**C++ Concepts:**

**Introduction:**

* **C++ is a computer language used to make programs.**
* It helps you build **fast**, **strong**, and **well-organized** software like games, apps, and tools.

**Difference between C and C++**

C++ was developed as an extension of [C](https://www.w3schools.com/c/index.php), and both languages have almost the same syntax.

The main difference between C and C++ is that C++ supports classes and objects, while C does not.

**Tool:**

DevC++

**C++ Statements:**

Programming instructions is called statements and these instructions can be executed by computer.

**Comments:**

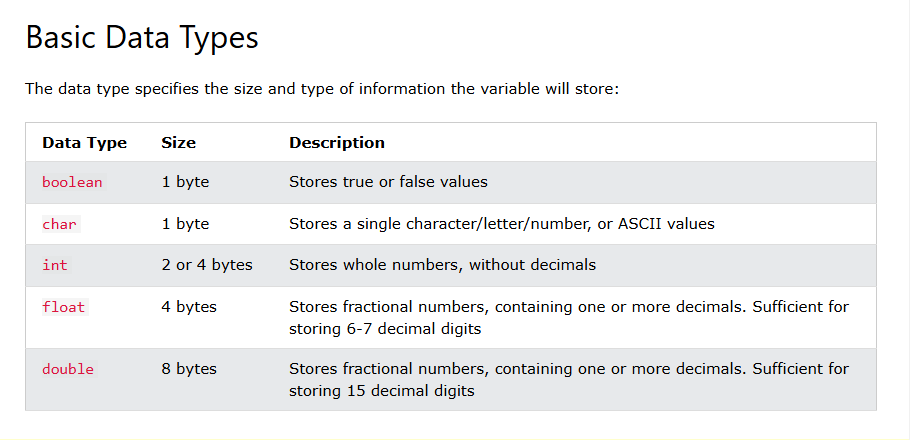
Single line comment: // for SLC

Multi-line comments: /\* \*/ for MLC

**Variables:**

Variables are containers for storing data values.

* int - stores integers (whole numbers), without decimals, such as 123 or -123
* double - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* string - stores text, such as "Hello World". String values are surrounded by double quotes
* bool - stores values with two states: true or false



**C++ Identifiers**

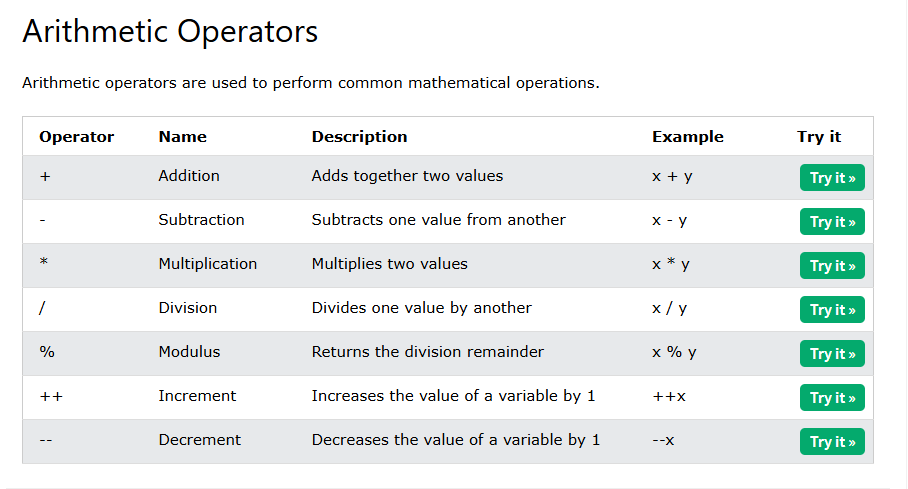
All C++ variables must be identified with unique names.

