



# Internet of Things: Theory and Applications

## **Module 1: C++ Programming – Part A**

*Introduced by: Eng. Mohamed Hatem*

Teaching Assistant at the Faculty of Computers and Data Science in Alexandria University

Graduate of Nanotechnology and Nano-electronics Engineering Program at the Zewail City of Sciences, Technology, and Innovation

Graduate Student Member at IEEE

# Table of Content

- Section 0: Training Brief
- Section 1: IoT in a Nutshell
- Section 2: Basics of Computer Science & Engineering
- Section 3: Basics of C++ Programming
- Section 4: Assignment



# Section 0

## Training Brief

**Internet of Things: Theory and Applications**

# Training Pivots

- The training is based on the following basis:

Hands-on  
Experience

Theoretical  
Basics

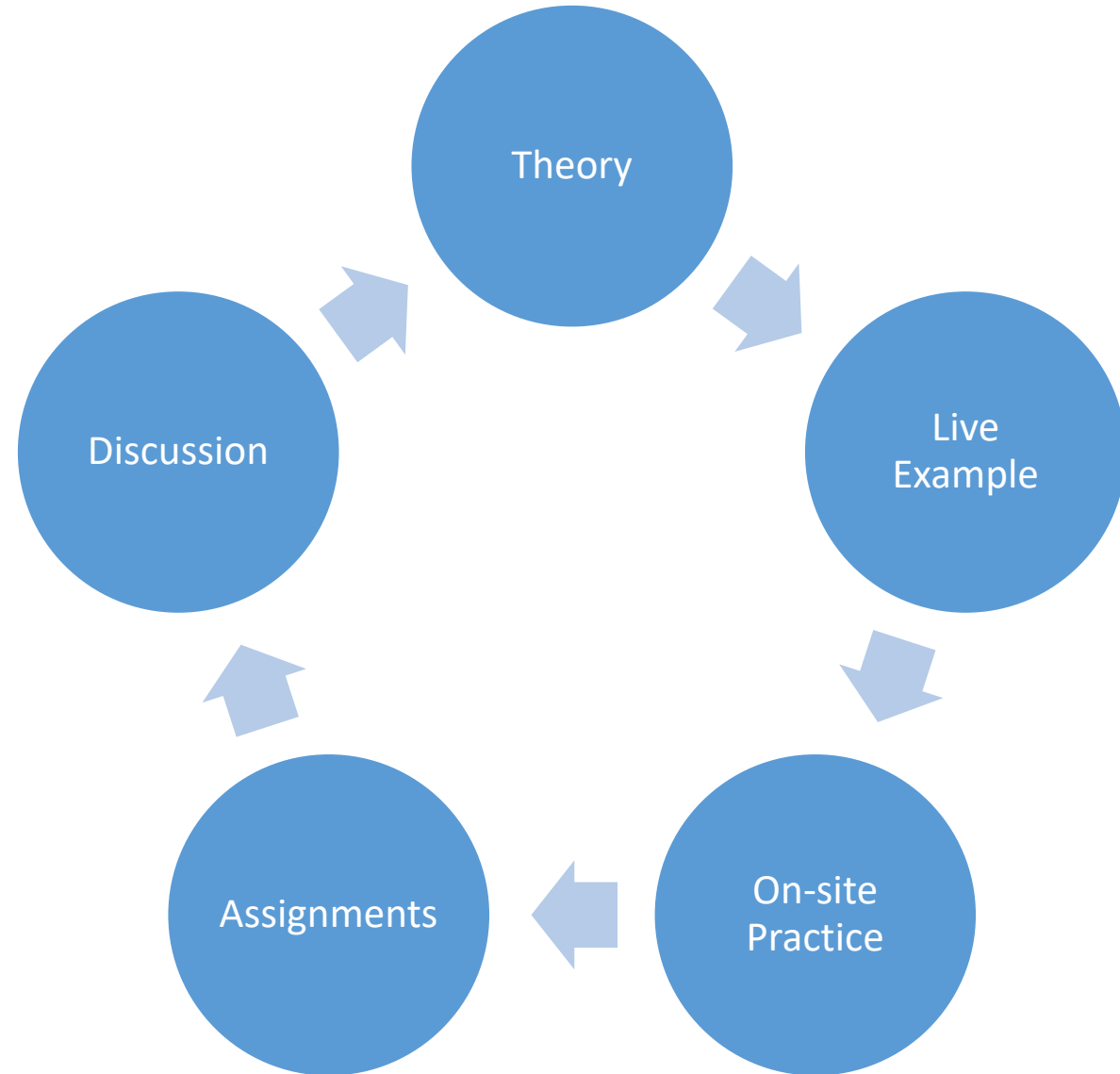
Documentation

Team-work

Learning by  
Discussion

# Training Life Cycle

- Every concept during the training is to pass by the following cycle:



# Training Plan

C++ Programming

ESP32 Based IoT Systems

Cloud Databases using Firebase

GUI using Flutter/Web Apps

Basics of Networks & IoT Security

# Training Platform

- The training official platform is Google Classroom at which you can:
  - Upload your assignments.
  - Check your grades.
  - Download training materials.
  - Communicate with your trainers.
  - Follow-up any updates.

# Training Evaluation

Item	Grade (100)
On-site Participation	12
Assignments & Discussion	18
Final Project	30
Final Report	40



# Training Duration is 60 Hours!

Item	Duration (Hours)
C++ Programming	6 On-site + 8 Team-work = 14
ESP32 based IoT Systems	6 Onsite + 8 Team-work = 14
Firebase	6 Onsite + 8 Team-work = 14
Flutter	6 Onsite + 8 Team-work = 14
Security	4 Onsite = 4

# Training To-Do

- Be on Time.
- Join your correct Google classroom.
- Late assignments policy is - 10% on each late day.
- Cheating and Plagiarism has ZERO grades!
- Working in teams from 3 to 5 members is preferred.
- Attend with your team in the same group.
- Bring in your laptop every session to save your work.
- Enjoy the journey!

# Training Team

- The training will be delivered through various trainers in either groups who are:

Saturday Group	Wednesday Group
Mohamed Hatem	Mohamed Hatem
Rowan Hassan	Mohamed Wael
Alaa Ahmed	Mahmoud Wael
Mariam Elsayed	Menna Ashraf
	Mostafa Ali



# Section 1

## IoT in a Nutshell

**Internet of Things: Theory and Applications**

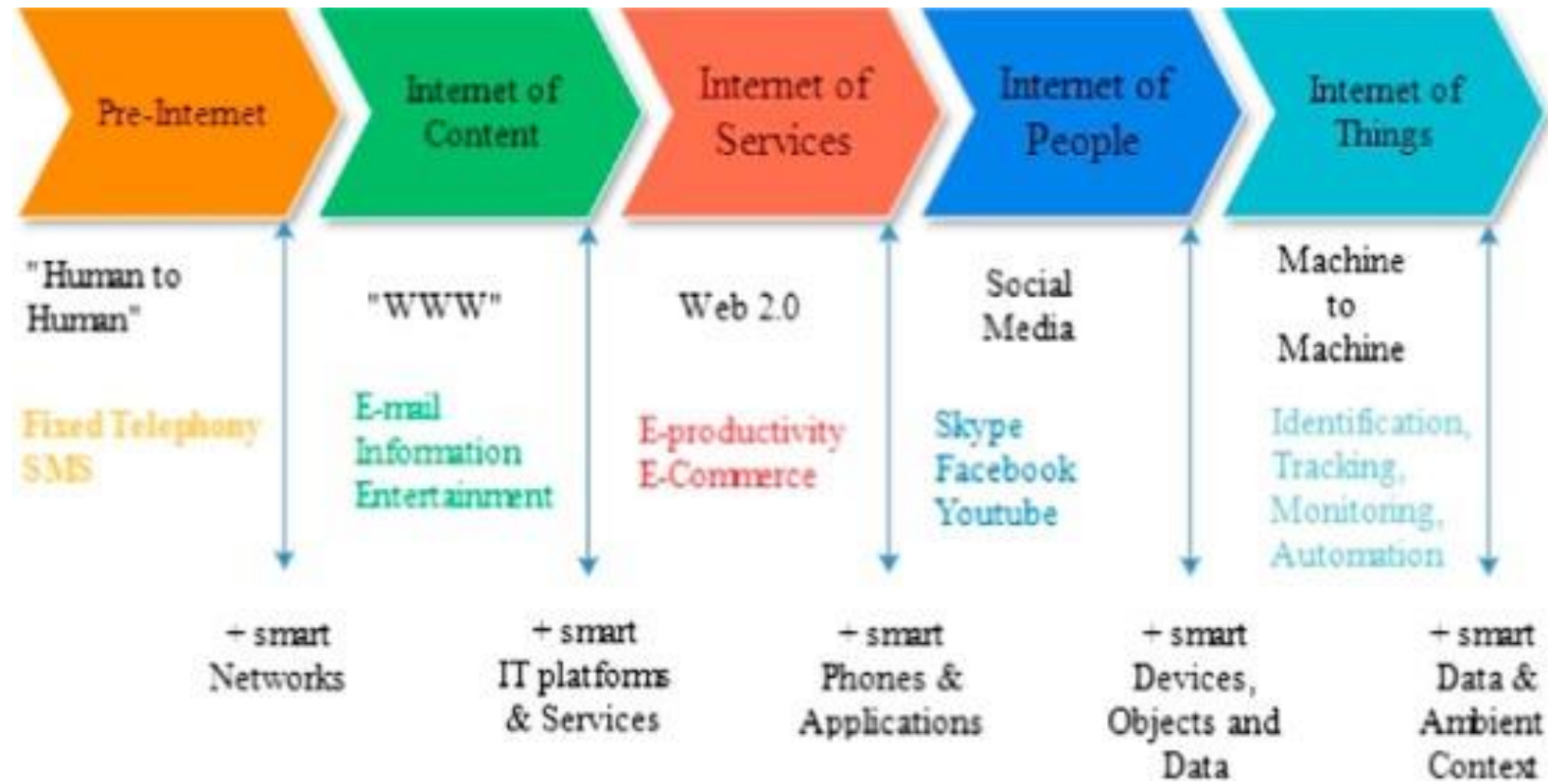
“In the twentieth century, computers were brains without senses—they only knew what we told them.”

“IoT is changing this paradigm; in the twenty-first century, computers are sensing things for themselves!

# Introduction

- Internet of Things (IoT) is a collection of smart devices that can sense (sensors) and control (actuators) the physical components around them remotely across the internet network, with main goal to **“connect the unconnected”**.
- These sensors and actuators are systemized to enable them to collect and exchange data without human intervention.
- IoT serves wide range of applications and industries such as *smart cities, smart factories, and connected vehicles*.
- IoT improves efficiency, accuracy, and automation.

# The evolution of IoT with reference to the technological progress in Internet conception



# Official IoT Definition

- IoT could be deemed as a system wherein things are connected in such a manner that they can intelligently interact with each other as well as to humans.



# IoT Characteristics

1. Connectivity
2. Scalability
3. Device heterogeneous
4. Sensors data collection, storage, processing
5. Security

# IoT Architectures

- The IoT architecture can be:
  - Three-layer IoT architecture
  - Five-layer IoT architecture
  - Six-layer IoT architecture
  - Seven-layer architecture

**In this training, we are working on the “Five-layer IoT architecture”**

# Mostly Known Five-layer IoT architecture



Business Layer

Application Layer

Processing Layer

Transport Layer

Perception Layer

# Five-layer IoT architecture

- The perception layer is the physical layer, which has sensors for sensing and gathering information about the environment. It senses some physical parameters or identifies other smart objects in the environment.
- The network layer is responsible for connecting to other smart things, network devices, and servers. Its features are also used for transmitting and processing sensor data.
- The service management layer (processing) layer is also known as the middleware layer. It stores, analyzes, and processes huge amounts of data that comes from the network layer. It employs many technologies such as databases, cloud computing, and big data processing.

# Five-layer IoT architecture

- The application layer is responsible for delivering application specific services to the user. It defines various applications in which the Internet of Things can be deployed, for example, smart homes, smart cities, and smart health.
- The business layer manages the whole IoT system, including applications, business and profit models, and users' privacy.

# IoT Building Blocks

- Sensors and Actuators: Physical devices that collect data from the environment and enable actions in the physical world.
- Connectivity: Various options like Wi-Fi, Bluetooth, or cellular networks that allow IoT devices to communicate with each other and the cloud.
- Microcontrollers and Embedded Systems: Provide computational power and interfaces to connect sensors and actuators, execute code, and transmit/receive data.

# IoT Building Blocks

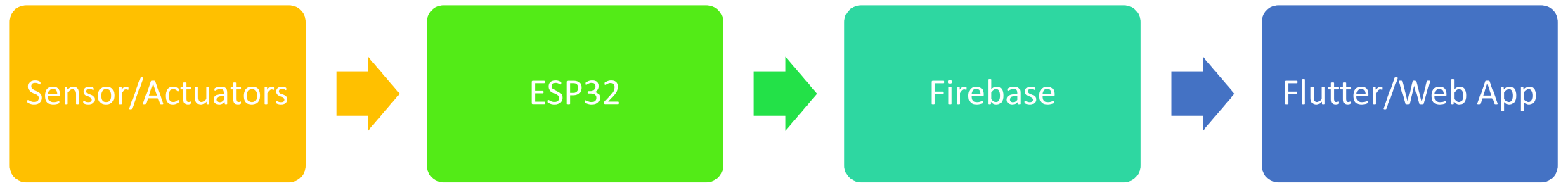
- Cloud Infrastructure: Scalable and centralized platform for data storage, processing, and management, including servers, databases, and analytics services.
- Security and Application Development: Security measures to protect data and devices, while application development frameworks enable building user interfaces and control systems for IoT solutions.

# Our Five-layers IoT Architecture – Alternative Architecture in this Training!

- **Physical Layer:** This layer involves the physical devices and sensors, such as the sensors connected to the ESP32 microcontroller. These sensors collect data from the physical environment.
- **Device Layer:** The **ESP32** acts as the device layer in this architecture. It is responsible for **reading the sensor data and processing** it locally. The ESP32 can perform data preprocessing, filtering, and other tasks before sending the data to the cloud.
- **Communication Layer:** In this layer, the ESP32 communicates with the Firebase backend. It sends the sensor data over the network to Firebase using a protocol such as MQTT or HTTP.
- **Cloud Layer:** Firebase serves as the cloud layer in this architecture. It receives the data sent by the ESP32 and stores it in its backend infrastructure. Firebase provides services like real-time database, authentication, and hosting.
- **Application Layer:** The Flutter application running on the mobile device represents the application layer. It connects to Firebase to retrieve the sensor data stored in the cloud and presents it to the user through the mobile application's user interface.



# Our Five-layers IoT Architecture – Alternative Architecture in this Training!



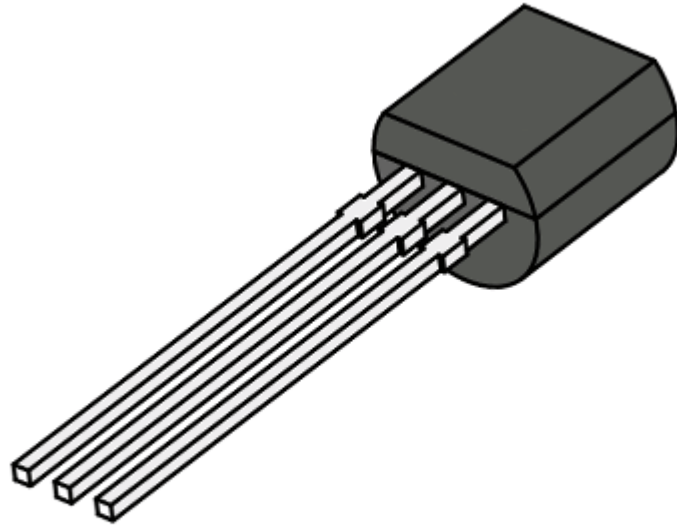


# Section 2

## Basics of Computer Science & Engineering

**Internet of Things: Theory and Applications**

# Computers are all about switches – Transistors!



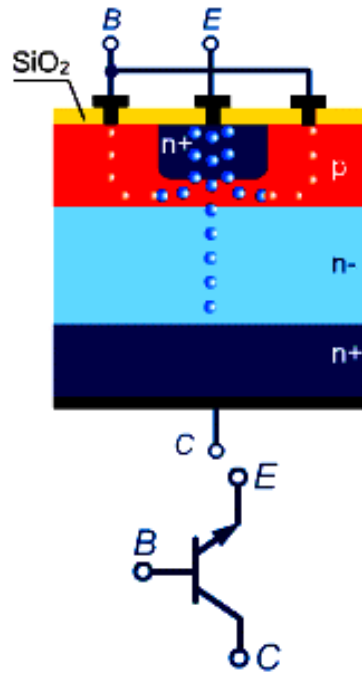
**TRANSISTOR**

# What is a Transistor?

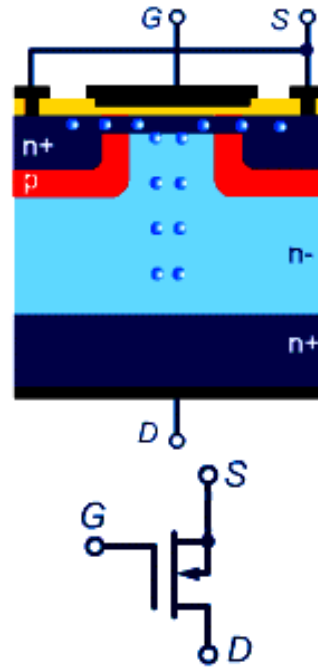
- Transistors are **electronic devices** that play a fundamental role in modern electronics.
- They are semiconductor devices that can **amplify or switch electronic signals**.
- Common Types of Transistors:
  - 1) Bipolar Junction Transistor (**BJT**): BJTs are the most common type of transistor. They come in two main varieties: **NPN (negative-positive-negative)** and **PNP (positive-negative-positive)**. BJTs use the flow of both electron and hole currents to amplify or switch signals. They have three terminals: the **emitter, base, and collector**.
  - 2) Field-Effect Transistor (**FET**): FETs are another major type of transistor. They **operate based on the voltage applied to their terminals**, controlling the flow of current through the device. There are two primary categories of FETs: **Junction Field-Effect Transistors (JFETs)** and **Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs)**.

