and double variables can also be assigned values in a similar way as shown below :

```
#include <iostream>
using namespace std;

int main()
{

  char gender;

  double height;

  gender = 'M';

  height = 1.7;

  cout << "Gender is " << gender;
  cout << " Height is " << height;
  return 0;
}
```



Output : **Gender is M Height is 1.7**

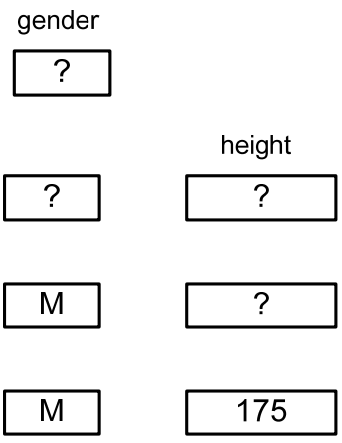
- As seen in the above program, a character variable can only contain a single character.

- **A character is always enclosed in single quotes (').**
  Example : 'F'

- **A string is always enclosed within double quotes (").**
  Example : "Hi there!"

- Double variables can be output in ***exponential format*** by using the output manipulator, **scientific** as shown below :

```
#include <iostream>
using namespace std;

int main()
{

  char gender;

  double height;

  gender = 'M';

  height = 175;


  cout << "Gender is " << gender;
  cout << " Height is " <<scientific<<height<<"cm.";
  return 0;
}
```



Output: **Gender is M Height is 1.750000e+002 cm.**

(1.750000e+002 means 1.75 x 10^2)

Examples

Decimal	Scientific
123.45	1.234500e+002
0.0051	5.100000e-003
1200000000	1.200000e+009

Newline

'\n' vs endl

Both will create a new lines in the output stream. The **'\n'** must always be between quotes and therefore can also be embedded within the output text string.

```
cout << "This \n string \n is \n displayed \n in \n seven \n lines \n";

cout << "This" << endl << "string" << endl << "is"  << endl << "also"
   << "on" << endl << "separate" << endl << "lines" << endl ;
```

Both methods can be used based on personal choice and convenience.

But you cannot create a new line with these statements :

```
cout << "This is line one " << \n ;
```

- without "" the compiler does not understand what is \n

```
cout << "This is line two " << "endl" ;
```

- This will just print : This is line two endl

Activity

1. Write a C++ statement to declare 3 variables to store the following information of a student:

   * **marks** – value with decimal point (*sample value : 82.37*)
   * **grade** – an alphabet (*sample value : C*)
   * **subject code** – 4-digit integer number (*sample value : 4001*)

   [*Read "Some Rules for naming Variables and Constants" on next pages*]

2. Using the variables created in Question 1.

   Assuming the student score 82 marks (grade is A) in subject code 1183, assign the values into the variables you have declared earlier.

3. What happens if you execute the following codes:

   a.
   ```
   double varA = 12.55; int varB;
   varB = varA;
   ```

   b.
   ```
   char varA = 'g'; int varB;
   varB = varA + 1;
   ```

4. Fill in the following table:

C++ representation	Decimal
-2.500000e-003	
	1234.5
	0.07

5. Determine the output of the following C++ statements:

```
cout << "Hi!\n" << "My name ";
cout << "is " << endl << "Andy << endl";
```

Using Constants

- Like variables, they must be declared with unique names.
- Their values must be assigned upon declaration.
- Once assigned, the values in the Constants cannot be changed in the program.
- Declaration and assignment of Constants are done outside the body of the program.

```
#include <iostream>
using namespace std;
#define KG2LBS 2.2

int main()
{
    double weightKG = 55.2;
    cout << "Weight in lbs =" << weightKG * KG2LBS;
    return 0;
}
```

Some Rules for naming Variables and Constants

- Use meaningful names for variables and constants.

  `zxq987` is a valid declaration, but it is meaningless.
  `totalWeight` is valid and more meaningful.

- No spacing within the names.
  `My Name` is invalid
  `myName` is valid

- No symbols except "_" within names.
  `you&me` is invalid
  `you_me` is valid

- Each name must begin with an alphabet.
  `3ply` is invalid
  `ply3` is valid

Some Rules for assigning values to Variables

- Do not put commas in values that you assign to variables.

  ```
  height = 1,000.00;      invalid statement.
  height = 1000.00;       valid statement.
  ```

- If you want to assign a character data to a character variable, the character must be enclosed in single quotes.

  e.g. `char status = 'M';`

- Be careful when you mix data types. If a variable is declared to be of the type integer, and you assigned a floating point value to it.

  ```
  int high;
  high = 1.5;
  ```

  The variable **`high`** will be only 1 and not 1.5.

Keywords (this is not a complete list)

The keywords are reserved words for the compiler because they have special meaning. So do not use them as variable names.

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

Example of wrong usage:

```
int long;
double short;
```

The “long” and “short” are keywords.

A full list of C++ keywords in given in Appendix.

Comments

- Comments are messages that you insert into your C++ program, explaining what is going on at that point in the program.

- A comment is a message for the person reading your program; it is not an instruction to the computer.

- **The C++ compiler ignores the comment lines.**

- A comment in C++ begins with **/*** and ends with ***/**

- Another way to write a comment is to use two slashes (//)
  From the two slashes till the end of that line, it is all a comment.

  Example:

```
/* Filename : prog1.cpp */
// Filename : prog1.cpp
```

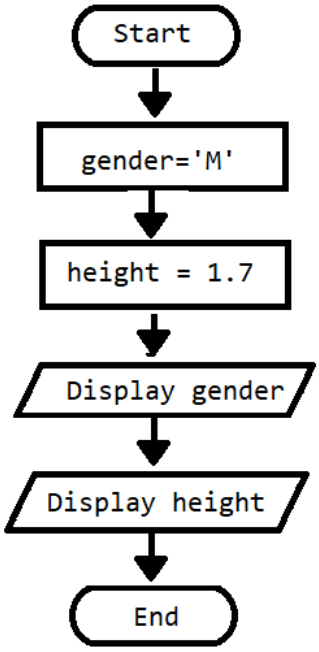
- Always insert your comments while you are writing your program. Do not put it off until later.

- Shown below is the previous program with comments included :

```
/* This program prints out the gender and height of a
person */

#include <iostream>
using namespace std;

int main()
{
  //----- Variable Declaration -------
  char gender;
  double height;

  //----- Assign values to variables --
  gender = 'M';
  height = 1.7;

  //---- Display gender and height -------
  cout << "Gender is " << gender << endl;
  cout << "Height is " << height;

  return 0;
}
```

```
Begin
   1. Declare gender, height
   2. Initialise variables
      2.1   gender = 'M'
      2.2   height = 1.7
   3. Display variables
      3.1   Display gender
      3.2   Display height
End
```

(Pseudo codes)



(Flowchart)

Keyboard Input

- Just as ***cout*** is used to output data to the screen, ***cin*** allows the user to input data from the keyboard and assign to a variable.

  cin >> variable;

- The input operation is often called the ***read*** operation.

Example:

```
int answer;
cin >> answer;
```

The value entered by the user from the keyboard is read in and assigned to the variable **answer**.

- When your program reaches the line containing ***cin*** it stops and waits for the user to input a value from the keyboard.

Activity

```
int age, weight;
cin >> age ;
cin >> weight;
```

Assume that you key in

20

64.9

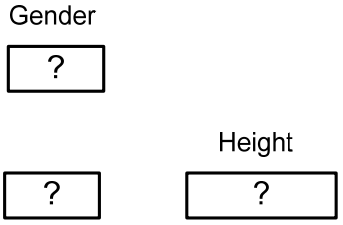
The value assigned to age will be 20, but the value assigned to weight will be only 64 because weight was typed integer.

How do you correct this problem?

_____

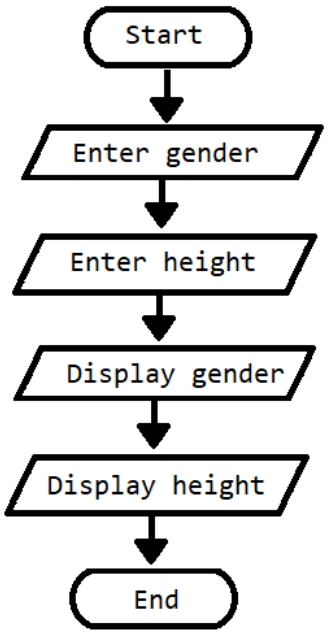
The following program demonstrates the use of ***cin***.

```
/* This program reads in the gender and height of a
 person and displays it on the screen */

#include <iostream>     //-- pre-processor directive
using namespace std;

int main()
{
  //---- variable declarations --

  char gender;

  double height;

  //---- read in gender and height from keyboard ----
  cout << "Please enter the gender (M or F) : ";

  cin >> gender;

  cout << "Please enter the height : ";

  cin >> height;
  //---- display gender and height ------------------
  cout << "Gender is " << gender << endl;
  cout << "Height is " << height << endl;
  return 0;
}
```



```
Begin
   1. Declare gender, height
   2. Get inputs
      2.1   Prompt user for gender
      2.2   Enter gender
      2.3   Prompt user for height
      2.4   Enter height
   3. Display variables
      3.1   Display gender
      3.2   Display height
End
```

(Pseudo codes)



(Flowchart)

Sample run

Please enter the gender (M or F) : F <ENTER>
Please enter the height : 1.6 <ENTER>
Gender is F
Height is 1.6

Chapter 3

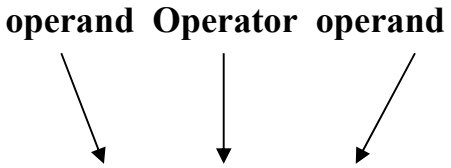
Operators

Operators

- C++ has four main groups of Operators :
    - Arithmetic Operators
    - Relational Operators
    - Logical Operators
    - Bitwise Operators

- Arithmetic Operators are used to perform arithmetic operations.

- The basic syntax of C++ Operators is as follows :



e.g. 3 + 5 **Binary Operator**

The sum of 3 and 5.

- The table below shows the symbols used for arithmetic operations and their meanings:

SYMBOL	MEANING
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

- Some sample results of these operations are shown below:

FORMULA	RESULT
4 * 2	8
64 / 4	16
11 % 3	2
12 + 9	21
80 - 15	65

- 11 % 3 is the remainder of 11/3 which is 2

  Example : 25 % 5 is 0

- When an integer is divided by an integer, the outcome is an integer.

  Example : 7/3 is 2 **(not 2.33)**

The Division Operator

- If integers appear on both sides of the slash (/), it will give an integer result and discard the remainder as shown below :

```
/* Tests the division operator */

#include <iostream>
using namespace std;
int main()
{
  int num1, num2;
  double result;

  num1 = 3;
  num2 = 2;

  result = num1/num2;

  cout << "The result of 3/2 is " << result;
  return 0;
}
```

result = 3/2;

Output:

The result of 3/2 is 1

integer / integer

⇩

the result is integer.
The remainder is lost.

- However, if at least ONE of the numbers is a double, it will give you a result together with the remainder as shown below :

```
/* Tests the division operator */

#include <iostream>
using namespace std;
int main()
{
  int num2;
  double num1,result;

  num1 = 3.0;
  num2 = 2;
                          result = 3.0/2
  result = num1/num2;

  cout << "The result of 3.0/2 is " << result;
  return 0;
}
```

Output:

The result of 3.0/2 is 1.5

- The modulus operator (%) requires that there should be ***integers*** on both sides of the symbol for it to work.

double / integer
integer / double
double / double

⇩

the results are double.

Activity

Complete the following program which converts time in minutes to hours and minutes. E.g. 150mins is 2:30

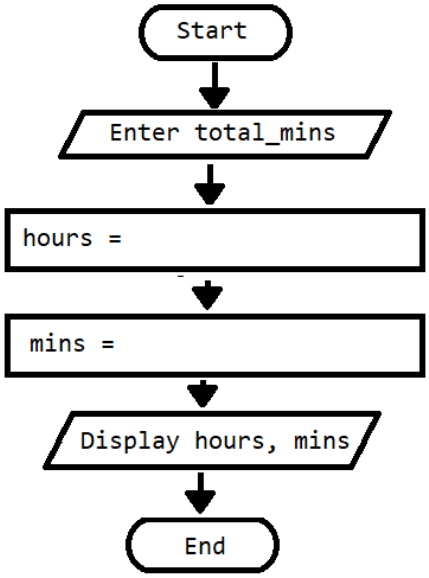
```
#include <iostream>
using namespace std;

int main()
{
    int total_mins, hours, mins;

    cout << "Enter time in minutes: ";
    cin >> total_mins;

    hours = _____________________________________;

    mins = _____________________________________;

    cout << _____________________________________;

    return 0;
}
```

Sample console output:

```
Enter time in minutes: 150
2:30
```

```
Begin
  1. Declare total_mins, hours, mins
  2. Get inputs
      2.1   Prompt user for total_mins
      2.2   Enter total_mins
  3. Compute
      3.1   Compute hours =
      3.2   Compute mins =
  4. Display hours and mins
End
```

(Pseudo codes)



(Flowchart)

Order of Precedence

- The order of precedence determines exactly how C++ computes arithmetic expressions.

- To see how the order of precedence works, try to determine the result of the arithmetic expression shown below :

  2 + 3 * 2

- If you interpret the formula from left to right, you would get 10.

- However, if you do the multiplication first, you would get 8, which is how C++ computes the answer.

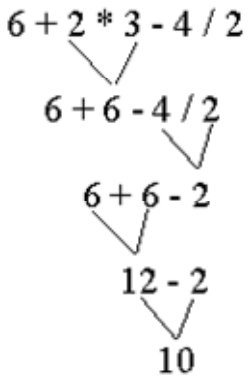
- C++ always performs multiplication, division and modulus first, and then addition and subtraction. This is summarized in the table below:

ORDER	OPERATION
First	Multiplication, Division, Modulus (* , / , %)
Second	Addition, Subtraction (+ , -)

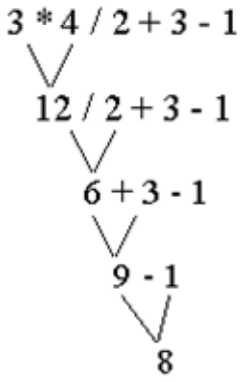
- The multiplication, division and modulus will be performed first, from left to right i.e. the left most operation is done first.

- The additions and subtractions are done next, from left to right i.e. the left most operation is done first.

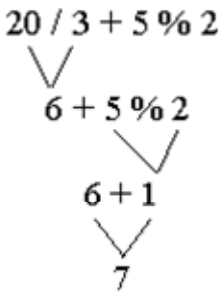
Examples



6+(2*3)-(4/2)



(3*4)/2 + 3 -1



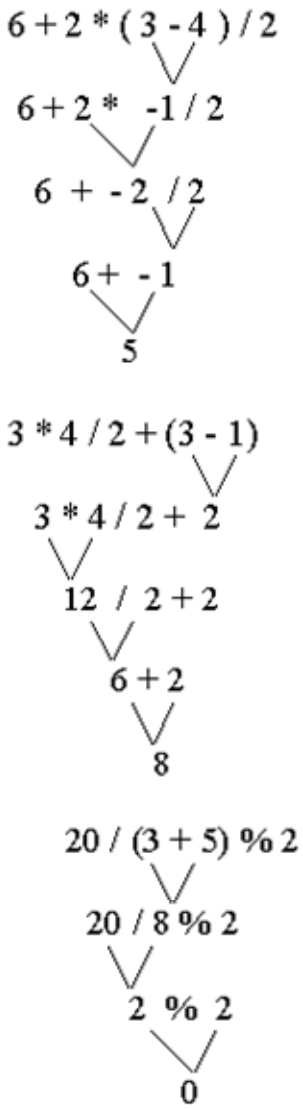
(20/3)+(5%2)

Using Parenthesis ( )

- Parenthesis ( ) have higher precedence than all other operations in the precedence table.

- In other words, any calculation within parenthesis is always done first before anything else.

- Shown on the next page are the same three expressions that are calculated differently because of parenthesis.



The table below shows the order of Precedence and Associativity of C++ Operators:

OPERATORS	TYPE	ASSOCIATIVITY
() [] .	Groups	left to right
! ~ ++ --	Unary	right to left
* / %	Multiplicative	left to right
+ -	Additive	left to right
<< >>	Shift	left to right
< > <= >=	Relational	left to right
== !=	Equality	left to right
&	Bitwise	left to right
^		
|		
&&	Logical	left to right
||		
= += -= *= /=	Assignment	right to left

- **Unary operators** require only one operand.

C++ has special unary operators

```
y++;        // increments the variable y by 1
i--;        // decrease the variable i by 1
```

So if y was 5 and i was also 5,

then after y++; and i--;

y will be 6 (incremented by 1) and i will be 4 (decremented by 1) .

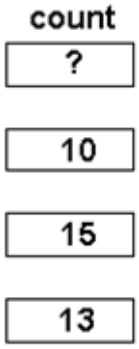
Combined Assignment and Arithmetic Operator

int count;

count = 10;

count += 5; // count = count + 5

count -= 2; // count = count - 2



Activity

1. **Evaluate the following expressions:**
   a. **34 / 5 + 7**

   b. **3 + 8 % 3**

   c. **2 * 7 / (2.5 – 5) + 8**

2. **Determine the result after the following code segment is executed.**
   a. **double varA = 4.8; varA++;**

   b. **char varA = 'S'; varA--;**

   c. **int varB = 2; varB -=5;**

   d. **int varA = 4; varA += 5+3;**

Predefined Math Functions

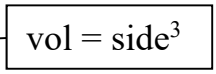
- Functions are a set of instructions that are executed only when they are activated.
- Provided as part of the C++ system.
- The cmath library contains many functions for common mathematical operations and transformation.
- The following pre-processor directive must be included in your program to direct the compiler to include the cmath header file.

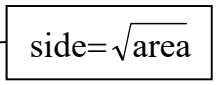
```
# include <cmath>
```

- Some common mathematical operations are:
  - pow – to raise a number to a power
  - sqrt – to compute the square root of a number
  - sin – to compute the sine of an angle
  - log – to compute the natural logarithm of a number.. etc

Example:

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double side, vol, area;

    // Using pow() to calculate volume
    cout << "Enter length of a cube: ";
    cin >> side;
    vol = pow(side,3);
    cout << "Volume of cube is " << vol << endl;
    // Using sqrt() to find side of a square
    cout << "Enter area of a square: ";
    cin >> area;
    side = sqrt(area);
    cout << "Length of the square is "
         << side << endl;
    return 0;
}
```





Displaying Several Variables

- *cout* can display more than one variable at a time.

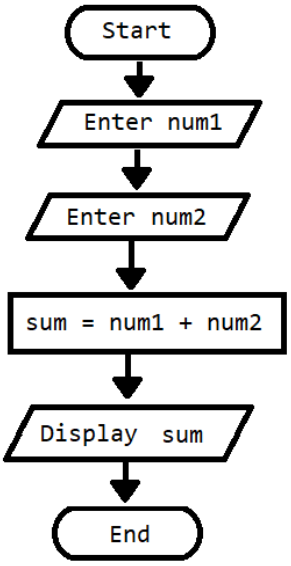
```
#include <iostream>
using namespace std;
int main()
{
  int num1, num2, sum;
  /* Read  Numbers */
  cout <<"\nWhat is the first number : ";

  cin >> num1;
  cout << "\nWhat is the second number : ";

  cin >> num2;

  sum = num1 + num2;
  /* Display Result */

  cout << "Sum of " << num1 <<"+" <<num2
                              << " is " << sum ;

  return 0;
  }
```

num1	num2	sum
?	?	?
5	?	?
5	10	?
5	10	15

Output: **Sum of 5+10 is 15**

The last statement can also be written as :

```
cout  << "Sum of "
      << num1
      << "+"
      << num2
      << " is "
      << sum ;
```



(Flowchart)

A closer look at character

- Characters are stored as integer values based on their ASCII codes (see pg 25). Hence it is valid to assign integer value to a character. Also simple arithmetic on a character variable is also allowed.

```
#include <iostream>
using namespace std;

int main()
{
   char a = 'M';

   char b;
   int c;

   cout << " a = " << a << endl;

   b = a+2;
   c = a+2;

  cout << " b = " << b << endl;
   cout << " c = " << c << endl;


  return 0;
}
```

Partial ASCII table (see pg 25)

DEC	CHR
75	K
76	L
77	M
78	N
79	O

Output: **a = M**
b = O
c = 79

- Note: Character is 1 byte. If a value greater than 255 (decimal) is being assigned to a character, only a single byte of data will be retained, i.e. range 0 - 255.

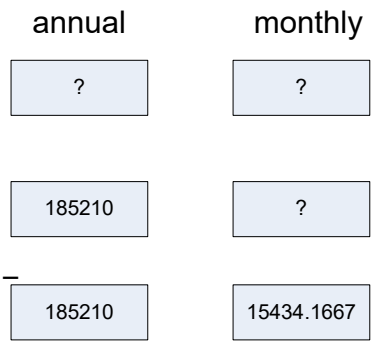
  e.g.

```
char a = 333;
cout << a ;  // display 'M' i.e. 333-256=77
```

Formatting Floating-point Outputs

- The following program reads in a person's Annual pay and displays the Monthly pay.

