**What is C++?**

1. C++ is a cross-platform language that can be used to create high-performance applications.
2. C++ was developed by Bjarne Stroustrup, as an extension to the C language.
3. C++ gives programmers a high level of control over system resources and memory.
4. The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++1
5. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
6. This C++ tutorial adopts a simple and practical approach to describe the concepts of C++ for beginners to advanced software engineers.

**Why Use C++**

1. C++ is one of the world's most popular programming languages.
2. C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
3. C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
4. C++ is portable and can be used to develop applications that can be adapted to multiple platforms.
5. C++ is fun and easy to learn!
6. As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice
7. C++ is one of the every green programming languages and loved by millions of software developers. If you are a great C++ programmer then you will never sit without work and more importantly you will get highly paid for your work.

**Applications of C++ Programming**

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

**Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.

**Programming Languages Development -** C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX’s C Shell, PHP and Python, and Verilog etc.

**Computation Programming -** C++ is the best friends of scientists because of fast speed and computational efficiencies.

**Games Development -** C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.

**Embedded System -** C++ is being heavily used in developing Medical and Engineering Applications like software’s for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great software’s. I highly recommend you to learn C++ and contribute great software’s to the community.

**C++ Syntax**

**Example**

#include <iostream>

using namespace std;

int main() {

cout << "Hello World!";

return 0;

}

### Example explained

**Line 1:** #include <iostream> is a header file library that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.

**Line 2:** using namespace std means that we can use names for objects and variables from the standard library.

**Line 3**: A blank line. C++ ignores white space.

**Line 4:** Another thing that always appear in a C++ program, is int main(). This is called a function. Any code inside its curly brackets {} will be executed.

**Line 5:** cout (pronounced "see-out") is an object used together with the insertion operator (<<) to output/print text. In our example it will output "Hello World".

**Note:** Every C++ statement ends with a semicolon ;.

**Note:** The body of int main() could also been written as:

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

**Remember:** The compiler ignores white spaces. However, multiple lines makes the code more readable.

**Line 6**: return 0 ends the main function.

**Line 7:** Do not forget to add the closing curly bracket } to actually end the main function.

**Omitting Namespace**

You might see some C++ programs that runs without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

#include <iostream>

int main() {

std::cout << "Hello World!";

return 0;

}

**C++ Comments**

Comments can be used to explain C++ code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Comments can be singled-lined or multi-lined.

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by the compiler (will not be executed).

**Example**

// This is a comment

cout << "Hello World!";

**C++ Multi-line Comments**

Multi-line comments start with /\* and ends with \*/.

Any text between /\* and \*/ will be ignored by the compiler:

**Example:**

/\* The code below will print the words Hello World!

to the screen, and it is amazing \*/

cout << "Hello World!";

**C++ Variables**

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −

Variables are containers for storing data values.

In C++, there are different types of variables (defined with different keywords), for example:

* **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

**Variable Scope in C++**

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

* Inside a function or a block which is called local variables,
* In the definition of function parameters which is called formal parameters.
* Outside of all functions which is called global variables.

**Local Variables**

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables –

Example:

#include <iostream>

using namespace std;

int main () {

// Local variable declaration:

int a, b;

int c;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

**Global Variables**

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

**Example**

#include <iostream>

using namespace std;

// Global variable declaration:

int g;

int main () {

// Local variable declaration:

int a, b;

// actual initialization

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;}

**C++ Constants/Literals**

Constants refer to fixed values that the program may not alter and they are called literals.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

**Manipulator**

In C++ programming, language manipulators are used in the formatting of output. The two most commonly used manipulators are: "endl" and "setw".

* "endl" is used for the next line.
* "setw" is used to specify the width of the output.

**Operator Precedence & Operator Associativity**

Operator precedence helps us to solve an expression. For example, in an expression "int c = a\*b+c" the multiplication operator's precedence is higher than the precedence of addition operator, so the multiplication between "a & b" first and then addition will be performed.

Operator associativity helps us to solve an expression; when two or more operators have the same precedence, the operator associativity helps us to decide that we should solve the expression from "left-to-right" or from "right-to-left".

**website:**

<https://en.cppreference.com/w/cpp/language>

**C++ Conditions/ C++ decision making statements**

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C++ programming language provides following types of decision making statements.

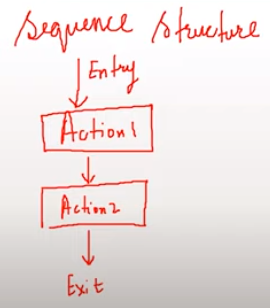
|  |  |
| --- | --- |
| **Sr.No** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/cplusplus/cpp_if_statement.htm)  An ‘if’ statement consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statement](https://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm)  An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the boolean expression is false. |
| 3 | [switch statement](https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm)  A ‘switch’ statement allows a variable to be tested for equality against a list of values. |
| 4 | [nested if statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_if.htm)  You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s). |
| 5 | [nested switch statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_switch.htm)  You can use one ‘switch’ statement inside another ‘switch’ statement(s |

Control Structures in C++

The work of control structures is to give flow and logic to a program. There are three types of basic control structures in C++.

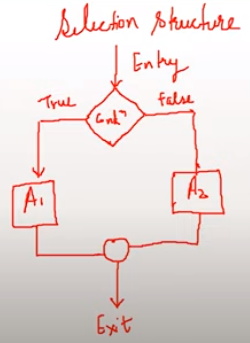
**Sequence Structure**

Sequence structure refers to the sequence in which program execute instructions one after another. An example diagram for the sequence structure is shown in figure 1



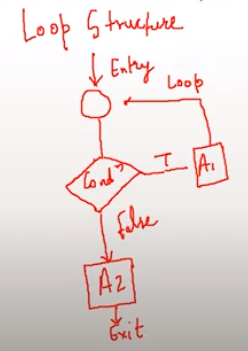
**Selection Structure**

Selection structure refers to the execution of instruction according to the selected condition, which can be either true or false. There are two ways to implement selection structures, by “if-else statements” or by “switch case statements”. An example diagram for selection structure is shown in figure 2.



**Loop Structure**

Loop structure refers to the execution of an instruction in a loop until the condition gets false. An example diagram for loop structure is shown in figure 3.



IF Else in C++

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

If else statements are used to implement a selection structure.

Switch Statement in C++

* In switch-case statements, the value of the variable is tested with all the cases.
* A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.
* The default keyword must be used as the last statement in the switch, and it does not need a break.

C++ Loop Types

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



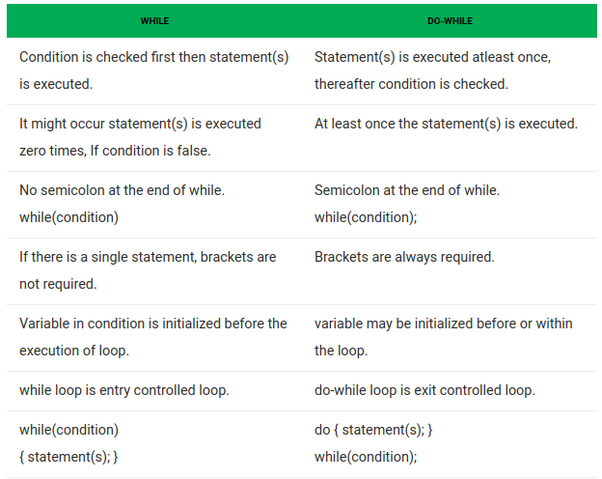
C++ programming language provides the following type of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/cplusplus/cpp_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm)  Like a ‘while’ statement, except that it tests the condition at the end of the loop body. |
| 4 | [nested loops](https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm)  You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop |

Loop Control Statements/jump

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

|  |  |
| --- | --- |
| **Sr.No** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/cplusplus/cpp_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.in this case of inner loop,it breaks only inner loop. |
| 2 | [continue statement](https://www.tutorialspoint.com/cplusplus/cpp_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating./skip **the current iteration loop(for , while , do while).**not for **switch** |
| 3 | [goto statement](https://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use goto statement in your program. |



# C++ Pointers

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 general form of a pointer variable declaration is −

type \*var-name;

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

#### **Pointers and Arrays**

Storing the address of an array into pointer is different than storing the address of a variable into the pointer because the name of the array is an address of the first index of an array. So to use ampersand "&" with the array name for assigning the address to a pointer is wrong.

* &Marks --> Wrong
* Marks --> address of the first block

An example program for storing the starting address of an array in the pointer is shown in code snippet 4.

int\* p = marks;

cout<<"The value of marks[0] is "<<\*p<<endl;

***ode Snippet 4: Pointer and Array Program***

As shown in code snippet 7, we have assigned the address of array “marks” to the pointer variable “\*p” and then printed the pointer “\*p”. The main thing to note here is that the value at the pointer “\*p” is the starting address of the array “marks”. The output for the following program is shown in figure 4.