# Types of Transistors

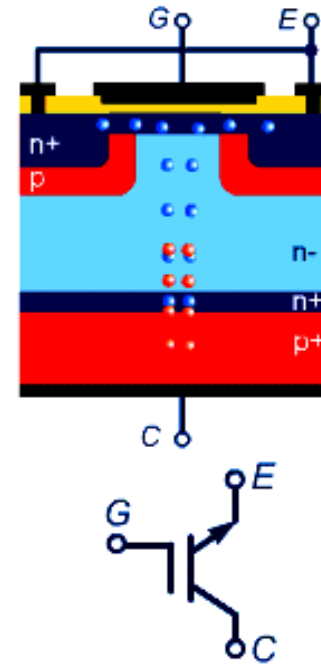
nnp - Power Bipolar



n - Channel Power MOSFET

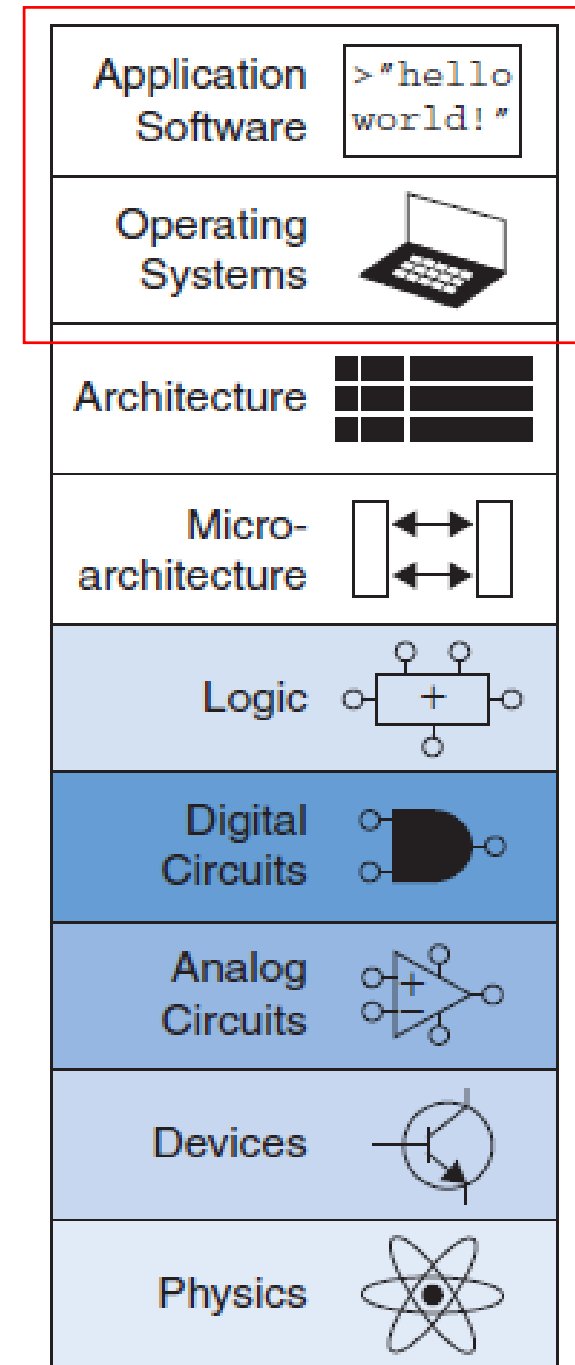


n - Channel Power IGBT



# Abstraction in Computer Science & Engineering

- Abstraction, in the context of computer engineering and computer science, refers to the process of simplifying complex systems by hiding unnecessary details and focusing on essential features. It involves creating higher-level concepts or models that capture the important aspects of a system while hiding the underlying implementation details.



# Introduction to Computers

- A **computer** is a **programmable electronic device** that is capable of executing a variety of tasks by following a set of instructions or **programs**.
- A **program** is a **set of instructions** for a computer to follow.
- The **collection of programs** used by a computer is referred to as the **software** for that computer.
- The actual **physical machines** that make up a computer installation are referred to as **hardware**.
- The computer can be thought of as having five main components (hardware):
  - **Input device(s)** i.e keyboard, mouse, mics, touchpads, sensors!
  - **Output device(s)** i.e monitors, speakers,
  - **Processor** (also called the CPU, for central processing unit) “the brain”
  - **Main memory** i.e RAM (random access)
  - **Secondary memory** which could be referred to as auxiliary memory, auxiliary storage, external memory, and external storage i.e hard disks, diskettes, CDs, DVDs, Blue ray (sequential access)

# Main Memory

- Main memory is often referred to as **RAM** or **random access memory**. It is called random access because the computer can immediately access the data in any memory location.





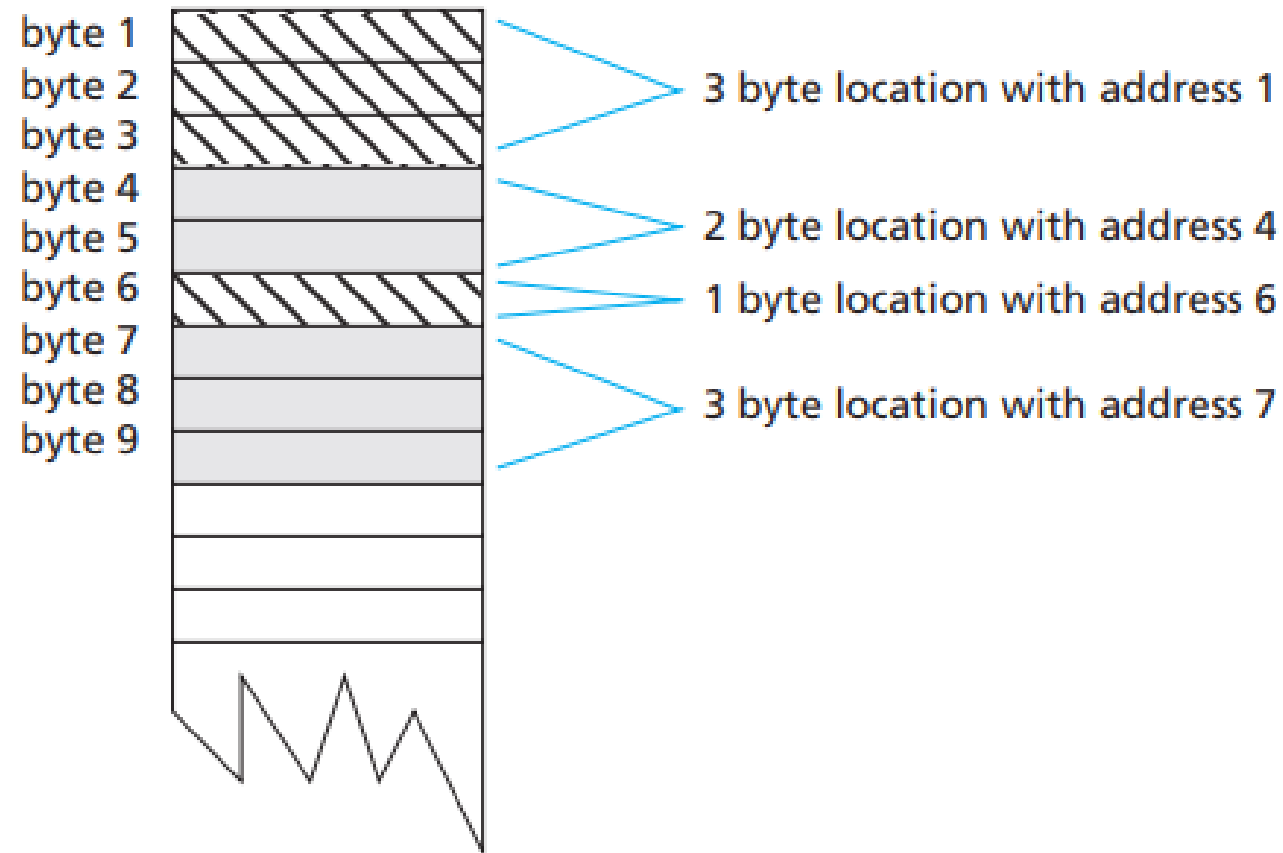
# Main Memory – Memory Location

- **Main memory** consists of a **long list of numbered locations called memory locations**.
- Each **memory location** typically holds a **fixed amount of data**.
- In programming, memory locations are often **represented by variables or data structures**.
- Each **memory location** contains a **string of 0s and 1s** which are named bits that represent this variable or data structure.
- In most computers, **all memory locations contain the same number of bits** that could store same amount of data.
- The memory locations in most computers **contain eight bits (or some multiple of eight bits)**. An eight bit portion of memory is called a **byte**, so we can refer to these numbered memory locations as bytes.

# Main Memory – Memory Address

- Then, the main memory as a long list of numbered memory locations called **bytes**. The number that identifies a byte is called its address.
- A memory address, is a **specific numeric identifier that uniquely identifies a memory location**.
- A data item, such as a number or a letter, can be stored in one or some of these bytes.
- In this case, the entire **chunk** of memory that holds the data item is still called a **memory location**.

# Memory Locations and Bytes



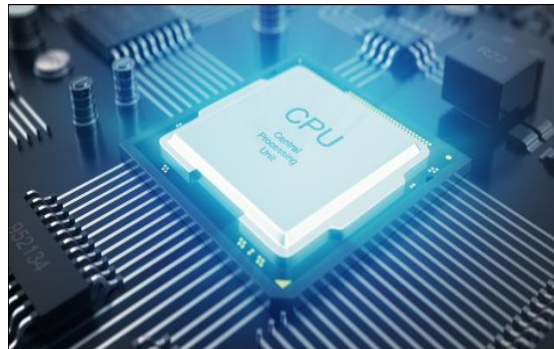
# Secondary Memory



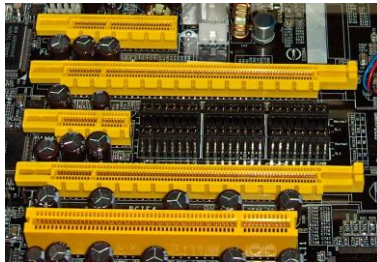
- **Secondary memory** is the memory that is used for **keeping a permanent record of information** after and before the computer is used.
- Several different kinds of secondary memory can be attached to a single computer. The most common forms of secondary memory are **hard disks, diskettes, CDs, DVDs, and removable flash memory drives**.
- Secondary memory often requires **sequential access**, which means that the computer must look through all (or at least very many) memory locations until it finds the item it needs.

# The Processor

- The processor can **add, subtract, multiply, and divide** and can **move things from one memory location to another**.
- **Example:** it can interpret strings of 0s and 1s as letters and send the letters to an output device. The processor also has some primitive ability to rearrange the order of instructions.



# Other Computer Components



Busses



Input and output  
devices

# Machine & Assembly Language inside a Processor

- The kind of language a computer can understand is called a **low level language**.
- Example: **ADD X Y Z**

This instruction might mean “Add the number in the memory location called X to the number in the memory location called Y, and place the result in the memory location called Z.”

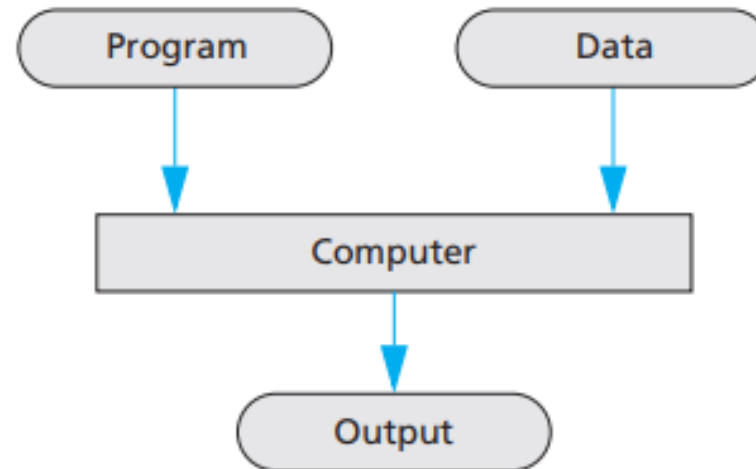
# Machine & Assembly Language

- The version of the instruction aforementioned that the computer ultimately follows would then be: **0110 1001 1010 1011**.
- For example, the word ADD might translate to 0110, the X might translate to 1001, the Y to 1010, and the Z to 1011.
- Programs written in the form of 0s and 1s are said to be written in **machine language** which consists of **binary digits (bit): 0 or 1** and **binary codes: a sequence of 0s and 1s**.
- Any high-level language program must be translated into machine language before the computer can understand and follow the program.



# Simple Scheme of a Program a Processor Run

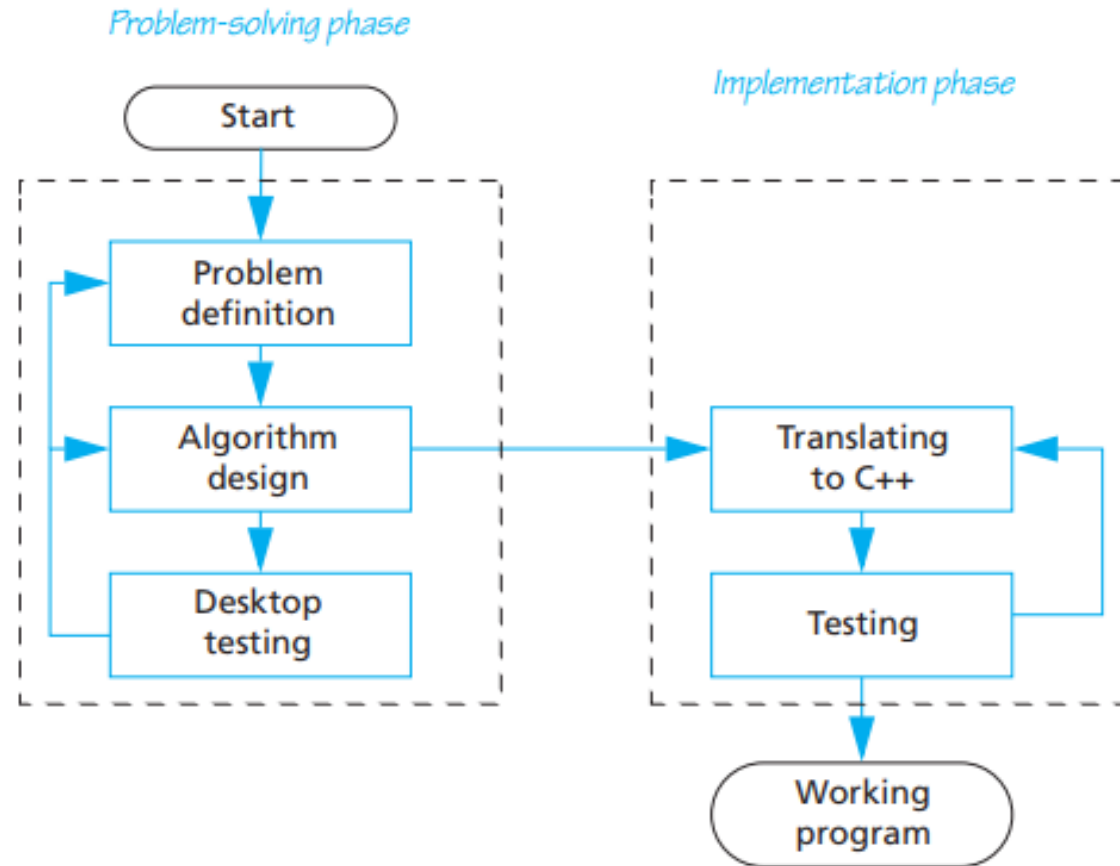
- Data is the input to the program, and both the program and the data are input to the computer (usually via the operating system).



# Program Design: Communicate with the Processor

- A **computer program** is simply an **algorithm** expressed in a language that a computer can understand.
- Program design process can be divided into two phases, the problem-solving phase and the implementation phase.
- The **result** of the **problem-solving phase** is an **algorithm**, expressed in English, for solving the problem.
- To produce a **program** in a programming language such as C++, the algorithm is translated into **the programming language** producing the final program from the algorithm is called the **implementation phase**.

# Program Design Scheme



# Then, what are Algorithms and Programming Languages?

- A **sequence of precise instructions** which leads to a solution is called an **algorithm**.
- The instructions may be expressed in a programming language or a human language.
- Our algorithms will be expressed in English and in the programming language C++.