```
/* Converts from Annual to Monthly pay */

#include <iostream>
using namespace std;
int main()
{
   //-- Variable Declarations ------------
   double annual, monthly;

   //-- Read Input ----------------
   cout << "Please enter Annual Pay :$";
   cin >> annual;

   //-- Calculate Monthly Pay ------------
   monthly = annual/12;

   //-- Output Monthly Pay ---------------
   cout << "Your Monthly Pay is " << monthly;
   return 0;
}
```



Output: Please enter Annual Pay :$185210
Your monthly pay is $15434.2

By default, the display is in 6 significant digits.
In order to have better control of the display,
we need to use the I/O manipulators.

(Pseudo codes)

```
Begin
 1. Declare annual, monthly
 2. Get inputs
   2.1 Prompt user for annual
   2.2 Enter annual
 3. Compute monthly = annual/12
 4. Display monthly
End
```


(Pseudo codes)

Manipulators

<iomanip>

resetiosflags	Clears the specified flags.
setbase	Set base for integers.
setfill	Sets the character that will be used to fill spaces in a right-justified display.
setiosflags	Sets the specified flags.
setprecision	Sets the precision for floating-point values.
setw	Specifies the width of the display field.

In order to use these manipulators, we need to
#include <iomanip>

<iostream>

endl	Terminates a line and flushes the buffer.
flush	Flushes the buffer.
dec	Specifies that integer variables appear in base 10 notation.
fixed	Specifies that a floating-point number is displayed in fixed-decimal notation.
hex	Specifies that integer variables appear in base 16 notation.
left	Causes text that is not as wide as the output width to appear in the stream flush with the left margin.
oct	Specifies that integer variables appear in base 8 notation.
right	Causes text that is not as wide as the output width to appear in the stream flush with the right margin.
scientific	Causes floating point numbers to be displayed using scientific notation.

In order to use these manipulators, we need to
#include <iostream>

- To change the output to 2 decimal places, we could modify the last *cout* statement as follows :

```
cout << "Your Monthly pay is $"
     << fixed << setprecision(2)
     << monthly << "\n";
```

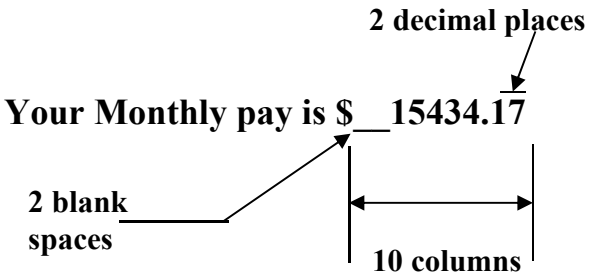
- This would produce an output of

Your Monthly pay is $15434.17

- To reserve 10 places for the entire number, we could modify the *cout* statement as follows :

```
cout << "\nYour Monthly Pay is $"
     << fixed << setprecision(2)
     << right << setw(10)<< monthly << "\n";
```

- This will produce the following output:



- To change the display into scientific :

```
cout<<"\nYour Monthly Pay is $"
    <<scientific  <<monthly<<"\n";
```

- The output is
  Your Monthly Pay is $1.543417e+004

Relational Operators

- Relational Operators are used to compare data.

- They are used to determine whether the result of a comparison is TRUE or FALSE.

- Shown below are some **Relational Operators** used in C++.

OPERATOR	MEANING
>	Greater than
>=	Greater than or Equal to
<	Less than
<=	Less than or Equal to
==	Equal to
!=	Not Equal to

- It is important to note that C++ uses a double equal sign (==) to test if two values are equal.

- The single equal sign (=) is used to assign a value to a variable.

  e.g.
  ```
  num1 = 5;
  ```

- Relational Operators have lower precedence than Arithmetic Operators.

  For example, the expression
  ```
  i < limit - 1
  ```
  is interpreted as
  ```
  i < (limit - 1)
  ```

- This shows that < has a lower precedence than -

Logical Operators

- Shown below are some **Logical Operators** used in C++.

OPERATOR	MEANING
&&	AND
||	OR

- Logical Operators have a lower precedence than Relational Operators.

  For example the expression

```
count<10 && step>0.1
```

  is equivalent to

```
(count<10) && (step>0.1)
```

- Amongst the Logical operators, the precedence of && is greater than that of ||

  This means that the expression
  ```
  a<0 || b>10 &&  c>20
  ```
  is interpreted as
  ```
  a<0 || (b>10 &&  c>20)
  ```

- Relational and Logical expressions are defined to have a value 1 if TRUE and 0 if FALSE.

```
cout << (5==5);
```
will display 1 because the expression (5==5) is TRUE and hence has a value 1

```
cout << (5==4);
```
will display 0 because the expression (5==4) is FALSE and hence has a value 0.

1. Evaluate the following expressions:
   a. (17 < 4 * 3 + 5) || (8 * 2 == 4 * 4) && (3 + 3 == 6)

   _____
   _____
   _____
   _____
   _____

   b. char ch = 'A'; cout << (ch >= 65);

   _____

2. Rewrite the following conditions in a single C++ expression:
   a. A is greater than B but less than C

   _____

   b. A is greater than or equal to B and A is equal to C

   _____

Chapter 4

Selection Constructs

The *if-else* Statement

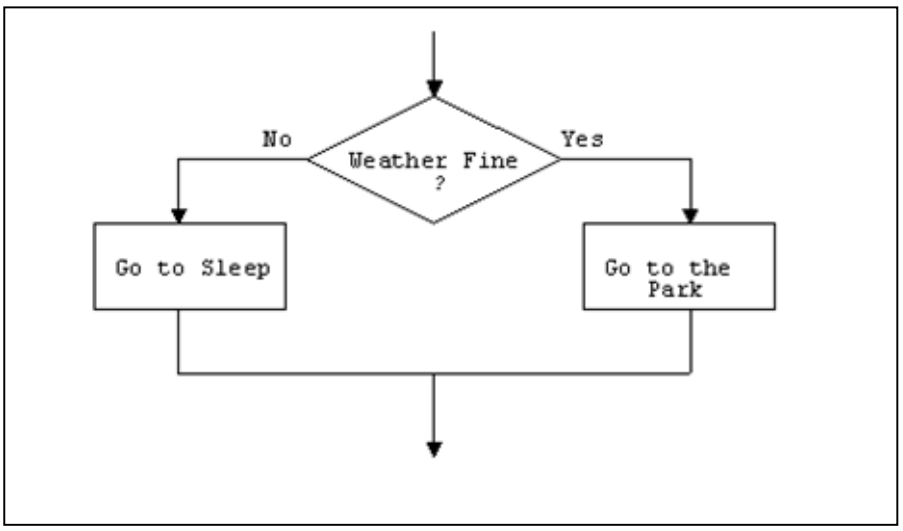
- An ***if-else*** statement is used in Selection Constructs.

- A condition is evaluated. If it is TRUE, certain actions are taken. If the condition is FALSE, a different set of actions are taken.

For example,

condition

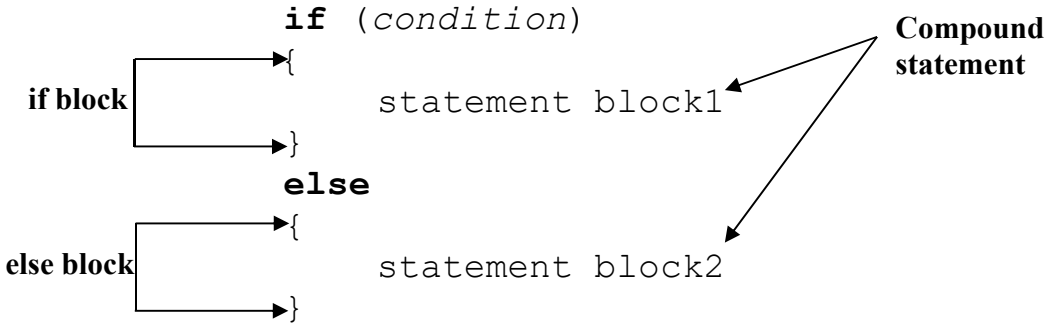
if (*the weather is fine*)
    I will go to the park
else
    I will go to sleep



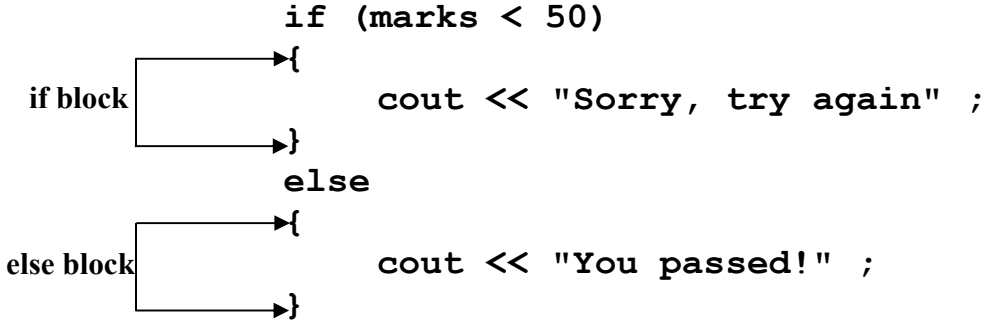
Flowchart for the Program

- In many applications, the program should be capable of evaluating conditions and making its own decisions.

- The syntax of the *if-else* statement is as follows :



- Example:



- Note that if the statement block has only one statement, the braces ({ }) can be omitted.

- Example:

```
if (marks < 50)
    cout << "Sorry, try again" ;
else
    cout << "You passed!" ;
```

However if the statement block consist of more than one statement,

```
if (marks < 50)
     cout << "Sorry, try again";
     // let's help by adding one mark
     marks = marks + 1;

else

     cout << "You passed!" ;
```

This is incorrect.

The correct version is

```
if (marks < 50)
{
     cout << "Sorry, try again" ;
     // let's help by adding one mark
     marks = marks + 1;
}
else

     cout << "You passed!" ;
```

The multiple statements must be encased with the curly braces.

Note:

Things that comes in pairs in C++.
{ }
"This is a string "
'A'
(marks < 50)
// comments
/* comments */

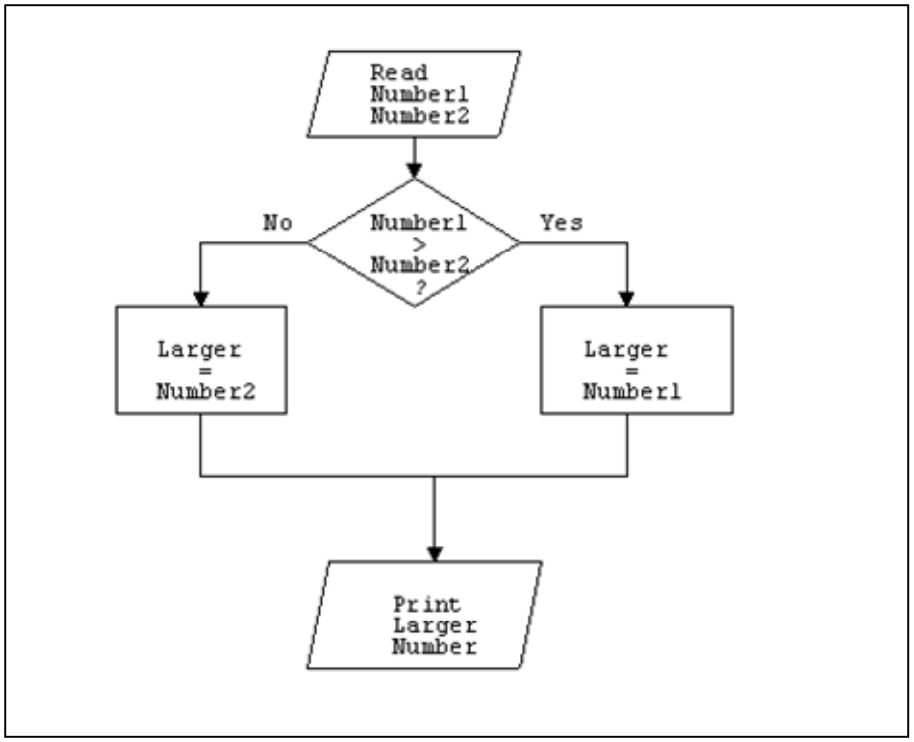
Example

Design a program that would read in two numbers from the keyboard and output the larger of the two numbers.

- The first-level pseudo-code would be as follows :

  1.0 Read in the two numbers
  2.0 Find the larger number
  3.0 Output the larger number

- The flowchart for the program is given below :



Flowchart for the Program

- The program is shown below :

```
/* Find and display the larger of the two numbers */

#include <iostream>
using namespace std;

int main()
{
    int n1, n2, ans;

    //---- Read in the two numbers -----------
    cout << "Enter first number : ";
    cin >> n1;
    cout << "Enter second number : ";
    cin >> n2;

    //---- Find the larger number ------------

    if(____________________)
    {
        ____________________________________ ;
    }
    else
    {
        ____________________________________ ;
    }

    //---- Output the larger number --------------
    cout << "\nThe larger number is " << ans;
    return 0;

}
```

if block condition is TRUE

else block condition is FALSE

- The condition of the ***if-else*** statement must be within parenthesis.

  e.g.
  ```
  if( x >= y )
  ```

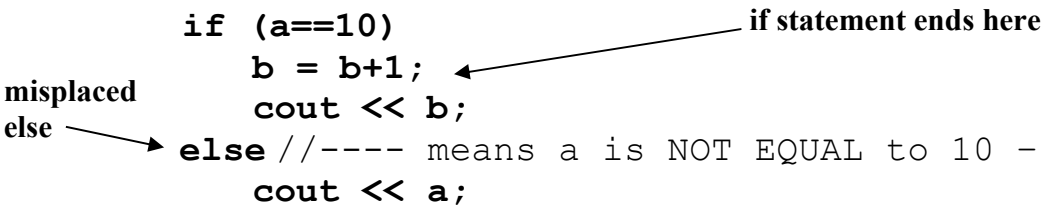
- The statement

  ```
  if x >= y
  ```
  is INVALID.

- Either the ***if*** part or the ***else*** part of the *if-else* statement is executed, not both.

- Compound statements in the ***if*** part and the ***else*** part, must be enclosed in braces { } .

- If the braces are missing, the ***if*** and ***else*** parts will only be valid for ONE statement following it.



- Always **indent** statements within a block. Non-indented programs, like the one shown below, are difficult to read and the blocks are not easily visible.

```
if(number1 > number2)
{
larger = number1;
cout << "number1 is larger than number2";
}
else
{
larger = number2;
cout << "\nnumber2 is larger than number1");
}
```

Be very careful when putting a semicolon (;) immediately after the if clause. The programming logic changes.

```
if(number1 > number2); <-- if statement terminates here
{
   larger = number1;
}
```

empty statement !

- The semicolon would terminate the ***if*** statement. The C++ Compiler would take it to mean :

  `if(number1 > number2)` do nothing;

- The else part of an if-else statement is optional and can be left out.

```
            if (marks==49)
            {
if              marks = 50;
block       cout << "Mark is " << marks;
            }
```

- But no **else** without an **if**
- There is no conditional else like **else(marks > 50)**

Activity

- Determine the results of the following multiple conditional *if* expressions :

  Given x = 4; y = 1.543; ch = 'x';

```
if ((x==4) && (y>2) || (ch!='x'))
```

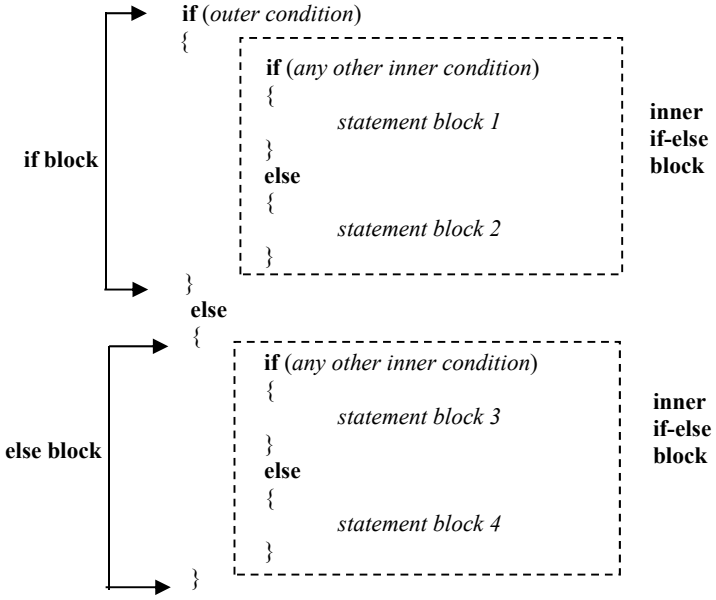
_____

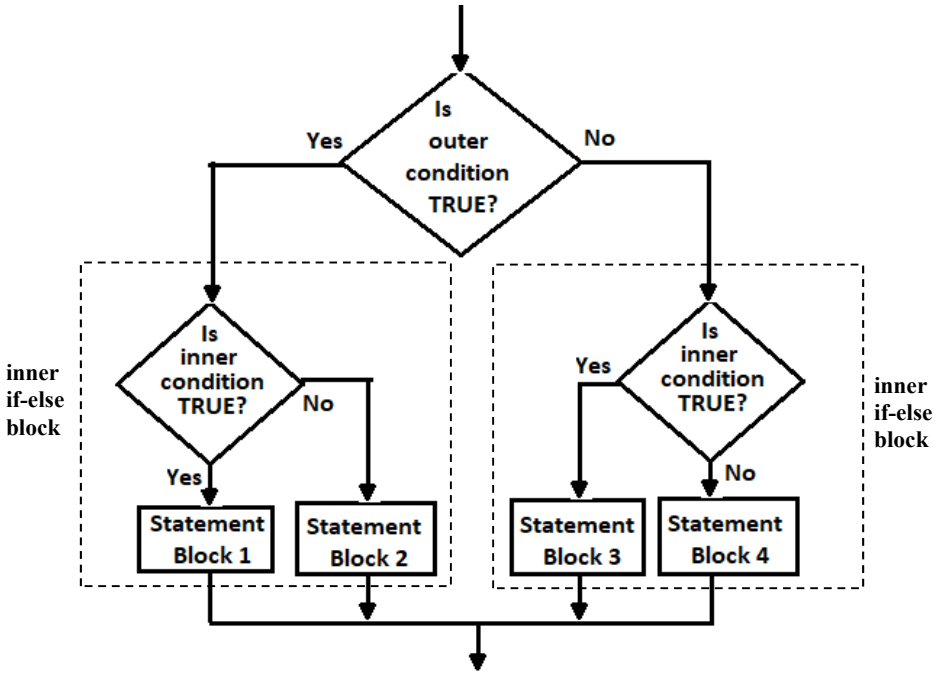
```
if ((x==4) && (y<2) || (ch!='x'))
```

_____

Nested if-else Statements

- A nested ***if-else*** statement is an ***if*** statement within an ***if*** statement.





Flowchart of the Nested if-else Statement

Activity

Write a program to read in a temperature value from the keyboard in degrees Centigrade. If the temperature is less than or equal to zero, it outputs the word **ICE**. If it is greater than zero and less 100, it outputs the word **WATER**. If it is greater than or equal to 100, it outputs the word **STEAM**. Note that this program uses **nested if-else statements.**

- Draw the flowchart in the space provided below :

```
/* Prints a message depending on the temperature */

#include <iostream>
using namespace std;

int main()
{
   int temperature;

   //--- Get the temperature ----------
   ____________________________________________________

   ____________________________________________________

   if(____________________)          //<-- Outer condition
   {
     ____________________________________
   }
   else
   {
     if(____________________)  //<-- inner condition
     {
       ____________________________________
     }
     else
     {
         ____________________________________
     }
   }
   return 0;
}
```

The *switch* Statement

- The ***switch*** statement is used for selecting a set of actions from several choices.

- The syntax of the ***switch*** statement is as follows :


```
switch (expression)
{
     case const1 : statement1(s) ;
                   break;
     case const2 : statement2(s) ;
                   break;
          :
          :
     default     : statementN(s) ;
                   break;
}
```

- The ***expression*** is first evaluated for its value. The value is then compared with the first constant value, ***const1***. If the value of the expression matches ***const1*** the corresponding ***statement1(s)*** are executed.

- If the match does not occur, the value of the expression is then compared with the next constant value, ***const2***. If the value of the expression matches ***const2*** the corresponding ***statement2(s)*** are executed.

- The ***break*** terminates the execution of the **switch** statement.

- The ***default*** case is executed only if the value of the expression does not match any of the constant values.

- The following program reads in an integer between 1 and 4, and displays the number in words. For any integer not in the range 1 to 4, the program outputs an error message.

```
#include <iostream>
using namespace std;

int main()
{
  int number;

  cout << "\nEnter a number between 1 and 4 : ";
  cin >> number;

  if(number == 1)
       cout << "ONE\n";
  else
       if(number == 2)
            cout << "TWO\n";
       else
            if(number == 3)
                 cout << "THREE\n";
            else
                 if(number == 4)
                      cout << "FOUR\n";
                 else
                      cout << "OUT OF RANGE\n";

  return 0;
}
```

- This program will read much better if written using a *switch* statement instead of the multiple *if-else* statements :

```
#include <iostream>
using namespace std;

int main()
{
  int number;

  cout << "\nEnter a number between 1 and 4 : ";
  cin >> number;

  switch(number)
  {
      case 1 : cout << "ONE\n";
               break;
      case 2 : cout << "TWO\n";
               break;
      case 3 : cout << "THREE\n";
               break;
      case 4 : cout << "FOUR\n";
               break;
      default : cout << "OUT OF RANGE\n";
                break;
  }
  return 0;
}
```

- The ***default*** case is executed only if none of the other cases are satisfied i.e. if the number is not in the range 1 to 4. It basically displays an error message.

- The ***default*** case is optional and may be left out of the *switch*.

- If it is left out, no error message will be displayed, should the number be not in the range 1 to 4.

- What happens if we remove all the ***break*** statements from the ***switch*** ?

```
#include <iostream>
using namespace std;

int main()
{
     int number;

     cout << "Enter a number between 1 and 4 : ";
     cin >> number;

     switch(number)
     {
          case 1 : cout << "ONE\n";
          case 2 : cout << "TWO\n";
          case 3 : cout << "THREE\n";
          case 4 : cout << "FOUR\n";
          default: cout << "OUT OF RANGE\n";
     }
     return 0;
}
```

Output:

```
Enter a number between 1 and 4 : 2 <ENTER>
TWO
THREE
FOUR
OUT OF RANGE
```

- As seen above, if the break statements are removed, the program will execute the statements corresponding to the matching case and all the cases that follow until it encounters a *break* or the end of the *switch* statement.

switch versus multiple *if-else*

- A **switch** statement cannot be used to evaluate *double* variables.

  So in the case of doubles, use ***if-else***

- Consider the following ***if*** condition :

```
if (number > 2 && number < 100)
```

  The above condition cannot be conveniently evaluated using ***switch*** unless you have case labels for each integer from 3 to 99.