***Figure 4: Pointer and Array Program Output***

As shown in figure 4, we have printed the value at pointer "\*p", and it has shown us the value of the first index of the array "marks" because the pointer was pointing at the first index of an array and the value at that index was "23". If we want to access the 2nd index of an array through the pointer, we can simply increment the pointer with 1. For example: "**\*(p+1)**" will give us the value of the 2nd index of an array. An example program to print the values of an array through the pointer is shown in code snippet 5.

int\* p = marks;

cout<<"The value of \*p is "<<\*p<<endl;

cout<<"The value of \*(p+1) is "<<\*(p+1)<<endl;

cout<<"The value of \*(p+2) is "<<\*(p+2)<<endl;

cout<<"The value of \*(p+3) is "<<\*(p+3)<<endl;

Copy

***Code Snippet 5: Pointer and Array Program 2***

As shown in code snippet 5, 1st we have printed the value at pointer “**\*p**”; 2nd we have printed the value at pointer “**\*(p+1)**”; 3rd we have printed the value at pointer “**\*(p+2)**”; 4th we have printed the value at pointer “**\*(p+3)**". This program will output the values at "0, 1, 2, 3" indices of an array "marks". The output of the following program is shown in figure 5.

C++ Arrays

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

## **Declaring Arrays**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

## **Initializing Arrays**

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

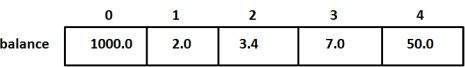
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



## **Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

#### **Structures in C++**

The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types. An example program for creating a structure is shown in Code Snippet 1.

struct employee

{

/\* data \*/

int eId;

char favChar;

float salary;

};

Copy

**Code Snippet 1: Creating a Structure Program**

As shown in Code Snippet 1, we have created a structure with the name “employee”, in which we have declared three variables of different data types (eId, favchar, salary). As we have created a structure now we can create instances of our structure employee. An example program for creating instances of structure employees is shown in Code Snippet 2.

int main() {

struct employee harry;

harry.eId = 1;

harry.favChar = 'c';

harry.salary = 120000000;

cout<<"The value is "<<harry.eId<<endl;

cout<<"The value is "<<harry.favChar<<endl;

cout<<"The value is "<<harry.salary<<endl;

return 0;

}

Copy

**Code Snippet 2: Creating Structure instances**

As shown in Code Snippet 2, 1st we have created a structure variable “harry” of type “employee”, 2nd we have assigned values to (eId, favchar, salary) fields of the structure employee and at the end we have printed the value of “salary”.

Another way to create structure variables without using the keyword “struct” and the name of the struct is shown in Code Snippet 3.

typedef struct employee

{

/\* data \*/

int eId; //4

char favChar; //1

float salary; //4

} ep;

Copy

**Code Snippet 3: Creating Structure Program 2**

As shown in Code Snippet 3, we have used a keyword “**typedef**” before struct and after the closing bracket of structure, we have written “ep”. Now we can create structure variables without using the keyword “struct” and name of the struct. An example is shown in Code Snippet 4.

int main(){

ep harry;

struct employee shubham;

struct employee rohanDas;

harry.eId = 1;

harry.favChar = 'c';

harry.salary = 120000000;

cout<<"The value is "<<harry.eId<<endl;

cout<<"The value is "<<harry.favChar<<endl;

cout<<"The value is "<<harry.salary<<endl;

return 0;

}

Copy

**Code Snippet 4: Creating Structure instance 2**

As shown in Code Snippet 4, we have created a structure instance “harry” by just writing “ep” before it.

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.

The general form of a C++ function definition is as follows −

return\_type function\_name( parameter list ) {

body of the function

}

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

**Return Type** − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.

**Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

**Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body** − The function body contains a collection of statements that define what the function does

#### **Functions in C++**

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions help us to reuse the code easily. An example program for the function is shown in Code Snippet 1.

int sum(int a, int b){

int c = a+b;

return c;

***Code Snippet 1: Function example***

As shown in Code Snippet 1, we created an integer function with the name of sum, which takes two parameters “int a” and “int b”. In the function, body addition is performed on the values of variable “a” and variable “b” and the result is stored in variable “c”. In the end, the value of variable “c” is returned to the function. We have seen how this function works now we will see how to pass values to the function parameters. An example program for passing the values to the function is shown in Code Snippet 2.

int main(){

int num1, num2;

cout<<"Enter first number"<<endl;

cin>>num1;

cout<<"Enter second number"<<endl;

cin>>num2;

cout<<"The sum is "<<sum(num1, num2);

return 0;

}

***Code Snippet 2: Passing Value to Function Parameters***

As shown in Code Snippet 2, we have declared two integer variables “num1” and “num2”, we will take their input at run time. In the end, we called the “sum” function and passed both variables “num1” and “num2” into sum function. “sum” function will perform the addition and returns the value at the same location from where it was called. The output of the following program is shown in figure 1.

**Figure 1: Function Output**

#### **Function Prototype in C++**

The function prototype is the template of the function which tells the details of the function e.g(name, parameters) to the compiler. Function prototypes help us to define a function after the function call. An example of a function prototype is shown in Code Snippet 3.

// Function prototype

int sum(int a, int b);

**Code Snippet 3: Function Prototype**

As shown in Code Snippet 3, we have made a function prototype of the function “sum”, this function prototype will tell the compiler that the function “sum” is declared somewhere in the program which takes two integer parameters and returns an integer value. Some examples of acceptable and not acceptable prototypes are shown below:

* int sum(int a, int b); //Acceptable
* int sum(int a, b); // Not Acceptable
* int sum(int, int); //Acceptable

##### **Formal Parameters**

The variables which are declared in the function are called a formal parameter. For example, as shown in Code Snippet 1, the variables “a” and “b” are the formal parameters.

##### **Actual Parameters**

The values which are passed to the function are called actual parameters. For example, as shown in Code Snippet 2, the variables “num1” and “num2” are the actual parameters.

The function doesn't need to have parameters or it should return some value. An example of the void function is shown in Code Snippet 4.

void g(){

cout<<"\nHello, Good Morning";

}

Copy

**Code Snippet 4: Void Function**

As shown in Code Snippet 4, void as a return type means that this function will not return anything, and this function has no parameters. Whenever we will call this function it will print “Hello, Good Morning”

## **Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

|  |  |
| --- | --- |
| **Sr.No** | **Call Type & Description** |
| 1 | [Call by Value](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Call by Pointer](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_pointer.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |
| 3 | [Call by Reference](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm)  This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

#### **Call by Value in C++**

Call by value is a method in C++ to pass the values to the function arguments. In case of call by value the copies of actual parameters are sent to the formal parameter, which means that if we change the values inside the function that will not affect the actual values. An example program for the call by value is shown in Code Snippet 1.

void swap(int a, int b){ //temp a b

int temp = a; //4 4 5

a = b; //4 5 5

b = temp; //4 5 4

}

Copy

**Code Snippet 1: Call by Value Swap Function**

As shown in Code Snippet 1, we created a swap function which is taking two parameters “int a” and “int b”. In function body values of the variable, “a” and “b” are swapped.  An example program is shown in Code Snippet 2, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

swap(x, y);

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

}

Copy

**Code Snippet 2: Passing Values to Swap Function**

As shown in Code Snippet 2, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed values of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 1.

**Figure 1: Call by Value Swap Function Output**

As shown in figure 3, the values of “a” and “b” are the same for both times they are printed. So the main point here is that when the call by value method is used it doesn’t change the actual values because copies of actual values are sent to the function.

#### **Call by Pointer in C++**

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for the call by reference is shown in Code Snippet 3.

// Call by reference using pointers

void swapPointer(int\* a, int\* b){ //temp a b

int temp = \*a; //4 4 5

\*a = \*b; //4 5 5

\*b = temp; //4 5 4

}

Copy

**Code Snippet 3: Call by Pointer Swap Function**

As shown in Code Snippet 3, we created a swap function which is taking two pointer parameters “int\* a” and “int\* b”. In function body values of pointer variables, “a” and “b” are swapped.  An example program is shown in Code Snippet 4, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

swapPointer(&x, &y); //This will swap a and b using pointer reference

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

}

Copy

**Code Snippet 4: Passing Values to Call by Pointer Swap Function**

As shown in Code Snippet 4, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed addresses of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 2.

**Figure 2: Call by Pointer Swap Function Output**

As shown in figure 2, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by pointer method is used it changes the actual values because addresses of actual values are sent to the function.

#### **Call by Reference in C++**

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for a call by reference is shown in Code Snippet 5.

void swapReferenceVar(int &a, int &b){ //temp a b

int temp = a; //4 4 5

a = b; //4 5 5

b = temp; //4 5 4

}

Copy

**Code Snippet 5: Call by Reference Swap Function**

As shown in Code Snippet 5, we created a swap function that is taking reference of “int &a” and “int &b” as parameters. In function body values of variables, “a” and “b” are swapped.  An example program is shown in Code Snippet 6, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

swapReferenceVar(x, y); //This will swap a and b using reference variables

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

}

Copy

**Code Snippet 6: Passing Values to Call by Reference Swap Function**

As shown in Code Snippet 6, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 3.

