**C++**

## Introduction:

## What is C++?

* C++ is a cross-platform language that can be used to create high- performance applications.
* C++ was developed by Bjarne Stroustrup, as an extension to the [C language](https://www.w3schools.com/c/index.php).
* **C++ programming language** was developed in 1980.
* C++ gives programmers a high level of control over system resources and memory.

## Why Use C++:

* C++ is one of the world's most popular programming languages.
* C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
* C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
* C++ is portable and can be used to develop applications that can be adapted to multiple platforms.
* C++ is fun and easy to learn!
* As C++ is close to [C#](https://www.w3schools.com/cs/index.php) and [Java](https://www.w3schools.com/java/default.asp), it makes it easy for programmers to switch to C++ or vice versa.

## Difference between C and C++:

* C++ was developed as an extension of [C](https://www.w3schools.com/c/index.php), and both languages have almost the same syntax.
* The main difference between C and C++ is that C++ support classes and objects, while C does not.

## Turbo C++ - Download & Installation:

There are many compilers available for C++. You need to download any one. Here, we are going to use Turbo C++. It will work for both C and C++. To install the Turbo C++ software, you need to follow following steps.

## Download Turbo C++

* Create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
* Double click on install.exe file
* Click on the tc application file located inside c:\TC\BIN to write the c program
* C++ Program
* Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

## Eg:

#include <iostream.h>

#include<conio.h>

void main() {

clrscr();

cout << "Welcome to C++ Programming.";

getch();

}

* #include<iostream.h> includes the standard input output library functions. It provides cin and cout methods for reading from input and writing to output respectively.
* #include <conio.h> includes the console input output library functions. The getch() function is defined in conio.h file.
* void main() The main() function is the entry point of every program in C++ language. The void keyword specifies that it returns no value.
* cout << "Welcome to C++ Programming." is used to print the data "Welcome to C++ Programming." on the console.
* getch() The getch() function asks for a single character. Until you press any key, it blocks the screen.

## Standard output stream (cout):

* The cout is a predefined object of ostream class.
* It is connected with the standard output device, which is usually a display screen.
* The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.

## Eg:

#include <iostream>

using namespace std;

int main( ) {

char ary[] = "Welcome to C++ ";

cout << "Value of ary is: " << ary << endl;

}

## Output:

Value of ary is: Welcome to C++

## Standard input stream (cin):

* The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard.
* The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

## Eg:

#include <iostream>

using namespace std;

int main( ) {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "Your age is: " << age << endl;

}

## Output:

Enter your age: 22

Your age is: 22

## Standard end line (endl):

The endl is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

## Eg:

#include <iostream>

using namespace std;

int main( ) {

cout << “Hello ";

cout << "World "<<endl;

cout << "End of line"<<endl;

}

## Output:

HelloWorld End of line

# Operators:

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

Types:

* [Arithmetic Operators](https://www.programiz.com/cpp-programming/operators#arithmetic)
* [Assignment Operators](https://www.programiz.com/cpp-programming/operators#assignment)
* [Relational Operators](https://www.programiz.com/cpp-programming/operators#relational)
* [Logical Operators](https://www.programiz.com/cpp-programming/operators#logical)
* [Bitwise Operator](https://www.programiz.com/cpp-programming/operators#bitwise)s
* [Other Operators](https://www.programiz.com/cpp-programming/operators#other-operators)

## Arithmetic Operators:

Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator Operation

+

Addition

-

Subtraction

\*

Multiplication

/

Division

% Modulo Operation (Remainder after division)

**Increment and Decrement Operators:**

C++ also provides increment and decrement operators: ++ and --

respectively.