```
switch(number)
{
     case 3 :
     case 4 :
     .
     .
     case 99 :
}
```

  Therefore it is more convenient to use the ***if*** statement.

  - For only 2 options, it may be more convenient to use an ***if-else*** statement.

  - However, if there are more than 2 options, a ***switch*** statement may be more convenient.

- The following program reads in the grade (A, B, C, D or F) obtained in an exam. The program then displays a word corresponding to the grade, as per the table shown below :

A	:	Excellent
B	:	Good
C	:	Satisfactory
D	:	Poor
F	:	Fail

For any other grade entered, the program displays an appropriate error message.

```
#include <iostream>
using namespace std;

int main()
{
     char grade;

     cout << "Enter the grade : ";
     cin >> grade;

     switch ( _________ )
     {









     }
     return 0;
}
```

Activity

What happens if the user enters **'a'** when the program runs?

_____

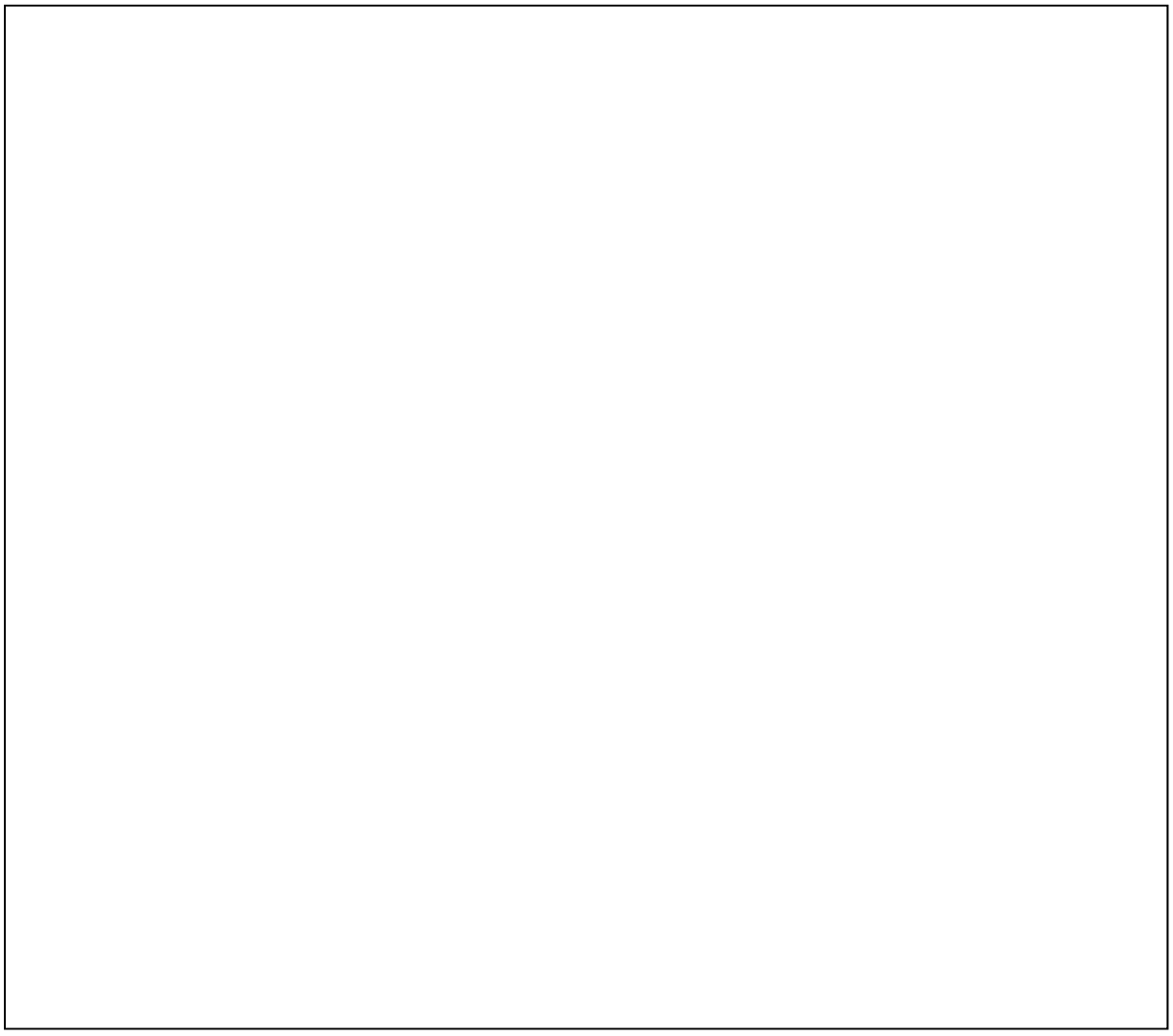
Modify the program to accept 'a', 'b', 'c', 'd' and 'f' as valid grades as well.

Activity

Write a program to calculate the total weekly pay which is dependent on the number of hours worked.
The normal rate is $4.00/hr.
If the hours worked exceed 40 in the week, they will be considered as overtime and will be paid at 1.5 times the normal rate.

Test Data:

Hours	Normal Pay	OT Pay	Total Pay
20	20 * 4 = 80	0	80
50	40 * 4 = 160	10 * 6 = 60	220



Chapter 5

Iteration (Repetition) Constructs

Introduction

- Computers are capable of doing things over and over again without getting bored or tired.

- An ***iterative***, or ***repetitive*** loop allows a set of statements to be repeated.

- There are three statements in C++ which can help a programmer to construct loops to repeat a set of statements :

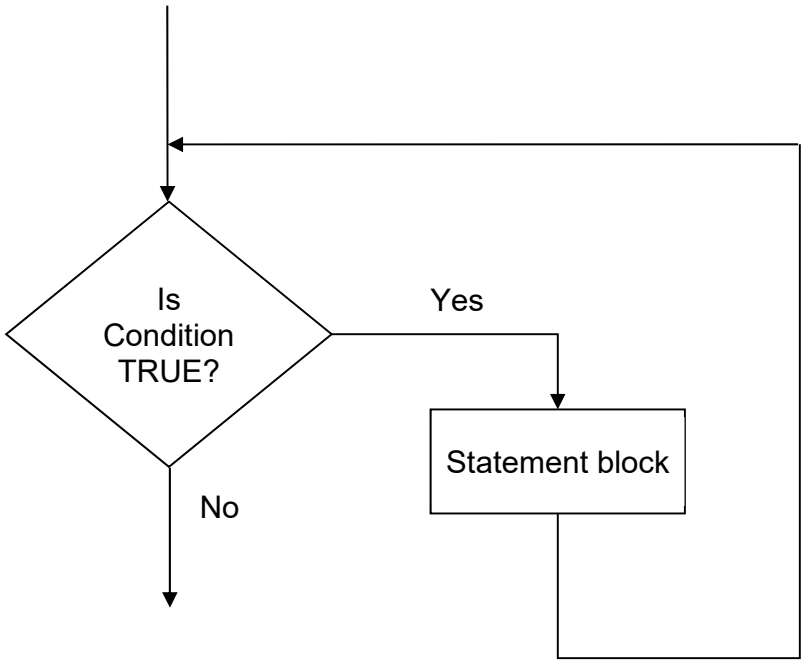
  - **while** – Loop

  - **for** – Loop

  - **do..while** – Loop

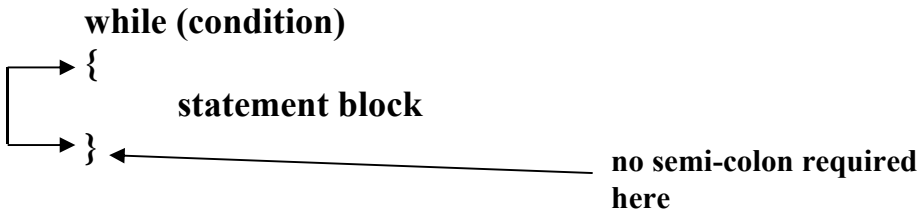
while - Loop

- The *while* statement allows a block of code to be executed while a specified condition is TRUE. It checks the condition at the start of the block; if this is TRUE the block is executed, else it will exit the loop.

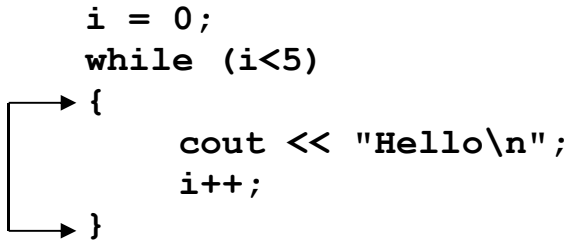


Flowchart Representation of the *while* Statement

- The syntax is :



Example:



```
i = 0;
while (i<5)
{
     cout << "Hello\n";
     i++;
}
```

- The *condition* normally uses relational operators. If multiple test conditions must be met, then the logical operators will also be used.

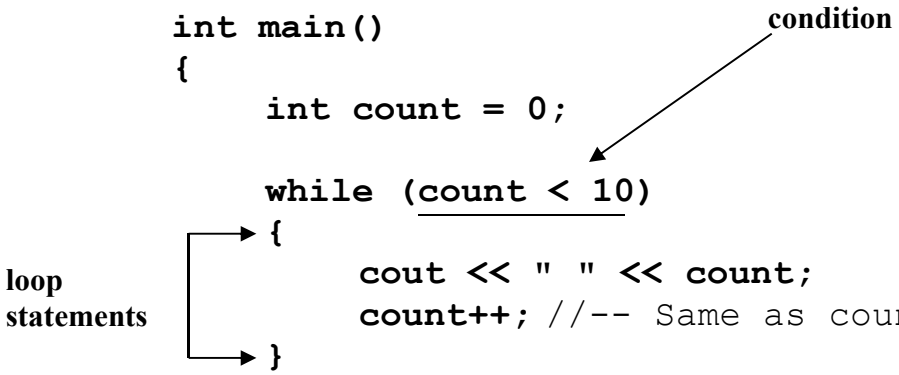
  Example:

```
while ( a<10 && b>5) ……
```

- If the statement block contains a single statement then the braces may be omitted.

Example:

```
//---- Program to print 0 to 9 -------

#include <iostream>
using namespace std;

int main()
{
     int count = 0;

     while (count < 10)
     {
          cout << " " << count;
          count++; //-- Same as count = count + 1
     }
     return 0;

}
```



Activity

1. Analyse the program and fill in the table below:

int count = 0;	0										
while(count<10)	True										
{											
cout<<count;	0										
count++;	1										
}											

2. Modify the program to display only the odd numbers from 1 up to 20.

- The following program uses multiple conditions with logical operators.

- The program allows the user to enter the age. If the user enters an age which is less than or equal to 0 OR greater than 100, the function will display an error message and prompt the user to re-enter the age.

```
/* Program allows the user to enter the age */

#include <iostream>
using namespace std;

int main()
{
    int age;

    cout << "\nPlease enter your age : ";
    cin >> age;
    while ( (age <= 0) || (age > 100) )
    {
        cout << "\nInvalid age.\n";
        cout << "\nPlease Re-enter age : ";
        cin >> age;
    }
    return 0;
}
```

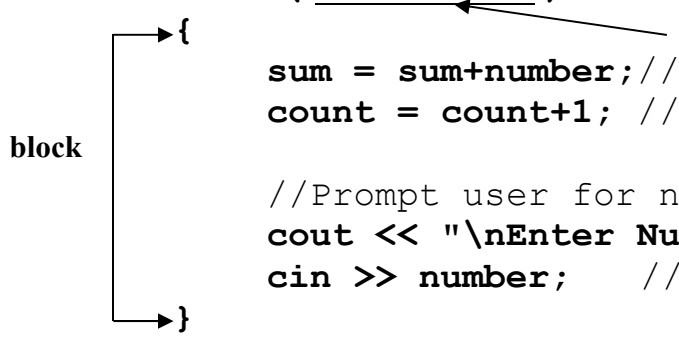
block

- If the user keeps on entering an invalid age, the function would keep on asking the user to en-enter the age until a valid age is entered.

Example

Write a program that would read in numbers from the keyboard, compute the average of the numbers, and display it. The sequence of numbers are to be terminated with a zero.

```
/* Find average of a series of numbers */

#include <iostream>
using namespace std;

int main()
{
   int count;
   double number, sum, average;

   count = 0;    //Initialize Count to 0
   sum = 0;      //Initialize Sum to 0

   //Prompt user to enter Number
   cout << "\nEnter Number (0 to terminate) : ";
   cin >> number;    //Read Number

   //While Number not equal to 0
   while(number != 0)
   {
        sum = sum+number;//Add Number to Sum
        count = count+1; //Increment Count

        //Prompt user for next Number
        cout << "\nEnter Number : ";
        cin >> number;    //Read Number
   }
   average = sum/count;  //Average = Sum/Count

   cout << "\nThe average of the " << count
        << " numbers is " << average << endl;
   return 0;
}
```



Activity

Analyze the program and fill in the table below, assuming that the following values are entered.

5, 7, 4, 2, 0

	`count =`					
	`sum =`					
	`number =`	5				
	`while(number!=0)`	True				
Loop	`{`					
	`sum = sum+number;`					
	`count = count+1;`					
	`cin >> number;`	7	4	2	0	
	`}`					
	`average=sum/count;`					

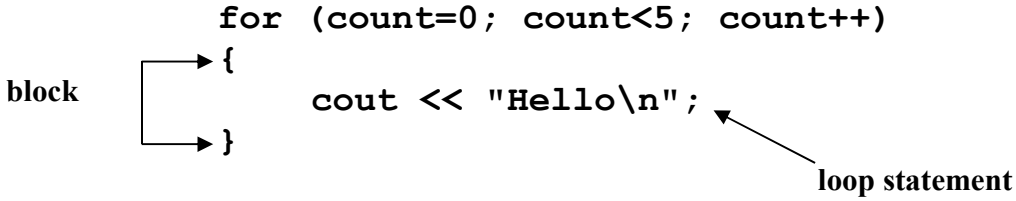
for - Loop

- The ***for*** loop allows the execution of a block of statements while a given condition is TRUE.

- The format of the ***for*** statement is as shown below :

```
for (initialisation; test condition; re-initialisation)
{
     statement block;
}
```

Example:



where:

initialisation	-	is the starting condition (usually initializing a variable) **count = 0**
test condition	-	if *test condition* is TRUE, the loop will continue execution **count<5**
re-initialisation	-	operation conducted at the end of the loop (usually up-dating the variable) **count**++

- If there is only one statement within the block then the braces can be omitted.

  Example :

```
for (count=0; count<5; count++)
     cout << "Hello\n";
```

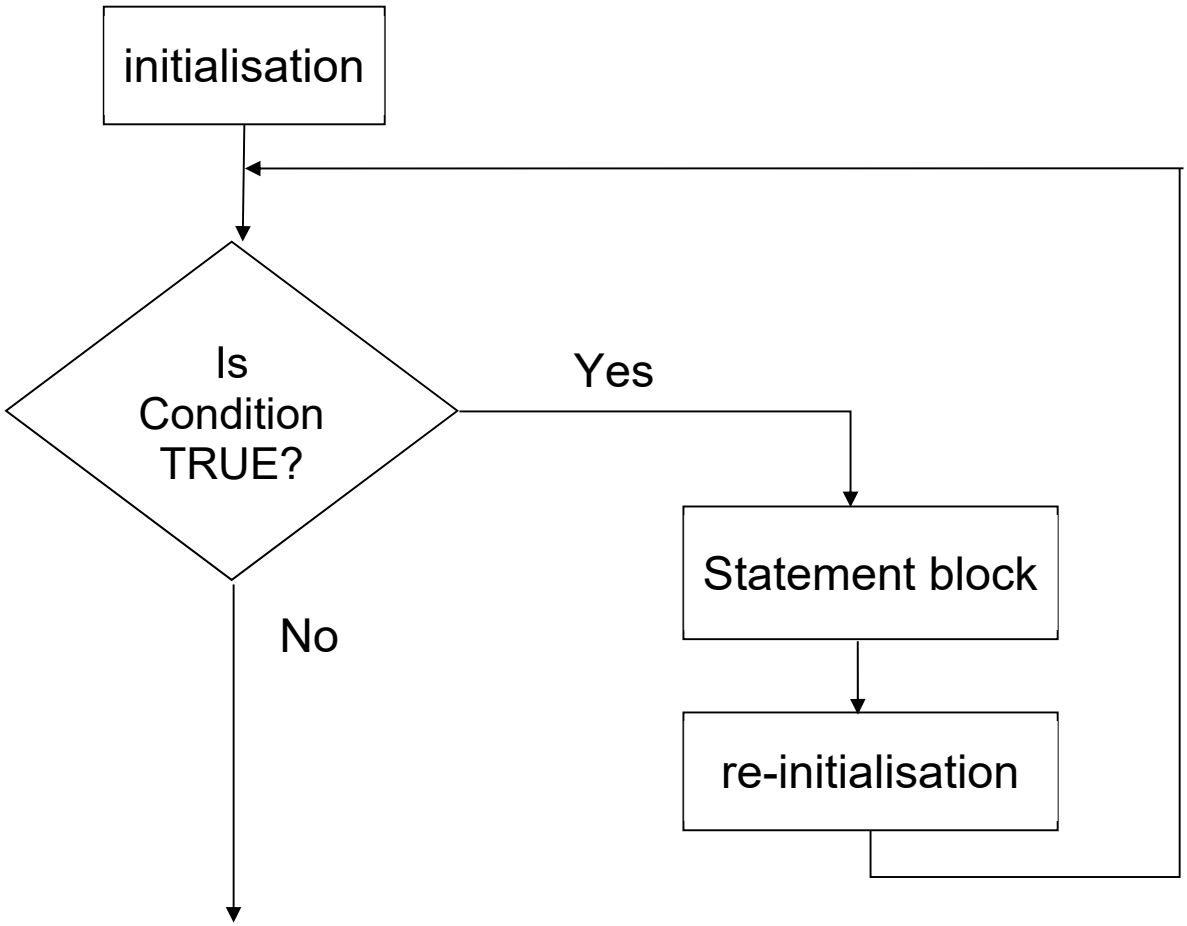
- Figure below shows the flow chart representation of this statement.

for (initialisation; test condition; re-initialisation)

{

statement block

}



Flowchart Representation of the *for* Statement

- The following example uses the *for* loop to print out numbers from 0 to 9.

```
// A sample for loop to print numbers from 0 to 9

#include <iostream>
using namespace std;

int main()
{
    int i;

    for (i = 0; i < 10; i++)
    {
        cout << " " << i;
    }
    return 0;

}
```

Activity

1. Analyze the program and fill in the table below:

`for(i=0;i<10;i++)`	True										
`{`											
`cout<<" " << i;`	0										
`}`											

2. What happens if the **for** statement is changed to the following?

```
for (i = 9; i >= 0; i--)
{
     cout << " " << i;
}
```

Analyze the program and fill in the table below:

`for(i=9;i>=0;i--)`											
`{`											
`cout<<" " << i;`											
`}`											

- The semicolons (;) inside the **for** statement's parenthesis are required as separators. All the other entries within the parenthesis are optional.

- The ***initialisation*** part can be taken out of the *for* statement and done before.

```
i = 0;   //-- initialisation done before the
         //-- for statement
for (  ; i < 10; i++)
{
     .....
}
```

- The ***re-initialisation*** part can also be taken out of the *for* statement and made a part of the loop statement.

```
i = 0;   //-- initialisation done before the
         //-- for statement
for (  ; i < 10;  )
{
     cout << "Hello\n";
     i++;   //-- re-initialisation done inside
            //-- the loop
}
```

- The ***condition*** part is also optional. The following *for* statement will become an **infinite loop**, also called an **endless loop**. An endless loop goes on and on forever.

```
for ( ; ; )
{
     cout << "Hello\n";
}
```

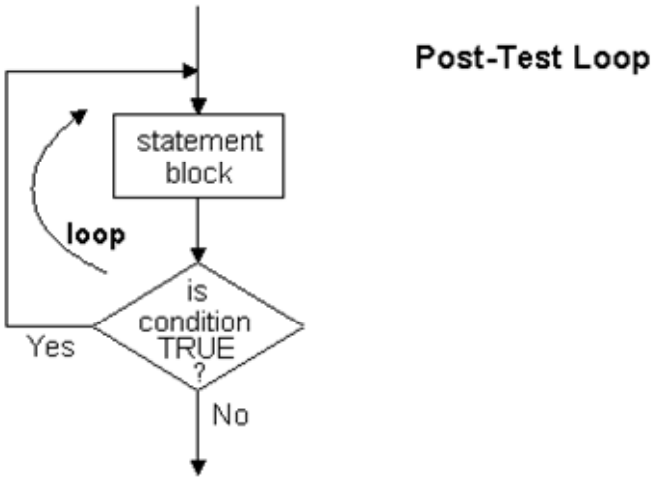
do..while - Loop

- The ***do..while*** statement is similar in its operation to ***while*** statement except that it tests the condition at the bottom of the loop.
- This allows the statement block to be executed at least once.
- The syntax is as follows :

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

Must put semicolon here

- A TRUE condition will cause the statement block to be repeated.
- As with ***for*** and ***while*** loops, the braces are optional for one statement only.
- The ***do..while*** loop requires a semicolon at the end of the loop, whereas the ***while*** does not.



Flowchart Representation of the *do..while* Statement

- The previous program which finds the average of a series of numbers could be improved using the **do..while** statement.