**Figure 3: Call by Reference Swap Function Output**

As shown in figure 3, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by reference method is used it changes the actual values because references of actual values are sent to the function.

#### **Recursion and Recursive Function**

When a function calls itself it is called recursion and the function which is calling itself is called a recursive function. The recursive function consists of a base case and recursive condition. It is very important to add a base case in recursive function otherwise recursive function will never stop executing. An example of the recursive function is shown in Code Snippet 1.

int factorial(int n){

if (n<=1){

return 1;

}

return n \* factorial(n-1);

}

Copy

**Code Snippet 1: Factorial Recursive Function**

As shown in Code Snippet 1, we created a “factorial” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller or equal to “1” if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a factorial. These are the steps which will be performed by recursive condition:

* **4 \* factorial( 4-1 )**
* **4 \* 3 \* factorial( 3-1 )**
* **4\* 3 \* 2 \* factorial( 2-1 )**
* **4 \* 3 \* 2 \* 1**

An example to pass the value to the recursive factorial function is shown in Code Snippet 2.

int main(){

int a;

cout<<"Enter a number"<<endl;

cin>>a;

cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;

return 0;

}

Copy

**Code Snippet 2: Factorial Recursive Function Call**

As shown in Code Snippet 2, we created an integer variable “a”, which takes input at the runtime and that value is passed to the factorial function. The output for the following program is shown in figure 1.

**Figure 1: Factorial Recursive Function Output**

As shown in figure 1, we input the value “4” and it gives us the factorial of it which is “24”. Another example of a recursive function for the Fibonacci series is shown in Code Snippet 3.

int fib(int n){

if(n<2){

return 1;

}

return fib(n-2) + fib(n-1);

}

Copy

**Code Snippet 3: Fibonacci Recursive Function**

As shown in Code Snippet 3, we created a “fib” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller than “2”, if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a Fibonacci number. An example to pass the value to the Fibonacci function is shown in Code Snippet 4.

int main(){

int a;

cout<<"Enter a number"<<endl;

cin>>a;

cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;

return 0;

}

Copy

**Code Snippet 4: Fibonacci Recursive Function Call**

As shown in Code Snippet 4, we created an integer variable “a”, which takes input at the runtime and that value is passed to the Fibonacci function. The output for the following program is shown in figure 2.

**Figure 2: Fibonacci Recursive Function Output**

As shown in figure 2, 1st we input the value “5” and it gives us the Fibonacci number at that place which is “8”. 2nd we input the value “6” and it gives us the Fibonacci number at that place which is “13”.

One thing to note here is that recursive functions are not always the best option. They perform well in some problems but not in every problem.

#### **Function Overloading in C++**

Function overloading is a process to make more than one function with the same name but different parameters, numbers, or sequence. An example program to explain function overloading is shown in Code Snippet 1.

int sum(float a, int b){

cout<<"Using function with 2 arguments"<<endl;

return a+b;

}

int sum(int a, int b, int c){

cout<<"Using function with 3 arguments"<<endl;

return a+b+c;

}

Copy

**Code Snippet 1: Sum Function Overloading Example**

As shown in Code Snippet 1, we have created two “sum” functions, the 1st “sum” function takes two arguments “int a”, “int b” and return the sum of those two variables; and the 2nd sum function is taking three arguments “int a”, “int b”, “int c” and return the sum of those three variables. Function call for these “sum” function is shown in Code Snippet 2.

int main(){

cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;

cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;

return 0;

}

Copy

**Code Snippet 2: Sum Function Call**

As shown in Code Snippet 2, we passed two arguments in the first function call and three arguments in the second function call. The output of the following program is shown in figure 1.

**Figure 1: Sum Function Output**

As shown in Code Snippet 3, both the “sum” function runs fine and gives us the required output. The main thing to note here is that the name of the function can be the same but the data type and the sequence of arguments need to be different as shown in the example program otherwise program will not run.

Another example of function overloading is shown in Code Snippet 3.

// Calculate the volume of a cylinder

int volume(double r, int h){

return(3.14 \* r \*r \*h);

}

// Calculate the volume of a cube

int volume(int a){

return (a \* a \* a);

}

// Rectangular box

int volume (int l, int b, int h){

return (l\*b\*h);

}

Copy

**Code Snippet 3: Volume Function Overloading Example**

As shown in Code Snippet 3, we have created three “volume” functions, the 1st “volume” function calculates the volume of the cylinder and has two arguments “double r” and “int h”; the 2nd “volume” function calculates the volume of the cube and has one argument “int a”; the 3rd “volume” function calculates the volume of the rectangular box and has three arguments “int l”, “int b” and “int h”. The function call for these “volumes” function is shown in Code Snippet 4.

int main(){

cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;

cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;

cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;

return 0;

}

Copy

**Code Snippet 4: Volume Function Call**

As shown in Code Snippet 4, we passed three arguments in the first function call, two arguments in the second function call, and one argument in the third function call. The output of the following program is shown in figure 2.

**Figure 2: Volume Function Output**

As shown in figure 2, all three “volume” functions run fine and give us the required output.

C++ Classes and Objects

## **C++ What is OOP?**

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

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

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

## **C++ What are Classes and Objects?**

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

## **class**

Fruit

## **objects**

Apple

Banana

Mango

Another example:

## **class**

Car

## **objects**

Volvo

Audi

Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and functions from the class.

#### **Why Object-Oriented Programming?**

Before we discuss object-oriented programming, we need to learn why we need object-oriented programming?

* C++ language was designed with the main intention of adding object-oriented programming to C language
* As the size of the program increases readability, maintainability, and bug-free nature of the program decrease.
* This was the major problem with languages like C which relied upon functions or procedure (hence the name procedural programming language)
* As a result, the possibility of not addressing the problem adequately was high
* Also, data was almost neglected, data security was easily compromised
* Using classes solves this problem by modeling program as a real-world scenario

#### **Difference between Procedure Oriented Programming and Object-Oriented Programming**

##### **Procedure Oriented Programming**

* Consists of writing a set of instruction for the computer to follow
* The main focus is on functions and not on the flow of data
* Functions can either use local or global data
* Data moves openly from function to function

##### **Object-Oriented Programming**

* Works on the concept of classes and object
* A class is a template to create objects
* Treats data as a critical element
* Decomposes the problem in objects and builds data and functions around the objects

#### **Basic Concepts in Object-Oriented Programming**

* **Classes -**Basic template for creating objects
* **Objects –**Basic run-time entities
* **Data Abstraction & Encapsulation –**Wrapping data and functions into a single unit
* **Inheritance –**Properties of one class can be inherited into others
* **Polymorphism –**Ability to take more than one forms
* **Dynamic Binding –**Code which will execute is not known until the program runs
* **Message Passing –**message (Information) call format

#### **Benefits of Object-Oriented Programming**

* Better code reusability using objects and inheritance
* Principle of data hiding helps build secure systems
* Multiple Objects can co-exist without any interference
* Software complexity can be easily managed

#### **Why use classes instead of structures**

Classes and structures are somewhat the same but still, they have some differences. For example, we cannot hide data in structures which means that everything is public and can be accessed easily which is a major drawback of the structure because structures cannot be used where data security is a major concern. Another drawback of structures is that we cannot add functions in it.

#### **Classes in C++**

Classes are user-defined data-types and are a template for creating objects. Classes consist of variables and functions which are also called class members.

#### **Public Access Modifier in C++**

All the variables and functions declared under public access modifier will be available for everyone. They can be accessed both inside and outside the class. Dot (.) operator is used in the program to access public data members directly.

#### **Private Access Modifier in C++**

All the variables and functions declared under a private access modifier can only be used inside the class. They are not permissible to be used by any object or function outside the class.

An example program to demonstrate classes, public and private access modifiers are shown in Code Snippet 1.

class Employee

{

private:

int a, b, c;

public:

int d, e;

void setData(int a1, int b1, int c1); // Declaration

void getData(){

cout<<"The value of a is "<<a<<endl;

cout<<"The value of b is "<<b<<endl;

cout<<"The value of c is "<<c<<endl;

cout<<"The value of d is "<<d<<endl;

cout<<"The value of e is "<<e<<endl;

}

};

void Employee :: setData(int a1, int b1, int c1){

a = a1;

b = b1;

c = c1;

}

Copy

**Code Snippet 1: Class Program**

As shown in Code Snippet 1, 1st we created an “employee” class, 2nd three integer variables “int a”, “int b”, and “int c” were declared under the private access modifier, 3rd two integer variables “int d” and “int e” was declared under the public access modifiers, 4th “setData” function was declared, 5th “getData” function was defined and values of all the variables are printed. 6th “setData” function was defined outside the “employee” class by using a scope resolution operator; “setData” function is used to assign values to the private member of the class. An example to create the object of the class and use its class members is shown in Code Snippet 2.

int main(){

Employee harry;

harry.d = 34;

harry.e = 89;

harry.setData(1,2,4);

harry.getData();

return 0;

}

Copy

**Code Snippet 2: Creating Object Example**

As shown in Code Snippet 2, 1st we created an object “harry” of the class “employee”; 2nd we assigned values to “int d” and “int e” which are public class members. If we try to assign values to the private class member’s compiler will throw an error. 3rd we passed the values to the function “setData” and at the end, we called “getData” function which will print the values of all the variables. The output for the following program is shown in figure 1.