These unique names are **called identifiers.**

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

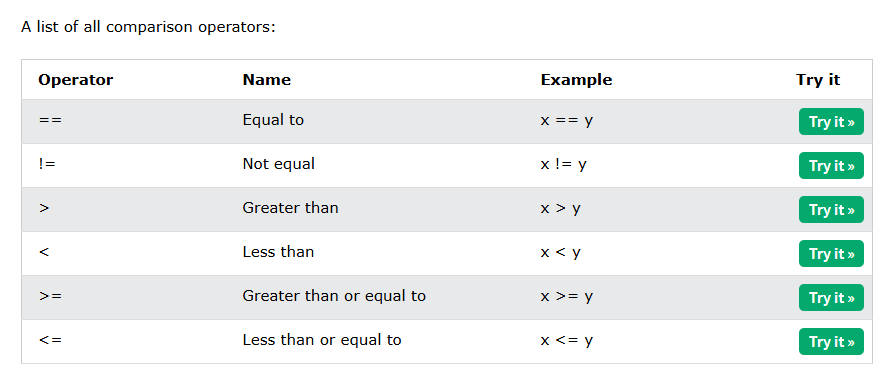
## Constants

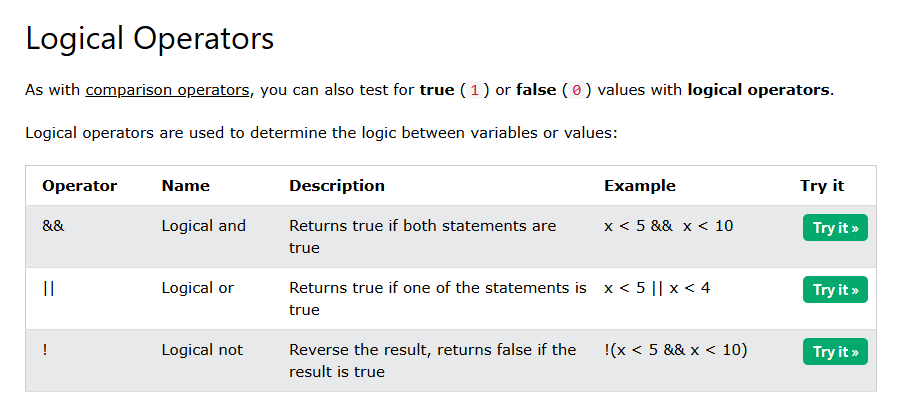
When you do not want others (or yourself) to change existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):



Assignment operator that is used for assign value to the variable.

e.g.: x=3





**String:**

Strings are used for storing text/characters.

For example, "Hello World" is a string.

**Concatenation:**

Used to add two strings (+)

**Append:**

Also used to add two strings

**String length:**

Used to find out the length of the string

**Getline function:**

when working with strings, we often use the getline() function to read a line of text.

**C++ if-else:**

If

else

If-else

Else-if

Ternary operator

int time = 20;  
string result = (time < 18) ? "Good day." : "Good evening.";  
cout << result;

**Loops in C++:**

**While-loop** (The while loop loops through a block of code as long as a specified condition is true)

**Do/while loop** (executed once before checking condition)

**for-loop** (When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop)

**nested loops** (loop inside another loop)

**foreach loop** ( foreach loop is a simple way to go through every item in a list or array, one by one.)

The break statement can also be used to jump out of a loop.

|  |  |
| --- | --- |
| **Break** | **Continue** |
| **Stops** the loop **completely** and jumps **out** of it. | **Skips** the **current step** and moves to the **next one** in the loop. |
| for (int i = 1; i <= 5; i++) {  if (i == 3) {  break; // Loop stops completely when i is 3  }  cout << i << endl;  }  // Output: 1 2 | for (int i = 1; i <= 5; i++) {  if (i == 3) {  continue; // Skip when i is 3  }  cout << i << endl;  }  // Output: 1 2 4 5 |

## C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

string cars[4];

**Fixed vs. dynamic array:**

## ****Fixed Array****

* **Size is fixed** at the time of creation.
* You **cannot change** the size later.
* Common in **C, C++**, and other low-level languages.

### 💡 Example in C:

int arr[5]; // Can hold only 5 elements

## ****Dynamic Array****

* **Size can grow or shrink** during runtime.
* Common in **JavaScript, Python**, etc.
* Easy to **add or remove** elements.

### 💡 Example in JavaScript:

Int size;

Cout<<”Enter size of array;

Cinn>>size;

int\* arr = new int[size]; // creates a dynamic array of runtime size declaration

## C++ Structures

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an [array](https://www.w3schools.com/cpp/cpp_arrays.asp), a structure can contain many different data types (int, string, bool, etc.).

**In short:**

The purpose of a structure is to **group related variables of different types** under one name, so we can manage complex data more easily and clearly.