```
/* Find average of a series of numbers */

#include <iostream>
using namespace std;

int main()
{
 int count;
 double number, sum, average;
 count = 0;    // Initialize Count to 0
 sum = 0;      // Initialize Sum to 0

 do
 {
      // Prompt user to enter Number
      cout << "\nEnter Number (0 to terminate): ";

      cin >> number;    // Read Number
      if (number != 0)
      {
           sum = sum + number;
           count = count + 1;
      }

 } while(number != 0);
 // Repeat while number not equal to 0

 // Average = Sum/Count
 average = sum/count;

 // Display Average
 cout << "\nThe average of the " << count
      << " numbers is " << average << endl;
 return 0;
}
```

- Note that the prompting message and the *input* statement outside the while loop are not required in the program.

Activity

Analyze the program and fill in the table below, assuming that the following values are entered.

5, 7, 4, 2, 0

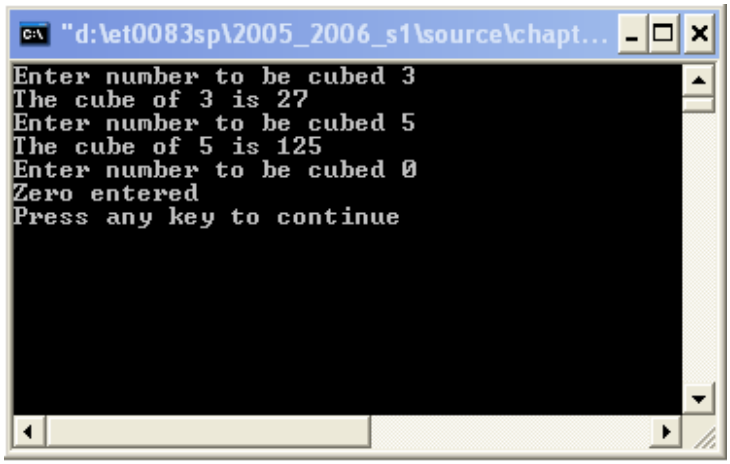
	`count =`					
	`sum =`					
	`do`					
Loop	`{`					
	`cin >> number;`	5	7	4	2	0
	`if (number !=0)`	True				
	`sum = sum + number;`					
	`count = count+1;`					
	`}`					
	`average=sum/count;`					

break

- The ***break*** statement is used to exit from a repetitive loop.
- It can be used with the ***for*** , ***while*** and ***do..while*** loops.
- Only the ***break*** statement can cause infinite loops (***while(1)***, ***do..while(1)*** *and* ***for(; ;)***) to terminate.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int i;
   for (; ;)     //---- endless loop ----
   {
        cout << "Enter number to be cubed ";
        cin  >> i;
        if (i == 0) break; //---- Stop infinite loop

        cout << "The cube of " << i
             << " is " << i*i*i << "\n";
   }
   cout << "Zero entered\n";
   return 0;
}
```



Chapter 6

Functions

Introduction

- All programs written so far have consisted of a single long function called *main()*.

- We will now learn to break up our program into several small functions (modules) which will be **called** (or **invoked**) from *main()*.

- A function is a block of instructions to perform a specific task.

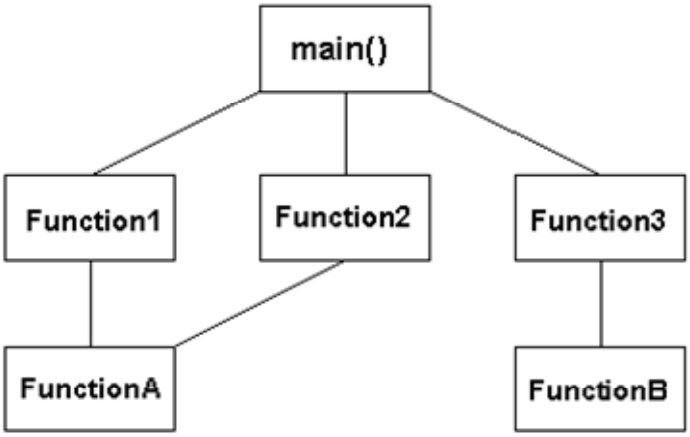
- Functions can be called from any part of a program and allow large programs to be split into smaller manageable tasks.

- By breaking up our program into several smaller functions, it will be easier for us to:

  - isolate problems
  - write programs faster
  - produce programs that are easier to maintain and upgrade.

- Functions are also useful in building libraries of routines that other programs can use **(reuse of code)**.

- The figure below shows a *main()* function calling several other functions (or modules).



Structure Chart

- A function can be thought of as a "black box" which takes in some value (if required) and outputs some result (if required). The internal details of the function can be hidden from the rest of the program, e.g. in **sqrt(x)** and **pow(y, z)** functions.

- Functions can be classified into two categories, namely, **Library Functions** and **User Defined Functions**.

- **sqrt(x)** and **pow(y, z)** functions you learned in Chapter 3 are examples of Library Functions which need **#include <cmath>** statement to be added in the program to use the functions.

- The main distinction between User Defined and Library Function is that the former is written by a user in a program while the latter can be found in the standard C++ libraries, or library functions provided by other parties.

- Shown below is an illustration of how a program can be re-written with **function calls** to perform the tasks.

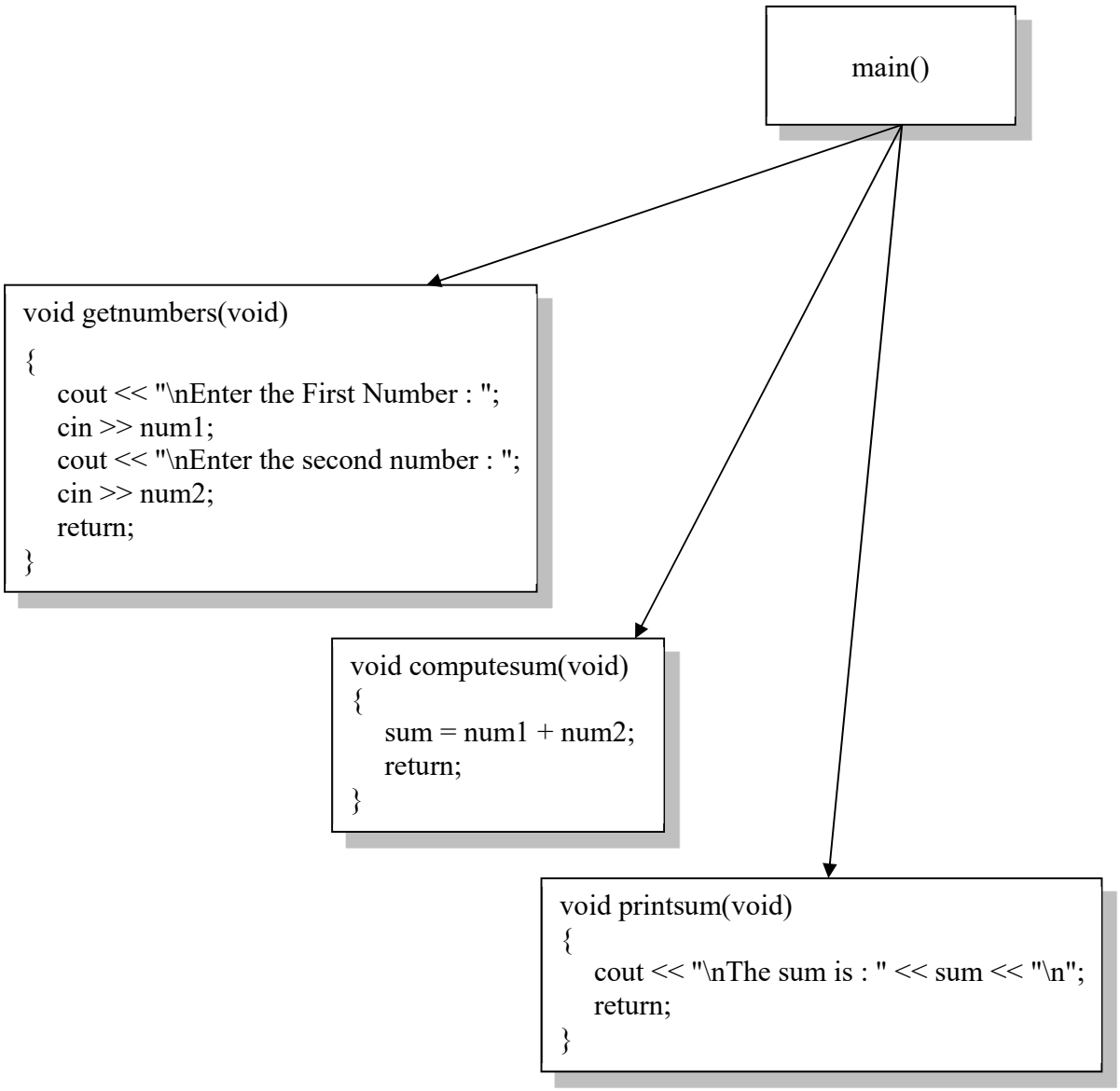
```
/* This program reads two integers from the keyboard
and displays out their sum  */

#include <iostream>
using namespace std;

int main()
{
   int num1, num2, sum;

   //---- Read Numbers -----------------
   cout << "\nEnter the first number : ";
   cin >> num1;
   cout << "\nEnter the second number : ";
   cin >> num2;

   //---- Compute answer ----------------
   sum = num1 + num2;

   //---- Display Result ----------------
   cout << "\nThe sum is : << sum <<"\n";
   return 0;
}
```

Get the numbers from the user

Compute the sum

Display the sum

- We shall now rewrite this program by breaking it up into functions.
- The **main()** function calls three other functions :
  - **getnumber()**
  - **computesum()**
  - **printsum()**



Structure Chart for the Problem

- The complete program is shown below :

```
#include <iostream>
using namespace std;

//----  Global Variables ----
int num1, num2, sum;

//---- Function Prototypes ----
void getnumbers(void);
void computesum(void);
void printsum(void);

int main()
{
    getnumbers();  // Get two numbers from User
    computesum();  // Compute the Sum
    printsum();     // Print the Sum
    return 0;
}

void getnumbers(void) // Get two numbers from user
{
    cout << "\nEnter the First Number : ";
    cin >> num1;  // Get first number
    cout << "\nEnter the second number : ";
    cin >> num2;  // Get second number
    return;
}

void computesum(void) // Compute the Sum
{
    sum = num1 + num2;  // Sum is 1st Num + 2nd Num
    return;
}

void printsum(void)   // Print the Sum
{
    cout << "\nThe sum is : " << sum << "\n";
    return;
}
```

- It is important to note that a program will always begin executing from ***main()***.

- Therefore every C++ program must have a ***main()*** function.

- A ***Function Prototype*** defines the format of a function before it is used or written.

- All functions ***must*** have a prototype.

- A call to a function must match its prototype.

- Each function prototype must be terminated by a semicolon (;) whereas the actual function header should not have a semicolon.

- Notice that the variables are now declared at the top of the program and outside any of the functions. These are known as ***Global variables***.

- ***Global Variables*** that are declared at the top of the program can be accessed by all the functions.

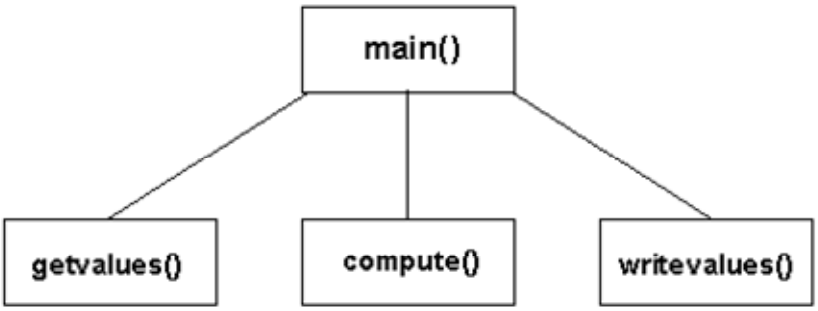
- If the variables are declared inside ***main()***, the variables cannot be accessed by other functions.

Example:

Write a program to allow the user to key in the length and width of a rectangle. Your program is to compute and print its area and perimeter.

Solution:

- We will divide the process among 4 functions:
  - main() – entry point of the C++ program
  - getvalues() – to ask user for length & width of rectangle
  - compute() – to calculate the area and perimeter
  - writevalues() – to display the results



Structure Chart for the Problem

Shown below is the complete program:

```
#include <iostream>
using namespace std;

double length, width, area, perim;
void getvalues(void);
void compute(void);
void writevalues(void);

int main()
{
    getvalues();
    compute();
    writevalues();
    return 0;
}
void getvalues(void)
{
    cout << "\nEnter length : ";
    cin >> length;
    cout << "\nEnter width : ";
    cin >> width;
    return;
}
void compute(void)
{
    area = length * width;
    perim = 2 * (length + width);
    return;
}
void writevalues(void)
{
    cout << "For a length of " << length
       << " and a width of " << width << ",\n";
    cout << "the Area is " << area
       << " and the Perimeter is " << perim << ".\n";
    return;
}
```

- Consider the function prototype

```
void getvalues(void);
```

- The ***void*** before the function name indicates that the function does not return any value.

void means empty!
(It means no data sent back to the **Calling Function**, main ( ) in this case, when the **Called Function** completes its task.)

- If a function returns nothing i.e. void, the last statement of the function must be :

```
return;
```

- The second ***void*** within the parenthesis indicates that the function does not have any arguments.

(We will be learning about arguments later).

- C++ does not allow you to put spaces within a function name.
  For example,

get values() is an INVALID function name.

- You may, however, join the words together as:

`getvalues()` OR `get_values()`

Be sure to use the underscore character (_) and NOT the hyphen (-) when naming functions and variables.

- In future all your programs should be written using functions.

Global versus Local Variables

- A ***Global variable*** is a variable that is declared outside any function.

- Also called ***public variable*** or ***external variable***.

- Since a Global variable is not declared within any particular function, all the functions following its declaration will have access to this Global variable and will be able to change the value of this variable.

- Shown below is a simple program that uses Global Variables.

- The program reads two numbers from the keyboard, finds their sum and displays the result.

```
/*
Read 2 numbers from the keyboard, find the sum of the two
numbers and prints out the result */

#include <iostream>
using namespace std;

int num1, num2, sum;//-- Global Variables -----

void printsum(void);//-- Function Prototypes --

int main()
{
    cout << "\nEnter the first number : ";
    cin >> num1;  //-- Get first number ---
    cout << "\nEnter the second number : ";
    cin >> num2;  //-- Get second number --

    sum = num1 + num2;//-- Compute sum --------

    printsum();        //-- Print the sum ------
    return 0;
}


void printsum(void)
{
    cout << "\nThe sum is " << sum << "\n";
    return;
}
```

Dangers in using Global Variables

- As seen in the above program, the Global variables can be accessed from *main()* and also from *printsum()*.

- Since the value of a Global Variable can be changed by any function, this global variable may be assigned a value in one function and accidentally overwritten by another function.

- Though the function ***printsum()*** is not supposed to modify the value of the global variable, *sum*, it can very easily do so since it has full access to it.

- If your program is large, and has, say, about 25 functions, you will have a difficult job trying to find out exactly which function has changed the value of this Global Variable.

- You should therefore do your best to avoid using Global variables in future.

Local Variables

- A ***Local Variable*** is a variable that is declared inside a particular function (usually after the opening brace {).

  Also called ***private variable*** or ***internal variable***.

- Since a Local Variable is declared within a particular function, they can only be accessed from within that particular function.

- Therefore, no other function will have access to this Local Variable and hence no other function will be able to change the value of this Local Variable.

- Therefore it is better to declare all your variables as Local Variables within the function that need to access them.

- What happens if you want to use a Local Variable in another functions?

- We can do it by **passing variables from one function to another**, in two ways:
  - **passing by value**
  - **passing by reference**

- Let's modify the previous program to use Local Variables instead of Global Variables.

```
/*
Reads two numbers from the keyboard, finds the sum of the
two numbers and displays the result */

#include <iostream>
using namespace std;

void printsum(void);   //-- Function Prototype --

int main()
{
    int num1, num2, sum;  //-- Local Variables --

    cout << "\nEnter the first number : ";
    cin >> num1;   //-- Get first number --
    cout << "\nEnter the second number : ";
    cin >> num2;   //-- Get second number --

    sum = num1 + num2;//-- Compute sum --

    printsum();            //-- Print the sum --
    return 0;
}


void printsum(void)
{
    int sum;      //-- Local Variable --

    cout << "\nThe sum is " << sum << "\n";
    return;
}
```

sum

30

sum

?

- In the above program, since we are calculating the sum in the *main()* function and printing the sum in the *printsum()* function, *sum* has to be used in both the functions.

- However, if the above program is executed, it will display an incorrect value for sum.

- This is because the *sum* declared in *printsum( )* is different from the *sum* declared in *main( )* though both variables have the same name.

- The program displays the value of *sum* that is local to *printsum( ).*

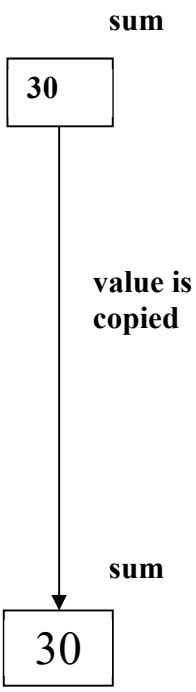
- Since this has not been given any value, it prints out an incorrect value.

- A local variable comes into existence inside the called function.

- It ***vanishes*** when the function is over.

- The solution is to have a single local variable called *sum* in *main()* and to pass this local variable as an "argument" to *printsum()* by passing a value (sending a copy) to the called function as shown below :

```
/*
Reads two numbers from the keyboard, finds the sum of the
two numbers and displays the result */

#include <iostream>
using namespace std;

void printsum(int sum);   //-- Function Prototype --

int main()
{
    int num1, num2, sum;  //-- Local Variables

    cout << "\nEnter the first number : ";
    cin >> num1;   //-- Get first number
    cout << "\nEnter the second number : ";
    cin >> num2;   //-- Get second number

    sum = num1 + num2;

    printsum(sum);          //-- Print the sum
    return 0;

}

void printsum(int sum)
{
    cout << "\nThe sum is " << sum << "\n";

    return;
}
```



actual parameter

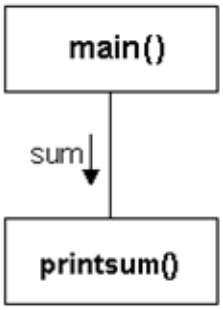
formal parameter

- The parameter with which the function *printsum()* is executed is called the ***actual parameter***.

- The parameter with which the function is declared is called the ***formal parameter***.

- The function *printsum()* receives the variable *sum* as a "parameter" from *main()*. The **value** of the actual parameter, *sum*, in the *main()* function **is copied** to the formal parameter, *sum*, in the function *printsum()*.

- The function *printsum()* treats the formal parameter, *sum,* as though it is a local variable, and prints out its value.



Structure Chart for the Problem

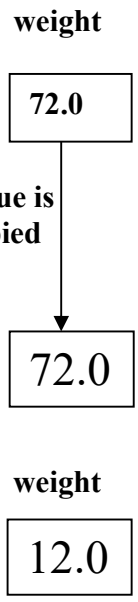
- Although it may seem easier to make all your variables Global, this should not be done.

- If variables are kept local only to the functions that need them, you are protecting their values, and you are also keeping the code and data of your programs fully modular.

Activity

The following program asks the user for his weight. It then passes the weight as an argument to a function that calculates the person's equivalent weight on the moon.

```
/*
Calculates a person's weight on the moon by passing the
weight on earth as an argument to another function */

#include <iostream>
using namespace std;

_____________________________   //-- Function Prototype --

int main()
{
    double weight;     //---- Local variable ----

    cout << "\nHow many kilos do you weigh ? ";
    cin >> weight;

    _____________________ // Calling Moon function
    return 0;

}

_____________________________
{

    //-- Weight on moon = 1/6th Weight on earth --

    _____________________ // Calculate weight on moon
    cout   << "You weigh " << _____________
           << " kilos on the moon\n";

    return;
}
```



- When the function ***moon()*** is called, the *formal parameter* ***weight*** comes into existence and gets its initial value from the actual parameter.

- When the function ***moon()*** is over, the *formal parameter* ***weight*** vanishes.

- **If the value of the *formal parameter* is changed, it does not affect the value of the *actual parameter*.**

- When a variable is passed to a function as an argument, the name of the *actual parameter* need not be the same as the name of the *formal parameter*.

Rewrite the moon function with a different name for the formal parameter. Note that you do not need to change anything in the main function.

```
_____________________________
{

    //-- Weight on moon = 1/6th Weight on earth --

    _____________________ // Calculate weight on moon
    cout   << "You weigh " << _______________
           << " kilos on the moon\n";

    return;
}
```

Review Questions

- Why is it desirable to avoid using *Global Variables*?

  _____

  _____

- What is a Local Variable?

  _____

  _____

- Suppose you have a Local Variable declared inside a particular function. If you want to use the value of this Local Variable in another function, how would you do it?

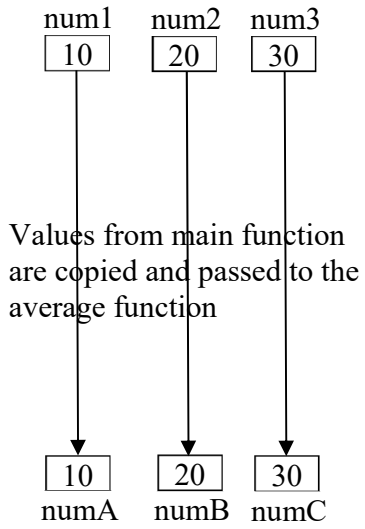
  _____

  _____

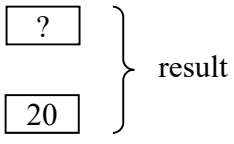
Functions that return a value

- Up till now, the functions we have written do not return any value using the *return* statement.
- Functions can also be used to return a value using the *return* statement.
- A *return* statement returns a single value from a called function to the calling function.

```
/* Calculates the average of three input values */
#include <iostream>
using namespace std;

//---- Function Prototype ----
double average(double num1, double num2, double num3);
//-- OR double average(double, double, double);

int main()
{
     double num1, num2, num3, avg;
     cout << "\nEnter first number : ";
     cin >> num1;
     cout << "\nEnter second number : ";
     cin >> num2;
     cout << "\nEnter third number : ";
     cin >> num3;
     avg = average(num1, num2, num3);
     cout << "\nThe average is " << avg;
     return 0;
}
```




```
double average(double numA,double numB,double numC)
{
     double result;

     result = (numA + numB + numC)/3;
     return(result);
}
```



- The function *average()* is similar to the functions we have used before except for the following differences :

  1. The function header has the word *double* before the function name.
     `double average(double numA,double numB,double numC);`

     This indicates that the function will return a value of type *double*.

     In this case, the function returns the average of the three numbers, which is a *double.*

     (If there is nothing before the function name, C assumes that the function will return an *int*)

  2. The *return* statement of the function *average()* now has some variable within the parenthesis.

     ```
     return(result);
     ```

     This is the *double* value that is sent back to *main(),* the calling function.