**Figure 1: Class Program Output**

As shown in figure 1, all the values of our data members are printed.

#### **Object-Oriented programming Recap**

* Stroustrup initially named C++ language as C with classes because C++ language was almost the same as C language but they added a new concept of classes in it.
* Classes are the extension of structures in C language.
* Structures had limitations such as; members are public and no methods.
* Classes have some additional futures than structures such as; classes that can have methods and properties.
* Classes have a feature to make class members as public and private.
* In C++ objects can be declared along with class deceleration as shown in Code Snippet 1.

class Employee{

// Class definition

} harry, rohan, lovish;

Copy

**Code Snippet 1: Declaring Objects with Class Declaration**

#### **Nesting of Member Functions**

If one member function is called inside the other member function of the same class it is called nesting of a member function. A program to demonstrate the nesting of a member function is shown below.

class binary

{

private:

string s;

void chk\_bin(void);

public:

void read(void);

void ones\_compliment(void);

void display(void);

};

Copy

**Code Snippet 2: Binary Class**

As shown in Code Snippet 2, we created a binary class that has, “s” string variable and “chk\_bin” void function as private class members; and “read” void function, “ones\_compliment” void function, and “display” void function as public class members. The definitions of these functions are shown below.

void binary::read(void)

{

cout << "Enter a binary number" << endl;

cin >> s;

}

Copy

**Code Snippet 3: Read Function**

As shown in Code Snippet 3, we have created a “read” function. This function will take input from the user at runtime.

void binary::chk\_bin(void)

{

for (int i = 0; i < s.length(); i++)

{

if (s.at(i) != '0' && s.at(i) != '1')

{

cout << "Incorrect binary format" << endl;

exit(0);

}

}

}

Copy

**Code Snippet 4: Check Binary Function**

As shown in Code Snippet 4 we have created a “chk\_bin” function. This ”for” loop in the function will run till the length of the string and “if” condition in the body of the loop will check the whole string that if there are any values in the string other than “1” and “0”. If there are values other than “1” and “0” this function will output “Incorrect binary format”.

void binary::ones\_compliment(void)

{

chk\_bin();

for (int i = 0; i < s.length(); i++)

{

if (s.at(i) == '0')

{

s.at(i) = '1';

}

else

{

s.at(i) = '0';

}

}

}

Copy

**Code Snippet 5: One's Compliment**

As shown in Code Snippet 5, in the body of the “ones\_compliment” function; the “chk\_bin” function is called, and as we have discussed above that if one member function is called inside the other member function of the same class it is called **nesting of a member function**. The “for” loop inside the “ones\_compliment” functions runs till the length of the string and the “if” condition inside the loop replaces the number “0” with “1” and “1” with “0”.

void binary::display(void)

{

cout<<"Displaying your binary number"<<endl;

for (int i = 0; i < s.length(); i++)

{

cout << s.at(i);

}

cout<<endl;

}

Copy

***Code Snippet 6: Display Function***

As shown in Code Snippet 6, the “for” loop inside display function runs till the length of the string and prints each value of the sting.

int main()

{

binary b;

b.read();

// b.chk\_bin();

b.display();

b.ones\_compliment();

b.display();

return 0;

}

Copy

**Code Snippet 7: Main Function**

As shown in Code Snippet 7, we created an object “b” of the binary data type, and the functions “read”, “display”, “ones\_compliment”, and “display” are called. The main thing to note here is that the function ”chk\_bin” is the private access modifier of the class so we cannot access it directly by using the object, it can be only accessed inside the class or by the member function of the class.

#### **Objects Memory Allocation in C++**

The way memory is allocated to variables and functions of the class is different even though they both are from the same class.

The memory is only allocated to the variables of the class when the object is created. The memory is not allocated to the variables when the class is declared. At the same time, single variables can have different values for different objects, so every object has an individual copy of all the variables of the class. But the memory is allocated to the function only once when the class is declared. So the objects don’t have individual copies of functions only one copy is shared among each object.

#### **Arrays in Classes**

Arrays are used to store multiple values of the same type. An array is very helpful when multiple variables are required, instead of making multiple variables one array can be used which can store multiple values. Array stores data in sequential order. An example program to demonstrate the use of arrays in classes is shown below.

class Shop

{

int itemId[100];

int itemPrice[100];

int counter;

public:

void initCounter(void) { counter = 0; }

void setPrice(void);

void displayPrice(void);

};

Copy

**Code Snippet 1: Shop Class**

As shown in Code Snippet 1, we created a shop class which has, “itemId[100]” and “itemPrice” as integer array variable and “counter” variable as private class members; and “initCounter” void function, “setPrice” void function, and “displayPrice” void function as public class members. The definitions of these functions are shown below.

void Shop ::setPrice(void)

{

cout << "Enter Id of your item no " << counter + 1 << endl;

cin >> itemId[counter];

cout << "Enter Price of your item" << endl;

cin >> itemPrice[counter];

counter++;

}

Copy

**Code Snippet 2: Set Price Function**

As shown in Code Snippet 2, we have created a “setPrice” function. This function will take input for “itemId” and “ItemPrice” from the user at runtime. The value of the counter will be incremented by one every time this function will run.

void Shop ::displayPrice(void)

{

for (int i = 0; i < counter; i++)

{

cout << "The Price of item with Id " << itemId[i] << " is " << itemPrice[i] << endl;

}

}

Copy

**Code Snippet 3: Display Price Function**

As shown in Code Snippet 3, the “for” loop inside the “displayPrice” function runs till the length of the counter and prints values of the array “itemId” and “ItemPrice”.

int main()

{

Shop dukaan;

dukaan.initCounter();

dukaan.setPrice();

dukaan.setPrice();

dukaan.setPrice();

dukaan.displayPrice();

return 0;

}

Copy

**Code Snippet 4: Main Function**

As shown in Code Snippet 4, we created an object “dukaan” of the shop data type, and the functions “initCounter” is called. The function “setPrice” is called three times. Loops can also be used to call the function multiple times. The “displayPrice” function is also called in the main function. The output of the following program is shown in figure 1.

**Figure 1: Program Output**

As shown in figure 1, for the item 1 we entered the ID “1001” and price “12”; for the item 2 we entered the ID “1002” and price “23”; for the item 3 we entered the ID “1003” and price “34”. The Output of the program has displayed the ID and the price of each item.

#### **Static Data Members in C++**

When a static data member is created, there is only a single copy of the data member which is shared between all the objects of the class. As we have discussed in our previous lecture that if the data members are not static then every object has an individual copy of the data member and it is not shared.

#### **Static Methods in C++**

When a static method is created, they become independent of any object and class. Static methods can only access static data members and static methods. Static methods can only be accessed using the scope resolution operator. An example program is shown below to demonstrate static data members and static methods in C++.

class Employee

{

int id;

static int count;

public:

void setData(void)

{

cout << "Enter the id" << endl;

cin >> id;

count++;

}

void getData(void)

{

cout << "The id of this employee is " << id << " and this is employee number " << count << endl;

}

static void getCount(void){

// cout<<id; // throws an error

cout<<"The value of count is "<<count<<endl;

}

};

Copy

**Code Snippet 1: Employee Class**

As shown in Code Snippet 1, we created an employee class that has integer “id” variable and “count” static integer variable as private class members; and “setData” void function, “getData” void function, and “getCount” static void function as public class members. These functions are explained below.

We have defined a “setData” function. This function will take input for “id” from the user at runtime and increment in the count. The value of the counter will be incremented by one every time this function will run.

We have defined a “getData” function. This function will print the values of the variables “id” and “count”.

We have defined a static “getCount” function. This function will print the value of the variable count”. The main thing to note here is that “getCount” function is static, so if we try to access any data members or member functions which are not static the compiler will throw an error.

// Count is the static data member of class Employee

int Employee::count; // Default value is 0

int main()

{

Employee harry, rohan, lovish;

// harry.id = 1;

// harry.count=1; // cannot do this as id and count are private

harry.setData();

harry.getData();

Employee::getCount();

rohan.setData();

rohan.getData();

Employee::getCount();

lovish.setData();

lovish.getData();

Employee::getCount();

return 0;

}

Copy

**Code Snippet 2: main Program**

As shown in Code Snippet 2:

* The count variable is declared whose default value is “0”.
* Then we created objects “harry”, “rohan”, and “lovish” of the employee data type
* The functions “setData”, “getData” are called by the object “harry”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
* The functions “setData”, “getData” are called by the object “rohan”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
* The functions “setData”, “getData” are called by the object “lovish”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.