We need it to:

* **Organize related data together**
* **Avoid using many separate variables**
* **Make code easier to read and maintain**

**C++ Reference:**

A reference variable is a "reference" to an existing variable, and it is created with the & operator:

string food = "Pizza";  // food variable  
string &meal = food;    // reference to food

Result: Now either access food or meal output remain same as “pizza”.

## Memory Address:

In the example from the previous page, the & operator was used to create a reference variable. But it can also be used to get the memory address of a variable; which is the location of where the variable is stored on the computer.

When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

To access it, use the & operator, and the result will represent where the variable is stored:

string food = "Pizza";  
  
cout << &food;

**in short:** when we use & operator with variable name then in this way we access the address of the variable where that variable store in the computer.

**C++ Pointers:**

Creating Pointers You learned from the previous chapter, that we can get the memory address of a variable by using the & operator:

string food = "Pizza"; // A food variable of type string  
  
cout << food;  // Outputs the value of food (Pizza)  
cout << &food; // Outputs the memory address of food (**0x6dfed4**)

A **pointer** however, is a variable that **stores the memory address as its value**.

A pointer variable points to a data type (like int or string) of the same type, and is created with the \* operator. The address of the variable you're working with is assigned to the pointer:

### Example

string food = "Pizza";  // A food variable of type string  
**string\* ptr = &food;**    // A pointer variable, with the name ptr, that stores the address of food  
  
// Output the value of food (Pizza)  
cout << food << "\n";  
  
// Output the memory address of food (0x6dfed4)  
cout << &food << "\n";  
  
// Output the memory address of food with the pointer (0x6dfed4)  
cout << ptr << "\n";

|  |  |  |  |
| --- | --- | --- | --- |
| Array | Structure | Pointers | References |
| Store multiple data but having same data type | A **structure** lets you group **different types of related data** under one name. | A **pointer** is a variable that holds the **address of another variable**, allowing indirect access to its value. | A **reference** is an **alias** (second name) for a variable that lets you access or modify it **directly** without copying. |
| int num[4]={10,20,30,40} | Struct person{  String name;  Int age;  Float marks;  }  /\* Now create a student \*/  Student s1;  s1.name = "Ali";  s1.age = 20;  s1.marks = 88.5; | int a = 10;  int\* p = &a;   a is a normal variable that stores the value 10.   &a means “**address of** a”.   p is a pointer that **stores the address** of a. | int a = 10;  int& ref = a;   a is a variable.   ref is a **reference** to a. It’s another name for a.  ref = 20;  cout << a; // Output will be 20  Because ref and a are the **same variable**. |

# C++ Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

|  |  |
| --- | --- |
| Parameters | Arguments |
| Information can be passed to functions as a parameter. Parameters act as variables inside the function.  Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma: Syntax void functionName(parameter1, parameter2, parameter3) {   // code to be executed } | void myFunction(**string fname**) {   cout << fname << " Refsnes\n"; }  int main() {   myFunction(**"Liam"**);   myFunction(**"Jenny"**);   myFunction(**"Anja"**);   return 0; }  // Liam Refsnes // Jenny Refsnes // Anja Refsnes are arguments |

**Default parameters value:**

When no argument pass when function is called then it set to the default value.

**Multiple paraenters:**

When more than one parameters in the function.

## Return Values

The void keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int, string, etc.) instead of void, and use the return keyword inside the function:

**int** myFunction(int x) {  
  **return** 5 + x;  
}  
  
int main() {  
  cout << myFunction(3);  
  return 0;  
}  
  
// Outputs 8 (5 + 3)

**Pass by Reference:**

# ****Pass by Value vs. Pass by Reference****

### ****1. Pass by Value****

* A **copy** of the variable is sent to the function.
* Changes **do not affect** the original variable.

### Example (C++):

void change(int a) {

a = 100;

}

int main() {

int x = 10;

change(x);

cout << x; // Output: 10 (original not changed)

}

### ****2. Pass by Reference****

* The function gets the **original variable**, not a copy.
* Changes **affect the original** value.