     Note that a function can return only a single value to the calling function.

- When the value is returned to function *main(), main()* must do something with that returned value.

  Notice that in *main()*, the function call is now on the right side of the assignment (=) operator.

  ```
  avg = average(num1, num2, num3);
  ```

  The returned value is therefore assigned to the local variable *avg.*

  For example, if the returned value is 25.8, the compiler "sees" this statement as

  avg = 25.8;

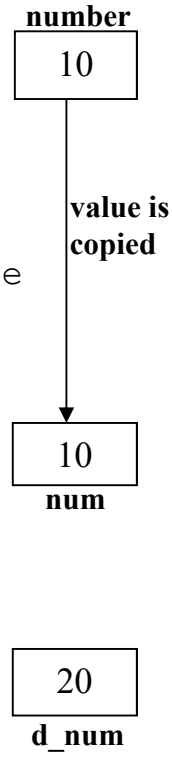
- Note that when a function is returning a value, you should not simply call the function as

  ```
  average(num1, num2, num3);
  ```

  If you do this, the compiler will have nowhere to put the returned value of 25.8, wasting the work done by the function.

- Shown below is a simple program that passes an integer as an argument to a function. The function takes the integer as a parameter and returns a value that is twice the integer.

```
/*  Doubles the user's number */

#include <iostream>
using namespace std;

int twice(int num);        //-- Function Prototype --

int main()
{
    int number;
    int result;

    cout << "What number do you want doubled? ";
    cin >> number;

    result = twice(number); //-- Assign returned value

    cout << number << " doubled is " << result;
    return 0;
}

int twice(int num)
{
    int d_num;

    d_num = num * 2;   //-- Double the number --
    return(d_num);          //-- Return the result --
}
```

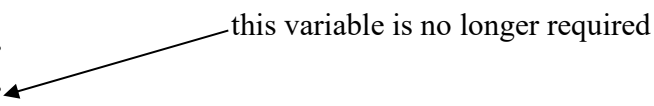


Output:

What number do you want doubled? 10 <ENTER>
10 doubled is 20

- The *main( )* function could be modified and written as follows :

```
/* Doubles the user's number */

#include <iostream>
using namespace std;

int twice(int num);    //-- Function Prototype --

int main()
{
 int number;
 int result;

 cout << "What number do you want doubled? ";
 cin >> number;
 cout << number << " doubled is " << twice(number) ;
 return 0;
}

int twice(int num)
{
 int d_num;

 d_num = num * 2; //-- Double the number --
 return(d_num);        //-- Return the result --
}
```



- Notice that in the *main( )* function, the value returned by the function *twice( )* is NOT assigned to the variable *result* before printing.
- The function is simply called within the *cout* statement itself and the value returned is directly displayed.
- Although this method has one variable less in *main( )*, it is often clearer to call the function and assign its return value to a variable before using it.

Activity

1. How many values is a function capable of returning? _____

2. If there is no data type before the function name,
   e.g. twice(int num);
   what is the data type of the value that is returned by this function? _____

3. How many arguments can be passed to a function? _____

4. What is wrong with the following code? _____

```
double average(int num)
{
  double sum, total = 0;
  // Some C++ statements
  return (total, sum);
}
```

5. Determine the function prototypes for the dummy functions :
   fun1(), fun2(), fun3() & fun4().

```
int main()
{
      int ivar=5; double dvar=2.4;
      fun1(ivar);
      ivar = fun2(dvar);
      dvar = fun3();
      fun4(ivar,dvar);
      return 0;
}
```

The function prototypes are best determined by observing how the functions are being used.
There are two parts

1. The parameters passed to the function,
   ie. whatever is within the ( ).
2. The return value from the function.

```
fun1(ivar);
```

The fun1( ) expects one parameter to be passed into the function and it is of type int. There is also no return value required.

So the prototype is _____

```
ivar = fun2(dvar);
```

The fun2( ) also expects one parameter to be passed into the function but it is of type double. It requires a return value that should be of the same data type as ivar, which is type int.

So the prototype is _____

```
dvar = fun3();
```

The fun3( ) expects no parameter to be passed into the function. It requires a return value that should be of the same data type as dvar, which is type double.

So the prototype is _____

```
fun4(ivar,dvar);
```

The fun4( ) expects two parameters to be passed into the function and their data types are int and double respectively. There is also no return value required.

So the prototype is _____

Chapter 7

Arrays & Strings

Introduction

- An array stores more than one values of the same data type, under a collective name.

- We have previously learnt how to declare variables

  e.g.
  ```
  char initial;  //-- A character variable
  int number;  //-- An integer variable
  double value;  //-- A floating point variable
  ```

  These variables can each be used to store one value only.

- However, if we want to store a name, will a single character variable be enough?

  For storing the marks for all the students in this class, will a single variable be enough?

- In C++, a string of characters or a set of numbers must be stored in an Array.

- An array is simply a group of variables of the **same data type**, which are stored next to each other in the memory.

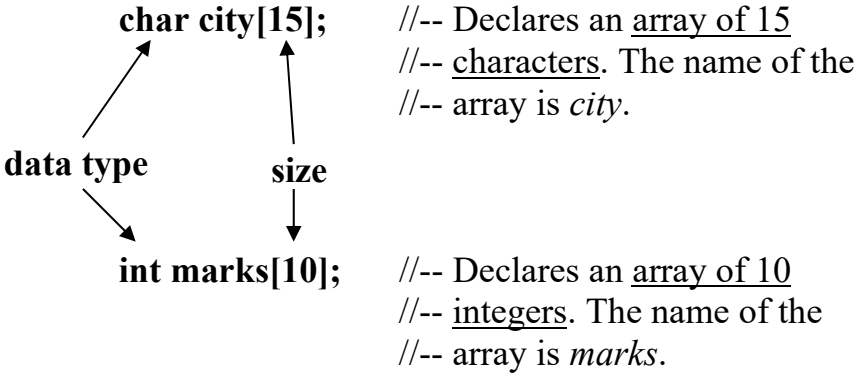
- The components of an array are often called the **elements** of the array.

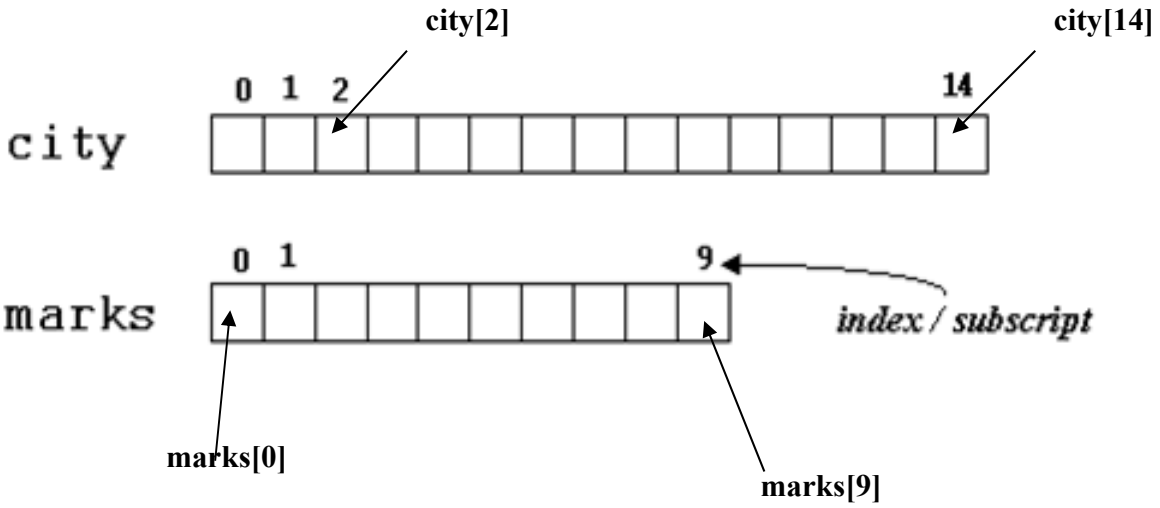
Single-subscripted Array Declaration

- The syntax for declaring an array is :

data_type array_name[size];

e.g.



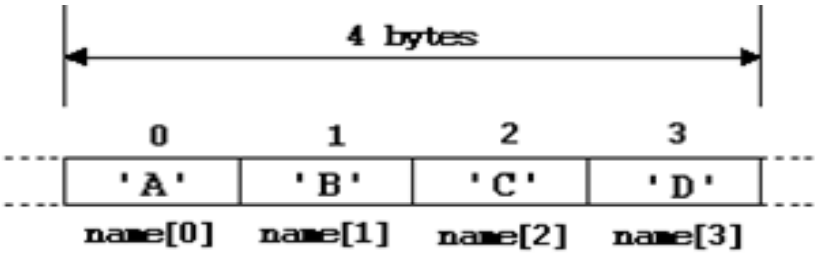


- Each element of the array ***city*** is a character data. Hence the total memory occupied by the array is 15 bytes.
- Each element of the array ***marks*** is an integer. Hence the total memory occupied by the array is 40 bytes.
- The **index** number (i.e. the **subscript**) **starts with 0**.

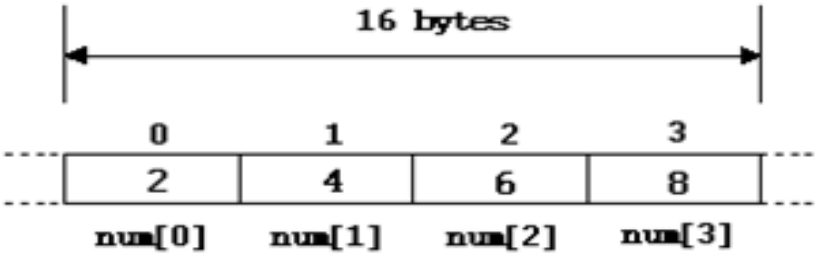
- When we declare an array, we may also assign values to it at the same time as follows :

```
#include <iostream>
using namespace std;
int main()
{
     char name[4] = {'A', 'B', 'C', 'D' };
     int num[4] = { 2, 4, 6, 8};
          :
          :
}
```

- The figures below show what the arrays look like in the memory of the computer.



Character Array called name



Integer Array called num

Character Arrays and Strings

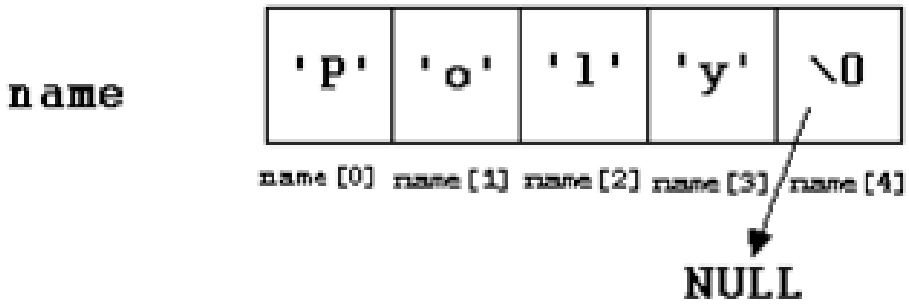
- In C++, a string of characters (e.g. a name or an address) must be stored in a character array.

- The example below shows the declaration of a character array followed by a string assignment to it at the same time.

```
#include <iostream>
using namespace std;
int main()
{
     char name[5] = "Poly";
          :
          :
}
```

A string must be enclosed within double-quotes

- The figure below shows what this character array looks like in the memory of the computer :



- Note that there is a NULL character (ASCII 0) at the end of the string. This is a string terminating character.

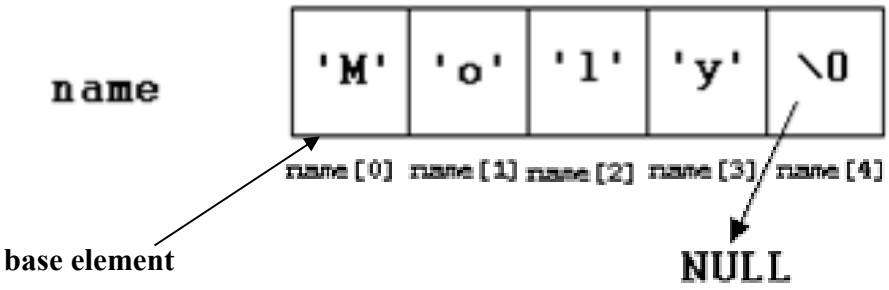
Accessing array elements

- We can access individual elements within an array, or we can access the array as a whole.

- Individual elements are accessed using the **index / subscript** :

  name[0] = 'M';

- This overwrites the *'P'* with a *'M'* and the array would now look like this :



- The number within the square brackets is called the index / subscript.

- All array subscripts start at zero. The zeroth element of the array is called the ***base element*** of the array.

- We can display the entire character array with a single *cout* statement as shown follows :

```
#include <iostream>
using namespace std;
int main()
{
     char name[5] = "Poly";
     cout << name;
          :
          :
}
```

- Notice that when we display a complete character array, we do not put the square brackets after the array name.

- What will be the output of the following *cout* << statement?

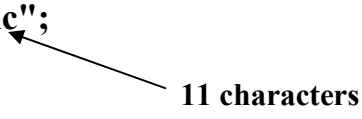
cout << name[1];

- We must always reserve enough array elements to hold the string, plus its terminating character, the NULL.

  e.g.

  char name[5] = "Polytechnic";

  would be incorrect.



- If the string contains 11 characters, the array size should be at least 12 in order to include the NULL character.

- If you leave empty brackets and assign a value to the character array at the same time, C++ will automatically reserve the correct amount of space.

  The declaration

  char school[4] = "EEE";

  and

  char school[] = "EEE";

  both achieve the same purpose.

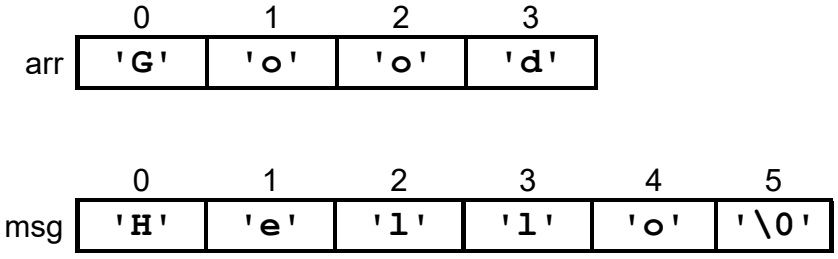
- However, you should never leave empty brackets unless you assign a value to the array *at the same time.*

  e.g.

  char school[];

  will not reserve any space for the array and is in fact illegal.

- Look at the two arrays shown below :



- The first array, called ***arr***, is a character array, but it does not contain a string.

- Instead of a string, it simply contains a list of several characters.

- The second array, called ***msg***, contains a string because it has a NULL character at the end.

- The first array could be initialized with 4 individual characters as follows :

  char arr[] = {'G', 'o', 'o', 'd'};

- The first array could also be initialized using assignment statements as follows :

```
char arr[4];

arr[0] = 'G';
arr[1] = 'o';
arr[2] = 'o';
arr[3] = 'd';
```

- Because this array does not contain a NULL, the array cannot be treated as if it were a string.

- The second array could be initialized with a string as follows :

  char msg[] = "Hello";

  (The NULL character will be inserted automatically).

- It could be initialized in the following manner as well :

  char msg[]={ 'H', 'e', 'l', 'l', 'o', '\0' };

  (The NULL character '\0' is inserted explicitly)

- It is important to note that you **cannot assign a string to a character array in a regular assignment statement**, except when you first declare it in the same array declaration statement.

```
/*---- Valid Program ----*/
#include <iostream>
using namespace std;
int main()
{
     char institution[15] = "Singapore Poly";
     cout << institution;

               :
               :
}
```

- The above program will work and display the string.

- However, the program below is invalid :

```
/*---- Invalid Program ----*/
#include <iostream>
using namespace std;
int main()
{
     char institution[15];
     institution = "Singapore Poly"; //-- Invalid
     cout << institution;
               :
               :
}
```

- in order to change a string mid-way in the program, we can use a function **strcpy(string1,string2)**. The function will copy string2 to string1.

- A string may be assigned to an array using the string copy function ***strcpy()*** as shown below :

```
/* Valid Program */
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
     char institution[15];

     //-- institution = "Singapore Poly";

     strcpy(institution, "Singapore Poly");
     cout << institution;
               :
               :
}
```

cstring must be included if you use the function strcpy()

Not possible

- You *must* ensure that the character array is large enough to hold the string.

Example

The following program reads in a person's first name, middle name and last name, it then displays the person's initials and his full name.

```
#include <iostream>
using namespace std;

int main()
{
    char first[10];          //---- Hold First Name
    char middle[10];         //---- Holds Middle Name
    char last[10];           //---- Holds Last Name

    cout << "\nWhat is your First Name : ";
    cin >> first;
    cout << "\nWhat is your Middle Name : ";
    cin >> middle;
    cout << "\nWhat is your Last Name : ";
    cin >> last;

    //---- Print Initials and name ----------------
    cout << "\nYour initials are "
       << first[0] << middle[0] << last[0] <<"\n";
    cout << "\nYour name is "
       << first << middle << last;
    return 0;
}
```

	0	1	2	3	4	5
first	'A'	'n'	'g'	'\0'	?	?

	0	1	2	3	4	5
second	'B'	'e'	'n'	'g'	'\0'	?

	0	1	2	3	4	5
third	'S'	'o'	'o'	'\0'	?	?

Important Notes

When assigning a string value to a character array, the value has to be enclosed in double quotes.

e.g.

```
char institution[15] = "Singapore Poly";
```

However, when assigning a character value, the value has to be enclosed in single quotes.

e.g.

```
char initial = 'R';

char institution[15];

institution[0] = 'S';
institution[1] = 'i';
        :
        :
        :
        :
institution[14] = '\0';
```

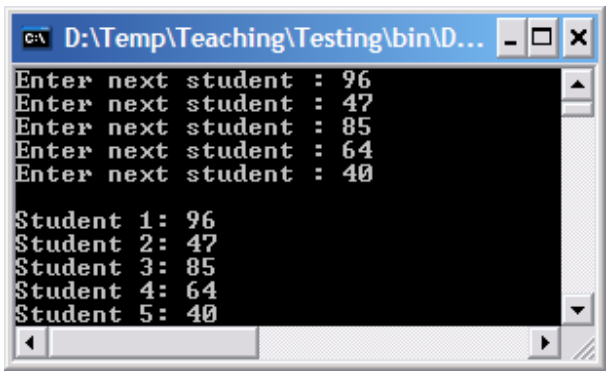
Examples of Arrays

- Arrays are useful when a list of values have to be processed. The following examples are related to the marks of a class of students.

- Write a program to allow a lecturer to capture the marks of a class of 5 students.

```
#include <iostream>
using namespace std;
int main()
{
     int marks[5]; // an array of 5 marks
     int i;
     // variable for other examples.
     int sum, gradeA;
     double average;
     // Getting the marks
     for ( i=0; i<5; i++)
     {
          cout << "Enter next student : ";
          cin >> marks[i];
     }
     // Printing the marks
     for ( i=0; i<5; i++)
     {
          cout << "\nStudent " << i+1 << ": " << marks[i];
     }
     return 0;
}
```

The sample inputs & outputs are :



Activity

1. Modify the program to find the average mark.

```
// adding all the marks
sum = 0;
for ( i=0; i<5; i++)
{
     sum = _____________________;
}
average = _____________________;
```

2. Modify the program to find the number of students with grade A. (Marks for Grade A is 80 and above).

```
gradeA = 0;
for ( i=0; i<5; i++)
{
     if (_____________________) gradeA++;
}
```

Multiple-Subscripted Array

- Arrays in C++ can have multiple subscripts.
- A common use of such arrays is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, two subscripts are required: The first index number (subscript) normally identifies the element's row and the second index number (subscript) normally identifies the element's column.
- To represent such tables, a minimum of two subscripts are required. This is known as double-subscripted arrays.

Double-subscripted Array Declaration

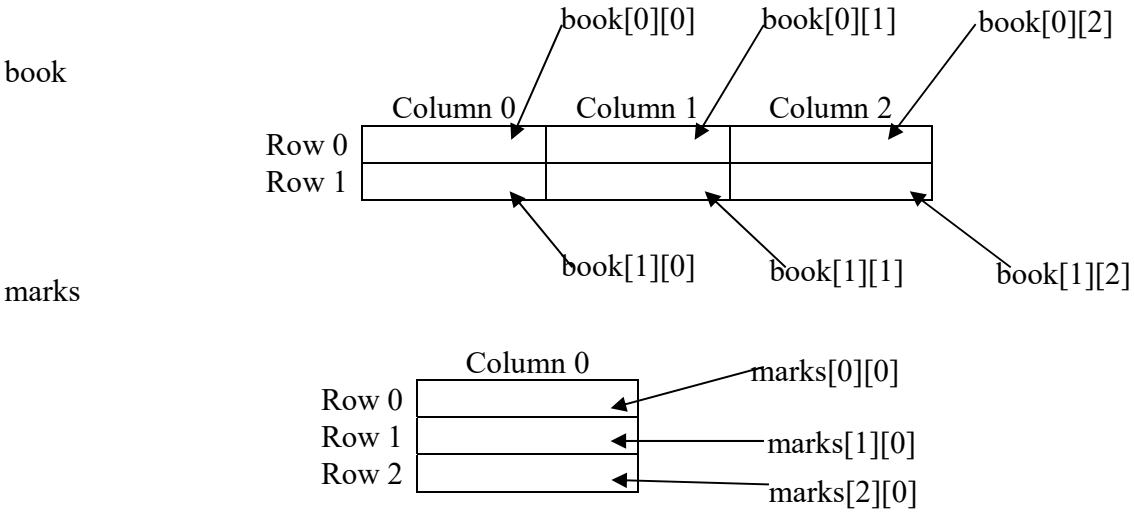
- The syntax for declaring such array is:

  data_type array_name[row_size][column_size];

e.g.

char book[2][3]; //-- Declares a double-subscripted array of 2 rows, 3 columns
//-- characters. The name of the array is book.

int marks[3][1]; //--Declares a double-subscripted array of 3 rows, 1 column
//--integers. The name of the array is marks.