# Example: Algorithm that determines how many times a name occurs in a list of names

1. Get the list of names.
2. Get the name being checked.
3. Set a counter to zero.
4. Do the following for each name on the list:
  - A- Compare the name on the list to the name being checked, and if the names are the same, then add one to the counter.
5. Announce that the answer which is the number indicated by the counter.

# What is Programming? – Implementation Phase

- It is the process of **designing and building an executable computer program** for accomplishing a specific computing task by giving computer instructions about what they should do next.

# What is a Programming Language?

- A **set of instructions** to create a computer programs that **implements** specific **algorithms**.
- It provides a linguistic framework for describing computations

# Levels of Programming Languages

- High-level Programming
- Low-level Programming
- Executable Machine Code

```
class Triangle {  
    ...  
    float surface()  
        return b*h/2;  
}
```

```
LOAD r1,b  
LOAD r2,h  
MUL r1,r2  
DIV r1,#2  
RET
```

```
0001001001000101  
0010010011101100  
10101101001...
```



# First Generation

- Low level language.
- Machine language.
- Very efficient and fast code but very difficult to write.

# Second Generation

- Low level language.
- Assembly languages.
- Very efficient code and easier to write.
- Each assembly language instruction is translated into one machine language instruction.

# Third Generation

- High level language.
- Closer to English but included simple mathematical notation.
- C, C++, Java.

# Fourth Generation

- High level language.
- Consists of statements similar to statements in a human language.
- Commonly used in database programming.
- Examples: PHP, Python, SQL.

# Fifth Generation

- High level language.
- Contain visual tools to help develop a program.

# How to analyze a problem?

- 1) **Understand** the Overall problem
- 2) Understand problem **requirements**
  - Does program require user interaction?
  - Does program manipulate data?
  - What is the output?
- 3) Implement the **algorithm**
- 4) If the problem is complex, divide it into **sub-problems**
  - Analyze each sub-problem as above

# Example: Write a pseudo code and flow chart to find the perimeter and area of a square

The perimeter and area of the square are given by the following formulas:

$$\text{perimeter} = \text{Side Length} * 4$$

$$\text{area} = \text{Side Length} * \text{Side Length}$$

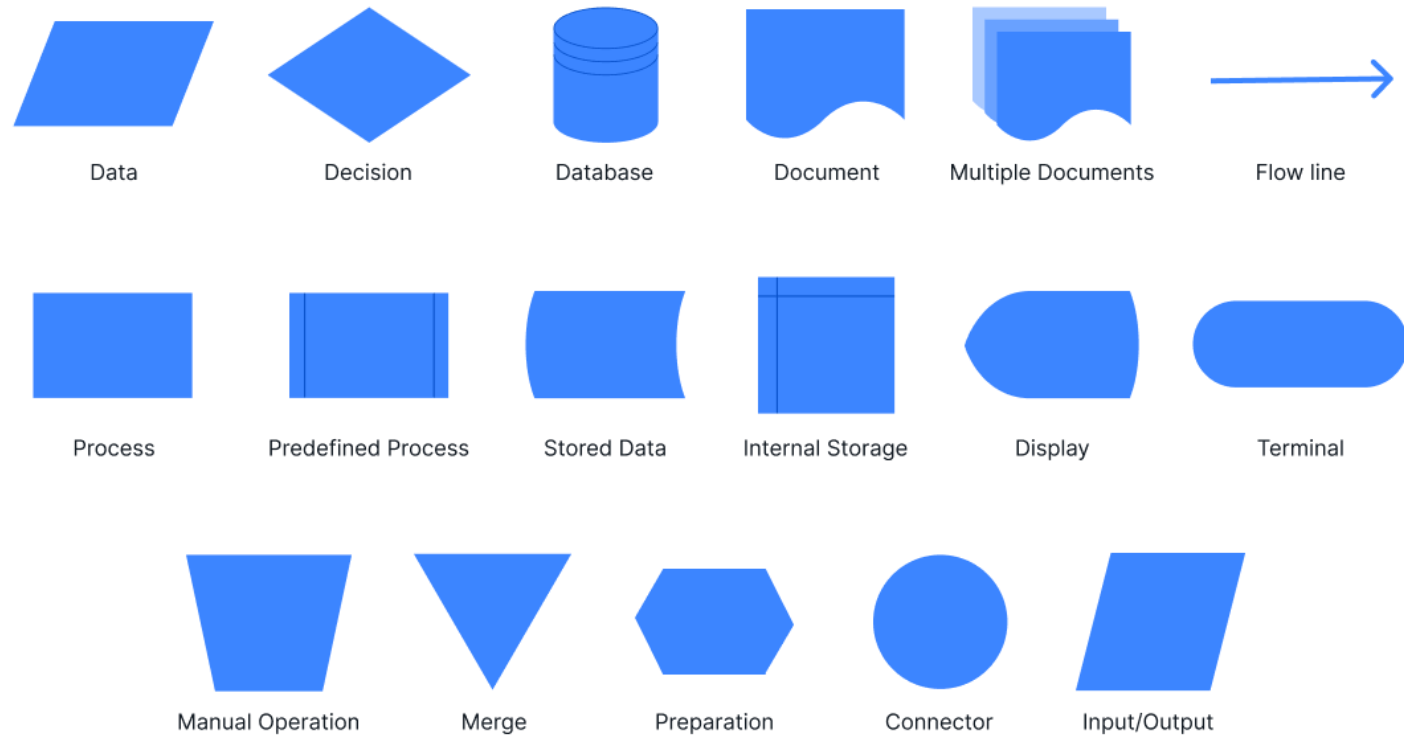
- Input:
  - Square Side Length
- Processing:
  - $\text{perimeter} = \text{Side Length} * 4$
  - $\text{area} = \text{Side Length} * \text{Side Length}$
- Output:
  - Print Out The Perimeter and Area.

# Pseudo Code

1. Input the length of a side of the square  
(side\_length)
2. Calculate the perimeter of the square:  $\text{perimeter} = 4 * \text{side\_length}$
3. Calculate the area of the square:  $\text{area} = \text{side\_length} * \text{side\_length}$
4. Output the calculated perimeter and area



# Flow Chart Rules to Remember!



Practice Problem:

Try it now!

# Syntax of a Programming Language

- English is a natural language. It has words, symbols and grammatical rules.
- A programming language also has words, symbols and rules of grammar.
- The grammatical rules are called syntax.
- Each programming language has a different set of syntax rules.

# Compilers & Interpreters

- **Compiler** is a program that translates a high-level language program, such as a C++ program, into a machine-language program that the computer can directly understand and execute it as one go.
- **Interpreter** is a program that **reads and executes the source code line by line**, translating and executing each line immediately without the need for a separate compilation step.

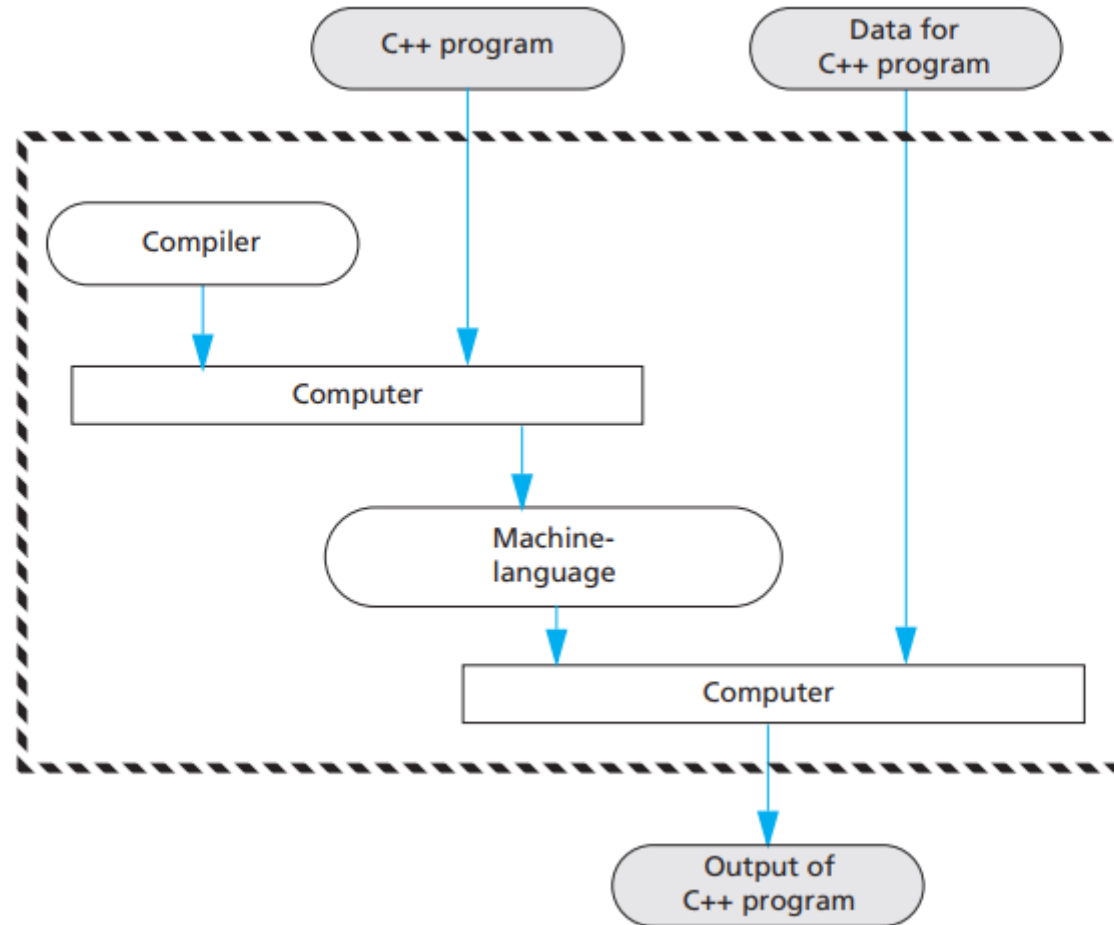
# Compilers & Interpreters

Compiler	Interpreter
<ul style="list-style-type: none"><li>• A compiler takes the entire program in one go.</li></ul>	<ul style="list-style-type: none"><li>• An interpreter takes a single line of code at a time.</li></ul>
<ul style="list-style-type: none"><li>• The compiler generates an intermediate machine code.</li></ul>	<ul style="list-style-type: none"><li>• The interpreter never produces any intermediate machine code.</li></ul>
<ul style="list-style-type: none"><li>• The compiler is best suited for the production environment.</li></ul>	<ul style="list-style-type: none"><li>• An interpreter is best suited for a software development environment.</li></ul>
<ul style="list-style-type: none"><li>• The compiler is used by programming languages such as C, C ++, C #, Scala, Java, etc.</li></ul>	<ul style="list-style-type: none"><li>• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.</li></ul>

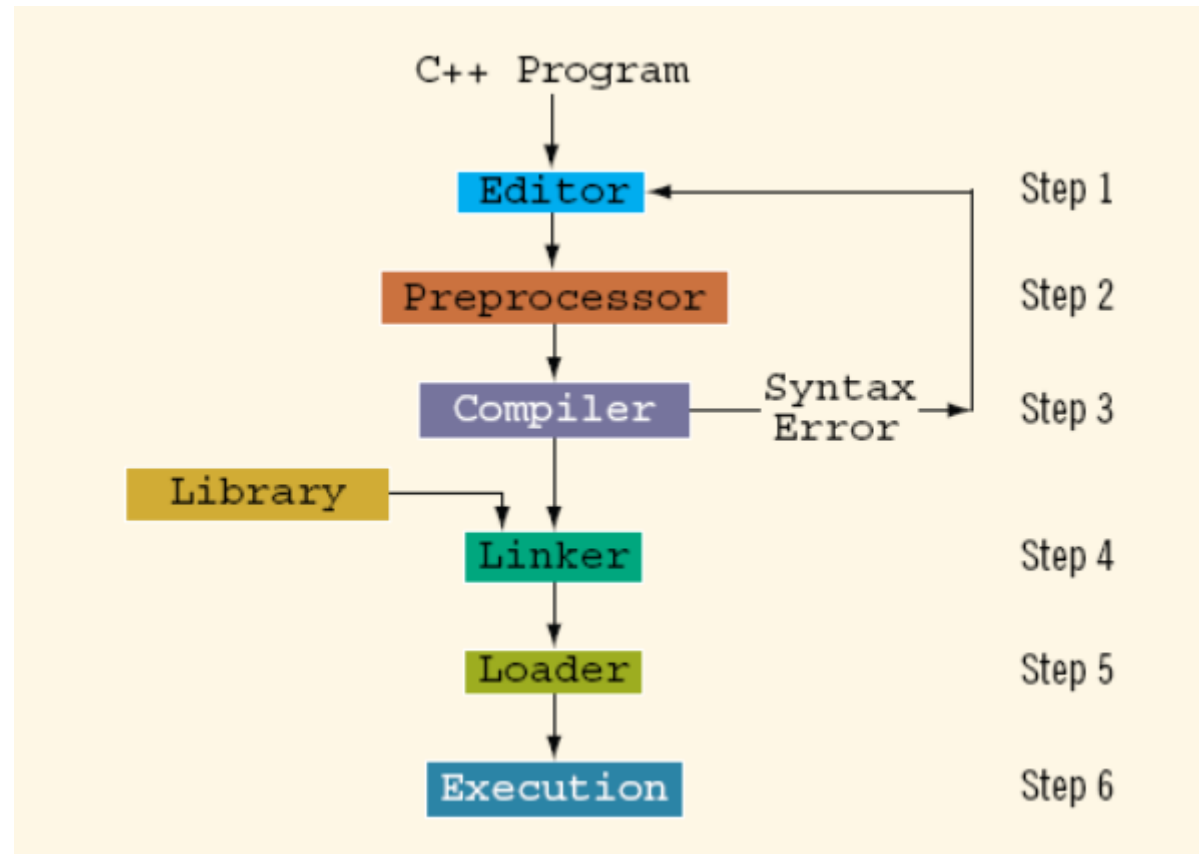
# Source Code & Object Code

- **Code** is frequently is used to mean a **program or a part of a program**, and this usage is particularly common when referring to object programs.
- **Source Code**: Source code refers to the **human-readable form of a program** written by a developer. It is typically written in a programming language such as C++, Python, Java, or JavaScript.
- **Object Code**: Object code, also known as **machine code**, is the compiled or translated version of the source code.

# Scheme of Running a C++ Program



# Processing in a C++ Program





# Processing in a C++ Program

- Pre-processor: It's a tool or component that processes the source code before it is compiled or interpreted by the compiler or interpreter. The preprocessor performs various tasks such as text manipulation, conditional compilation, and inclusion of header files. The preprocessor directives, typically starting with a '#' symbol, provide instructions to the preprocessor, which then modifies the source code accordingly.

# Processing in a C++ Program

- Linker: In C++, the linker is a crucial component of the compilation process. It is responsible for combining multiple object files and libraries into a single executable or library file. The linker's primary role is to resolve references between different parts of a program and ensure that all necessary code and data are linked together correctly.

# Processing in a C++ Program

- Loader: In the context of a C++ program, a loader is responsible for loading and preparing an executable file or a dynamically linked library (DLL) into memory for execution. The loader is an integral part of the operating system's runtime environment and is involved in the process of running a program.

# Software Development Life Cycle

Designers of large software systems, such as compilers and operating systems, divide the software development process into **six phases collectively** known as the **software development life cycle**. The six phases of this life cycle are:

1. Analysis and specification of the task (problem definition)
2. Design of the software (object and algorithm design)
3. Implementation (coding)
4. Testing
5. Maintenance and evolution of the system
6. Obsolescence



# Section 3

# Basics of C++ Programming

## Internet of Things: Theory and Applications

“Programming is the literacy of engineers nowadays!”

# What is C/C++?

- Bell laboratories developed C++ in the early 1980s. Stroustrup designed C++ to be a better C.
- The C language was derived from the B language, and C++ was derived from the C language.
- It is a **general purpose** programming language.
- It is an **object oriented** programming language.
- It provides facilities for **low-level memory manipulation**.

# Integrated Development Environment (IDE) for C++ Programming



**1- Download here:**

<https://sourceforge.net/projects/codeblocks/files/Binaries/20.03/Windows/codeblocks-20.03mingw-setup.exe/download>

**2- Install, select & agree to all**

**3- Set the GNU GCC Compiler as the default compiler**

**4- Set as default the IDE as default for C++ (yes all) when opens, and open it.**



# Download the GNU GCC Compiler:

- Download from here: <https://sourceforge.net/projects/mingw-w64/>
- Unzip the file and install the compiler (select & agree to all)
- Copy the folder inside the zipped folder into the C drive.
- Go to Code Block, settings > compiler > tool chain executables > auto-detect > OK

# Start New C++ Program in Code Blocks

- Follow these steps:
  1. File
  2. New
  3. Project
  4. Console Application
  5. Go
  6. Then, the Console Application Wizard will appear:
    1. Select Next
    2. Select C++ then Next
    3. Type Project Title, and Browse to a folder then select the path. Then, Next.
    4. Make sure you use GNU GCC Compiler and Check Release & Debug checkboxes then Next.