### Example (C++):

void change(int& a) {

a = 100;

}

int main() {

int x = 10;

change(x);

cout << x; // Output: 100 (original is changed)

}

|  |  |
| --- | --- |
| Function Overloading | Function overriding |
| With**function overloading**, multiple functions can have the same name with different parameters: Example int myFunction(int x) float myFunction(float x) double myFunction(double x, double y) | Imagine a base class called **Teacher** with a function called introduce().  Then a child class called **MathTeacher** overrides that function to introduce themselves in a more specific way.  #include <iostream>  using namespace std;  // Base class  class Teacher {  public:  void introduce() {  cout << "I am a teacher." << endl;  }  };  // Derived class  class MathTeacher : public Teacher {  public:  void introduce() {  cout << "I am a math teacher." << endl;  }  };  int main() {  Teacher t;  MathTeacher mt;  t.introduce(); // Output: I am a teacher.  mt.introduce(); // Output: I am a math teacher.  return 0;  } |

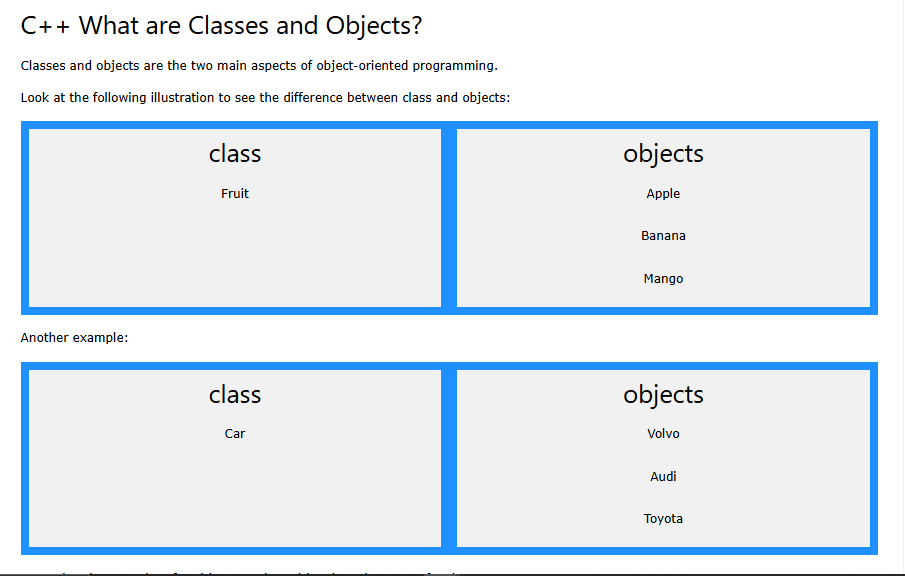
|  |  |
| --- | --- |
| **Local variable** | **Global variable** |
| A variable created inside a function belongs to the local scope of that function, and can only be used inside that function | A variable created outside of a function, is called a **global variable** and belongs to the global scope. |

**C++ OOP**

OOP stands for Object-Oriented Programming.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time



**Class**: Blueprint or template for creating objects; defines data and behavior.

**Object**: Real-world instance of a class; uses the class's structure.

**Class Methods**

Methods are **functions** that belongs to the class.

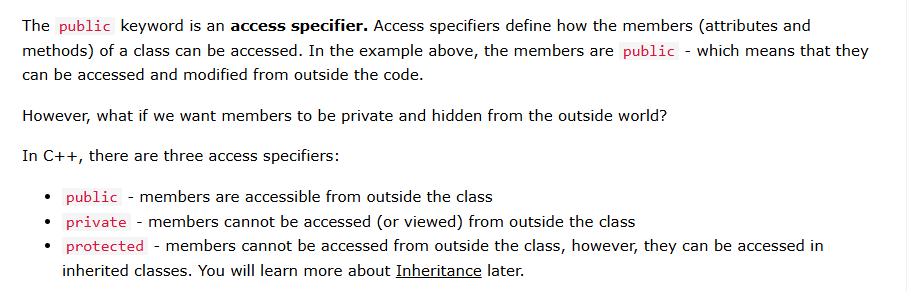
There are two ways to define functions that belongs to a class:

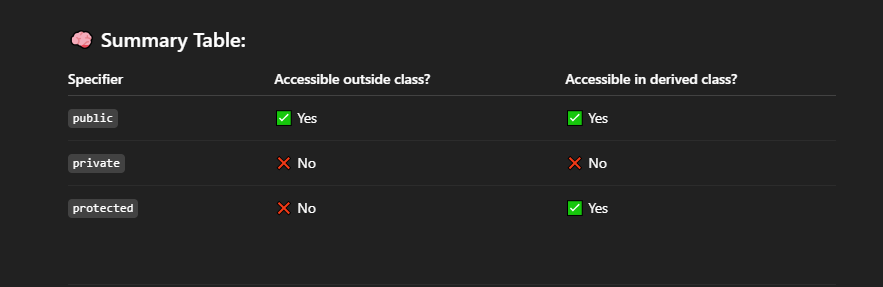
* Inside class definition(Function is defined directly in the class body (implicitly inline).)
* Outside class definition(Function is declared in the class, defined later using ClassName::FunctionName() syntax.)

# C++ Constructors

A **constructor** is a **special function** inside a class that **automatically runs** when you create (make) an object.

## Access Specifiers





**The 4 main pillars of OOP (Object-Oriented Programming) are:**

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

# Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

**Abstraction**

**????**

# Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **derived class** (child) - the class that inherits from another class
* **base class** (parent) - the class being inherited from

**Inheritance** is an object-oriented programming feature in C++ that allows a **class (child or derived class)** to acquire the **properties and behaviors (data and functions)** of another class (parent or base class).  
It promotes **code reusability** and supports **hierarchical relationships** between classes.

**In short:** A class (child or derived class) that can inherit features from another class (base class or parent).

**Types of inheritance**:

### 1. ****Single Inheritance****

One child class inherits from **one parent** class.

class Parent {};

class Child : public Parent {};

### 2. ****Multiple Inheritance****

One child class inherits from **more than one parent** class.

class A {};

class B {};

class C : public A, public B {};

### 3. ****Multilevel Inheritance****

A class inherits from a class which is already derived from another class.

class A {};

class B : public A {};

class C : public B {};

### 4. ****Hierarchical Inheritance****

**Multiple child classes** inherit from **one parent** class.

class Parent {};

class Child1 : public Parent {};

class Child2 : public Parent {};

### 5. ****Hybrid Inheritance****

A mix of **two or more** types of inheritance (like multiple + multilevel).

class A {};

class B : public A {};

class C {};

class D : public B, public C {};

## Polymorphism

**Polymorphism** means "**many forms**" — the ability of a function or object to behave **differently** based on the situation.

**Polymorphism** is a key feature of object-oriented programming in C++ that means **"many forms."** It allows the **same function name or operator** to behave **differently based on the context** (i.e., the type or number of arguments, or the object calling it).

Polymorphism increases **flexibility and reusability** of code.

// Base class  
class Animal {  
  public:  
    void animalSound() {  
      cout << "The animal makes a sound \n";  
    }  
};  
  
// Derived class  
class Pig : public Animal {  
  public:  
    void animalSound() {  
      cout << "The pig says: wee wee \n";  
    }  
};  
  
// Derived class  
class Dog : public Animal {  
  public:  
    void animalSound() {  
      cout << "The dog says: bow wow \n";  
    }  
};

Pig myPig;

Dog myDog;

myPig.animalSound(); // Output: The pig says: wee wee

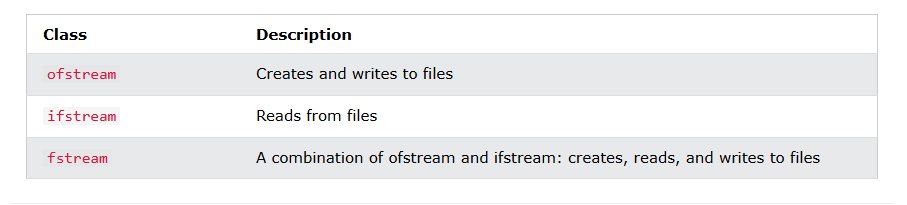
myDog.animalSound(); // Output: The dog says: bow wow

## C++ Files

## Files in C++ are used to store data permanently on disk, which is essential when we want to preserve information beyond program execution. They're commonly used in real-world applications like banking systems, logging tools, and report generation.

The fstream library allows us to work with files.