Assigning values to double-subscripted Array

- As with single-subscripted array, we can assign values to double-subscripted array when we declare it at the same time:

```
#include <iostream>
using namespace std;
int main()
{
    char book[2][3] = {{'A','B','C'},{'X','Y','Z'}};
    // or char book[2][3] = {'A','B','C','X','Y','Z'};

    int marks[3][1] = { {300}, {400}, {500} };
    // or int marks[3][1] = { 300, 400, 500 };
    :
    :
}
```

What happens if you do not assign any value to some elements?

```
#include <iostream>
using namespace std;
int main()
{
     char book1[2][3] = {{'A','B'},{'X','Y','Z'}};
     char book2[2][3] = {'A','B','X','Y','Z'};

     int marks1[3][1] = {{300},{},{500}};
     int marks2[3][1] = {300,500};

     :
     :
}
```

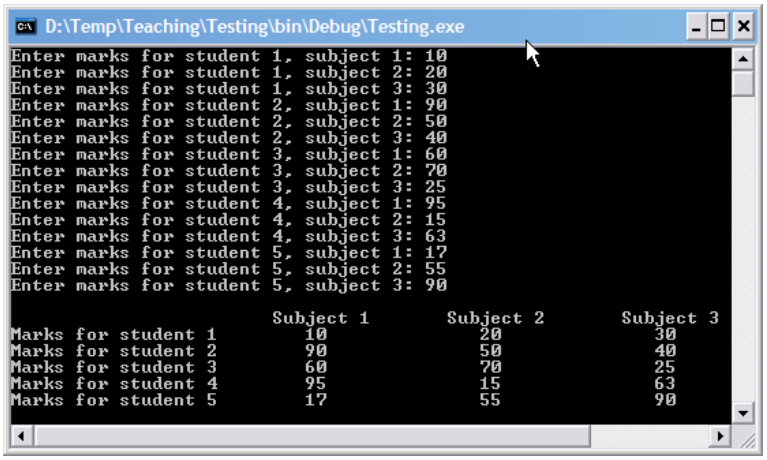
- What are the values of in book1 and book2? _____
- What are the values of in mark1 and mark2? _____
- Are they the same? _____
- What can you conclude from this activity? _____

Example of double-subscripted array

- Double-subscripted arrays are useful when a list of values in table form have to be processed. The following example is related to the marks of a number of students in a class.

```
#include <iostream>
using namespace std;

int main()
{
    int student = 5, subject = 3;
    int marks[student][subject];
    int i, j;

    // Getting the marks
    for (i=0; i < 5; i++)
    {
        for (j=0; j<3; j++)
        {
            cout << "Enter marks for student " << i+1;
            cout << ", subject " << j+1 << ": ";
            cin >> marks[i][j];
        }
    }
    // Display the marks
    cout << "\n\t\t\tSubject 1\tSubject 2\tSubject 3"
         << endl;
    for (i=0; i < 5; i++)
    {
        cout << "Marks for student " << i+1;
        cout << "\t";
        for (j=0; j<3; j++)
        {
            cout << "   " << marks[i][j] << "\t\t";
        }
        cout << "\n";
    }
    return 0;
}
```

The sample input & outputs are:



Passing Array as Arguments

- Arrays become a problem when used with functions. Since arrays can be very large, a different way of passing arguments to a function is required.

- The following program passes a single-subscripted array as an argument to the function add5marks(). The function adds 5 to every students' marks.

- Note that there is no need to return the values of the modified marks. This method of passing argument is called passing by reference. The address of the array is passed to the function and whatever in the array is changed in the function will also be affected in the main program.

```
#include <iostream>
using namespace std;
void add5marks(int marks[], int sz);

int main()
{
    int size=5;
    int i, marks[size] ;
    for ( i=0; i<size; i++)
    {
        cout << "Enter next student : ";
        cin >> marks[i];
    }
    add5marks(marks, size);
    for ( i=0; i<size; i++)
        cout << "Student " << i+1 << ": "
            << marks[i] << "\n";
    return 0;
}

void add5marks(int marks[], int s)
{
    int i;
    for(i=0; i<s; i++)
        marks[i] = marks[i] + 5;
    return;
}
```

Group Presentation Activity

For this activity, you have to divide yourselves into 4 groups. Each group will have to do the following:

1. Find out about C++ file I/O using the fstream header. (Refer to books or search the web)

2. Understand basic file I/O operation with C++. You only need to know how to:
   a. Open a text file for sequential access
   b. Store numbers / characters into the file (output)
   c. Retrieve numbers / characters from the file (input)
   d. Close the text file

3. Prepare PowerPoint slides on what you have learned.

4. Prepare a C++ program that uses the file I/O operation. Your program only needs to implement input or output. The program should be complete with user interface and allow user to interact with the stored or entered data.

5. Give a short group presentation to your class and also demo your C++ program during your presentation. You group must be able to explain to the class in detail how your program works.

6. Refer to some suggestions below for ideas on what your program can do.

   Program with file output operation:
   - User to enter a group of students' names and marks for a few subjects they have taken.
   - User to enter a list of groceries purchased. It should include date, description, cost, etc.
   - User to enter his / her daily expenses. It should include date, description, amount, etc.
   - User to enter his / her car fuel consumption history. It should include date, amount, brand, odometer reading, etc.
   - Any interesting program you want to try.

   You can show content of your saved file by opening it with notepad.

For file input operation, prepare a text file with relevant data using notepad.
Program with file input operation:

- Retrieve the students' names and all subject marks. Display all details and calculate average or number of failures / passes etc.
- Retrieve a list of groceries purchased. Display all details and calculate total cost for each trip.
- Retrieve daily expenses. Display all details and calculate expenses per month.
- Retrieve car fuel consumption history. Display all details and calculate average litres per km, etc.
- Any interesting program you want to try.

Note: Simple File I/O using fstream header may be tested in Quiz 7.

Beware of Plagiarism

Plagiarism in academia is considered as academic dishonesty or fraud. Offenders are subjected to academic censure or expulsion. It is important to know what is plagiarism.

To find out more, click on the following links:

- http://www.youtube.com/watch?v=atTRlg6iaGo&feature=related
- https://www.youtube.com/watch?v=6PeOTOO98BE

Do remember to cite the source if your program makes use of the codes or ideas from certain websites / books.

Chapter 8

Bitwise Operators

Bitwise Operators

- C++ has six Bitwise Operators :
  - One's Complement, or Bitwise Negation: ~
  - Bitwise AND: &
  - Bitwise OR: |
  - Bitwise EXCLUSIVE OR: ^
  - Left Shift: <<
  - Right Shift: >>

One's Complement, or Bitwise Negation: ~

- The unary operator ~ changes each 1 to a 0 and each 0 to a 1

  Example:

~(10011010) *is evaluated as* (01100101)

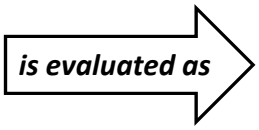
Bitwise AND: &

- The binary operator & produces a new value by making a bit-by-bit comparison between two operands.
- For each bit position, the resulting bit is 1 (true) only if both corresponding bits in the operands are 1.
- Example:

(10010011) & (00111101) *is evaluated as* (00010001)

Bitwise OR: |

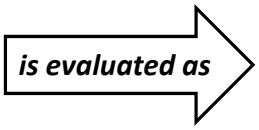
- The binary operator | produces a new value by making a bit-by-bit comparison between two operands.
- For each bit position, the resulting bit is 1 (true) if either of the corresponding bits in the operands is 1.
- Example:

(10010011) | (00111101) 

(10111111)

Bitwise EXCLUSIVE OR: ^

- The binary operator ^ makes a bit-by-bit comparison between two operands.
- For each bit position, the resulting bit is 1 (true) if one or the other (but not both) of the corresponding bits in the operands is 1.
- Example:

(10010011) ^ (00111101) 

(10101110)

Usage of Bitwise Logical Operators

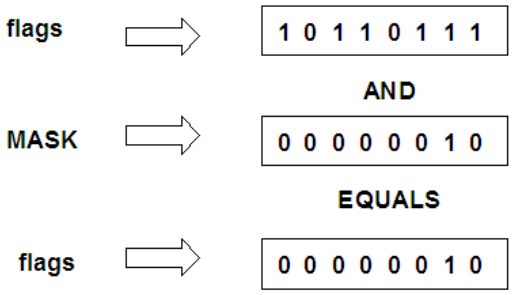
- The bitwise logical operators have the following usage:
  - Masks – hide the state of a bit
  - Turning Bits On – turn on particular bit
  - Turning Bits Off – turn off Particular bit
  - Toggling Bits – turning bit from **off to on** or **on to off**
  - Checking the value of a bit – determine the status of a bit

Masks

- The bitwise AND operator is often used with a mask. A mask is a bit pattern with some bits that are set to ON (1) and some bits to OFF (0).
- For example, suppose MASK is define to be 2, i.e. 00000010. Then the statement

  flags = flags & MASK;

  would cause all the bits of flags (except bit 1, second bit from the rightmost bit) to be set to 0.
- This process is called using a mask because the zeros in the mask hide the corresponding bits in flags

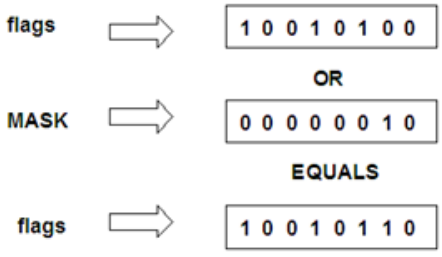


Turning Bits On

- Sometimes you may need to turn on particular bits in a value while leaving the remainder bits unchanged.
- This can be done with the bitwise OR operator. For example, MASK which has bit 1 set to 1. The statement

  flags = flags | MASK;

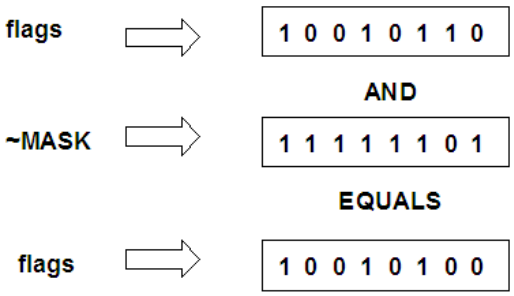
  will set bit number 1 in flags to 1 and leave all the other bit unchanged.



Turning Bits Off

- Just as it's useful to be able to turn on particular bits without disturbing the other bits, it's useful to be able to turn them off.
- Suppose you want to turn off bit 1, then in the variable flags. Once again, MASK is 00000010. You can do this:

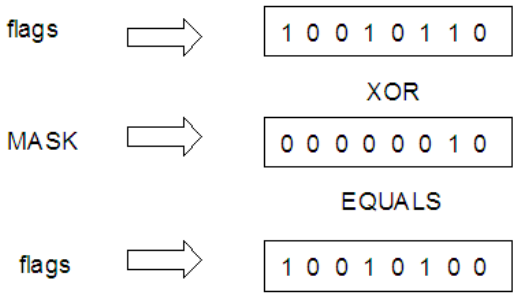
  flags = flags & ~MASK;
- Since MASK is all 0s except for bit 1, then ~MASK is all 1s except for bit1. A 1 AND with any bit will give the same value as that bit, so our statement leaves all the bits other than bit 1 unchanged.



Toggling Bits

- Toggling a bit means turning it OFF if it is ON, and turning it ON if it is OFF.
- You can use bitwise EXCLUSIVE OR operator to toggle a bit. The idea is that if b is a bit setting to (1 or 0), then 1^b is 0 if b is 1 and is 1 if b is 0. Also 0^b is b, regardless of its value.
- Therefore, if you EXCLUSIVE OR a value with a mask, values corresponding to 1s in the mask are toggled, and values corresponding to 0s in the mask are unaltered.
- To toggle bit 1 in flags, you can use:

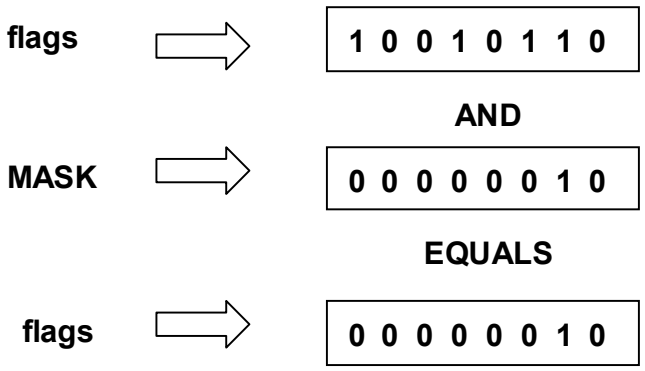
  flags = flags ^ MASK;



Checking the Value of a Bit

- Suppose that you want to check the value of a bit. For example, does flags have bit 1 set to 1?
- You must mask the other bits in flags so that you can compare only bit 1 of flags with MASK:

```
if ((flags & MASK) == MASK)
        cout << "Wow!";
```

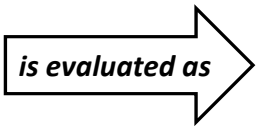


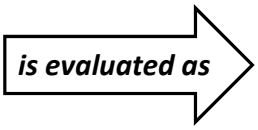
```
/* Tests the bitwise logical operators */

#include <iostream>
using namespace std;
#define MASK 0x2
int main()
{
     int flags;


     flags = 0x96;     // value in Hexadecimal

     cout << "The one's complement of flags is "
          << hex <<  (~flags) << "\n";
     cout << "The flags AND MASK is "
          << hex <<  (flags&MASK) << "\n";
     cout << "The flags OR MASK is "
          << hex <<  (flags|MASK) << "\n";
     cout << "The flags ^ MASK is "
          << hex <<  (flags^MASK) << "\n";
     return 0;
}
```

Output:

The one's complement of flags is ffffff69
The flags AND MASK is 2
The flags OR MASK is 96
The flags ^ MASK is 94

Bitwise Shift Operators

- The bitwise shift operators shift bits to the left or right.
- Left Shift: <<
  - The << left shift operator shifts the bits of the left operand to the left by the number of places given by the right operand.
  - Thus, (10001010) << 2 

(00101000)
  - The vacated positions are filled with 0s, and bits moved past the end of the left operand are lost.
- Right Shift: >>
  - The >> right shift operator shifts the bits of the left operand to the right by the number of places given by the right operand.
  - Thus, (10001010) >> 2 

(00100010)
  - For unsigned types, the places vacated at the left end are replaced by 0s. For signed types, the result is machine dependent.

Bitwise Shift Operators Usage

- The bitwise shift operators can provide swift efficient multiplication and division by powers of 2:

Number << n	Multiplies number by 2 to the nth power.
Number >> n	Divides number by 2 to the nth power if number is not negative.

```
/* Tests the bitwise shift operators */

#include <iostream>
using namespace std;

int main()
{
    int number;

    number = 0x96;

    cout << "The result of number << 2 is "
        << hex << (number << 2) <<"\n";

    cout << "The result of number >> 2 is "
        << hex << (number >> 2) << "\n";
    return 0;
}
```

Output:

The result of number << 2 is 258
The result of number >> 2 is 25

Exercise 1: Computer Basics & Designing Programs

1. A computer system is made up of _____and _____

2. What are the 4 main parts in a basic computer architecture?

3. What is a compiler used for?

4. Why is it easier to program in high-level languages than in machine language or assembly language?

5. What is an algorithm?

6. In the following questions, apply the design & development process up to Stage 4 only. Use both pseudocode and flowcharts in your algorithm.

   a. Design a program to add three integer numbers.
   b. Design a program to calculate the sum and difference of two whole numbers.

7. Design an algorithm in flowchart to display the smaller number of any 2 numbers input to the program (or keyed in by the user).

Exercise 2a: Using Code::Blocks

Learning Outcomes:

The student should gain the following basic knowledge:

- Use Code::Blocks
- Edit / Compile programs
- Single step & watch variables

Installing Code::Blocks

Download the program from eLearning@SP->ET0083->Learning Resources->Resources->CodeBlocks. Unzip and install the program. Use default setup during installation.

Using Code::Blocks

There are four steps to writing a program using CodeBlocks.

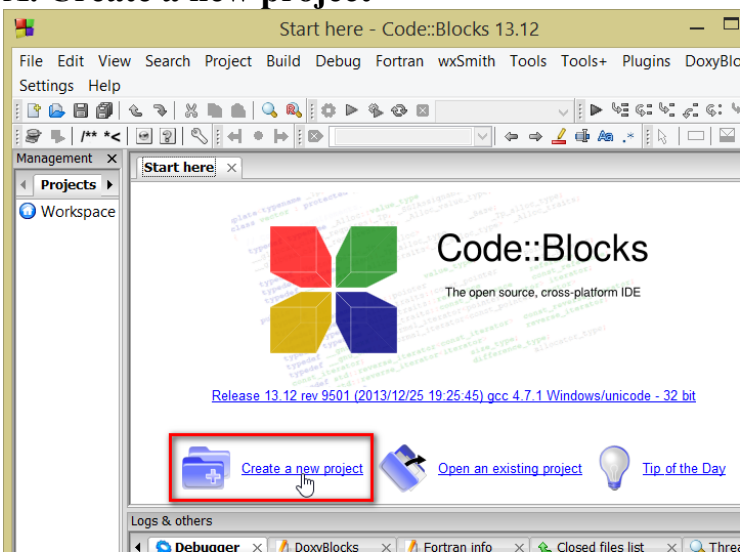
1. Create a new project : every program needs a project
2. Enter the C++ codes
3. Build project, (and if no errors)
4. Run the program

To Start CodeBlocks:

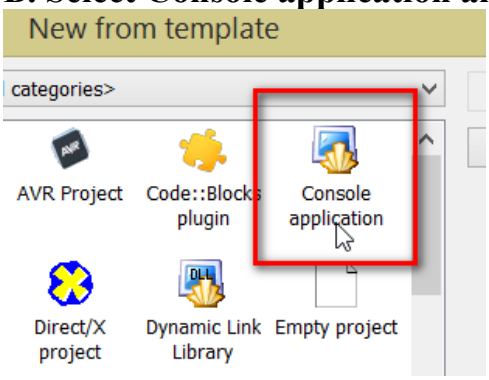
Look for the icon 

and double click it.

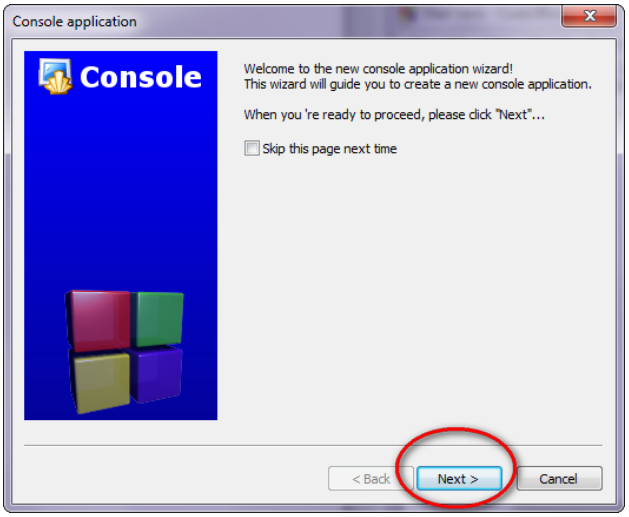
A. Create a new project



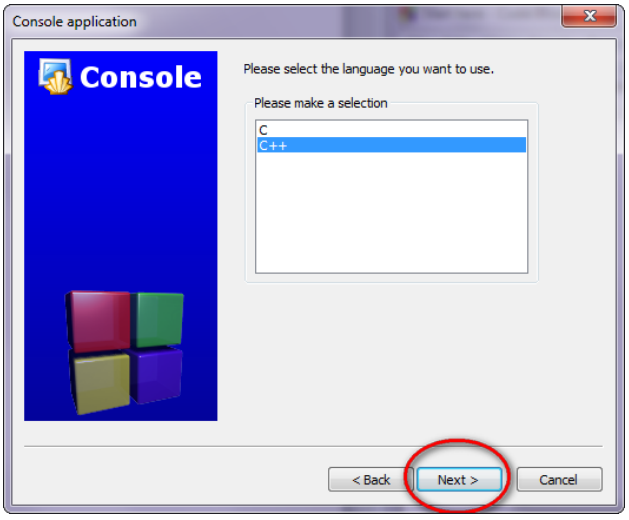
B. Select Console application and click the Go button



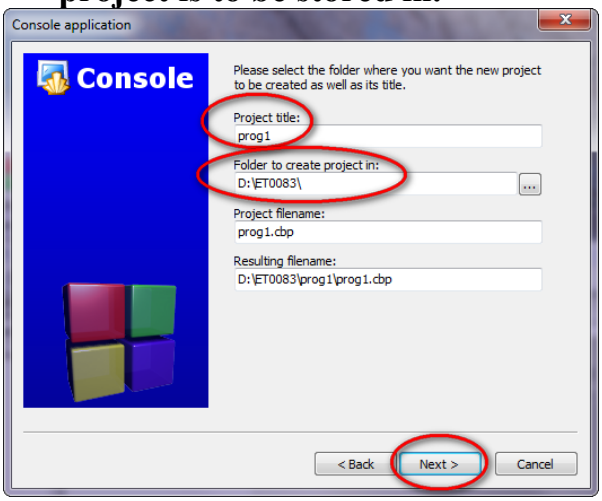
C. Click the Next> button



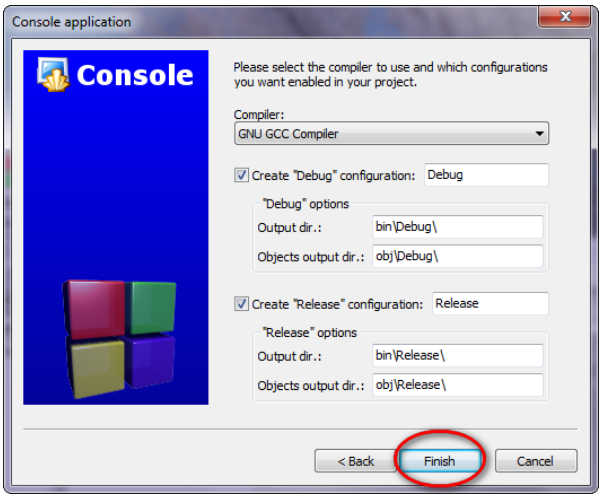
D. Click the Next> button



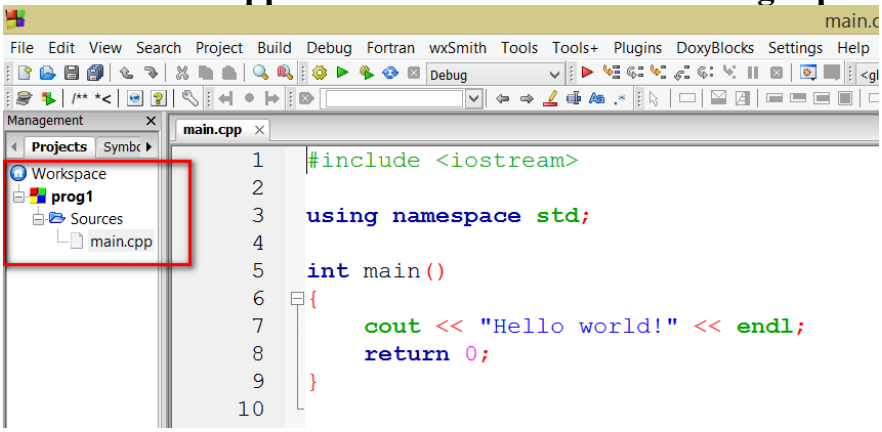
E. Enter the Project title "prog1". You have to select a folder where your project is to be stored in.