The output of the following program is shown in figures 1 and 2.

**Figure 1: Program Output 1**

**Figure 2: Program Output 2**

As shown in figures 1 and 2, for the “harry” object we entered the ID “1”; for the “rohan” object we entered the ID “2”; and for the “lovish” object we entered the ID “3”. The Output of the program has displayed the ID and the count of each employee.

#### **An array of Objects in C++**

An array of objects is declared the same as any other data-type array.  An array of objects consists of class objects as its elements. If the array consists of class objects it is called an array of objects. An example program to demonstrate the concept of an array of objects is shown below.

class Employee

{

int id;

int salary;

public:

void setId(void)

{

salary = 122;

cout << "Enter the id of employee" << endl;

cin >> id;

}

void getId(void)

{

cout << "The id of this employee is " << id << endl;

}

};

Copy

**Code Snippet 1: Employee Class**

As shown in Code Snippet 1, we created an employee class that has integer “id” variable and “salary” integer variable as private class members; and “setId” void function, “getId” void function as public class members. These functions are explained below.

We have defined a “setId” function. In this function, the “salary” variable is assigned by the value “122” and the function will take input for “id” from the user at runtime. We have defined a “getId” function. This function will print the values of the variables “id”.

int main()

{

Employee fb[4];

for (int i = 0; i < 4; i++)

{

fb[i].setId();

fb[i].getId();

}

return 0;

}

Copy

**Code Snippet 2: main program**

As shown in Code Snippet 2, we created an array “fb” of size “4” which is of employee data-type. The “for” loop is used to run “setId” and “getId” functions till the size of an array. The main thing to note here is that the objects can also be created individually but it is more convenient to use an array if too many objects are to be created. The output of the following program is shown in figure 1.

**Figure 1: Employee Program Output**

As shown in figure 1. As we input the Id for an employee it gives us the output of the employee Id.

#### **Passing Object as Function Argument**

Objects can be passed as function arguments. This is useful when we want to assign the values of a passed object to the current object. An example program to demonstrate the concept of passing an object as a function argument is shown below.

class complex{

int a;

int b;

public:

void setData(int v1, int v2){

a = v1;

b = v2;

}

void setDataBySum(complex o1, complex o2){

a = o1.a + o2.a;

b = o1.b + o2.b;

}

void printNumber(){

cout<<"Your complex number is "<<a<<" + "<<b<<"i"<<endl;

}

};

Copy

**Code Snippet 3: Complex Class**

As shown in Code Snippet 3, we created a complex class that has integer “a” variable and “b” integer variable as private class members; and “setData” void function, “setDataBySum” void function, and “printNumber” void function as public class members. These functions are explained below.

We have defined a “setData” function. In this function the values are assigned to the variables “a” and “b” because they are private data members of the class and values cannot be assigned directly. We have defined a “setDataBySum” function. In this function, the values of two objects are added and then assigned to the variables “a” and “b”. We have defined a “printNumber” function. In this function, the values of the variable “a” and “b” are being printed.

int main(){

complex c1, c2, c3;

c1.setData(1, 2);

c1.printNumber();

c2.setData(3, 4);

c2.printNumber();

c3.setDataBySum(c1, c2);

c3.printNumber();

return 0;

}

Copy

**Code Snippet 4: main program 2**

As shown in Code Snippet 4:

* We have created object “c1”, “c2”, and”c3” of complex data-type.
* The object “c1” calls the “setData” and “printNumber” functions.
* The object “c2” calls the “setData” and “printNumber” functions.
* The object “c3” calls the “setDataBySum” and “printNumber” functions.

The output of the following program is shown in figure 2.

**Figure 2: Complex Program Output**

#### **Friend Function in C++**

Friend functions are those functions that have the right to access the private data members of class even though they are not defined inside the class. It is necessary to write the prototype of the friend function. One main thing to note here is that if we have written the prototype for the friend function in the class it will not make that function a member of the class. An example program to demonstrate the concept of friend function is shown below.

class Complex{

int a, b;

public:

void setNumber(int n1, int n2){

a = n1;

b = n2;

}

// Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)

friend Complex sumComplex(Complex o1, Complex o2);

void printNumber(){

cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;

}

};

Complex sumComplex(Complex o1, Complex o2){

Complex o3;

o3.setNumber((o1.a + o2.a), (o1.b+o2.b))

;

return o3;

}

Copy

**Code Snippet 1: Complex Class**

As shown in Code Snippet 1, we created a complex class that has integer “a” variable and “b” integer variable as private class members; and “setNumber” void function, “printNumber” void function as public class members. The “sumComplex” friend function prototype is written as well in the complex class. These functions are explained below.

We have defined a “setNumber” function. In this function the values are assigned to the variables “a” and “b” because they are private data members of the class and values cannot be assigned directly. We have defined a “printNumber” function. In this function, the values of the variable “a” and “b” are being printed. We have defined a “sumComplex” friend function. In this function, the object “o3” is created which calls the “setNumber” function and passes the values of two objects after performing addition on them.

int main(){

Complex c1, c2, sum;

c1.setNumber(1, 4);

c1.printNumber();

c2.setNumber(5, 8);

c2.printNumber();

sum = sumComplex(c1, c2);

sum.printNumber();

return 0;

}

Copy

**Code Snippet 2: main Program**

As shown in Code Snippet 2:

* We have created object “c1”, “c2”, and”sum” of complex data-type.
* The object “c1” calls the “setNumber” and “printNumber” functions.
* The object “c2” calls the “setNumber” and “printNumber” functions.
* The function “sumComplex” is called and the values are assigned to the “sum”.
* The object “sum” calls the “printNumber” functions.

The output of the following program is shown in figure 1.

**Figure 1: Complex Program Output**

As shown in figure 1, the output of the complex number program is printed.

#### **Properties of Friend Function**

* Not in the scope of the class
* Since it is not in the scope of the class, it cannot be called from the object of that class, for example, **sumComplex()**is invalid
* A friend function can be invoked without the help of any object
* Usually contain objects as arguments
* Can be declared under the public or private access modifier, it will not make any difference
* It cannot access the members directly by their names, it needs (object\_name.member\_name) to access any member.

#### **Member Friend Functions in C++**

Friend functions are those function which has access to private members of the class in which they are declared. The main thing to note here is that only that function can access the member function which is made friend of the other class. An example of a friend function is shown below.

class Complex

{

int a, b;

// Individually declaring functions as friends

friend int Calculator ::sumRealComplex(Complex, Complex);

friend int Calculator ::sumCompComplex(Complex, Complex);

public:

void setNumber(int n1, int n2)

{

a = n1;

b = n2;

}

void printNumber()

{

cout << "Your number is " << a << " + " << b << "i" << endl;

}

};

int Calculator ::sumRealComplex(Complex o1, Complex o2)

{

return (o1.a + o2.a);

}

int Calculator ::sumCompComplex(Complex o1, Complex o2)

{

return (o1.b + o2.b);

}

Copy

**Code Snippet 1: Friend function example**

As shown in a code snippet 1, the complex class is created which consist of two friend functions “sumRealComplex” and “sumCompComplex” of the calculator class. The main thing to note here is that “sumRealComplex” and “sumCompComplex” are the friend functions of complex class so they can access all the private members of the complex class.

#### **Friend Classes in C++**

Friend classes are those classes which have the permission to access private members of the class in which they are declared. The main thing to note here is that if the class is made friend of another class then it can access all the private members of that class. An example program to demonstrate friend classes in C++ is shown below.