# Where to Code your Program & Hello World!

- Go to workplace:
  - Go to your project, named with you project title.
  - Go to sources.
  - Go to main and browse to find the following screen:
  - Press run and build and the black screen of the output appears.

NB: Make sure to run only one program at a time in one main function.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

Hello world!

Process returned 0 (0x0) execution time : 0.063 s  
Press any key to continue.

# Simple Layout of the C++ Program

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      Variable_Declarations
7
8      Statement_1
9      Statement_2
10     ...
11     Statement_Last
12
13     return 0;
14 }
```

# Example Program

```
#include <iostream> // iostream is the library that contains the definitions of the routines that handle input from the keyboard and output to the screen
using namespace std; // This line says that the names defined in iostream are to be interpreted in the “standard way” (std is an abbreviation of standard)
int main( ) // The correct term is main function, rather than main part
{int number_of_weeks, sections_per_week, total_sections; //variable declaration
cout << "Press return after entering a number.\n"; //statement to show the user a text and enters a new line
cout << "Enter the number of weeks:\n";
cin >> number_of_weeks; //the programmer prepares the code to receive a user input
cout << "Enter the number of sections in a week:\n";
cin >> sections_per_week;
total_sections = number_of_weeks * sections_per_week; //logical equation
cout << "If you have ";
cout << number_of_weeks;
cout << " weeks\n";
cout << "and ";
cout << sections_per_week;
cout << " sections in each week, then\n";
cout << "you have ";
cout << total_sections;
cout << " sections in all the weeks.\n";
return 0; // end the program when you get to here
}
```

# Output

```
Press return after entering a number.  
Enter the number of weeks:  
10  
Enter the number of sections in a week:  
14  
If you have 10 weeks  
and 14 sections in each week, then  
you have 140 sections in all the weeks.  
  
Process returned 0 (0x0)   execution time : 4.995 s  
Press any key to continue.
```

# Let us Learn Syntax?

- The **syntax** for a programming language (or any other kind of language) is the **set of grammar rules** for that language.
- For example, when we talk about the syntax for a variable declaration, we are talking about the rules for writing down a well-formed variable declaration which the compiler understand and operates.
- If you follow all the syntax rules for C++, then the **compiler will accept your program**, otherwise it will give an error.

# Variables and Operators



# What are Variables?

- Programs manipulate data such as numbers and letters, these data shall be constructed for effective manipulation.
- C++ and most other common programming languages use programming constructs known as variables to name and store data.
- The number or other type of **data stored** in a variable is called its **value**.
- **Variables** in **C++** have the same meaning as **variables** in **algebra**. That is, they represent some unknown value.

$$x = a + b$$

$$z + 2 = 3(y - 5)$$

# Variables in Memory Location

- The compiler assigns **a memory location (of the kind discussed in to each variable name)** in the program.
- The **value** of the variable, in **a coded form consisting of 0s and 1s, is kept in the memory location** assigned to that variable.
- Each **location** of these variables is given **an address**.

# Identifiers

- The **name of a variable** (or other item you might define in a program) is called an **identifier which is representations containing multiple characters for the variables**.
- Identifiers are case sensitive in C++.

# Identifiers

- There are rules for these representations or identifiers which are:
- **Variable names in C++:**
  - May only consist of letters, digits, and underscores.
  - May be as long as you like, but only **the first 31 characters are significant**.
  - May **not begin with a number**.
  - May not be a **C reserved word (keyword)**.

# C Reserved Words

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

# Naming Conventions of C Identifiers

- C programmers generally agree on the following **conventions** for naming variables.
  - **Begin** variable names with **lowercase letters**.
  - Use **meaningful** identifiers.
  - **Separate** “words” within identifiers with **underscores** or mixed upper and lower case which named the camel case.
- Examples: **surfaceArea**                      **surface\_Area**  
    **surface\_area**
- Be consistent and use the same in all!

# Declaring Variables – Marking Identifiers

- **Before using a variable**, you must give the compiler some information about the variable; i.e., you must **declare** it.
- The **declaration statement** includes the **data type and data name (identifier)** of the variable:

Type\_Name Variable\_Name\_1, Variable\_Name\_2,

- Examples of variable declarations:

```
int   number_of_students;  
float area_of_class ;
```

*NB: When there is more than one variable in a declaration, the variables are separated by commas. Also, note that each declaration ends with a semicolon.*

# What are Data Types?

- C has three basic predefined data types:
  - **1) Integers (whole numbers)**
    - int, long int, short int, unsigned int
  - **2) Floating point (real numbers)**
    - float, double
  - **3) Characters**
    - char

At this point, you need only be concerned with the data types that are: char, double, and int.



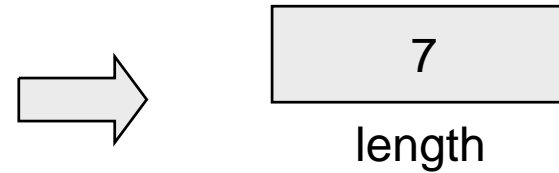
# Data Types and Memory

- The **value of any variable** is coded as a string of **0s and 1s**.
- Different **types of variables** require **different sizes of memory locations and different methods for coding their values** as a string of 0s and 1s.

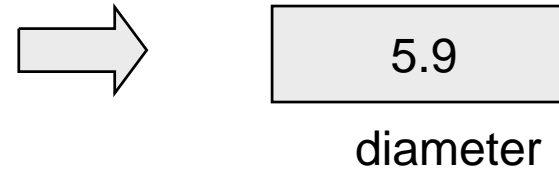
# Variable Initialization

- Variables **may be** given initial values, or **initialized**, when declared.
- Examples:

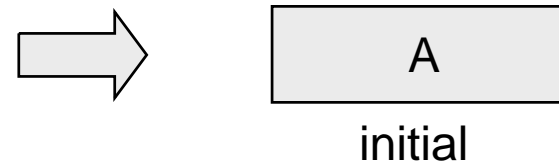
```
int length = 7 ;
```



```
float diameter = 5.9 ;
```



```
char initial = 'A' ;
```



# Alternative Syntax for Initializing In Declaration

## Syntax:

```
Type_Name Variable_Name_1 (Expression_for_Value_1),  
Variable_Name_2 (Expression_for_Value_2), . . .;
```

## Example:

```
int count(0), limit(10), fudge_factor(2);  
double distance(999.99);
```

# Assignment Declaration – Mapping Identifier to a Value

- An assignment statement always consists of a **variable on the left-hand side of the equal sign and an expression on the right-hand side**.
- An assignment statement ends with a **semicolon**.
- The expression on the right-hand side of the equal sign may be a **variable, a number, or a more complicated expression made up of variables, numbers, and arithmetic operators such as \* and +**.

## Syntax

`Variable = Expression;`

## Examples:

```
total_weight = one_weight + number_ofBars;
```

```
total_weight = one_weight;
```

```
number_ofBars = 37;
```

# Symbolic Constants – One Identifier, One Value

- In C++, a symbolic constant refers to a **named value that cannot be changed during program execution**.
- It is created using the **const** keyword that provides a meaningful name for a constant value.
- In constant naming, we almost use all uppercase for **symbolic constants** (a naming convention).
- Symbolic constants are also known as named **constants** as:

```
const int MAX_VALUE = 100;
```

# Another Method for Symbolic Constants

- Used in **#define** preprocessor directives.
- Examples:

```
#define PI 3.14159  
#define AGE 52
```

# What are Directives & Namespaces?

- In C++, directives are **special instructions that provide guidance to the preprocessor**, a component of the compiler.
- Directives **begin with a hash symbol "#"** and are used to **include header files, define constants, control compilation behavior**, and perform other preprocessor-related tasks.

# What are Directives & Namespaces?

- C++ divides names into **namespaces**.
- A namespace is a **collection of names**, such as the names **cin** and **cout** which the user could input or output.
- The reason that C++ has namespaces at all is because there are so many things to name. As a result, sometimes two or more items receive the same name; that is, a **single name can get two different definitions**.
- A statement that specifies a namespace in the way illustrated by the following is called a **using directive**.



# Start with Including Important Directives

- **Include directives** is used to **add library** files to our programs.
- To make the definitions of the **cin** and **cout** available to the program we shall use:

**#include <iostream>**

- **Using directives** include a collection of defined names as namespace.
- To make the names **cin** and **cout** available to our program:

**using namespace std;**

# How to Print Text in C++?

- **cout** is an **output stream** sending data to the monitor.
- The **insertion operator** "<<" **inserts data into cout**

- **Example:**

```
cout << number_of_bars << " candy bars\n";
```

- **This line sends two items to the monitor:**

- The value of `number_of_bars`
- The quoted string of characters " candy bars\n"
- ***Notice the space before the 'c' in candy for better reading.***
- The '\n' causes a new line to be started following the 's' in bars

*NB: A new insertion operator is used for each item of output*

# How to Print Text in C++?

- This produces the same result as the previous sample

```
cout << number_of_bars ;  
cout << " candy bars\n";
```

- Here is an arithmetic is performed in the cout statement:

```
cout << "Total cost is $" << (price + tax);
```

- Quoted strings are enclosed in double quotes ("Walter")
- NB: Don't use two single quotes ('')
- A blank space can also be inserted with:

```
cout << " " ;
```

# Input Text using C++

- cin is an input stream bringing data from the keyboard
- The **extraction operator (>>)** removes data to be used

# cin Important Functions

- **cin.ignore():**
- **cin.ignore() is used to skip or discard characters in the input buffer.**
- It is commonly used after using cin to read input, especially when switching between different types of input.
- One common usage is to clear the newline character (\n) from the input buffer before using getline() to read a line of text.
- **cin.peek():**
- **cin.peek() allows you to look at the next character in the input stream without removing it from the stream.**
- It helps you *examine the upcoming character before deciding how to handle the input.*
- You can use it to check the next character and make decisions based on its value.
- It returns the next character in the input stream or -1 if there are no more characters.

# cin.ignore example:

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int number;
```

```
    char newline;
```

```
    cout << "Enter a number: ";
```

```
    cin >> number;
```

# cin.ignore example:

```
cin.ignore(); // Ignore the newline character in the input buffer

    cout << "Enter a sentence: ";
    string sentence;
    getline(cin, sentence);

    cout << "Number: " << number << endl;
    cout << "Sentence: " << sentence << endl;

    return 0;
}
```

# cin.peek example

```
#include <iostream>
using namespace std;
int main() {
    char ch;
    cout << "Enter a character: ";
    ch = cin.peek();
    if (ch == '\n') {
        cout << "You entered a newline character." << endl;
    } else {
        cout << "The next character is: " << ch << endl;
    }
    return 0;}
```



# Using cout & cin Together

```
cout << "Enter the number of bars in \n";  
cout << " and the weight in ounces of one bar.\n";  
cin >> number_of_bars;  
cin >> one_weight;
```

- This code prompts the user to enter data then **reads two data items from cin**
- The first value read is stored in number\_of\_bars
- The second value read is stored in one\_weight
- *NB: Data is separated by spaces when entered*

# Escape Sequence

***NB: The backslash, \, preceding a character tells the compiler that the character following the \ does not have the same meaning as the character appearing by itself. such a sequence is called an escape sequence.***

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Use to print a single quote character.
\"	Double quote. Used to print a double quote character.

# Comments

- For one line comment use //
- For multi-lines comment use /\* \*/

# Number Types: int & double

- Conceptually, the numbers 2 and 2.0 are the same number. But C++ considers them to be of **different types**.
- Numbers of type *int* are stored **as exact values**.
- The precision with which *double* values are stored varies from one computer to another, but you can expect them to be stored **with 14 or more digits of accuracy**.
- NB: This **e notation** is used because keyboards normally have no way to write exponents as superscripts.

# Formatting for Numbers with a Decimal Point

- When the computer outputs a value of type **double**, the format may not be what you would like.
- For example, the following simple cout statement can **produce any of a wide range of outputs**:

```
cout << "The price is $" << price << endl;
```

The output:

If price has the value 78.5, the output might be

The price is \$78.500000

or it might be

The price is \$78.5

# Example

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

int main() {
    double budget = 100.50;

    cout << "For a price of $" << budget << endl;

    return 0;
}
```

# Comment on Example

- In the previous example, the message "For a price of \$" is **concatenated** with the value of the variable budget using the << operator.
- The **endl manipulator is used to insert a newline character** after the output, ensuring the next output appears on a new line.
- The resulting string is then printed to the console using cout as previously stated.

# Some cout Function – Double Handling

- **cout.setf(ios::fixed); sets the floating-point number output format to fixed-point notation.** This means that floating-point numbers will be displayed with a fixed number of decimal places, regardless of their actual precision.
- **cout.setf(ios::showpoint); enables the display of the decimal point and trailing zeros for floating-point numbers,** even if they are not necessary for the given precision.
- **cout.precision(2); sets the precision of floating-point numbers to 2 decimal places.** This means that when displaying floating-point numbers using cout, they will be rounded to two decimal places.



# Example for cin Functions

```
#include <iostream>

using namespace std;

int main() {
    double number = 3.14159;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    cout << "Number: " << number << endl;

    return 0;
}
```

# Number Types

Type Name	Memory Used	Size Range	Precision
<i>short</i> (also called <i>short int</i> )	2 bytes	-32,768 to 32,767	(not applicable)
<i>int</i>	4 bytes	-2,147,483,648 to 2,147,483,647	(not applicable)
<i>long</i> (also called <i>long int</i> )	4 bytes	-2,147,483,648 to 2,147,483,647	(not applicable)
<i>float</i>	4 bytes	approximately $10^{-38}$ to $10^{38}$	7 digits
<i>double</i>	8 bytes	approximately $10^{-308}$ to $10^{308}$	15 digits
<i>long double</i>	10 bytes	approximately $10^{-4932}$ to $10^{4932}$	19 digits

# Char Data Type

- A variable of type *char* can hold any single character on the keyboard. so, for example, the variable `symbol` could hold an 'A' or a '+' or an 'a'.
- Note that uppercase and lowercase versions of a letter are considered different characters.
- Example:

```
char symbol, letter;
```

# Example on Char

```
#include <iostream>
using namespace std;
int main( )
{
char symbol1, symbol2, symbol3;
cout << "Enter two initials, without any periods:\n";
cin >> symbol1 >> symbol2;
cout << "The two initials are:\n";
cout << symbol1 << symbol2 << endl;
cout << "Once more with a space:\n";
symbol3 = ' ';
cout << symbol1 << symbol3 << symbol2 << endl;
cout << "That's all.";
return 0;
}
```

# Example Output

```
Enter two initials, without any periods:  
M H  
The two initials are:  
MH  
Once more with a space:  
M H  
That's all.  
Process returned 0 (0x0)   execution time : 1.586 s  
Press any key to continue.
```

# Bool Datatype

- Expressions of type bool are called Boolean after the English mathematician George Boole (1815–1864), who formulated rules for mathematical logic which **is based only on true or false or 0 or 1.**
- **True & False, 0 & 1 could be used interchangeably.**

# String Class

- To use the string class we must first include the string library:  
**#include <string>**
- This is to be discussed more in the next classes.

# Example on Strings

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string second_name, first_name;
    string all_name;
    cout << "Enter your middle name and the name of your pet.\n";
    cin >> first_name;
    cin >> second_name;
    all_name = first_name + " " + second_name;
    cout << "The name is ";
    cout << all_name << "." << endl;
    return 0;
}
```



# Output

```
Enter your middle name and the name of your pet.  
mohamed  
hatem  
The name is mohamed hatem.  
  
Process returned 0 (0x0)   execution time : 2.990 s  
Press any key to continue.
```

# Unsigned Data Types

- In C++, the unsigned keyword is used to **declare variables that can hold only positive values or zero.**
- It is typically used with **integer data types, such as int, short, long, long long, char,** etc., to specify that the variable should only store non-negative values.
- Here's an example that demonstrates the usage of unsigned with different data types:

# Example

```
#include <iostream>
using namespace std;
int main() {
    unsigned int positiveInt = 10;
    unsigned short positiveShort = 20000;
    unsigned long positiveLong = 1234567890;
    unsigned long long positiveLongLong = 9876543210;
    unsigned char positiveChar = 'A';
```

# Example

```
cout << "Unsigned int: " << positiveInt << endl;
    cout << "Unsigned short: " << positiveShort << endl;
    cout << "Unsigned long: " << positiveLong << endl;
    cout << "Unsigned long long: " << positiveLongLong <<
endl;
    cout << "Unsigned char: " << positiveChar << endl;
    return 0;
}
```

# Output

```
Unsigned int: 10
Unsigned short: 20000
Unsigned long: 1234567890
Unsigned long long: 9876543210
Unsigned char: A

Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

# Type Casting

- In C++, type casting refers to the conversion of one data type to another. It allows you to **change the interpretation or representation of a value from one type to another.** There are several ways to perform type casting in C++, including:
  - **1) Implicit Type Conversion**

```
int num1 = 10;  
double num2 = num1; // Implicit conversion from int to  
double
```

# Type Casting

- **2) Explicit Type Conversion**

```
double num1 = 3.14;  
int num2 = static_cast<int>(num1); // Explicit  
conversion from double to int
```

# Type Casting

- 3) C-Style Type Casting:

```
double num1 = 3.14;  
int num2 = (int)num1; // C-style cast from double  
                      to int
```



# Operators in C++

## Operators in C++

Unary operator

Binary operator

Ternary operator

Operator	Type
++, --	Unary operator
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
&&,   , !	Logical operator
&,  , <<, >>, ~, ^	Bitwise operator
=, +=, -=, *=, /=, %=	Assignment operator
?:	Ternary or conditional operator

# Increment and Decrement Operators

- Increment operator: **increment variable by 1:**
  - Pre-increment: ++variable
  - Post-increment: variable++
- Decrement operator: **decrement variable by 1:**
  - Pre-decrement: --variable
  - Post-decrement: variable --
- Examples:
  - ++K , K++ → **k= K+1**
  - K , K-- → **K= K-1**

# Increment and Decrement Operators

- If the value produced by ++ or -- is not used in an expression, it **does not matter whether it is a pre or a post increment (or decrement)**.
- When ++ (or --) is used **before** the variable name, the computer first increments (or decrements) the value of the variable and **then uses its new value to evaluate the expression**.
- When ++ (or --) is used **after** the variable name, the computer **uses the current** value of the variable to evaluate the expression, and then it increments (or decrements) the value of the variable.
- **There is a difference between the following**

```
x = 5;  
Cout << ++x;
```

```
x = 5;  
Cout << x++;
```

# Special Assignment Statement

- C++ has special assignment statements called compound assignments

**+= , -= , \*= , /= , % =**

- Example:

**X +=5 ;** means **x = x + 5;**

**x \*=y;** means **x = x \* y;**

**x /=y;** means **x = x / y;**

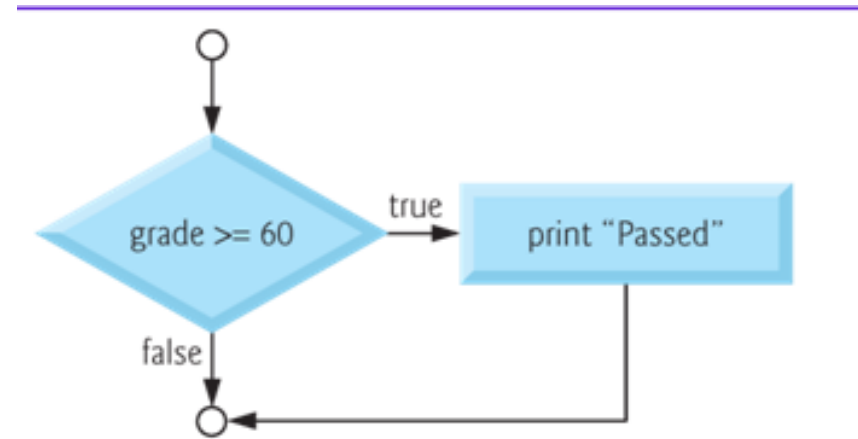
# Conditions

# Flow of Control to Selection Statement

- The **order** in which statements are executed is often referred to as **flow of control**.
- **Selection statements** are used to choose **among alternative courses of action in a flow of control**.
- **Flow control has many types which are:**
  - **Condition statements > if else, switch**
  - **Loops > do while, while, for**
  - **Jump statements > break, continue, return, go to**

# If Conditions Concept & Flow Chart

- For example, suppose the passing mark on an exam is 60.
- The pseudocode statement is “ **If student’s marks is greater than or equal to 60 Then Print “Passed”** ”
- The follow chart of the If condition is as the following:



Flowcharting the single-selection if statement.

# If Statement Syntax in C++

- The **Expression** can be **any valid expression** including a **relational expression, logical expressions, and even arithmetic expression**
- In case of **using arithmetic expressions** , a **non-zero** value is considered to be **true**, whereas a **0** is considered to be **false**

```
if ( Expression)
{
    action statement 1 ;
    action statement 2 ;
    .
    .
    action statement n ;
}
```

```
if ( grade >= 60 )
    cout <<"Passed\n";
```



# Relational Expressions and Relational Operators

- **Relational Expression** is an expression which **compares 2 operands** and returns a **TRUE (1) or FALSE (0) answer.**

Example :

a >= b

a == c

a >= 99

'A' > 'a'

- **Relational Expressions** are used to **test the conditions** in selection, and looping statements.

# Comparison Operators in Relational Expressions

Operator	Means
==	Equal To
!=	Not Equal To
<	Less Than
<=	Less Than or Equal To
>	Greater Than
>=	Greater Than or Equal To

# More Comparison Operators

Math Symbol	English	C++ Notation	C++ Sample	Math Equivalent
<code>=</code>	equal to	<code>==</code>	<code>x + 7 == 2 * y</code>	$x + 7 = 2y$
<code>≠</code>	not equal to	<code>!=</code>	<code>ans != 'n'</code>	$\text{ans} \neq 'n'$
<code>&lt;</code>	less than	<code>&lt;</code>	<code>count &lt; m + 3</code>	$\text{count} < m + 3$
<code>≤</code>	less than or equal to	<code>&lt;=</code>	<code>time &lt;= limit</code>	$\text{time} \leq \text{limit}$
<code>&gt;</code>	greater than	<code>&gt;</code>	<code>time &gt; limit</code>	$\text{time} > \text{limit}$
<code>≥</code>	greater than or equal to	<code>&gt;=</code>	<code>age &gt;= 21</code>	$\text{age} \geq 21$

# Logical Operators

- The “AND” Operator &&

You can form a more elaborated Boolean expression by combining two simple tests (almost relational expressions) using **the “and” operator &&**.

- The “OR” Operator ||

You can form a more elaborated Boolean expression by combining two simple tests (almost relational expressions) using the **“or” operator ||**.

- The “NOT” operator is !.

# Logical Operators – Truth Tables

<b>AND</b>		
<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1 &amp;&amp; Exp_2</i>
true	true	true
true	false	false
false	true	false
false	false	false

<b>OR</b>		
<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1    Exp_2</i>
true	true	true
true	false	true
false	true	true
false	false	false

<b>NOT</b>	
<i>Exp</i>	<i>!(Exp)</i>
true	false
false	true

# Short Form if

In C++, the short form of the if statement is known as the ternary operator. It allows you to write conditional expressions in a more concise way. Here's an example:

```
#include <iostream>
using namespace std;
int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    string result = (number % 2 == 0) ? "Even" : "Odd";
    cout << "The number is " << result << endl;
    return 0;}

```

# Short Form if

- In this example, the ternary operator ? : is used to check if the number is even or odd. If the condition (number % 2 == 0) is true, the value assigned to result is "Even", otherwise, it is assigned "Odd".
- This short form of the if statement helps to write the code more concisely and in a single line.

# If else Syntax

```
• if(condition)
    {
        statementA1
        statementA2
        ...
    }
    else {
        statementB1
        statementB2
        ...
    }
```



# Nested If Syntax

Nested If : means to write an if statement within another if statement.

```
1  if (count > 0)
2      if (score > 5)
3          cout << "count > 0 and score > 5\n";
4      else
5          cout << "count > 0 and score <= 5\n";
```

```
if ( Expression)
{
    action statement n ;

    if ( Expression)
    {
        action statement 1 ;
        action statement 2 ;
        .
        .
        action statement n ;
    }
}
```

**Example :** Write a program that accepts an integer number from the user. In case the number is Positive , check and print out whether it is Even or Odd number.

```
int main() {  
    int number;  
    cout <<"Please Enter any number \n";  
    cin >>number;  
    if ( number >=0)  
        if (number % 2 == 0)  
            cout <<" This is an Even number \n";  
        else  
            cout <<"This is an Odd number \n \n";  
}
```

# If-else if – Take care about braces!

**ELSE IF : means to write an if statement with multiple choices**

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
    .
    .
    .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

# Combining more than One Condition

To combine more than one condition we use the logical operators.

Operator	Means	Description
<b>&amp;&amp;</b>	<b>And</b>	The Expression Value Is true If and Only IF both Conditions are true
<b>  </b>	<b>OR</b>	The Expression Value Is true If one Condition Is True

Example : check whether num1 is between 0 and 100

```
if ( (num1 >= 0) && (num1 <=100) )  
    cout <<“The Number Is between 1 and 100” ;  
Else  
    Cout <<“ The Number Is Larger Than 100”;
```

# Switch Statement

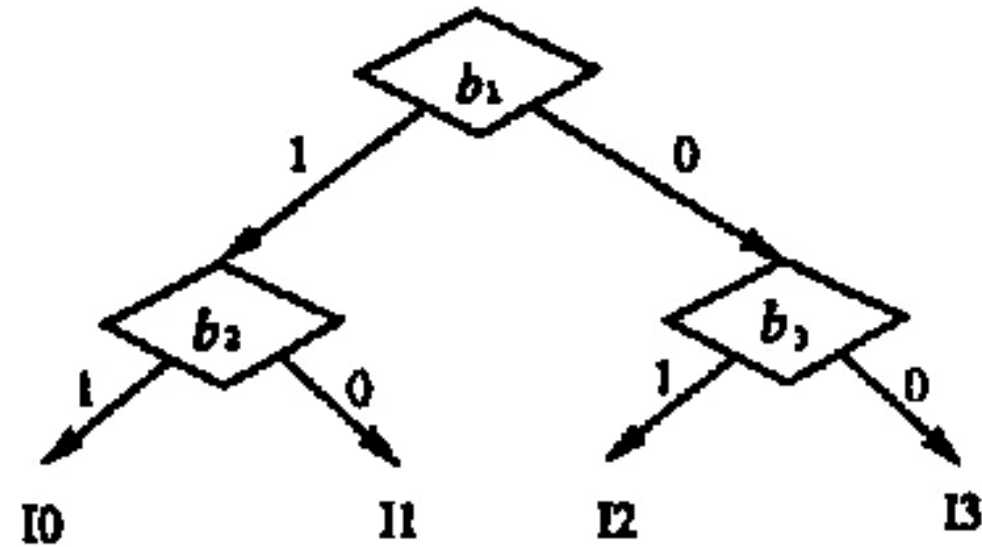
- You have seen ***if-else* statements** used to construct multiway branches.
- The ***switch* statement** is another kind of C++ statement that also implements **multiway branches**.
- Syntax:

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
```

```
    case Constant_2:
        Statement_Sequence_2
        break;
    .
    .
    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
```

# Multiway Branches

```
IF  $b_1$  THEN
    IF  $b_2$  THEN I0
    ELSE I1
ELSE
    IF  $b_3$  THEN I2
    ELSE I3
END
```





# Practice Problem

Write a C++ program that takes two numbers as input from the user and performs the following operations based on the values:

- If both numbers are even, compute their sum and display it.
- If both numbers are odd, compute their product and display it.
- If one number is even and the other is odd, compute the difference (subtract the odd number from the even number) and display it.

# Solution:

```
#include <iostream>

using namespace std;

int main() {
    int num1, num2;

    cout << "Enter the first number: ";
    cin >> num1;

    cout << "Enter the second number: ";
    cin >> num2;

    if (num1 % 2 == 0 && num2 % 2 == 0) {
        int sum = num1 + num2;

        cout << "Sum of the two even numbers: " << sum << endl;
    }
}
```

# Solution:

```
    else if (num1 % 2 != 0 && num2 % 2 != 0) {  
        int product = num1 * num2;  
        cout << "Product of the two odd numbers: " << product << endl;  
    }  
    else {  
        int difference = num1 - num2;  
        cout << "Difference between the even and odd numbers: " << difference << endl;  
    }  
    return 0;  
}
```

# Practice Problem:

Write a C++ program that calculates the final grade for a student based on their scores in three exams. The program should take the scores as input and calculate the average. Then, it should assign a letter grade based on the following scale:

- Average score  $\geq 90$ : A
- Average score  $\geq 80$ : B
- Average score  $\geq 70$ : C
- Average score  $\geq 60$ : D
- Average score  $< 60$ : F

# Solution:

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

int main()
{
    int score1, score2, score3;
    double average;
    char grade;

    cout << "Enter score for Exam 1: ";
    cin >> score1;

    cout << "Enter score for Exam 2: ";
    cin >> score2;
```

# Solution:

```
cout << "Enter score for Exam 3: ";  
cin >> score3;
```

```
    average = (score1 + score2 + score3) / 3.0;
```

```
    if (average >= 90)  
{  
        grade = 'A';  
    } else if (average >= 80) {  
        grade = 'B';  
    } else if (average >= 70) {  
        grade = 'C';  
    }
```

# Solution:

```
else if (average >= 60) {  
    grade = 'D';  
} else {  
    grade = 'F';  
}  
  
cout << "Average score: " << average << endl;  
cout << "Letter grade: " << grade << endl;  
  
return 0;  
}
```

# Loops



# Repetition Statements

- Some times we need to **repeat** a specific course of actions either a **specified number of times or until a particular condition is being satisfied.**
- **For example :**
  - To calculate the Average grade for 10 students
  - To calculate the bonus for 10 employees
  - To sum the input numbers from the user as long as the user enters positive numbers.

There are three methods by way of which we can repeat a part of a program.

- (a) Using a for statement**
- (b) Using a do-while statement**
- (C) Using a while statement**

# The While Loop Syntax

```
While ( continuation condition)
{
    Action statement 1 ;
    Action statement 2 ;
    .
    .
    Action statement n ;
}
```

# The While Loop

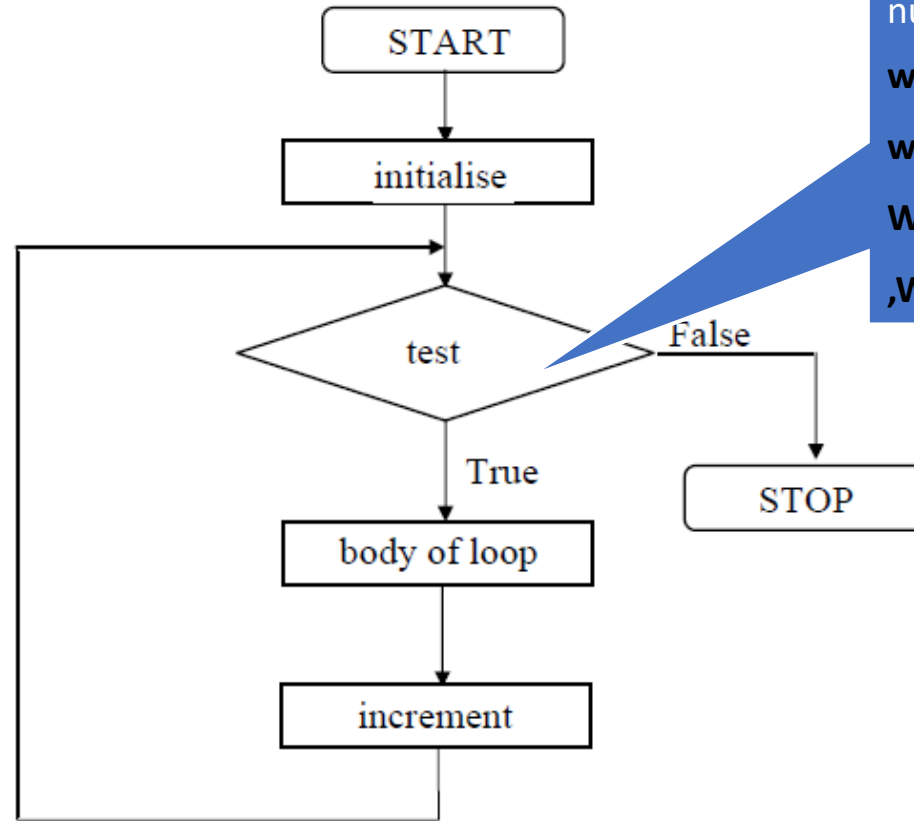
1- The **loop counter** is any **numeric variable** (int , float ,....).

2- When using a **counter** , the while loop is called **counter –controlled loop**.

3- The **initial value of the loop counter is up to you**, but you have to increment to reach the condition value avoid endless loops.

```
initialise loop counter ;  
while ( test loop counter using a condition )  
{  
    do this ;  
    and this ;  
    increment loop counter ;  
}
```

# The While Loop Flow Chart



The condition being tested may be relational, logical or even numeric expression :

**while ( i <= 10 )**

**while ( i >= 10 && j <= 15 )**

**While (3) ,While (3 + 5)**

**,While (true)**

# Example

```
#include <iostream>
using namespace std;
int main( )
{
    int count_down;
    cout << "How many greetings do you want? ";
    cin >> count_down;
    while (count_down > 0)
    {
        cout << "Hello ";
        count_down = count_down - 1;
    }
    cout << endl;
    cout << "That's all!\n";
    return 0;
}
```

```
How many greetings do you want? 10
Hello Hello Hello Hello Hello Hello Hello Hello Hello
That's all!