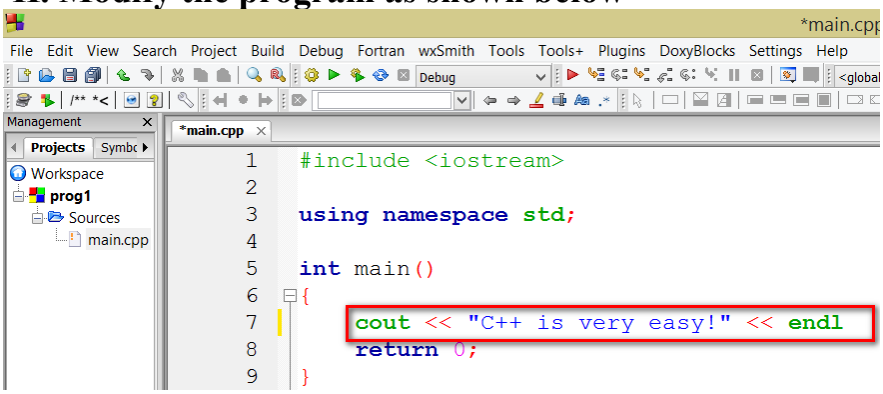
F. Click the Finish button



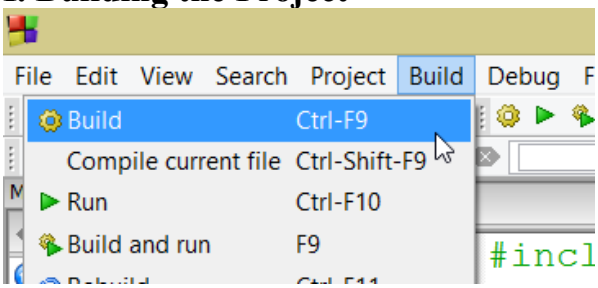
G. Double click Sources on the left panel and you should see main.cpp. Double click main.cpp reveals the C++ code on the right panel



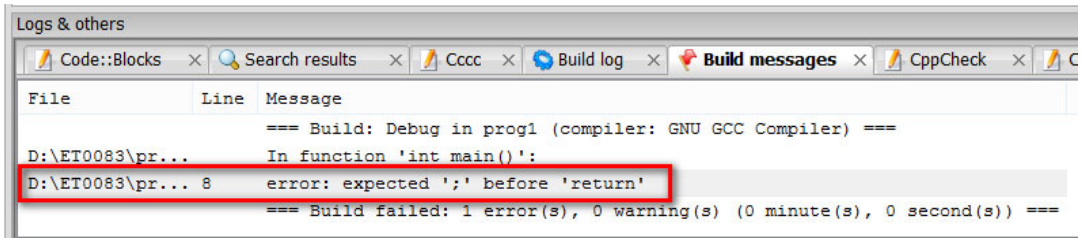
H. Modify the program as shown below



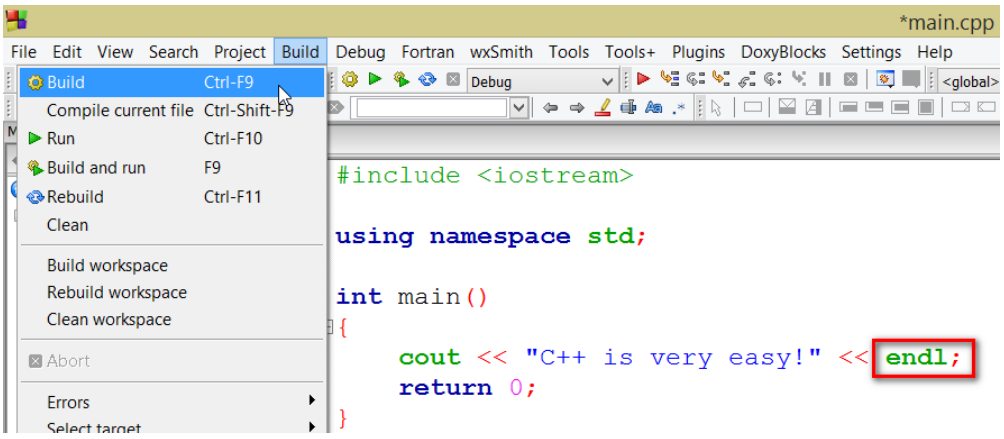
I. Building the Project



Before we can execute the program, we must build it first.

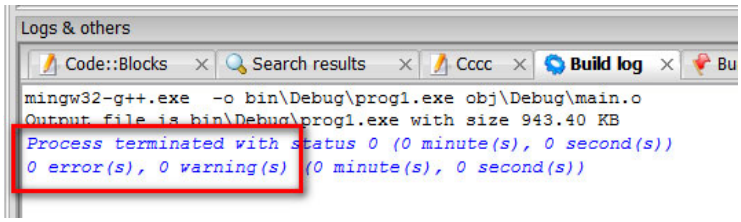


There should be 1 error (missing semicolon after the endl).

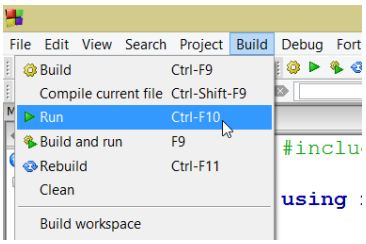


Place a semicolon after endl and build again.

You should have 0 errors.



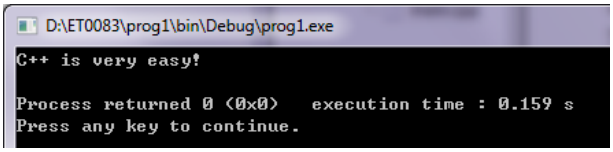
J. Run the program



Once there are no errors, we are ready to RUN or execute the program.

The result of this program is the statement,

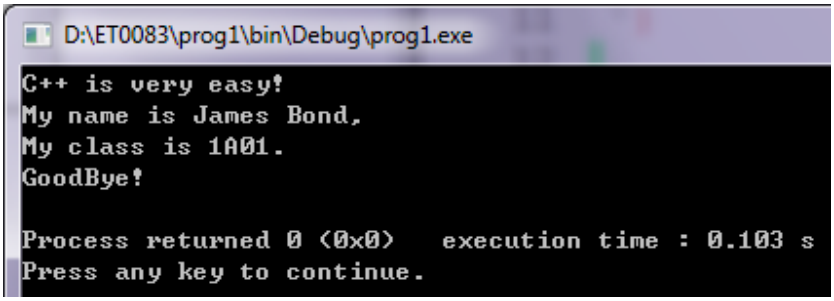
C++ is very easy!



"**Process returned 0 (0x0) execution time: 0.159s. Press any key to continue.**" is a prompt generated by the system.

Activity: Introduce yourself

Modify the program to produce the following result:



Edit / Compile Programs

Create a new project in CodeBlocks call "prog2" and type in the following codes:

```
#include <iostream>
using namespace std;

int main()
{
      int value;
      cout << "Enter a decimal value : ";
      cin >> value;
      cout << "The value entered is " << value ;
      return 0;
}
```

Build and run program to see the result.

Spelling Errors:

1. Spell the word "**decimal**" as "**desimal**". Re-build your program. Are there any error messages? If not, why?

2. Spell the word "**cout**" as "**Cout**". Re-build your program. Record the error messages.

Multiple Errors:

1. Make two errors:
   (i) Spell one of the word "**value**" as "**values**" and
   (ii) Delete any semicolon.

   Re-build your program. Record the error messages.

Ambiguous errors

Remove the " before Enter in the following statement:
cout << "Enter a decimal value : ";

1. Look at the error messages. Note that one mistake had generated so many error messages.

2. Briefly explain the importance of the " you removed.

Single Step and Watch Variables

Run time errors and warnings

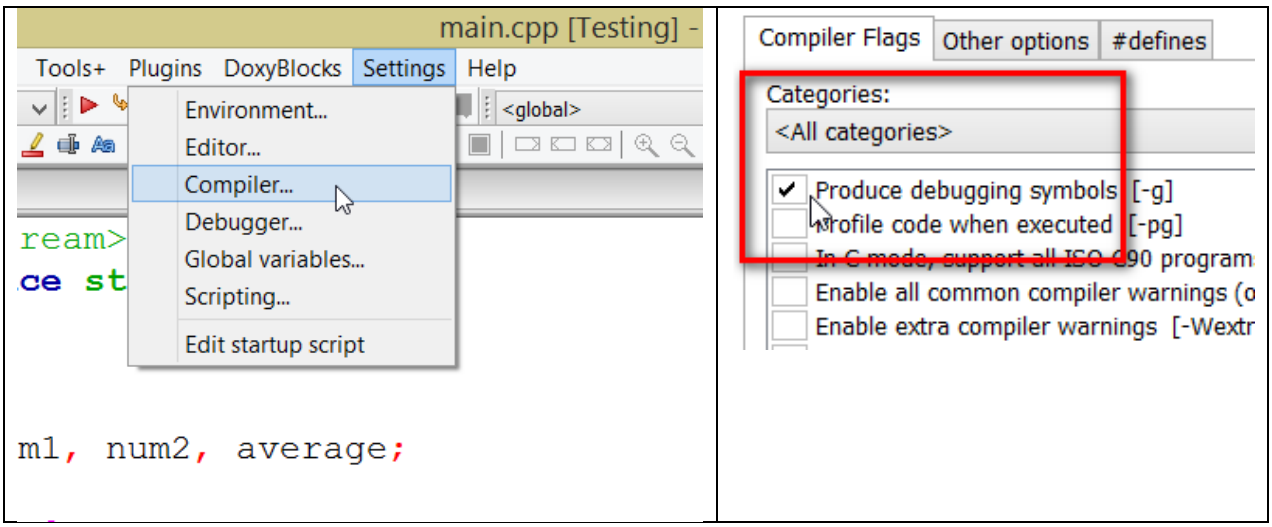
Some errors cannot be detected at all during compilation. Such errors affect the program only when the program is executed. These errors are called **run time errors** or **logical errors** and may result in error messages, wrong outputs or wrong operation of the computer.
Warnings do not normally stop the compiler thus they too may result in run time errors. The following activity explores run time errors.

Create a new project in CodeBlocks call "prog3".

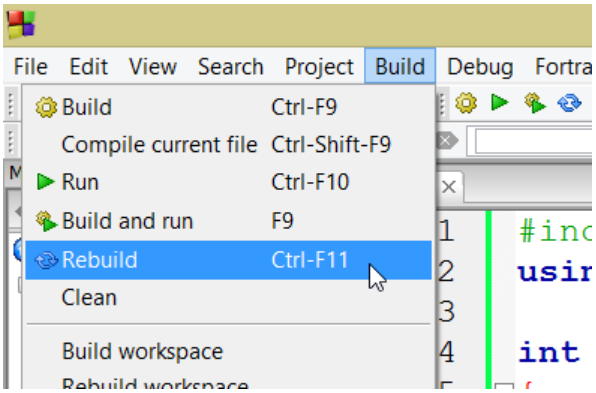
```
#include<iostream>
using namespace std;

int main()
{
      double num1, num2, average;

      num1 = 10.0;
      num2 = 20.0;
      average = num1 + num2 / 2;
      cout << "Average value is " << average << "\n" ;
      return 0;
}
```

Build and run program to see the result.
The result of 20 is not correct, it should be 15.

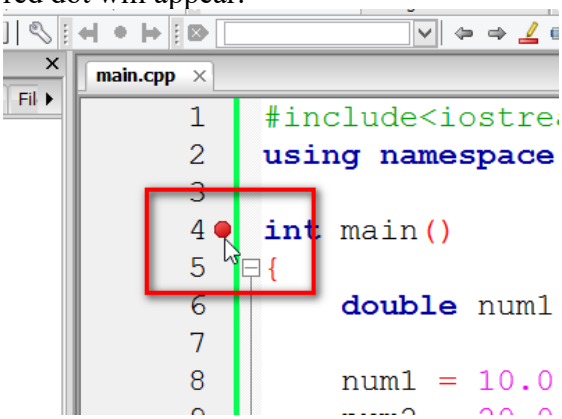
There is run time error in this program. Single stepping can be used to determine the location of the erroneous code statement(s). To debug a program by single stepping, the compiler options need to be changed.



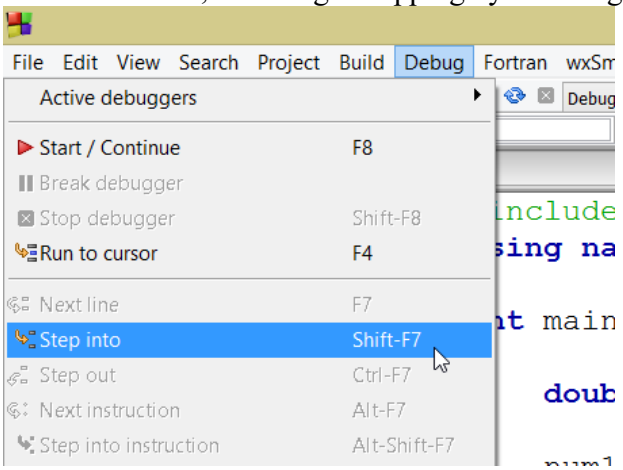
Then do a Build -> Re-Build



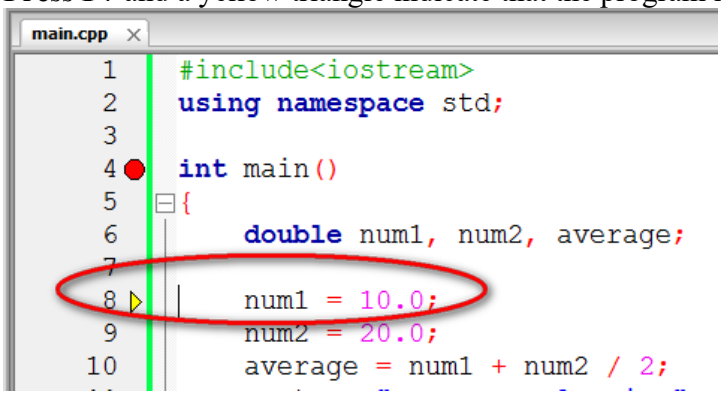
Before debugging, set a break point that the main function by clicking beside the line number. A red dot will appear.



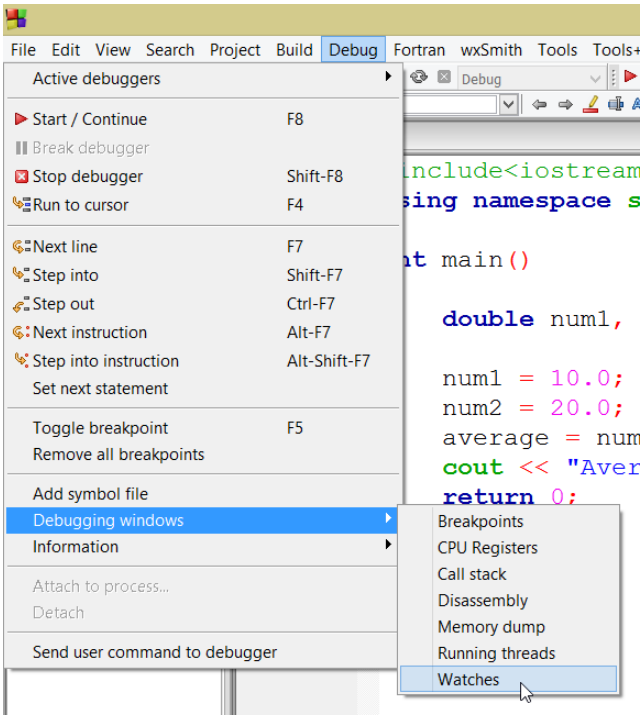
Instead of RUN, start single stepping by selecting Debug->Step into or press **Shift F7**.



Press F7 and a yellow triangle indicate that the program is at the starting position.



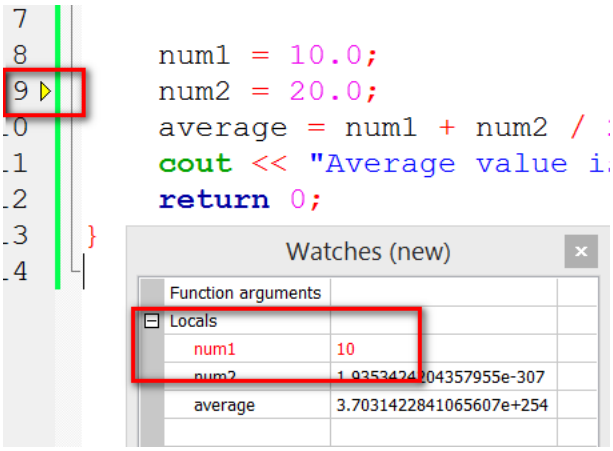
If the Watch window is not enable, turn it on by selecting Debug → Debugging windows → Watches



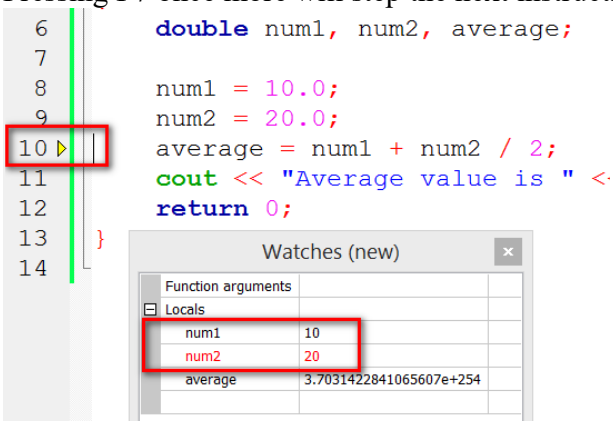
Click on Local Variables in the Watches window to display all the local Variables to see the values of the variables as the program is single stepping.

At the beginning of the program, the variables contain random values.

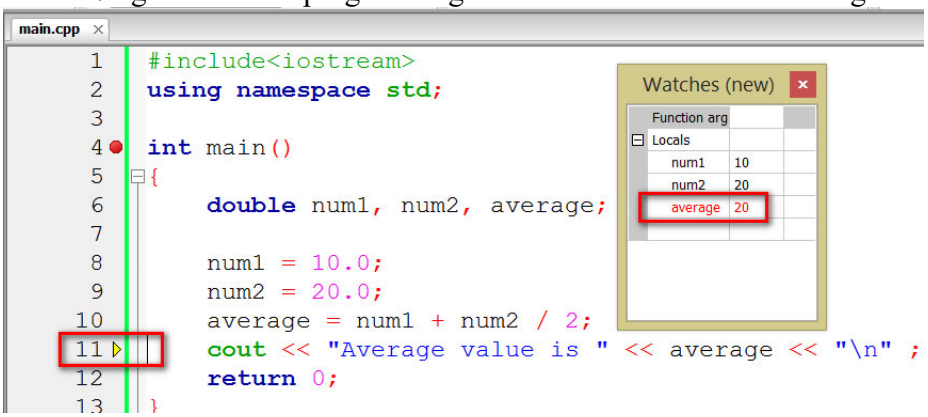
Press F7 once and the statement at the yellow triangle will be executed. The yellow triangle will move to the next statement. Pressing F7 again will execute that statement and move the yellow triangle to the next line. At the same time, the affected variable will be updated whenever a statement is executed. Press F7 until the value of num1 in the Watches changes to 10.



Pressing F7 once more will step the next instruction.



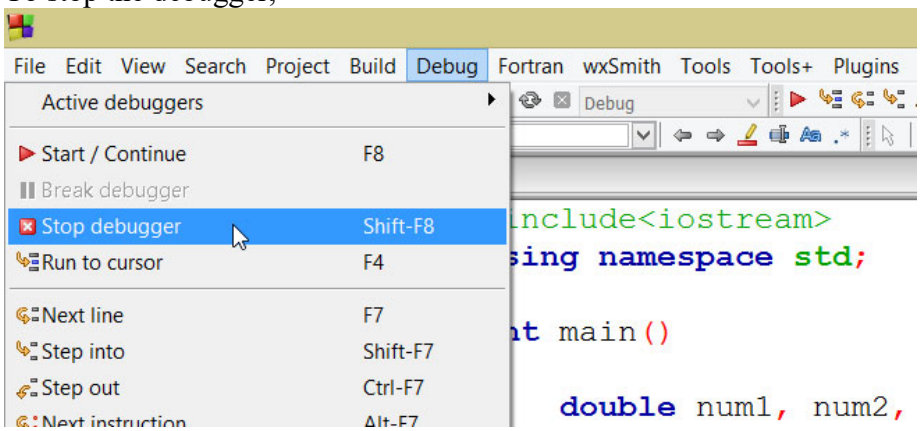
Press F7 again until the program to get the calculated result in average.



Correct error and re-do single stepping

Can you see where the problem is?

To stop the debugger,



Edit the program in order to get the correct result.