// Forward declaration

class Complex;

class Calculator

{

public:

int add(int a, int b)

{

return (a + b);

}

int sumRealComplex(Complex, Complex);

int sumCompComplex(Complex, Complex);

};

Copy

**Code Snippet 2: Calculator Class**

As shown in code snippet 2, complex class is declared at the top which is known as forward declaration. Forward declaration hints the compiler that this class is declared somewhere forward in the code. After that calculator class is defined this consists of three public member functions, “add”, “sumRealComplex”, and “sumCompComplex”. The “add” function will add the values of “a” and “b” and return the value. The “sumRealComplex” and “sumCompComplex” are taking two objects of the complex class. Code for the complex class is shown below.

class Complex

{

int a, b;

// Aliter: Declaring the entire calculator class as friend

friend class Calculator;

public:

void setNumber(int n1, int n2)

{

a = n1;

b = n2;

}

void printNumber()

{

cout << "Your number is " << a << " + " << b << "i" << endl;

}

};

int Calculator ::sumRealComplex(Complex o1, Complex o2)

{

return (o1.a + o2.a);

}

int Calculator ::sumCompComplex(Complex o1, Complex o2)

{

return (o1.b + o2.b);

}

Copy

**Code Snippet 3: Complex Class**

As shown in code snippet 3, complex class is defined which consists of, two private data members “a” and “b”, and two public member functions “setNumber” and “printNumber”. The function “setNumber” will assign the values to the variables “a” and “b”. The function “printNumber” will print the values of the variables “a” and “b”. Two functions “sumRealComplex” and “sumCompComplex” are defined at the end. The function “sumRealComplex” will add the real values and the function “sumCompComplex” will add the complex value. The main program is shown below.

int main()

{

Complex o1, o2;

o1.setNumber(1, 4);

o2.setNumber(5, 7);

Calculator calc;

int res = calc.sumRealComplex(o1, o2);

cout << "The sum of real part of o1 and o2 is " << res << endl;

int resc = calc.sumCompComplex(o1, o2);

cout << "The sum of complex part of o1 and o2 is " << resc << endl;

return 0;

}

Copy

**Code snippet 4: Main Program**

As shown in code snippet 4, 1st two objects “o1” and “o2” of the “complex” data type are declared. 2nd “setNumber” function is called with the “o1” and “o2” objects and the values are passed. 3rd object “calc” of the calculator data type is declared. 4th “sumRealComplex” function is called by the “calc” object and the object “o1” and “o2” are passed to it. 5th “sumCompComplex” function is called by the “calc” object and the object “o1” and “o2” are passed to it. The output of the following program is shown in figure 1.

## **Constructors**

A constructor in C++ is a **special method** that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses ():

**Properties of constructor:**

1. Constructor is a special type of functions.
2. Constructor has the same name as that of the class it belongs.
3. It has no return type not even void.
4. It is called automatically.

**Types of Constructors**

1. [**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters.

**Note:**Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

**2. Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

* **Uses of Parameterized constructor:**
  1. It is used to initialize the various data elements of different objects with different values when they are created.
  2. It is used to overload constructors.
* **Can we have more than one constructor in a class?**  
         Yes, It is called [Constructor Overloading](https://www.geeksforgeeks.org/constructor-overloading-c/).

**3. Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on [Copy Constructor](https://www.geeksforgeeks.org/copy-constructor-in-cpp/).

Whenever we define one or more non-default constructors( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

Destructors in C++

* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 26 Oct, 2020

**What is destructor?**   
Destructor is a member function which destructs or deletes an object.

**Syntax:**

**~**constructor-name();

**Properties of Destructor:**

* Destructor function is automatically invoked when the objects are destroyed.
* It cannot be declared static or const.
* The destructor does not have arguments.
* It has no return type not even void.
* An object of a class with a Destructor cannot become a member of the union.
* A destructor should be declared in the public section of the class.
* The programmer cannot access the address of destructor.

**When is destructor called?**   
A destructor function is called automatically when the object goes out of scope:   
(1) the function ends   
(2) the program ends   
(3) a block containing local variables ends   
(4) a delete operator is called

**How destructors are different from a normal member function?**   
Destructors have same name as the class preceded by a tilde (~)   
Destructors don’t take any argument and don’t return anything

**Can there be more than one destructor in a class?**   
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

**When do we need to write a user-defined destructor?**   
If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

**Can a destructor be virtual?**   
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function

C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

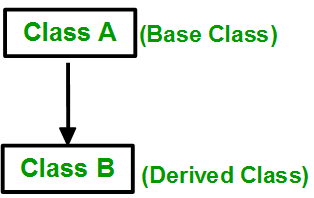
Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

A derived class inherits all base class methods with the following exceptions −

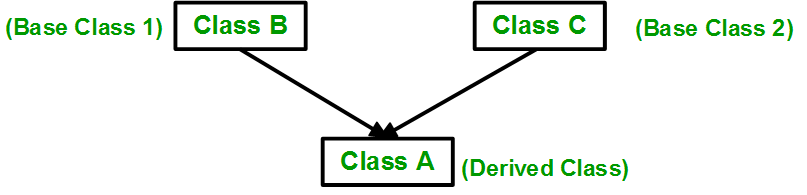
1. Constructors, destructors and copy constructors of the base class.
2. Overloaded operators of the base class.
3. The friend functions of the base class.

**Types of Inheritance in C++**

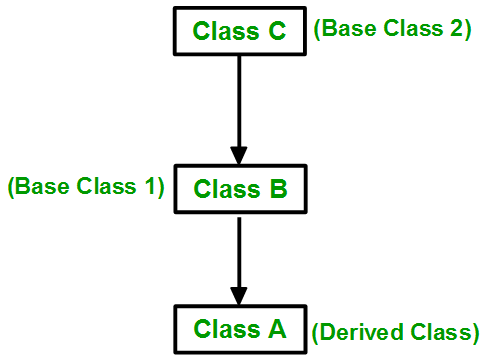
**1. Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



**2. Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



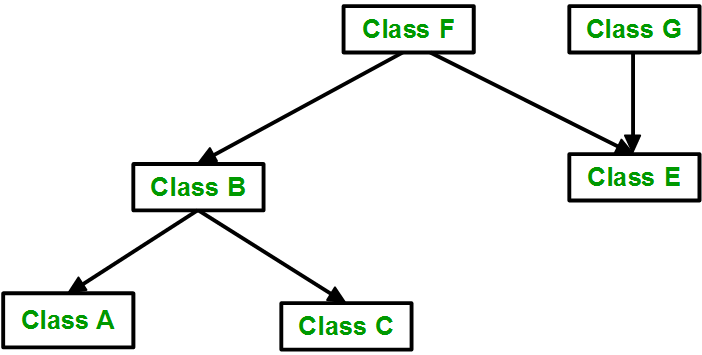
3.**Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.

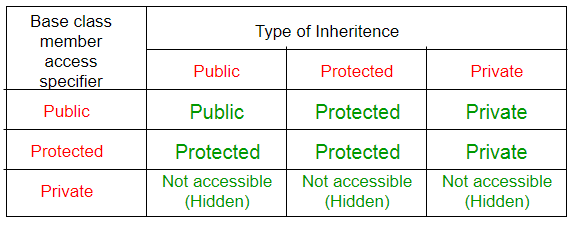


**4. Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



**5. Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.   
Below image shows the combination of hierarchical and multiple inheritance:





C++ Polymorphism

Polymorphism is an important feature of object-oriented programming. Polymorphism refers to “one name” “multiple forms”. Polymorphism means a program contains the same function name performs different functionality.

Polymorphism is classified into two types. These are:

1. Compile time polymorphism

**Compile time polymorphism:** It is achieved by function overloading or operator overloading. This is also known as **early binding or static binding.**

1. Run time polymorphism

**Run time polymorphism:** It is achieved by function overriding. This is also known as **late binding or dynamic binding .or virtual function.**

## **Real Life Example Of Polymorphism**

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

# C++ Overloading -Compile time polymorphism

C++ Overloading is a condition when two or more members have the same name with different parameter type or a different number of parameter. C++ overloading is two types:

1. **Function overloading**
2. **Operator overloading**

## C++ Function Overloading

When two or more function having the same name are different in their parameter is called function overloading.

## C++ Operator Overloading

C++ provides the functionality to redefine or overload the existing operators in C++. Operator overloading is used to perform an operation on user define the data type.

# C++ Overriding- Run time polymorphism

## C++ Function Overriding

When the base and derive class both contain the same function name and calling the function through derived object invokes derived class function called **function overriding**. Function overloading is achieved in inheritance.

Function Overloading -

In Function overloading, in which we have more than one method, with same name but different parameter list.

Compiler will invoke correct method depending on name and number of parameters.. if two methods having same name and parameter list, compiler will get confused which method to call, so it'll raise an error but if they have different parameter list, compiler will check for how many parameter have been passed to it.depending on the parameter it will call the method.

For Example.

class A {

public void mip(int n) {

System.out.println("One parameter");

public void mip(int n, int m) {

System.out.println("Two parameter");

}

class MainA {

public static void main (String args[]) {

A a1 = new A();

a1.mip(5); // here compiler goes in class A and check of mip method, if there is a method mip and it accepts one parameter compiler will invoke that method and show the output.

a1.mip(3,5); here compiler goes in class A and check of mip method, if there is a method mip and it accepts two parameter compiler will invoke that method and show the output.

}

}

Method overriding -

it's said to be occur in inheritance, when a method in subclass has same name and same parameter list as of super class.. then that method is said to be override super class method.. if method call is within the subclass then subclass method is called.. if the method call is from outside of superclass as well as subclass, in this case correct method will be executed as per the invoking object, if the object belongs to subclass, subclass method will be executed or if the object belongs to superclass then superclass method will be executed..