Process returned 0 (0x0)   execution time : 0.830 s
Press any key to continue.
```

# Example: Write a program that calculates and prints out the Average grade for 6 students .

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

int main() {
    int grade = 0, sum = 0;
    for (int counter = 1; counter <= 6; counter++) {
        cout << "Enter Grade: ";
        cin >> grade;
        sum += grade;
    }
    cout << "The Average grade is: " << sum / 6.0 << endl;

    return 0;
}
```

```
Enter Grade: 10
Enter Grade: 10
Enter Grade: 10
Enter Grade: 10
Enter Grade: 10
Enter Grade: 10
The Average grade is: 10

Process returned 0 (0x0)   execution time : 4.922 s
Press any key to continue.
```

Example : Write a program To print out the sum of the numbers entered from the user as long as he/she enters positive numbers.

```
#include <iostream>
using namespace std;
int main() {
    int number;
    int sum = 0;
    while (true) {
        cout << "Enter a positive number (or a negative number to exit):
";
        cin >> number;
        if (number < 0) {
            break;
        }
        sum += number;
    }
    cout << "Sum of the entered positive numbers: " << sum << endl;
    return 0;
}
```



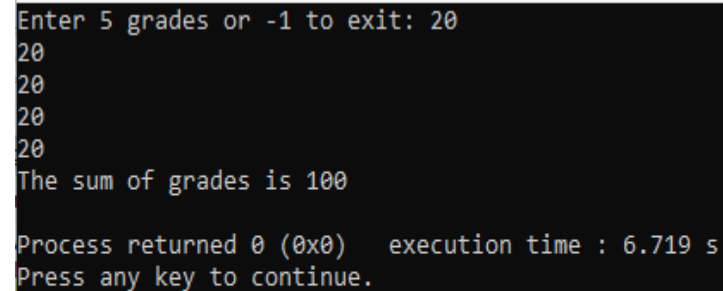
# Output

```
Enter a positive number (or a negative number to exit): 1
Enter a positive number (or a negative number to exit): 2
Enter a positive number (or a negative number to exit): 5
Enter a positive number (or a negative number to exit): 6
Enter a positive number (or a negative number to exit): 6
Enter a positive number (or a negative number to exit): 7
Enter a positive number (or a negative number to exit): 8
Enter a positive number (or a negative number to exit): 7
Enter a positive number (or a negative number to exit): -1
Sum of the entered positive numbers: 42

Process returned 0 (0x0)   execution time : 9.624 s
Press any key to continue.
```

**Example** : Write a program that calculates and prints out the average grade for 5 students or ends the program by entering -1

```
#include <iostream>
using namespace std;
int main() {
    int grade = 0, counter = 1, sum = 0;
    cout << "Enter 5 grades or -1 to exit: ";
    while (counter <= 5 && grade != -1) {
        cin >> grade;
        if (grade != -1) {
            sum += grade;
            counter++;
        }
    }
    cout << "The sum of grades is " << sum << "\n";
    return 0;
}
```



```
Enter 5 grades or -1 to exit: 20
20
20
20
20
The sum of grades is 100

Process returned 0 (0x0)   execution time : 6.719 s
Press any key to continue.
```

# Do While Loop Syntax

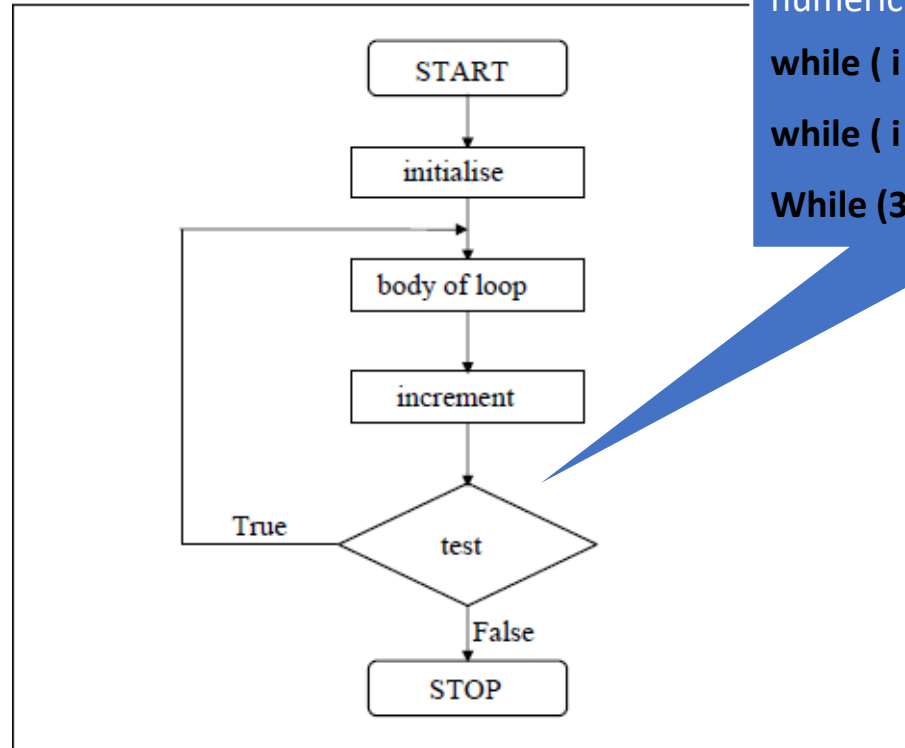
```
do
{
    this ;
    and this ;
    and this ;
    and this ;
} while ( this condition is true ) ;
```

```
while ( this condition is true )
{
    this ;
    and this ;
    and this ;
    and this ;
}
```

# Do While Loop Syntax

- 1- Loop counter is **any numeric variable** ( int , float ,....).
- 2- The initial value of the loop counter is up to you, but you have to **increment it to avoid endless loops**.
- 3- The Loop Body is Executed **at least one Time**.

# Do While Loop Flow Chart



The condition being tested may be relational, logical or even numeric expression :

**while ( i <= 10 )**

**while ( i >= 10 && j <= 15 )**

**While (3) , While (3 + 5)**

**Example :** Write a program that calculates and prints out the Average grade for 6 students .

```
#include <iostream>
using namespace std;
int main() {
    int counter = 1;
    int grade = 0, sum = 0;
    do {
        cout << "Enter grade for student no " << counter << ": ";
        cin >> grade;
        sum += grade;
        counter++;
    } while (counter <= 6);
    cout << "Average Grade is " << static_cast<double>(sum) / (counter - 1) << "\n";
    return 0;
}
```

```
Enter grade for student no 1: 10
Enter grade for student no 2: 10
Enter grade for student no 3: 10
Enter grade for student no 4: 10
Enter grade for student no 5: 10
Enter grade for student no 6: 10
Average Grade is 10
```

```
Process returned 0 (0x0)   execution time : 4.250 s
Press any key to continue.
```

# For Loop Syntax

- For Loop is probably **the most popular** looping instruction.
- General form of for statement is

```
for ( initialise counter ; test counter ; increment counter )  
{  
    do this ;  
    and this ;  
    and this ;  
}
```

```
for (Initialization_Action; Boolean_Expression; Update_Action)  
    Body_Statement
```

# The for allows us to specify three things about a loop in a single line:

- (a) Setting a **loop counter** to an initial value.
- (b) Testing the loop counter **to detect whether its value reached** the number of repetitions desired.
- (c) Increasing the value of loop counter **each time the program segment** within the loop has been executed.



**Example** : Write a program that prints out numbers from 0 to 10;

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

int main() {
    for (int i= 0 ; i <=10 ;
        i++)
        cout << i << endl;

    return 0;
}
```

```
0
1
2
3
4
5
6
7
8
9
10
Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

**Example** : Write a program that prints out numbers from 0 to 10 in descending order;

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

int main() {
    for (int i = 10 ; i >=0 ; i--
    )
        cout << i <<endl;

    return 0;
}
```

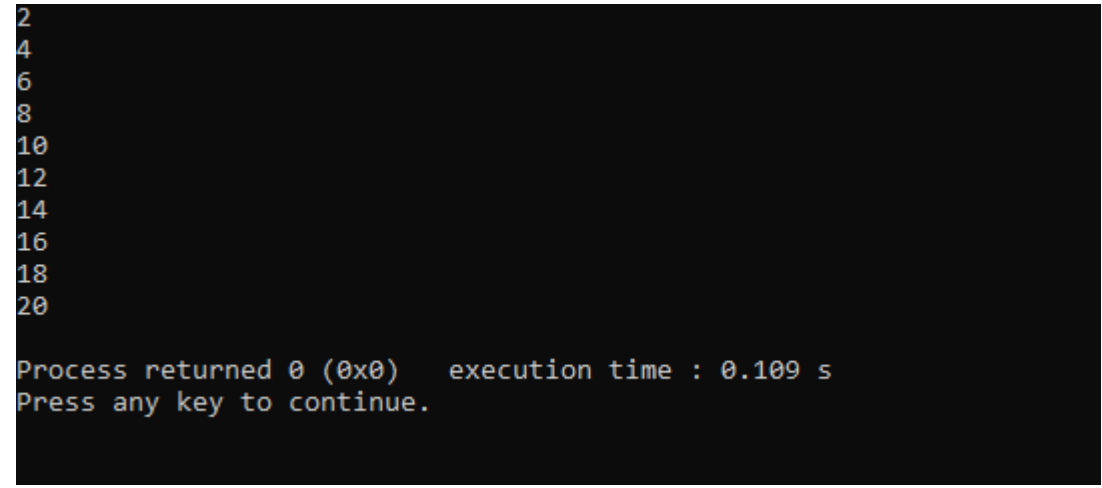
```
10
9
8
7
6
5
4
3
2
1
0
Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

**Example** : Write a program that prints out the even numbers from 2 to 20; - Using single for loop

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

int main() {
    for (int i = 2 ; i <=20 ;
        i+=2 )
        cout << i << endl;

    return 0;
}
```

A screenshot of a terminal window with a black background. The output of the program is a list of even numbers from 2 to 20, each on a new line. At the bottom, it shows the process return status and execution time.

```
2
4
6
8
10
12
14
16
18
20

Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

**Example :** Write a program that calculates and prints out the Average grade for 6 students .

```
#include <iostream>
using namespace std;
int main() {
    int grade = 0, sum = 0;
    for (int counter = 1; counter <= 6; counter++) {
        cout << "Enter grade for student no " << counter << ": ";
        cin >> grade;
        sum += grade;
    }
    cout << "Average Grade is " << sum / 6 << "\n";
    return 0;
}
```

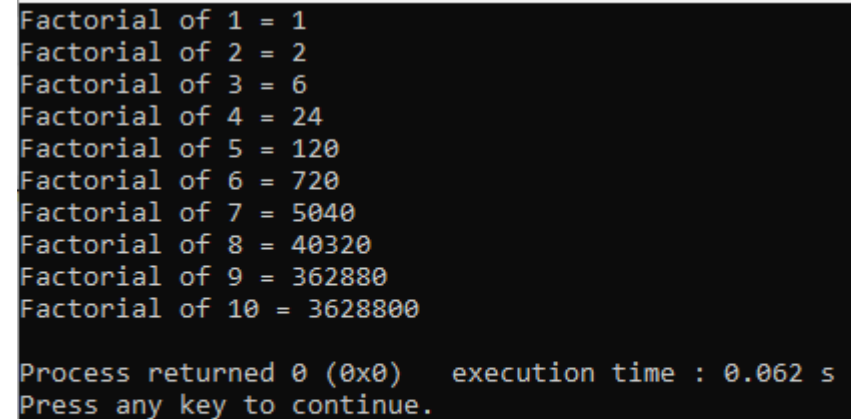
```
Enter grade for student no 1: 10
Enter grade for student no 2: 10
Enter grade for student no 3: 10
Enter grade for student no 4: 10
Enter grade for student no 5: 10
Enter grade for student no 6: 10
Average Grade is 10

Process returned 0 (0x0)   execution time : 5.000 s
Press any key to continue.
```

# Nested Loops

Example : Write a program that calculates the Factorial for numbers from 1 to 10;

```
#include <iostream>
using namespace std;
int main() {
    for (int number = 1; number <= 10; number++) {
        int factorial = 1;
        for (int i = 1; i <= number; i++) {
            factorial = factorial * i;
        }
        cout << "Factorial of " << number << " = " << factorial << "\n";
    }
    return 0;
}
```



```
Factorial of 1 = 1
Factorial of 2 = 2
Factorial of 3 = 6
Factorial of 4 = 24
Factorial of 5 = 120
Factorial of 6 = 720
Factorial of 7 = 5040
Factorial of 8 = 40320
Factorial of 9 = 362880
Factorial of 10 = 3628800

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

- **Example** : Write a program that prints out the following pattern.

```
*  
**  
***  
****  
*****  
*****
```

```
#include <iostream>  
using namespace std;  
  
int main() {  
    for (int i = 1; i <= 5; i++) {  
        for (int j = 1; j <= i; j++) {  
            cout << "*";  
        }  
        cout << endl;  
    }  
    return 0;  
}
```

# Write in a Good Programming Style

- Naming Conventions
- Indentation
- Comments

# Practice Problem

- Write a C++ program to print a pattern of numbers in a pyramid shape. The program should prompt the user to enter the number of rows and then display the pyramid pattern.

Enter the number of rows: 5

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



# Solution

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

int main() {
    int rows;

    cout << "Enter the number of rows: ";
    cin >> rows;

    for (int i = 1; i <= rows; i++) {
        // Print spaces
        for (int j = 1; j <= rows - i; j++) {
            cout << " ";
        }
    }
}
```

# Solution

```
// Print numbers
for (int k = 1; k <= i; k++) {
    cout << i << " ";
}

cout << endl;
}

return 0;
}
```

# Practice Problem

- Write a C++ program to print the Fibonacci series up to a given number using a do-while loop. The program should prompt the user to enter a number and then display the Fibonacci series.

# Solution

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

int main() {
    int number;

    cout << "Enter a number: ";
    cin >> number;

    int previous = 0, current = 1, next;

    cout << "Fibonacci series: ";
    cout << previous << " " << current << " ";
```

# Solution

```
do {  
    next = previous + current;  
    if (next > number)  
        break;  
    cout << next << " ";  
  
    previous = current;  
    current = next;  
} while (true);  
  
cout << endl;  
  
return 0;  
}
```

# Problem

Write a C++ program that simulates a simple calculator. The program should repeatedly prompt the user to enter two numbers and an operator (+, -, \*, or /). It should then perform the corresponding operation on the numbers and display the result. The program should continue running until the user chooses to exit.

# Solution

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

int main() {
    float num1, num2, result;
    char op;

    do {
        cout << "Enter the first number: ";
        cin >> num1;

        cout << "Enter the second number: ";
        cin >> num2;

        cout << "Enter an operator (+, -, *, /) or 'q' to quit: ";
        cin >> op;
```

# Solution

```
// Check if user wants to quit
    if (op == 'q' || op == 'Q') {
        cout << "Exiting the program." << endl;
        break;
    }

    cout << "Enter the second number: ";
    cin >> num2;
```



# Solution

```
switch (op) {  
    case '+':  
        result = num1 + num2;  
        cout << "Result: " << result << endl;  
        break;  
    case '-':  
        result = num1 - num2;  
        cout << "Result: " << result << endl;  
        break;  
    case '*':
```

# Solution

```
result = num1 * num2;
    cout << "Result: " << result << endl;
    break;
case '/':
    if (num2 != 0) {
        result = num1 / num2;
        cout << "Result: " << result << endl;
    } else {
        cout << "Error: Division by zero is
not allowed." << endl;
    }
```

# Solution

```
break;
        default:
            cout << "Error: Invalid operator."
<< endl;
        }
    } while (true);

    return 0;
}
```

# Output

```
Enter the first number: 4
Enter an operator (+, -, *, /) or 'q' to quit: *
Enter the second number: 45
Result: 180
Enter the first number:
```

# Functions

# Divide & Conquer Technique

Experience has shown that the best way to develop and maintain a large program is to **construct it from smaller pieces or modules, each of which is more manageable than the original program.**

- This technique is called **divide and conquer**.
- **A Function** : is a **self-contained block** of statements that perform a **specific task**.
- Using a **function** is something like hiring a person to do a specific job for you.

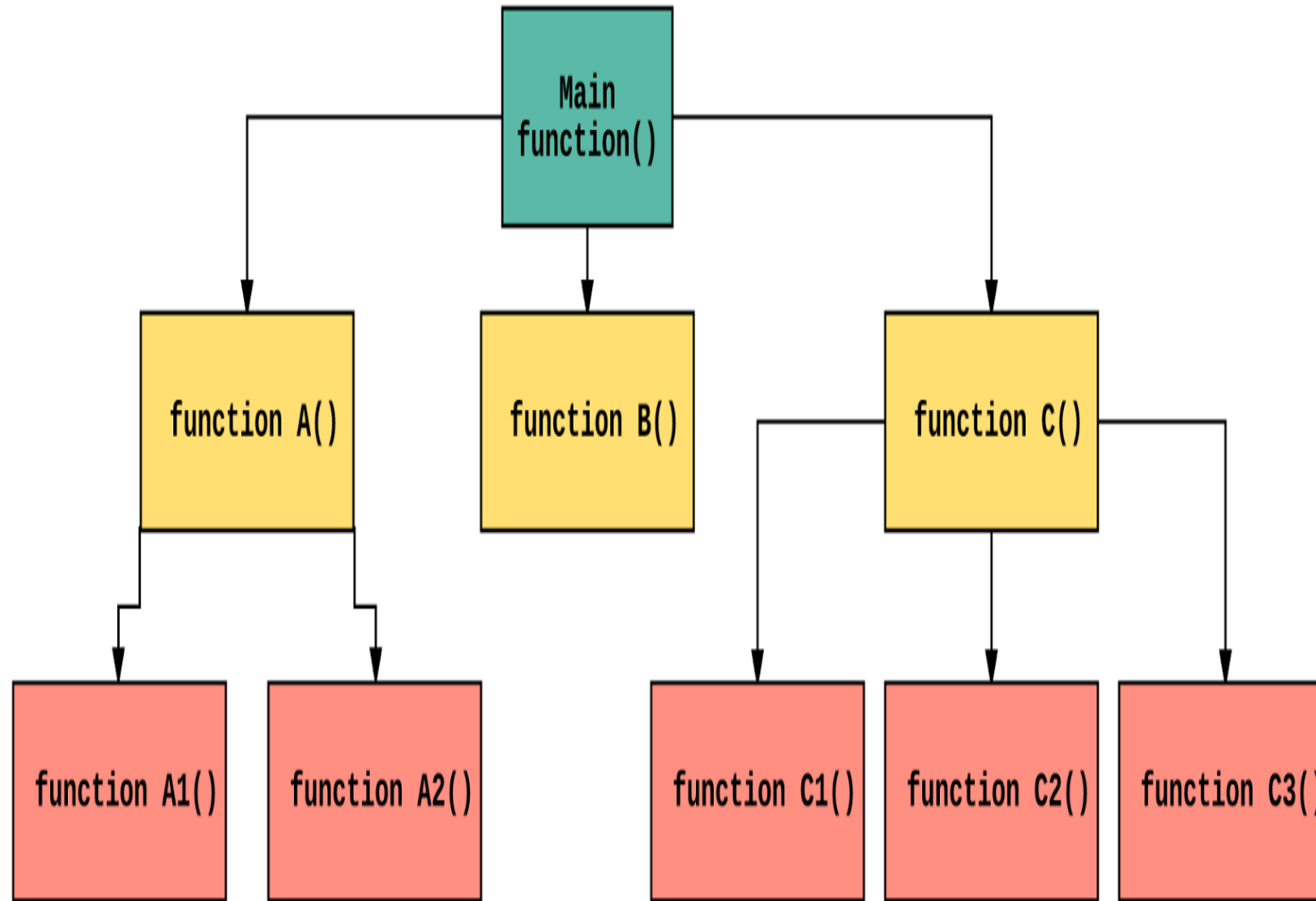
# Why to use Functions?