Exercise 2b: Input / Output Operations & Data Types

1. What are the characters that indicate the **begin** and **end** of instructions in a C++ Program?

Where is the error?

Create a new project and call it "prog4"

```
#include<iostream>
using namespace std;

int main()
{
    double radius, PI;
    int area;

    radius = 12.5;
    PI = 22 / 7;
    area = PI * radius * radius;
    cout << "Area of circle is " << area << "\n" ;
    return 0;
}
```

Build and run the program to see the result.
The correct value for the area is 491.071.

Where is/are the error(s)?

2. What is the difference between **variables** and **constants**?
   Name two data items, which are best represented as constants and two that are best represented as variables.

3. How much memory (in bytes) is occupied in each of the following statements?

```
double number;
int count, index;
char yesno;
```

4. Which of the following variable declarations are correct? If a variable declaration is not correct, give the reason(s) and provide the correct variable declaration.
   a. n=12;
   b. char letter =;
   c. int one = 5, two;
   d. double x, y, z;

5. Spot and correct the errors in the following code:

```
#include <iostream>
using namespace std;
#define PI=3.14;
int Main()
{
    int count;
    double colorVal

    cout << "Enter an integer number;
    cin << "count";
    colorVal = PI*count;
    cout << "\nThe color value is : "
        <<  colorVal;
    return 0;
}
```

6. In the following questions, apply the design & development process. Give both the first and second level pseudocode and flowcharts.

   a. The volume of a cylinder is given by the following equation:

      volume = $\pi r^2 l$ r is the radius
      l is the length of the cylinder.

      Write a program that will prompt the user to enter the radius and the length of the cylinder. The program will calculate and display the volume.

   b. Enhance the above program such that it also calculates and displays the total surface area of the cylinder.
      The total surface are of the cylinder is given by the following equation:

      total surface area = $2\pi r^2 + 2\pi rl$

Exercise 3: Operators

1. Data variables x, y, z are declared as follows:
   int x = 5, y = 6, z = 4;
   double w = 3.5;

   Evaluate each of the following statements, if possible. If it is not possible, state the reason.
   a. (x + z) % y
   b. (x + y) % w

2. Do a walk-through to find the value assigned to **e**. Assume that all variables are integers and they are properly declared.

```
a = 3;
b = 4;
c = (a % b) * 6;
d = c / b;
e = (a + b + c + d) / 4;
```

3. Which of the following are valid C++ assignment statements? Assume that i, x and percent are *double* variables.
   a. i = i + 5;
   b. x + 2 = x;
   c. x = 2.5 * x;
   d. percent = 10%;

4a. Determine the output of the following code segment.

```
int x, y, z=4;

y = (7+6) % 5 / 2;      cout << y << "\n" ;
z *= 3 + 2;             cout << z << "\n" ;
x=y==z;                 cout << x << "\n" ;
```

4b. Determine the output of the following code segment.

```
char x, y;
int z= 61;
x = 'a' + 10;           cout << x << "\n" ;
y = z +10;              cout << y << "\n" ;
                        cout << (z+10) << "\n" ;
```

5. Spot and correct the errors in the following code:

```
#include <iostream>
using namespace std;

int main()
{
   int weight;
   int result;

   cout << "Enter your weight in lbs: ";
   cin >> "weight";
   result == weight x 2.2;
   cout << "\nYour weight is : " <<  setprecision
        << result << endl;
   return 0;
}
```

6a. The equivalent resistance of two resistors connected in parallel is given by the equation:

R = R1*R2/(R1+R2)

Write a program that prompts the user to enter the value of the two resistors. The program then calculates and displays the equivalent resistance.

6b. Write a program that calculates the equivalent resistances of two resistors connected in series and parallel. The user will be prompted to enter the values of the two resistors. A sample run of the code is shown below:

Program to calculate equivalent resistances in series and parallel.

Enter the value of the first resistor : 5
Enter the value of the second resistor: 20

The equivalent resistance in series is 25 Ohms.
The equivalent resistance in parallel is 4 Ohms.

Exercise 4: Selection Constructs

1. Draw flow chart for the following code segments:

```
if ( I am late )
        I will take a cab.
 else
        I will take a bus.
```

2. What is wrong (if anything) with the following code?

```
x=80;
if (x = 100);
   cout << "Excellent" ;
```

3. Determine the errors in the following program.

```
int main()
{
   double a=5.0, b=10.0;
   int operation;
   cout<<"Enter the operation [+, -, * or /]:);
   cin >> operation ;
   switch (operation)
        case = '+'      :    c = a+b;
        case = '-'      :    c = a-b;
        case = '*'      :    c = a*b;
        case = '/'      :    c = a/b;
   };
   cout << a << operation << b << "=" <<  c;
   return 0;
}
```

4. Write a program that calculates the equivalent resistance of two resistors connected either in series or parallel. The user will be prompted to enter his choice of calculation and then the values of the two resistors. A sample run of the code is shown below:

Program to calculate equivalent resistance.

1. Series connection
2. Parallel connection

Enter your choice: 2

Enter the value of the first resistor: 5
Enter the value of the second resistor: 20

The equivalent resistance is 4 Ohms.

(The program must be tested with choice 1 and choice 2.)

5. Write a menu driven program that calculates the voltage, current or resistance using the Ohm's Law (V = IR).
The program first displays a menu prompting the user to enter the choice of calculation. If he chooses voltage calculation, he will then be asked to enter the value of current and resistance. If he chooses current, he will then be prompted to enter voltage and resistance and so on. A sample run of the program is given below:

Ohms Law

1. Voltage Calculation.
2. Current Calculation.
3. Resistance Calculation.

Enter your choice : 3

Resistance Calculation
==========
Enter voltage : 12
Enter current : 1.5

The resistance is 8 Ohms

(Use a **switch** statement for selection construct)

Exercise 5: Iteration Constructs

1. Fill in the blanks for the following programs to produce the respective outputs shown.

(a)

```
int m;
for(m=1; __________; m++)
{
  cout << "We've Got a problem.\n";
}
```

Console output:

```
We've Got a Problem.
We've Got a Problem.
We've Got a Problem.
```

(b)

```
int p =5;
while (p>=1)
{
  cout << p <<"*****\n";

  _____________;
}
```

Console output:

```
5*****
4*****
3*****
2*****
1*****
```

(c)

```
int limit = _____;
do
{
  cout << limit << endl;

  ___________________;
} while (limit<=10000);
```

Console output:

```
10
100
1000
10000
```

(d)

```
int layerOut,layerIn;

for(layerOut=0; layerOut<______; layerOut++)
{
  for(layerIn=0; layerIn<______; layerIn++)
  {
       cout << layerOut << "-" << layerIn << endl;
  }
  cout <<"*****\n";
}
```

Console output:

```
0-0
0-1
0-2
0-3
*****
1-0
1-1
1-2
1-3
*****
```

2a. What will be the output of the following code:

```
int i;
for(i=10; i<20; i=i+2)
{
    cout << i*10 ;
}
```

2b. Rewrite the code in question 1 using a **while** loop.

3. Spot and correct the errors in the following code segments:

(a)

```
int num;
for(num=1, num<=5, num=num+1)
{
  cout << "Square of num = << num*num << endl ;
}
```

Console output:

```
Square of 1 = 1
Square of 2 = 4
Square of 3 = 9
Square of 4 = 16
Square of 5 = 25
```

(b)

```
char input;
double voltage current;
while (input = y);
{
  cout << "Enter the voltage and current: ";
  cin >> voltage >> current;
  cout << "The resistance is " << voltage/current << endl;
  cout << Do you wish to continue [y/n]: ";
  input << cin ;
}
```

4. What is the output of the following code?

```
int count = 1, odd = 0;
do
{
    if ( (count %  2) != 0 )
         odd++;
         count++;
}while (count<10);
cout << "odd = " << odd;
```

5a. Write a program, which prompts the user to enter an integer. The program then displays the corresponding multiplication table.

A sample run is shown below:

Enter an integer : 8

8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80

5b. Modify your program in (a) above so that after displaying the multiplication table, the program repeats, asking the user to enter another number. If the number entered is non-zero, the multiplication table for the number is displayed and the program repeats. The program terminates if the number entered is zero.

6. The gain of a RC active filter is given by the following equation:

Gain = 1/(2πfRC)

Write a program, which prompts the user to enter the value of the resistor (R) and the capacitor (C). It then displays a table of *frequencies* and *Gains* for frequencies from f=0.1Hz to f=1GHz in decade steps (i.e for each iteration, the frequency is multiplied by 10).

Write your program using a
a) **for** loop
b) **while** loop
c) **do-while** loop

Exercise 6: Functions

1. What are the advantages of writing modular programs?

2. Fill in the blanks.
   (a)

```
_____________________;      //-- function prototype
int main()
{
   cout << "Laugh out loud 5 times.\n";
   laugh();
   return 0;
}

void laugh(void)
{
   for (int i=0;i<5;i++)
     cout << "LOL\n";
}
```

Sample console output:

```
Laugh out loud 5 times.
LOL
LOL
LOL
LOL
LOL
```

   (b)

```
_____________________;      //-- function prototype
int main()
{
   int x;
   cout << "Enter a number : ";
   cin >> x;
   cout << "Laugh out loud " << x << " times.\n";

   _____________________;    //-- call the function
   return 0;
}


void laugh(int _______)
{
   for (int i=0;i<num;i++)
     cout << "LOL\n";
}
```

Sample console output#1:

```
Enter a number: 3
Laugh out loud 3 times.
LOL
LOL
LOL
```

Sample console output#2:

```
Enter a number: 6
Laugh out loud 6 times.
LOL
LOL
LOL
LOL
LOL
LOL
```

(c) The function **validateMark** checks if the number entered by the user is within 0 – 100.

```
_____________________________;      //-- function prototype
int main()
{
   int x;
   cout << "Enter a number : ";
   cin >> x;
   if (validateMark(____)== _____ )
      cout << "Invalid marks.\n";
   else
      cout << "Thank U.\n";
   return 0;
}


_____ validateMark (_____________)
{
   if (num>=0 && num<=100)
      return 1;
   else return 0;
}
```

Sample console output#1:

```
Enter a number: 3
Thank U.
```

Sample console output#2:

```
Enter a number: -5
Invalid marks.
```

Sample console output#3:

```
Enter a number: 88
Thank U.
```

Sample console output#4:

```
Enter a number: 101
Invalid marks.
```

3. The following program has some syntax and a logical error, because of this it gives an incorrect output. Spot and correct the errors.

```
int displaySum(void);      //-- function prototype
int sum;                   //-- global variable
int main()
{
   int num1, num2;
   cout << "Enter a number : ";
   cin >> num1;
   cout >> "Enter another number : ";
   cin >> num2;
   sum = num1 + num1;
   displaySum()
   return 0;
}

void displaySum(void);
{
   int sum;
   cout << "The sum is : " << sum <"\n";
   return;
}
```

4. Write a program that will analyze, for your class, the grades obtained by all the students for Structured Programming. The program will prompt the user to enter the grade for each student. Valid grades are A, B, and C. The program calculates and displays the total number of As, Bs and Cs. The user should be able to enter the grades in uppercase or lowercase. You may assume there are only 10 students in your class.

   Your program must be modular. Write a function to read and total the grades and another to print the results. A skeleton of the program is given below:

```
char grade; //-- global variables
int totalA, totalB, totalC;

int main()
{
    readandTotalGrades();
    displayTotals();
    return 0;
}
```

Sample console output:

```
Please enter grade for student 1:A
Please enter grade for student 2:B
Please enter grade for student 3:C
Please enter grade for student 4:A
Please enter grade for student 5:B
Please enter grade for student 6:C
Please enter grade for student 7:C
Please enter grade for student 8:C
Please enter grade for student 9:C
Please enter grade for student 10:C
Total no. of grade A students : 2
Total no. of grade B students : 2
Total no. of grade C students : 6
```

5a. Write a program, which prompts the user to enter three integer numbers. It then finds and displays the smallest of the three numbers. The program outline is:

```
int main()
{
  int num1, num2, num3, smallest;

  cout << "Enter the first number : ";
  cin >> num1;
  cout << "Enter the second number : ";
  cin >> num2;
  cout << "Enter the third number : ";
  cin >> num3;
  smallest = findSmallest(num1, num2, num3);
  cout << "The smallest number is : " << smallest;
  return 0;
}
```

Sample console output:

```
Enter the first number : 200
Enter the second number : -2
Enter the third number : 8
The smallest number is : -2
```

5b. Create two more functions, similar to the function *findSmallest()*, one to find the largest and another to find the average of the three numbers. The function prototypes are as follows:

```
int findLargest(int, int, int);
double findAverage(int, int, int);
```

Sample console output:

```
Enter the first number : 1
Enter the second number : -20
Enter the third number : 100
The smallest number is : -20
The largest number is : 100
The average is : 27
```

Exercise 7: Arrays and Strings

1. An array is used to hold several data items together as one entity. Can data of different types be stored in a single array?

2. Explain the difference between the character '5' and the string "5" in C++ language.

3. Study the code shown below:

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
   char myname[] = "Barney";
   char yourname[20];
   int i, length;

   length = strlen(myname);
   for(i=0; i<length+1; i++)
        yourname[i] = myname[i];

   cout << yourname;
   return 0;
}
```

   What does the program do? What is the output?

4. Given the following declarations:

```
int num[4];
int count, max, zero_count;
```

   a. Using a for-loop, initialise each element of the array with the value entered by the user. The user has to enter 4 numbers one by one.

   b. Using a while-loop, display the value of each element of the array ***num***.

   c. Write the code to find the largest value in the array and store it in the variable ***max***. Display ***max*** on the screen.

d. Write the code to count the number of zeros in the array and store it in the variable ***zero_count***. Display ***zero_count*** on the screen.

Write the entire program as one main() function.

5. Which of the following statements about arrays are true/false?

a. The array index starts with 0.
b. Once an array has been initialised, its elements can never be modified.
c. You cannot define a character array with only one element.
d. When you define an array without any size, you must initialise it in the same statement.
e. You must initialize all elements in a double-scripted array.

6. Work on the Group Presentation Activity (refer to Chapter 7- page 146.)

Exercise 8: Bitwise Operations

1. The bits in the result of an expression using the ____ operator are set to 1 if at least one of the corresponding bits in either operand is set to 1. Otherwise, the bits are set to zero.

2. The bits in the result of an expression using the ____ operator are set to 1 if the corresponding bits in each operand are set to 1. Otherwise, the bits are set to zero.

3. The bits in the result of an expression using the ____ operator are set to 1 if exactly one of the corresponding bits in either operand is set to 1. Otherwise, the bits are set to zero.

4. The bitwise AND operator & is often used to _____ bits, that is to select certain bits from a bit string while zeroing others.

5. The ____ and ____ operators are used to shift the bits of a value to the left or to the right, respectively.

6. Determine and verify the results of the following hexadecimal values:

   a. 3 & 5 =
   b. E & E =
   c. F & 7 =
   d. 3 | 5 =
   e. E | E =

7. Determine and verify the results of the following operations:

   a. 0x1234 << 3 =
   b. 0x1234 >> 2 =

8. Write a program that will perform Shift Left or Shift Right operation depending on the choice entered by the user. If the choice is 1, shift left is performed or 2, which will shift right. The user will then enter the value and the number of time to be shifted. The result will be displayed on the screen.

   A sample run is as follows:

   Choice:
   1. **Shift Left**
   2. **Shift Right**

   Enter choice: 1
   Enter value: 8
   Enter times: 2

   The value 8 shifted 2 times is 32

Common C++ Operators: Precedence and Associativity

OPERATORS	TYPE	ASSOCIATIVITY
() [] .	Groups	left to right
! ~ ++ --	Unary	**right to left**
* / %	Multiplicative	left to right
+ -	Additive	left to right
<< >>	Shift	left to right
< > <= >=	Relational	left to right
== !=	Equality	left to right
&	Bitwise	left to right
^		
|		
&&	Logical	left to right
||		
= += -= *= /=	Assignment	**right to left**

Standard ASCII Table

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

C++ Keywords

Keyword	Description
asm	insert an assembly instruction
auto	declare a local variable
bool	declare a boolean variable
break	break out of a loop
case	a block of code in a switch statement
catch	handles exceptions from throw
char	declare a character variable
class	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
continue	bypass iterations of a loop
default	default handler in a case statement
delete	make memory available
do	looping construct
double	declare a double precision floating-point variable
dynamic_cast	perform runtime casts
else	alternate case for an if statement
enum	create enumeration types
explicit	only use constructors when they exactly match
export	allows template definitions to be separated from their declarations
extern	tell the compiler about variables defined elsewhere
false	the boolean value of false
float	declare a floating-point variable
for	looping construct
friend	grant non-member function access to private data
goto	jump to a different part of the program
if	execute code based on the result of a test
inline	optimize calls to short functions
int	declare an integer variable
long	declare a long integer variable
mutable	override a const variable
namespace	partition the global namespace by defining a scope

Keyword	Description
new	allocate dynamic memory for a new variable
operator	create overloaded operator functions
private	declare private members of a class
protected	declare protected members of a class
public	declare public members of a class
register	request that a variable be optimized for speed
reinterpret_cast	change the type of a variable
return	return from a function
short	declare a short integer variable
signed	modify variable type declarations
sizeof	return the size of a variable or type
static	create permanent storage for a variable
static_cast	perform a nonpolymorphic cast
struct	define a new structure
switch	execute code based on different possible values for a variable
template	create generic functions
this	a pointer to the current object
throw	throws an exception
true	the boolean value of true
try	execute code that can throw an exception
typedef	create a new type name from an existing type
typeid	describes an object
typename	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location
unsigned	declare an unsigned integer variable
using	import complete or partial namespaces into the current scope
virtual	create a function that can be overridden by a derived class
void	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified unexpectedly
wchar_t	declare a wide-character variable
while	looping construct

The School of Electrical & Electronic Engineering at Singapore Polytechnic offers the following full-time courses.

1. Diploma in Aerospace Electronics (DASE)
The Diploma in Aerospace Electronics course aims to provide students with a broad-based engineering curriculum to effectively support a wide spectrum of aircraft maintenance repair and overhaul work in the aerospace industry and also to prepare them for further studies with advanced standing in local and overseas universities.

2. Diploma in Computer Engineering (DCPE)
This diploma aims to train technologists who can design, develop, setup and maintain computer systems; and develop software solutions. Students can choose to specialise in two areas of Computer Engineering & Infocomm Technology, which include Computer Applications, Smart City Technologies (IoT, Data Analytics), Cyber Security, and Cloud Computing.

3. Diploma in Electrical & Electronic Engineering (DEEE)
This diploma offers a full range of modules in the electrical and electronic engineering spectrum. Students from 2019/20 Year 1 intake can choose one of the six available specialisations (Biomedical, Communication, Microelectronics, Power, Rapid Transit Technology and Robotics & Control) for their final year. Students from earlier intakes and direct-entry 2nd year students can choose one of the seven available double-specialisation tracks (Aerospace + Communication, Biomedical + Robotics & Control, Computer + Communication, Microelectronics + Nanoelectronics, Microelectronics + Robotics & Control, Power + Control and Rapid Transit Technology + Communication) for their final year.

4. Diploma in Energy Systems & Management (DESM)*
The Diploma in Energy Systems & Management course aims to equip students with the knowledge and expertise in three specialisations: clean energy, power engineering and energy management, so as to design clean and energy efficient systems that will contribute to an economically and environmentally sustainable future.

5. Diploma in Engineering Systems (DES)*
The Diploma in Engineering Systems course aims to provide students with a broad-based engineering education to support activities and future challenges requiring interdisciplinary engineering systems capabilities. The course leverages on the experience and expertise of two schools, namely the School of Electrical & Electronic Engineering and the School of Mechanical & Aeronautical Engineering.

6. Diploma in Engineering with Business (DEB)
Diploma in Engineering with Business provides students with the requisite knowledge and skills in engineering principles, technologies, and business fundamentals, supported by a strong grounding in mathematics and communication skills, which is greatly valued in the rapidly changing industrial and commercial environment.

7. Common Engineering Program (DCEP)
In Common Engineering Program, students will get a flavour of electrical, electronics and mechanical engineering in the first semester of their study. They will then choose one of the 8 engineering courses specially selected from the Schools of Electrical & Electronic Engineering and Mechanical & Aeronautical Engineering.

*Course is applicable only for AY2018 intake and earlier

School of Electrical & Electronic Engineering

More than 60 Years of solid foundation	8 Tech Hubs	Unique PTN Scheme	More than 35,000+ Alumni

SINGAPORE POLYTECHNIC SP

Electives offered by SP | SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

All SP students, including EEE students are free to choose electives offered by ANY SP schools, subject to meeting the eligibility criteria.

Like all schools, School of Electrical and Electronic Engineering offers electives for:

- EEE students only
- *and* for all SP students

EEE students are required to complete 3 electives, starting from Year 2 to Year 3 (one elective per semester).

Electives Choices for All SP students

Mod Code	Module Title
EP0400	Unmanned Aircraft Flying and Drone Technologies
EP0401	Python Programming for IoT*
EP0402	Fundamentals of IoT*
EP0403	Creating an IoT Project*
EP0404	AWS Cloud Foundations
EP0405	AWS Cloud Computing Architecture

Certificate in IoT (Internet of Things)

* A certificate in IoT would be awarded if a student completes the 3 modules: EP0401, EP0402 and EP0403

Electives Choices for EEE students

Mod Code	Module Title
EM0400	Commercial Pilot Theory
EM0401	Autonomous Electric Vehicle Design
EM0402	Artificial Intelligence for Driverless Cars
EM0403	Autonomous Mobile Robots
EM0404	Smart Sensors and Actuators
EM0405	Digital Manufacturing Technology
EM0406	Linux Essential
EM0407	Advanced Linux
EM0408	Linux System Administration
EM0409	Rapid Transit System
EM0410	Rapid Transit Signalling System
EM0411	Smart City Systems Design
EM0412	Data Analytics
EM0413	Mobile App Development
EM0414	Client-Server App Development
EM0415	Machine Learning & Artificial Intelligence
EM0416	Solar Photovoltaic System Design
EM0417	Energy Management and Auditing
EM0418	Integrated Building Energy Management System
EM0419	Digital Solutioning Skills