To use the fstream library, include both the standard <iostream> **AND** the <fstream> header file:



**C++ Exceptions**

## **Exception handling** in C++ is a mechanism that allows a program to **detect and handle runtime errors (exceptions)** in a **safe and controlled manner** without crashing the program. It helps maintain program **stability, reliability, and user-friendly error messages**.

## Keywords in C++ Exception Handling:

| **Keyword** | **Meaning** |
| --- | --- |
| try | Code that might cause an error (risky code). |
| Throw | Raise an exception (signal the error). |
| Catch | Handle the exception (what to do if error happens) |
|  |  |

#include <iostream>

using namespace std;

int main() {

int a = 10;

int b = 0;

try {

if (b == 0) {

throw "Can't divide by zero!";

}

cout << a / b;

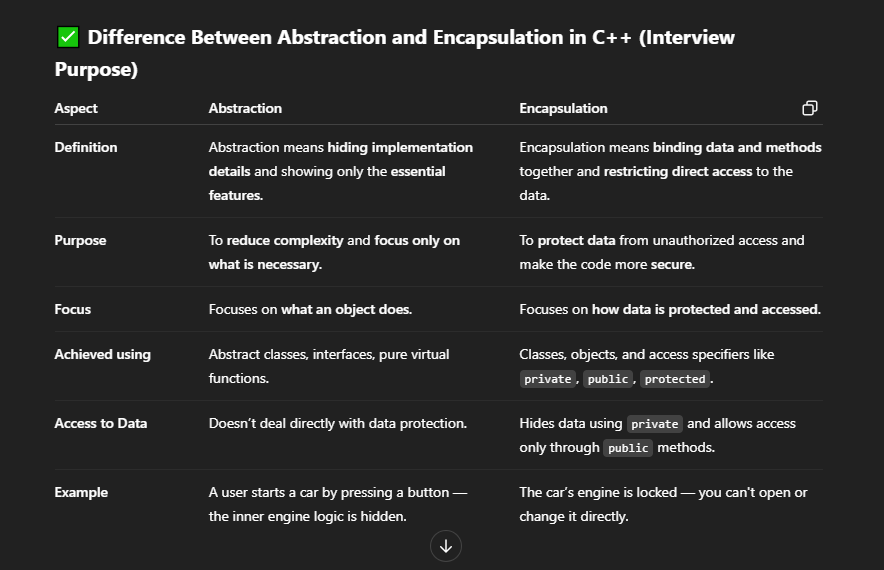
} catch (const char\* error) {

cout << "Error: " << error << endl;

}

return 0;

}



## ✅ One-Line Core Difference:

* **Encapsulation** = **Hiding data** (to protect it).
* **Abstraction** = **Hiding complexity** (to simplify it).

## 🔍 Think of it like this:

| **Concept** | **What is hidden?** | **Why?** |
| --- | --- | --- |
| **Encapsulation** | Data (variables/properties) | So others can't misuse it |
| **Abstraction** | Details of how something works | So others don’t need to understand it |

## 🎯 Everyday Example: Using a Mobile Phone

### ****Encapsulation****:

* You can't access the phone’s hardware directly (battery, processor).
* It's **encapsulated**.
* You interact through a **safe interface** (like buttons, apps).

### ****Abstraction****:

* You **tap a button** to take a photo.
* You don't need to know how the lens, sensor, and software process the image.
* That’s **abstraction** — hiding all the internal steps.

## ✅ Summary:

They **work together**:

* **Encapsulation** protects the **data**.
* **Abstraction** hides the **logic**.

**Diff b/w virtual and abstract function?**

**Virtual Function**:

A virtual function is a function in a **base (parent) class** that **can be overridden** by a **derived (child) class**.

* It has **some implementation** in the parent class.
* The child class can **override** it to change its behavior.

### Example (C++):

class Animal {

public:

virtual void sound() {

cout << "Animal makes sound";

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks";

}

};

The function sound () is **virtual** in the base class and **can be overridden** in child classes.

## 2. ****Abstract Function**** (Pure Virtual Function in C++, Abstract Method in Java/C#)

An abstract function is a function that has **no body** in the base class and only **declared in base class and end with =0, and defines later on in the derived class**.

### Example (C++):

class Animal {

public:

virtual void sound() = 0; // Pure virtual (abstract) function

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks";

}

};

✅ Here, sound() is **abstract** (pure virtual), so **must** be implemented in any derived class.