- **Functions:**
  - Allows to write **more manageable units of code**.
  - **Avoids** rewriting the same code **over and over**.
  - Are **easy** for **maintenance** for programs.

## Important Tips:

- 1- **Don't try to cram the entire logic in one function.** It is a very bad style of programming.
- 2- Instead, **break a program into small units and write functions** for each of these isolated subdivisions.
- 3- Don't hesitate to write functions that are **called only once**.
- 4- Each function should be designed, coded, and tested as a separate unit from the rest of the program. This is the essence of the **top-down design strategy at which you can design functions inside functions**.

# Functional Programming





# Functions Syntax

- C++ comes with libraries of predefined functions that you can use in your programs as **sqrt** included in **cmath** library.
- A function call is an expression consisting of the function name followed by arguments enclosed in parentheses. If there is more than one argument, the arguments are separated by commas.

```
Function_Name(Argument_List)
```

where the *Argument\_List* is a comma-separated list of arguments:

```
Argument_1, Argument_2, . . . , Argument_Last
```

```
side = sqrt(area);  
cout << "2.5 to the power 3.0 is "  
      << pow(2.5, 3.0);
```

# General Functions Syntax

## Function Definition

```
return-value-type function-name( parameter-list )  
{  
  Lines of code to be  
  executed  
  ...  
  ...  
  ...  
  (Body)  
}
```

**Parameter-List** : Is a list of data values supplied from the calling program as input to the function. It is **Optional**

***return-value-type***:  
Is the data type for the returned value from the function to the calling program. It is **Mandatory**, if no returned value expected , we use keyword **Void**.

# Functions Types

## 1. Built in/Pre-defined Functions (Ready Made).

- For Example : Math Library Functions <cmath> Like

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	<i>double</i>	<i>double</i>	sqrt(4.0)	2.0	cmath
pow	powers	<i>double</i>	<i>double</i>	pow(2.0,3.0)	8.0	cmath
abs	absolute value for <i>int</i>	<i>int</i>	<i>int</i>	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for <i>long</i>	<i>long</i>	<i>long</i>	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for <i>double</i>	<i>double</i>	<i>double</i>	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	<i>double</i>	<i>double</i>	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	<i>double</i>	<i>double</i>	floor(3.2) floor(3.9)	3.0 3.0	cmath
srand	Seed random number generator	<i>none</i>	<i>none</i>	srand()	none	cstdlib
rand	Random number	<i>none</i>	<i>int</i>	rand()	0-RAND_MAX	cstdlib

# Pre-defined Function

- To correctly use the built in functions we must know well its **header (prototype)**.
- A function prototype, also known as a **function declaration**, is a statement that describes a **function's name, return type, and parameters without providing the actual function implementation**, such as:

*return-value-type* **function-name**( *parameter-list* )

## Examples for Function Prototypes:

**int abs (int number) ;**

**double pow (double base, double exponent) ;**

**Double floor (double number);**

**Double sqrt (double number);**

# Example: Type Power, Floor, and Ceil of any Number

```
#include <iostream>
#include <cmath>
using namespace std;
int main ( )
{
    int i = -5 ;
    double x = 5.0;
    double y = 2.0;
    double z = 5.21;
```

```
    cout << "The absolute value of i is " << abs (i) << "\n \n";
```

```
    cout << "The power of x to the power y is  " << pow (x,y) << "\n \n";
```

```
    cout << "The Floor for Z is " << floor (z) << "\n \n";
```

```
    cout << "The Ceil for Z is " << ceil (z) << "\n \n";
```

```
}
```

```
The absolute value of i is 5
The power of x to the power y is  25
The Floor for Z is 5
The Ceil for Z is 6

Process returned 0 (0x0)   execution time : 0.408 s
Press any key to continue.
```

# Example

Computes the size of a dog house that can be purchased given the user's budget.

# Solution

```
//Computes the size of a dog house that can be  
purchased  
//given the user's budget.  
#include <iostream>  
#include <cmath>  
using namespace std;  
int main( )  
{
```

# Solution

```
const double COST_PER_SQ_FT = 10.50;
double budget, area, length_side;
cout << "Enter the amount budgeted for your dog house $";
cin >> budget;
area = budget / COST_PER_SQ_FT;
length_side = sqrt(area);
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```



# Solution

```
cout << "For a price of $" << budget << endl  
<< "I can build you a luxurious square dog house\n"  
<< "that is " << length_side  
<< " feet on each side.\n";  
return 0;  
}
```

```
Enter the amount budgeted for your dog house $10000  
For a price of $10000.00  
I can build you a luxurious square dog house  
that is 30.86 feet on each side.  
  
Process returned 0 (0x0)   execution time : 2.500 s  
Press any key to continue.
```

# User Defined Functions

## 2. User Defined Functions ( Tailored by Programmers).

- Working with user defined functions is done in three steps :
  - 1- **Function Declaration (Prototype** , in C++ written before Main ( ) ).
  - 2- **Function Definition (Body**, in C++ written After Main ( ) ) which include the function header which is mentioned in the function declaration and the function body then a return statement at the end of the function which returns the function output.
  - 3- **Invoking Function (Calling**, From The Main ( ) )

# Building Functions in C++:

## 1- Function Declaration (Written before The main() )

*return-value-type* **function-name**( *Formal parameter-list* )

**Formal Parameter-List** : Is a list of input parameters with their data types .

**Int abs ( int x );**

## 2- Function Definition (Written after The main() )

*return-value-type* **function-name**( *Formal parameter-list* )  
{  
  *Lines of code to be executed*  
}

**Actual Parameter-List** : Is a list of input parameters without their data types .

**Cout << abs ( i );**

## 3- Invoking Function ( From The main() )

*Function name* ( *Actual Parameter- List* );

# Another Way of Declaration

- You are not required to list formal parameter names in a function declaration.
- The following two function declarations are equivalent:

```
double total_cost(int number_par, double price_par);
```

and

```
double total_cost(int, double);
```

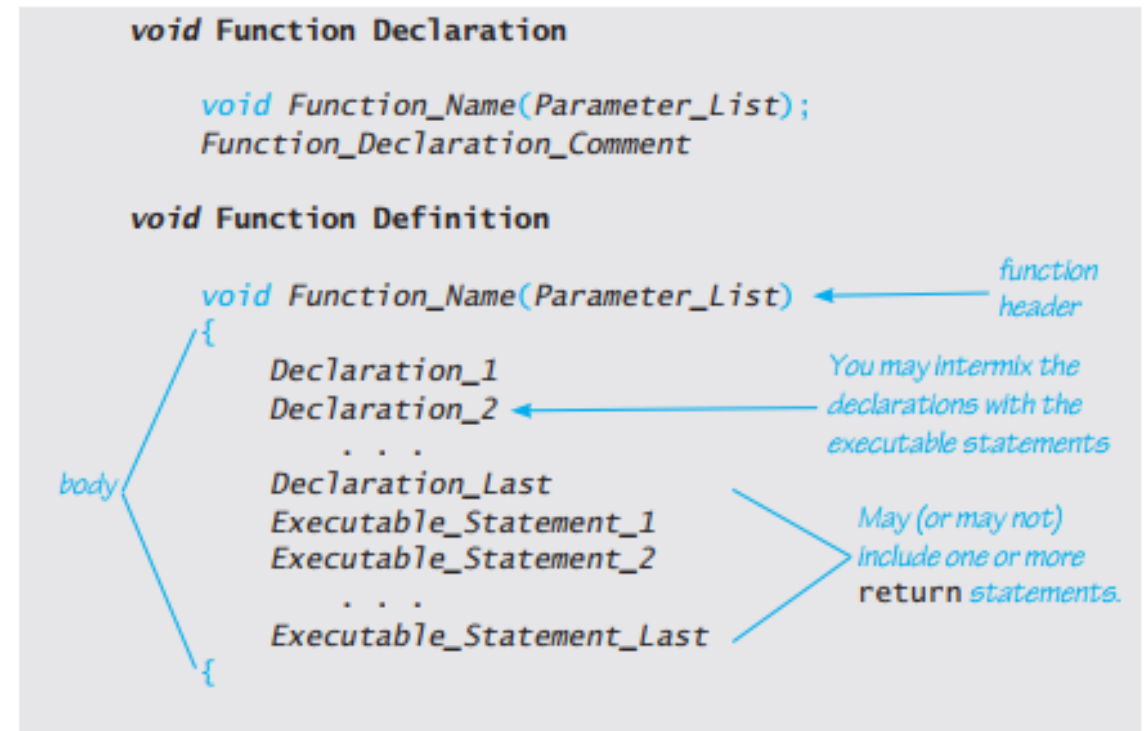
# Types of User Defined Functions:

- **1) Value-returning functions: have a return type**
  - Return a value of a specific data type using the return statement.
  - The returned value can be used in one of the following scenarios:
    - **Save the value for further calculation**  
Int result;  
result = sum ( x , y );
    - **Use the value in some calculation**  
result = sum(x , y) /2;
    - **Print the value**  
Cout << sum(x , y);

# Types of User Defined Functions

## 2) Void functions: do not have a return type

- Do not use a return statement to return a value
- This is a scheme for the void functions:
- Return is optional in void functions.



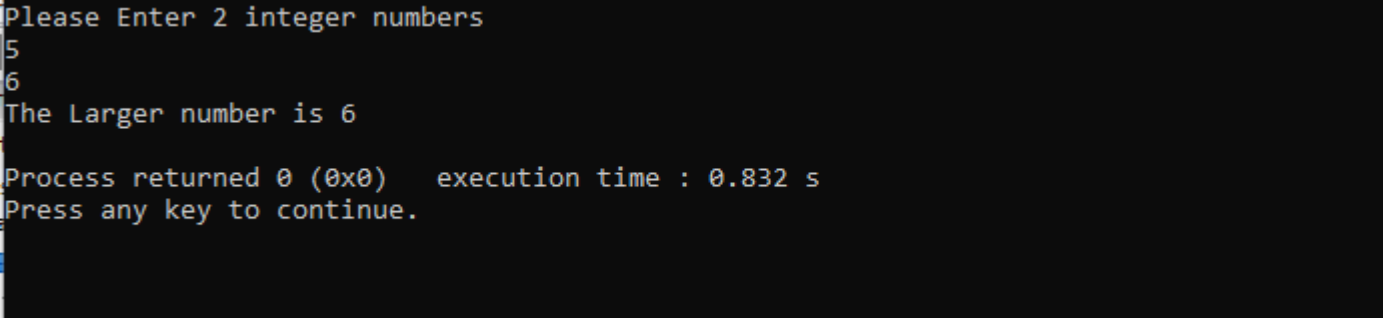
**For Example :** write a program that ask the user to Enter 2 integer numbers and print out the larger of them.

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

int larger(int num1, int num2); // Function prototype

int main() {
    int n1, n2, result;
    cout << "Please Enter 2 integer numbers" << '\n';
    cin >> n1 >> n2;
    result = larger(n1, n2);
    cout << "The Larger number is " << result << "\n";
    return 0;
}

int larger(int num1, int num2) {
    int max;
    if (num1 >= num2)
        max = num1;
    else
        max = num2;
    return max;
}
```



```
Please Enter 2 integer numbers
5
6
The Larger number is 6

Process returned 0 (0x0)   execution time : 0.832 s
Press any key to continue.
```

**For Example:** Write a program to calculate the Area and volume for a sphere:

Note that:

1. The area of sphere =  $4 * \text{PI} * \text{Radius} * \text{Radius}$ .
2. The Volume of sphere =  $\frac{4}{3} * \text{PI} * \text{Radius} * \text{Radius} * \text{Radius}$ .

**Note : PI = 3.14**



# Solution

```
#include <iostream>

using namespace std;

// Function declarations (prototypes)
double calculateSphereArea(double radius);
double calculateSphereVolume(double radius);

int main() {
    double radius;
    cout << "Enter the radius of the sphere: ";
    cin >> radius;

    // Function invocations
    double area = calculateSphereArea(radius);
    double volume = calculateSphereVolume(radius);
```

# Solution

```
cout << "Area of the sphere: " << area << endl;
    cout << "Volume of the sphere: " << volume << endl;
    return 0;
}
```

```
// Function definitions
```

```
double calculateSphereArea(double radius) {
    const double PI = 3.14;
    return 4 * PI * radius * radius;
}
```

```
double calculateSphereVolume(double radius) {
    const double PI = 3.14;
    return (4.0 / 3) * PI * radius * radius * radius;}
}
```

# Output

```
Enter the radius of the sphere: 4  
Area of the sphere: 200.96  
Volume of the sphere: 267.947  
  
Process returned 0 (0x0)   execution time : 1.806 s  
Press any key to continue.
```

For Example : write a program that ask the user to Enter 3 integer numbers and print out their sum and Average. int sum (int num1 , int num2, int num3);

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

```
int sum(int num1, int num2, int num3 = 90);
```

```
int main() {
    int n1, n2, n3;
    cout << "Please Enter 3 integer numbers \n";
    cin >> n1 >> n2 >> n3;
    cout << "The sum of the 3 numbers is " << sum(n1,
n2, n3) << "\n";
    return 0;
}
```

```
int sum(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}
```

```
Please Enter 3 integer numbers
1
2
3
The sum of the 3 numbers is 6

Process returned 0 (0x0)   execution time : 1.838 s
Press any key to continue.
```

# Function Parameter's Default Value

- Function parameters with default values are parameters that are assigned a predefined value in the function declaration.
- These default values are used when the corresponding argument is not provided during the function call.
- In other words, default parameter values allow you to make certain function parameters optional.

# Example:

Write a program that calculates the area of a rectangle. The program should prompt the user to enter the length and width of the rectangle. The width can be optionally provided, with a default value of 5 if not entered by the user. The program should then calculate and display the area of the rectangle. The program should properly handle cases where the user does not enter a value for the width (by pressing enter) and use the default value in that case.

Note: You can assume that the user will input valid numerical values for the length and width.

# Solution

```
#include <iostream>
using namespace std;
// Function declaration
int calculateRectangleArea(int length, int width = 5);
int main() {
    int length, width;
    cout << "Enter the length of the rectangle: ";
    cin >> length;
    cout << "Enter the width of the rectangle (or press enter to use default
value which is 5): ";
    cin.ignore(); // Ignore the newline character in the input buffer
    if (cin.peek() == '\n') {
        width = 5; // Use the default value if the user presses enter without
entering a width
    } else {
        cin >> width;
    }
    int area = calculateRectangleArea(length, width);
    cout << "Area of the rectangle: " << area << endl;
    return 0;
}
// Function definition
int calculateRectangleArea(int length, int width) {
    return length * width;}
```

# Output

```
Enter the length of the rectangle: 3
Enter the width of the rectangle (or press enter to use default value):
Area of the rectangle: 15

Process returned 0 (0x0)   execution time : 8.533 s
Press any key to continue.
```



# Function Return Boolean Value

In C++, a function can return a Boolean value, which is a value that can be either true or false. The return type of such a function is specified as `bool`. The function performs some logic or computation and returns a Boolean value based on the result of that logic.

# Example

Write a program that determines whether a given number is even or odd. The program should prompt the user to enter a number, and based on the input, it should display a message indicating whether the number is even or odd. Include a Boolean function as a method to determine odd and even numbers.

# Example:

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

bool isEven(int num) {
    if (num % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

```
int main() {  
    int number;  
    cout << "Enter a number: ";  
    cin >> number;  
  
    if (isEven(number)) {  
        cout << number << " is even." << endl;  
    } else {  
        cout << number << " is odd." << endl;  
    }  
  
    return 0;  
}
```


# Scope of Variable

Scope is the context within a program in which a variable is valid and can be used.