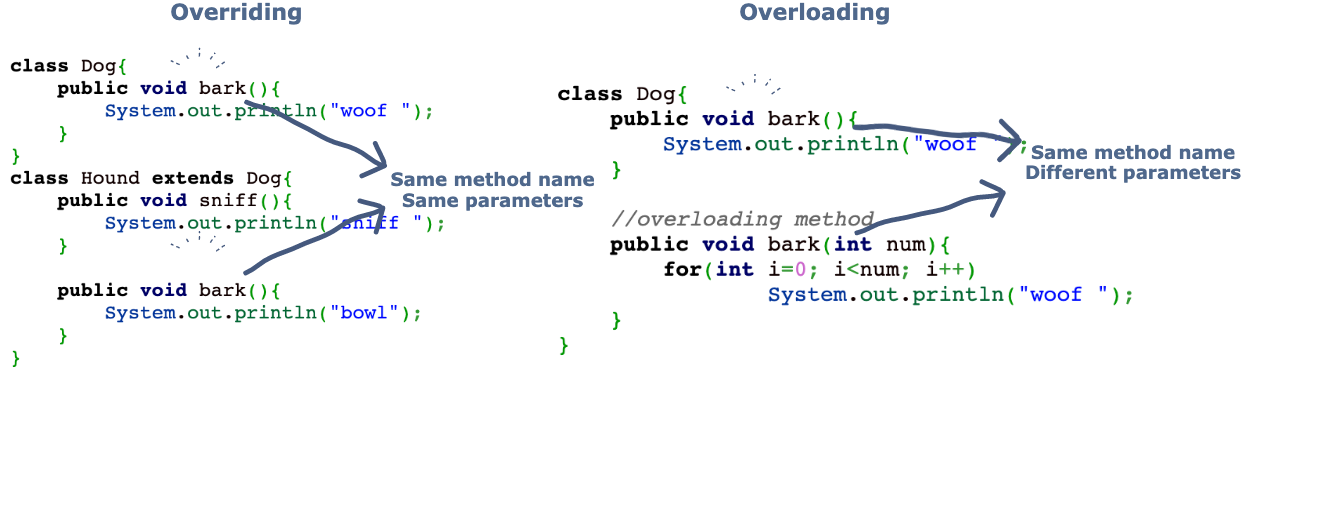
Method overriding occurs when method have same name and same signatures (parameter list), if the signatures are different, then it would become method overloading..

And two method involve in method overriding, one should be in sub class and one should be in superclass..

Method Overloading Vs Method Overriding

1. Both are used to achieve polymorphism.

2. Method overloading - same name but different parameter list and in method overriding - same name and same parameter list.



# C++ Virtual Function

A **virtual function** is such function which is declared inside the base class and redefined by the derive class. C++ uses **a virtual** keyword to make a function as a virtual function. The virtual function is always defined in the base class.

The concept of polymorphism is also achieved using the virtual function. If two functions with the same name present in base and derive class then the pointer of the base class would call the functions associated only with the base class.

# **C++ virtual function**

* A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
* It is used to tell the compiler to perform dynamic linkage or late binding on the function.
* There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
* A 'virtual' is a keyword preceding the normal declaration of a function.
* When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

## **Late binding or Dynamic linkage**

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

**Rules of Virtual Function**

* Virtual functions must be members of some class.
* Virtual functions cannot be static members.
* They are accessed through object pointers.
* They can be a friend of another class.
* A virtual function must be defined in the base class, even though it is not used.
* The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
* We cannot have a virtual constructor, but we can have a virtual destructor
* Consider the situation when we don't use the virtual keyword

A member function in the base class which is declared using virtual keyword is called virtual functions. They can be redefined in the derived class. To demonstrate the concept of virtual functions an example program is shown below

#include<iostream>

using namespace std;

class BaseClass{

public:

int var\_base=1;

virtual void display(){

cout<<"1 Dispalying Base class variable var\_base "<<var\_base<<endl;

}

};

class DerivedClass : public BaseClass{

public:

int var\_derived=2;

void display(){

cout<<"2 Dispalying Base class variable var\_base "<<var\_base<<endl;

cout<<"2 Dispalying Derived class variable var\_derived "<<var\_derived<<endl;

}

};

Copy

***Code Snippet 1: Virtual Function Example Program***

As shown in code snippet 1,

1. We created a class “BaseClass” which contains public data member “var\_base” which has the value “1” and member function “display”. The member function “display” will print the value of data member “var\_base”
2. We created another class “DerivedClass” which is inheriting “BaseClass” and contains data member “var\_derived” which has the value “2” and member function “display”. The member function “display” will print the values of data members “var\_base” and “var\_derived”

The code for the main program is shown below

int main(){

BaseClass \* base\_class\_pointer;

BaseClass obj\_base;

DerivedClass obj\_derived;

base\_class\_pointer = &obj\_derived;

base\_class\_pointer->display();

return 0;

}

Copy

***Code Snippet 2: Main Program***

As shown in code snippet 2,

1. We created a pointer “base\_class\_pointer” of the data type “Baseclass”
2. Object “obj\_base” of the data type “BaseClass” is created.
3. Object “obj\_derived” of the data type “DerivedClass” is created
4. Pointer “base\_class\_pointer” of the base class is pointing to the object “obj\_derived” of the derived class
5. The pointer “base\_class\_pointer” is pointed to the object “obj\_derived” of the derived class.
6. The function “display” is called using the pointer “base\_class\_pointer” of the base class.

The main thing to note here is that if we don’t use the “virtual” keyword with the “display” function of the base class then beside of the point that we have pointed our base call pointer to derived class object still the compiler would have called the “display” function of the base class because this is its default behavior as we have seen in the previous tutorial.

But we have used the “virtual” keyword with the “display” function of the base class to make is **virtual function**so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

#### **Virtual Functions Example in C++**

As we have seen in the previous tutorial that how virtual functions are used to implement run-time polymorphism. In this tutorial, we will see an example of virtual functions.

class CWH{

protected:

string title;

float rating;

public:

CWH(string s, float r){

title = s;

rating = r;

}

virtual void display(){}

};

Copy

**Code Snippet 1: Code with Harry Class**

As shown in a code snippet 1,

1. We created a class “CHW” which contains protected data members “title” which has a “string” data type and “rating” which has a “float” data type.
2. The class “CWH” has a parameterized constructor which takes two parameters “s” and “r” and assign their values to the data members “title” and “rating”
3. The class “CHW” has a virtual function void “display” which does nothing

class CWHVideo: public CWH

{

float videoLength;

public:

CWHVideo(string s, float r, float vl): CWH(s, r){

videoLength = vl;

}

void display(){

cout<<"This is an amazing video with title "<<title<<endl;

cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;

cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;

}

};

Copy

**Code Snippet 2: Code with Harry Video Class**

As shown in a code snippet 2,

1. We created a class “CHWVideo” which is inheriting the “CWH” class and contains private data members “videoLength” which has a “float” data type.
2. The class “CWHVideo” has a parameterized constructor which takes three parameters “s”, “r” and “vl”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “vl” will be assigned  to the data members “videoLength”
3. The class “CHWVideo” has a function void “display” which will print the values of the data members “title”, “rating” and “videoLength”

class CWHText: public CWH

{

int words;

public:

CWHText(string s, float r, int wc): CWH(s, r){

words = wc;

}

void display(){

cout<<"This is an amazing text tutorial with title "<<title<<endl;

cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;

cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;

}

};

Copy

**Code Snippet 3: Code with Harry Text Class**

As shown in a code snippet 3,

1. We created a class “CHWText” which is inheriting the “CWH” class and contains private data members “words” which has an “int” data type.
2. The class “CWHText” has a parameterized constructor which takes three parameters “s”, “r” and “wc”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “wc” will be assigned  to the data members “words”
3. The class “CHWText” has a function void “display” which will print the values of the data members “title”, “rating” and “words”

int main(){

string title;

float rating, vlen;

int words;

// for Code With Harry Video

title = "Django tutorial";

vlen = 4.56;

rating = 4.89;

CWHVideo djVideo(title, rating, vlen);

// for Code With Harry Text

title = "Django tutorial Text";

words = 433;

rating = 4.19;

CWHText djText(title, rating, words);

CWH\* tuts[2];

tuts[0] = &djVideo;

tuts[1] = &djText;

tuts[0]->display();

tuts[1]->display();

return 0;

}

Copy

**Code Snippet 4: Main Program**

As shown in a code snippet 4,

1. We created a string variable “title”, float variables “rating”, “vlen” and integer variable “words”
2. For the code with harry video class, we have assigned “Django tutorial” to the string “title”, “4.56” to the float “vlen” and “4.89” to the float “rating”.
3. An object “djVideo” is created of the data type “CWHVideo” and the variables “title”, “rating” and “vlen” are passed to it.
4. For the code with harry text class, we have assigned “Django tutorial text” to the string “title”, “433” to the integer “words” and “4.19” to the float “rating”.
5. An object “djText” is created of the data type “CWHText” and the variables “title”, “rating” and “words” are passed to it.
6. Two pointers array “tuts” is created of the “CWH” type
7. The address of the “djVideo” is assigned to “tuts[0]” and the address of the “djText” is assigned to “tuts[1]”
8. The function “display” is called using pointers “tuts[0]” and “tuts[1]”

The main thing to note here is that if we don’t use the “virtual” keyword with the “display” function of the base class then the “display” function of the base class will run.