- **Local variable**: declared within a function (or block)
- Local variables can be **accessible only within the function** or from declaration to the end of the block.

```
int sum (int x , int y )  
{  
    int result ;  
  
    result = x + y;  
    return result;  
}
```

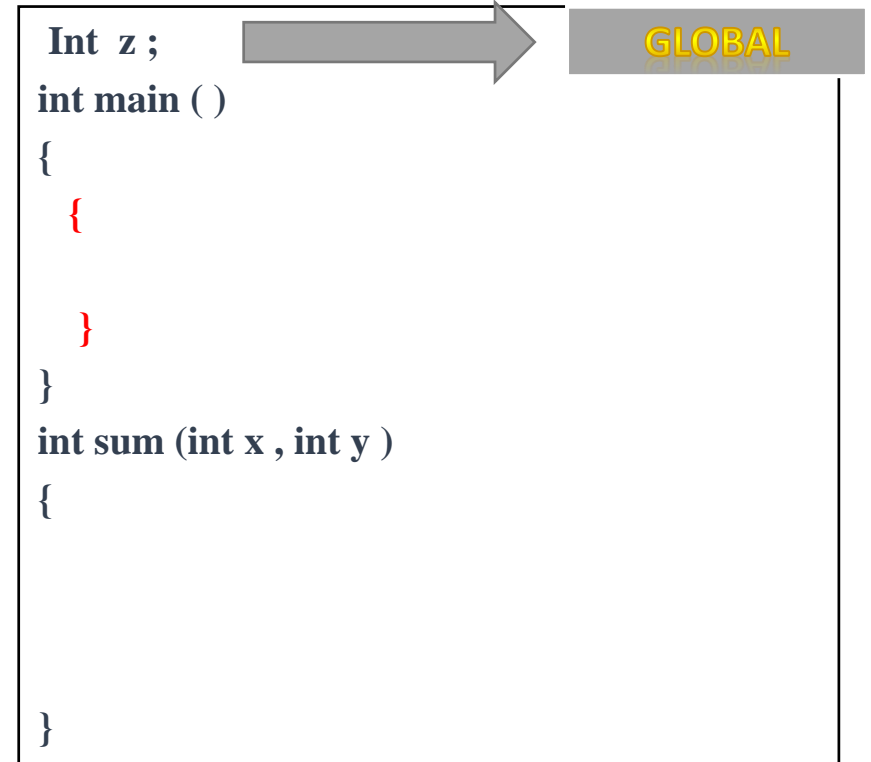
```
int main ()  
{  
    {  
        Int x ;  
    }  
}
```



# Scope of Variable

- Global variable: declared outside of every function definition.

Can be accessed from any function that has no local variables with the same name. In case the function has a local variable with the same name as the global variable ,



The diagram illustrates the scope of a global variable. It shows a code block with the following content:

```
Int z ;  
int main ()  
{  
    {  
    }  
}  
int sum (int x , int y )  
{  
  
}
```

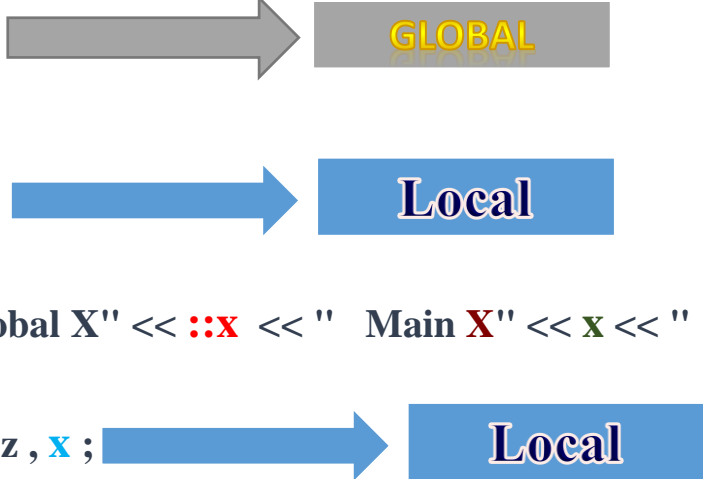
A grey arrow points from the declaration `Int z ;` to a grey box labeled **GLOBAL**, indicating that the variable `z` is globally accessible throughout the program.

# More on Global Variables

- Using global variables causes **side effects**
  - A function that uses global variables is **not independent**.
  - Sometimes more than one function uses the same global variable and something goes **wrong**!
  - It is difficult to find what went wrong and where problems caused in one area of the program may appear to be from another area.
  - **To prevent the global variable to be modified use the const Keyword.**

**i.e const float pi = 3.14;**

```
int x = 100 ;  
int main ( )  
{  
  int x = 10;  
  int y = 50;  
  cout << " Global X" << ::x << " Main X" << x << " " << y;  
  {  
    int z , x ;  
    z=100;  
    y=100;  
    x= 250;  
    cout <<" inner block " << x ;  
  }  
}
```





# Call by Value

- Call by value is the default behavior in C++ for passing arguments to a function.
- When arguments are passed by value, a **copy of the value is made and passed to the function.**
- Any modifications made to the parameters inside the function **do not affect the original values of the arguments passed.**
- **Changes** made to the parameters inside the function are **local to that function** and do not affect the variables in the calling code.
- It is useful when you want to work with the values without modifying the original variables.

# Call by Reference

- To make a formal parameter a **call-by-reference parameter, append the ampersand sign & to its type name.**
- The corresponding argument in a call to the function should then be a **variable, not a constant or other expression.**
- When the function is called, the corresponding **variable argument (not its value) will be substituted for the formal parameter.** Any **change** made to the formal parameter in the function body will be **made to the argument variable** when the function is called.

# Call by Reference

- Call by reference allows the function to **directly access and modify the original variables passed as arguments.**
- When arguments are passed by reference, the **function receives a reference (or a memory address) to the original variable.**
- Any **modifications** made to the parameters inside the function **affect the original variables in the calling code.**
- To pass arguments by reference, you need to **use references or pointers as the function parameters.**

# Example Value vs Reference

Try to calculate the square of any number in a simple  $x*x$  call by value and call by reference function and show the difference.

# Solution

```
#include <iostream>
using namespace std;
void squareByValue(int x) {
    x = x * x;}
void squareByReference(int& x) {
    x = x * x;}
int main() {
    int num1 = 5;
    int num2 = 5;
    squareByValue(num1);      // Call by value
    squareByReference(num2);  // Call by reference
    cout << "num1: " << num1 << endl;    // Output: num1: 5 (Unchanged)
    cout << "num2: " << num2 << endl;    // Output: num2: 25 (Modified)
    return 0;}
```

# Output

```
num1: 5  
num2: 25  
  
Process returned 0 (0x0)   execution time : 0.062 s  
Press any key to continue.
```

# Parameters and Arguments

1. The **formal parameters** for a function are **listed in the function declaration and are used in the body of the function definition.** A formal parameter (of any sort) is a kind of blank or place holder that is filled in with something when the function is called.
2. An **argument** is something that is used to **fill in a formal parameter.** When you write down a function call, the arguments are listed in parentheses after the function name. When the function call is executed, the arguments are “plugged in” for the formal parameters.

# Parameters and Arguments

3. The terms *call-by-value* and *call-by-reference* refer to the mechanism that is used in the “plugging in” process.

- In the **call-by-value** method, only the value of the argument is used.
- In this **call-by-value** mechanism, the **formal parameter is a local variable that is initialized to the value of the corresponding argument**.
- In the **call-by-reference** mechanism, the argument is a variable and the entire variable is used.
- In the **call-by-reference** mechanism, the **argument variable is substituted for the formal parameter** so that any change that is made to the formal parameter is actually made to the argument variable.



# Functions Calling Functions

A function body may contain a call to another function. The situation for these sorts of function calls is exactly the same as it would be if the function call had occurred in the main function of the program; the only restriction is that the function declaration should appear before the function is used.

# Overloading Function Names

If you have two or more function definitions for the same function name, that is called **overloading**.

When you overload a function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types.

When there is a function call, the compiler uses the function definition whose number of formal parameters and types of formal parameters match the arguments in the function call.

# Pre-condition & Post Condition

The **precondition** states what is assumed to be true when the function is called. The function should not be used and cannot be expected to perform correctly unless the precondition holds.

The **Post Condition** describes the effect of the function call; that is, the post condition tells what will be true after the function is executed in a situation in which the precondition holds.

# Abstraction of Functions

- When applied to a function definition, the principle of procedural abstraction means that your function should be written so that it can be used like a black box. This means that the programmer who uses the function should not need to look at the body of the function definition.

# Functional Programming Testing & Debugging

- Keep an Open Mind
- Localize the Error
- Check Common Errors are:
  - (1) uninitialized variables
  - (2) off-by-one errors
  - (3) exceeding a data boundary
  - (4) automatic type conversion
  - (5) using = instead of ==.

# A Tip of using Loops & Functions

- Whenever you have a loop nested within a loop, or any other complex computation included in a loop body, make the loop body a function call.
- This way you can separate the design of the loop body from the design of the rest of the program.

# Practice Problem – Try it!

Write a program that takes a positive integer as input from the user and performs the following tasks:

1. Function `isPrime()`: Implement a function that checks whether a number is prime or not. The function should return `true` if the number is prime, and `false` otherwise.
2. Function `reverseDigits()`: Implement a function that reverses the digits of a given number. For example, if the input is 12345, the function should return 54321.
3. Function `countDigits()`: Implement a function that counts the number of digits in a given number.
4. In the `main()` function, prompt the user to enter a positive integer. Perform the following tasks:  
Check if the number is prime using the `isPrime()` function. Display an appropriate message.  
Reverse the digits of the number using the `reverseDigits()` function. Display the reversed number.  
Count the number of digits in the number using the `countDigits()` function. Display the count.

# Output

```
Enter a positive integer: 234
234 is not a prime number.
Reversed number: 432
Number of digits: 3

Process returned 0 (0x0)   execution time : 0.801 s
Press any key to continue.
```



# Solution

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

bool isPrime(int number) {
    if (number <= 1)
        return false;
```

# Solution

```
for (int i = 2; i * i <= number; i++) {  
    if (number % i == 0)  
        return false;  
}
```

```
    return true;  
}
```

```
int reverseDigits(int number) {  
    int reversedNumber = 0;
```

# Solution

```
while (number != 0) {  
    int remainder = number % 10;  
    reversedNumber = reversedNumber * 10 +  
remainder;  
    number /= 10;  
}  
  
return reversedNumber;  
}
```

# Solution

```
int countDigits(int number) {  
    int digitCount = 0;  
  
    do {  
        digitCount++;  
        number /= 10;  
    } while (number != 0);  
  
    return digitCount;  
}
```

# Solution

```
int main() {  
    int number;  
    cout << "Enter a positive integer: ";  
    cin >> number;  
  
    // Check if the number is prime  
    if (isPrime(number))  
        cout << number << " is a prime number." << endl;  
    else
```

# Solution

```
cout << number << " is not a prime number." << endl;

// Reverse the digits of the number
int reversed = reverseDigits(number);
cout << "Reversed number: " << reversed << endl;

// Count the number of digits in the number
int digitCount = countDigits(number);
cout << "Number of digits: " << digitCount << endl;

return 0;
}
```

# Module 1 Assignment – Part A – Problem 1

- Compare Advantages and Disadvantages of IoT.
- Define what's the Gateway and explains its role in IoT layers.
- Compare fog computing to cloud computing, then comment what is better for the IoT Application?
- State the difference between HTTP and MQTT in IoT systems and which is preferred in most cases?
- Mention the three main kinds of program errors and compare them to each others.
- State difference between source code and object code.
- What is enum data type is used for?

# Module 1 Assignment – Part A – Problem 2

- Write a program that allows the user to enter a time in seconds and then outputs how far an object would drop if it is in freefall for that length of time. Assume that the object starts at rest, there is no friction or resistance from air, and there is a constant acceleration of 32 feet per second due to gravity. Use the equation:

$$\text{distance} = \frac{\text{acceleration} \times \text{time}^2}{2}$$

- NB: You should first compute the product and then divide the result by 2



# Module 1 Assignment – Part A – Problem 3

Write a program that inputs a character from the keyboard and then outputs a large block letter “C” composed of that character. For example, if the user inputs the character “X,” then the output should look as follows:

```
      X X X
     X   X
    X
    X
    X
    X
    X
   X     X
  X X X
```

# Module 1 Assignment – Part A – Problem 3

Sound travels through air as a result of collisions between the molecules in the air. The temperature of the air affects the speed of the molecules, which in turn affects the speed of sound. The velocity of sound in dry air can be approximated by the formula:

$$\text{velocity} \approx 331.3 + 0.61 \times T_c$$

where  $T_c$  is the temperature of the air in degrees Celsius and the velocity is in meters/second.

Write a program that allows the user to input a starting and an ending temperature. Within this temperature range, the program should output the temperature and the corresponding velocity in 1° increments. For

example, if the user entered 0 as the start temperature and 2 as the end temperature, then the program should output as the following:

At 0 degrees Celsius the velocity of sound is 331.3 m/s

At 1 degrees Celsius the velocity of sound is 331.9 m/s

At 2 degrees Celsius the velocity of sound is 332.5 m/s

# Module 1 Assignment – Part A – Problem 4

Write a program to compute the interest due, total amount due, and the minimum payment for a revolving credit account. The program accepts the account balance as input, then adds on the interest to get the total amount due. The rate schedules are the following: The interest is 1.5 percent on the first \$1,000 and 1 percent on any amount over that. The minimum payment is the total amount due if that is \$10 or less; otherwise, it is \$10 or 10 percent of the total amount owed, whichever is larger. Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.

# Module 1 Assignment – Part A – Problem 5

Write a program that takes a positive integer as input from the user and performs the following tasks:

1. Function `isPerfectSquare()`: Implement a function that checks whether a number is a perfect square. The function should return `true` if the number is a perfect square, and `false` otherwise.
2. Function `reverseDigits()`: Implement a function that reverses the digits of a given number. For example, if the input is 12345, the function should return 54321.
3. Function `calculateSum()`: Implement a function that calculates the sum of all digits in a given number.
4. In the `main()` function, prompt the user to enter a positive integer. Perform the following tasks:
5. Check if the number is a perfect square using the `isPerfectSquare()` function. Display an appropriate message.
6. Reverse the digits of the number using the `reverseDigits()` function. Display the reversed number.
7. Calculate the sum of all digits in the number using the `calculateSum()` function. Display the sum.

# Module 1 Assignment – Part A – Problem 6

A liter is 0.264179 gallons. Write a program that will read in the number of liters of gasoline consumed by the user's car and the number of miles traveled by the car and will then output the number of miles per gallon the car delivered. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the number of miles per gallon. Your program should use a globally defined constant for the number of liters per gallon.

# Module 1 Assignment – Part A – Problem 7

The gravitational attractive force between two bodies with masses  $m_1$  and  $m_2$  separated by a distance  $d$  is given by:

$$F = \frac{Gm_1m_2}{d^2}$$

where  $G$  is the universal gravitational constant:

$$G = 6.673 \times 10^{-8} \left( \frac{cm^3}{g \times sec^2} \right)$$

Write a function definition that takes arguments for the masses of two bodies and the distance between them and that returns the gravitational force. Since you will use the preceding formula, the gravitational force will be in dynes. One dyne equals

$$\left( \frac{g \times cm}{sec^2} \right)$$

You should use a globally defined constant for the universal gravitational constant. Embed your function definition in a complete program that computes the gravitational force between two objects given suitable inputs. Your program should allow the user to repeat this calculation as often as the user wishes.

# Module 1 Assignment – Part A – Bonus

The large “economy” size of an item is not always a better buy than the smaller size. This is particularly true when buying pizzas. Pizza sizes are given as the diameter of the pizza in inches. However, the quantity of pizza is determined by the area of the pizza, and the area is not proportional to the diameter. Most people cannot easily estimate the difference in area between a 10-inch pizza and a 12-inch pizza and so cannot easily determine which size is the best buy—that is, which size has the lowest price per square inch. In this case study we will design a program that compares two sizes of pizza to determine which is the better buy?

# Module 1 Assignment – Part A – Bonus

## **Input:**

The input will consist of the diameter in inches and the price for each of two sizes of pizza.

## **Output:**

The output will give the cost per square inch for each of the two sizes of pizza and will tell which is the better buy, that is, which has the lowest cost per square inch. (If they are the same cost per square inch, we will consider the smaller one to be the better buy.)



# Module 1 Assignment – Part A – Bonus

## **Analysis of the Problem**

We will use top-down design to divide the task to be solved by our program

into the following subtasks:

**Subtask 1:** Get the input data for both the small and large pizzas.

**Subtask 2:** Compute the price per square inch for the small pizza.

**Subtask 3:** Compute the price per square inch for the large pizza.

**Subtask 4:** Determine which is the better buy.

**Subtask 5:** output the results.

# Module 1 Assignment – Part A – Bonus

## **Note about sub-task 2 & 3:**

1. They are exactly the same task. The only difference is that they use different data to do the computation. The only things that change between subtask 2 and subtask 3 are the size of the pizza and its price.
2. The result of subtask 2 and the result of subtask 3 are each a single value:  
the price per square inch of the pizza.

# Module 1 Assignment – Part A – Double Bonus

- Write the algorithm and C++ code for bubble sorting using functional programming only – Double Bonus

Thank you