But we have used the “virtual” keyword with the “display” function of the base class to make is a **virtual function**so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

The output of the following program is shown in figure 1

**Figure 1: Program Output**

#### **Rules for virtual functions**

1. They cannot be static
2. They are accessed by object pointers
3. Virtual functions can be a friend of another class
4. A virtual function in the base class might not be used.
5. If a virtual function is defined in a base class, there is no necessity of redefining it in the derived class

## **What are Pointers?**

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## **Using Pointers in C++**

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand.

# **Interfaces in C++ (Abstract Classes)**

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

## **C++ Abstract class**

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Shape
4. {
5. **public**:
6. **virtual** **void** draw()=0;
7. };
8. **class** Rectangle : Shape
9. {
10. **public**:
11. **void** draw()
12. {
13. cout < <"drawing rectangle..." < <endl;
14. }
15. };
16. **class** Circle : Shape
17. {
18. **public**:
19. **void** draw()
20. {
21. cout <<"drawing circle..." < <endl;
22. }
23. };
24. **int** main( ) {
25. Rectangle rec;
26. Circle cir;
27. rec.draw();
28. cir.draw();
29. **return** 0;
30. }

Output:

drawing rectangle...

drawing circle...

# **Data Abstraction in C++**

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

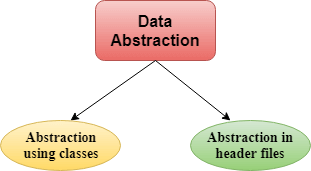
For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

* Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
* Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
* Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.
* C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

**Data Abstraction can be achieved in two ways:**

* Abstraction using classes
* Abstraction in header files.



**Abstraction using classes:** An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

**Abstraction in header files:** An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

**Access Specifiers Implement Abstraction:**

* **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
* **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

## **Advantages Of Abstraction:**

* Implementation details of the class are protected from the inadvertent user level errors.
* A programmer does not need to write the low level code.
* Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
* The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
* Internal implementation can be changed without affecting the user level code.

## Introduction to Encapsulation in C++

Encapsulation means the process of wrapping up the data and functions in a single capsule. It also safeguards the data from other classes by limiting access. Basically it hides the data. If we take a real-world example of college, we have different departments like Physics, Chemistry, and Biology, etc. A situation may arise where in, the Head of the Physics department need some information from the Biology department, he can’t access the data from that department directly. First, he should contact the Head of the Biology department, then request him to give the data. This is how encapsulation works.

### How to Use Encapsulation in C++?

In order to achieve this we have to follow the below steps:

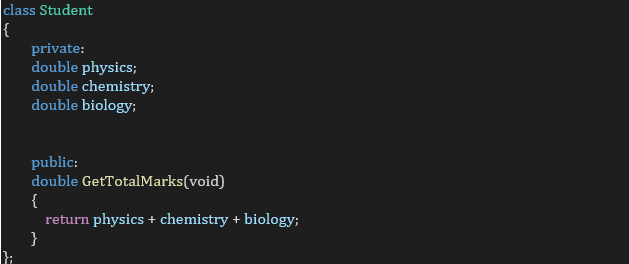
1. First, we need to make all the data members private.
2. Then public getter and setter functions should be created for each data member in such a way that, get function gets the value of data member and [the set function](https://www.educba.com/python-set-function/) sets the value of data member.

Encapsulation and data hiding can be achieved in C++, by using user-defined types called Classes. The [access specifiers](https://www.educba.com/access-specifiers-in-c-plus-plus/) in classes can be private, protected or public. By default all the items in a class are private. According to the need, we can change the access levels. Three levels of access specifiers are as below:

* **Private**: Members of the same class can access the data.
* **Public**: All the classes can access the data.
* **Protected**: Access of data is permitted to members of the same class or derived classes.

The best use of encapsulation is done only when we use either private or protected. When using public we have to make sure that, we know its proper need in the code.

To explain this we will take a look at the below class.



Here physics, chemistry, and biology are of the type double and are private variables. GetTotalMarks ( ) is a public method used to retrieve the total marks of all the three subjects. We can’t access each subject in another class because of its protection level. But we can access the method and can be used to retrieve the total marks by passing individual subject marks. We can set the marks of each subject by using the setter method, which we will look in the next example.

### Example of Encapsulation in C++ with Steps

Below is the step by step instruction to implement the encapsulation.

1) Let us consider the scenario where we need to calculate the total marks of the student by calculating the sum in three subjects i.e. Physics, Chemistry, and Biology. But the condition is such that another class

2) First, include the iostream for the input and output functionality.

 iostream for the input and output functionality-1.2

3) Use the name space std

 name space std -1.3

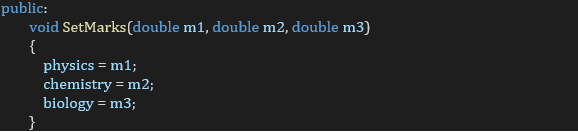
4) Write a class to hold all the data and functions. Let it be Student.



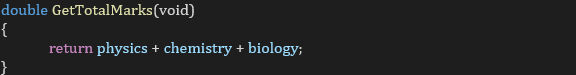
5) Declare the private variables physics, chemistry, and biology. This tells that we can’t access the variables outside this Student class. But we can access them locally and modify the value if needed. In order to modify the value, we can write a public setter function and access it in another class.



6) Write a function called SetMarks, which will be used to set the values of the private variables. Since we can’t access the private variables physics, chemistry and biology outside the student class we need this function to set these values from the external class. This method will be public as we need to access it outside the class. We need three parameters to get the values for each subject. Inside the function, we will set the value of the private variables.



7) We need one more method to get the total marks from all the three subjects by adding the marks. This method is also public. Just call this method outside the class and we will get the sum of the marks of the three subjects.



8) Now outside this class, we will write the main function.



9) Inside the main function first, we will access the class student and create an object of the class.

Encapsulation in C++-1.9

10) In the next step, we will call the function to set the marks of the three subjects. Here we will pass the marks as parameters to the function. Below is the piece of code to achieve this.

Encapsulation in C++-1.10

11) Next, we will get the total marks and print the value. And return 0 at the end.

cout

12) Can you guess the output of the code? You are right. The output is shown below.

**Output: 240**

13) Now we will analyze how we got this output. Since we could not access the private variables, we wrote a Setter function which takes values from other class and modifies the variables. Getter function fetches the variables, adds them and sends the total marks of the student. Each time when we pass three different values to setter function, we can see that total marks we get will also varies accordingly.

14) From this, we can say that [encapsulation helps](https://www.educba.com/encapsulation-in-javascript/) us to protect some of the needed data also setter and getter functions are used to alter their values. By doing this. Protection is also achieved without hampering our purpose.

### Conclusion

In this article, we have learned about how encapsulation can be achieved in C++. By doing this we can achieve the following benefits.

* Classes after encapsulation increase the readability and reduce complexity.
* It helps in protecting the data.
* The privacy of the data in the class can be changed without modifying the whole code by [using access modifiers](https://www.educba.com/access-modifiers-in-c-plus-plus/).

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and [access modifiers](https://www.geeksforgeeks.org/access-modifiers-in-c/). Look at the below program:

|  |
| --- |
| // c++ program to explain  // Encapsulation    #include<iostream>  using namespace std;    class Encapsulation  {      private:          // data hidden from outside world          int x;        public:          // function to set value of          // variable x          void set(int a)          {              x =a;          }            // function to return value of          // variable x          int get()          {              return x;          }  };    // main function  int main()  {      Encapsulation obj;        obj.set(5);        cout<<obj.get();      return 0;  } |

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

**Role of access specifiers in encapsulation**

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public**access specifier

# **C++ this Pointer**

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

* It can be used **to pass current object as a parameter to another method.**
* It can be used **to refer current class instance variable.**
* It can be used **to declare indexers.**

## **C++ this Pointer Example**

Let's see the example of this keyword in C++ that refers to the fields of current class.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Employee {
4. **public**:
5. **int** id; //data member (also instance variable)
6. string name; //data member(also instance variable)
7. **float** salary;
8. Employee(**int** id, string name, **float** salary)
9. {
10. **this**->id = id;
11. **this**->name = name;
12. **this**->salary = salary;
13. }
14. **void** display()
15. {
16. cout<<id<<"  "<<name<<"  "<<salary<<endl;
17. }
18. };
19. **int** main(**void**) {
20. Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21. Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22. e1.display();
23. e2.display();
24. **return** 0;
25. }

Output:

101 Sonoo 890000

102 Nakul 59000