 ++ increases the value of the operand by 1

* --decreases it by 1

## Assignment Operators:

In C++, assignment operators are used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | Operator | Example | Equivalent to |
|  |  |  |  |
|  | = | a = b; | a = b; |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | += | a += b; | a = a + b; |
|  |  |  |  |
|  | -= | a -= b; | a = a - b; |
|  |  |  |  |
|  | \*= | a \*= b; | a = a \* b; |
|  |  |  |  |
|  | /= | a /= b; | a = a / b; |
|  | %= | a %= b; | a = a % b; |

## Relational Operators:

A relational operator is used to check the relationship between two operands.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Operator | Meaning | Example |
|  | == | Is Equal To | 3 == 5 gives us false |
|  |  |  |  |
|  | != | Not Equal To | 3 != 5 gives us true |
|  |  |  |  |
|  | > | Greater Than | 3 > 5 gives us false |
|  | < | Less Than | 3 < 5 gives us true |
|  | >= | Greater Than or Equal To | 3 >= 5 give us false |
|  |  |  |  |
|  | <= | Less Than or Equal To | 3 <= 5 gives us true |

## Logical Operators:

Logical operators are used to check whether an expression is true or false. If the expression is true, it returns 1 whereas if the expression is false, it returns 0.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && | expression1 && expression2 | Logical AND.  True only if all the operands are true. |
| || | expression1 || expression2 | Logical OR.  True if at least one of the operands is true. |
| ! | !expression | Logical NOT.  True only if the operand is false. |

## Bitwise Operators:

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside char and int data types.

|  |  |  |
| --- | --- | --- |
|  | Operator | Description |
|  | & | Binary AND |
|  | | | Binary OR |
|  | ^ | Binary XOR |
|  | ~ | Binary One's Complement |
|  |  |  |
|  | << | Binary Shift Left |
|  |  |  |
|  | >> | Binary Shift Right |

## Other C++ Operators:

Here's a list of some other common operators available in C++.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
|  |  |  |
| sizeof | returns the size of data type | sizeof(int); // 4 |
| ?: | returns value based on the condition | string result = (5 > 0) ? "even" : "odd"; // "even" |
| & | represents memory address of the operand | &num; // address of num |
| . | accesses members of struct variables or class objects | s1.marks = 92; |
| -> | used with pointers to access the class or struct variables | ptr->marks = 92; |
| << | prints the output value | cout << 5; |
| >> | gets the input value | cin >> num; |

# Condition Statement:

In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

* if statement
* if-else statement
* nested if statement
* if-else-if ladder

## IF Statement:

The C++ if statement tests the condition. It is executed if condition is true.

## Stntax:

if(condition){

//code to be executed

}

## Eg:

#include <iostream>

using namespace std;

int main () {

int num = 10;

if (num % 2 == 0)

{

cout<<"It is even number";

}

return 0;

}

## Output:

It is even number

## IF-else Statement:

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

## Syntax:

if(condition){

//code if condition is true

}else{

//code if condition is false

}

## Example

#include <iostream> using namespace std; int main () {

int num = 11;

if (num % 2 == 0)

{

cout<<"It is even number";

}

else

{

cout<<"It is odd number";

}

return 0;

}

## Output:

It is odd number

## IF-else-if ladder Statement:

The C++ if-else-if ladder statement executes one condition from multiple statements.

## Syntax:

if(condition1){

//code to be executed if condition1 is true

}else if(condition2){

//code to be executed if condition2 is true

}

else if(condition3){

//code to be executed if condition3 is true

}

...

else{

//code to be executed if all the conditions are false

}

## Example:

#include <iostream>

using namespace std; int main () {

int num;

cout<<"Enter a number to check grade:";

cin>>num;

if (num <0 || num >100)

{

cout<<"wrong number";

}

else if(num >= 0 && num < 50){

cout<<"Fail";

}

else if (num >= 50 && num < 60)

{

cout<<"D Grade";

}

else if (num >= 60 && num < 70)

{

cout<<"C Grade";

}

else if (num >= 70 && num < 80)

{

cout<<"B Grade";

}

else if (num >= 80 && num < 90)

{

cout<<"A Grade";

}

else if (num >= 90 && num <= 100)

{

cout<<"A+ Grade";

}

}

## Output:

Enter a number to check grade:66 C Grade

## Output:

Enter a number to check grade:-2 wrong number

## switch:

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

## Syntax:

switch(expression){ case value1:

//code to be executed;

break; case value2:

//code to be executed;

break;

default:

//code to be executed if all cases are not matched;

break;

}

## Example

#include <iostream> using namespace std; int main () {

int num;

cout<<"Enter a number to check grade:";

cin>>num;

switch (num)

{

case 10: cout<<"It is 10"; break;

case 20: cout<<"It is 20"; break;

case 30: cout<<"It is 30"; break;

default: cout<<"Not 10, 20 or 30"; break;

}

}

## Output:

**Loop:**

Enter a number: 10

It is 10

**Output:**

Enter a number: 55

Not 10, 20 or 30

#include <iostream>

**Example:**

**While loop:**

while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop. **Syntax:**

while(condition){

//code to be executed

}

using namespace std;

int main() {

int i=1;

while(i<=10)

{

cout<<i <<"\n";

i++;

}

}

## Output:

1

2

3

4

5

6

7

8

9

10

## Do-While Loop:

Do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

Syntax:

do{

//code to be executed

}while(condition);

Example:

#include <iostream>

using namespace std;

int main() {

int i = 1;

do{

cout<<i<<"\n";

i++;

} while (i <= 10) ;

} Output:

1

2

3

4

5

6

7

8

9

10

## For Loop:

For loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do- while loops.

Syntax:

for(initialization; condition; incr/decr){

//code to be executed

}

Example:

#include <iostream>

using namespace std;

int main() {

for(int i=1;i<=10;i++){

cout<<i <<"\n";

}

} Output:

1

2

3

4

5

6

7

8

9

10

## Nested For Loop:

#include <iostream> using namespace std;

int main () {

for(int i=1;i<=3;i++){

for(int j=1;j<=3;j++){

cout<<i<<" "<<j<<"\n";

}

}

}

## Output:

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

## Infinite For Loop:

If we use double semicolon in for loop, it will be executed infinite times.

## Example:

#include <iostream> using namespace std;

int main () {

for (; ;)

{

cout<<"Infinitive For Loop";

}

}

## Output:

Infinitive For Loop Infinitive For Loop Infinitive For Loop Infinitive For Loop Infinitive For Loop ctrl+c

# Functions:

* A function is a block of code which only runs when it is called.
* You can pass data, known as parameters, into a function.
* Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Syntax:

void myFunction() {

// code to be executed

}

## Passing arguments:

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ 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. Consider the function swap() definition as follows.

## Syntax:

void swap(int x, int y) {

int temp;

temp = x; /\* save the value of x \*/

x = y; /\* put y into x \*/

y = temp; /\* put x into y \*/

return;

}

## Example:

#include <iostream> using namespace std;

// function declaration void swap(int x, int y);

int main () {

// local variable declaration:

int a = 100;

int b = 200;

cout << "Before swap, value of a :" << a << endl;

cout << "Before swap, value of b :" << b << endl;

// calling a function to swap the values.

swap(a, b);

cout << "After swap, value of a :" << a << endl;

cout << "After swap, value of b :" << b << endl;

return 0;

}

## Output:

Before swap, value of a :100 Before swap, value of b :200 After swap, value of a :100 After swap, value of b :200

## Function prototype:

* + A function prototype is a declaration in C and C++ of a function, its name, [parameters](https://www.thoughtco.com/definition-of-parameters-958124) and return type before its actual declaration.
  + This enables the compiler to perform more robust type checking.
  + Unlike a full function definition, the prototype terminates in a semi-colon.

## Example:

intgetsum(float \* value) ;

## Purposes:

* A function prototype ensures that calls to a function are made with the correct number and types of arguments.
* A function prototype specifies the number of arguments.
* It states the data type of each of the passed arguments.
* It gives the order in which the arguments are passed to the function.
* The function prototype tells the compiler what to expect, what to give to the function and what to expect from the function.

## Benefits:

* Prototypes save debugging time.
* Prototypes prevent problems that occur when you compile using functions that were not declared.
* When function overloading occurs, the prototypes distinguish which function version to call.

## Default argument initializers:

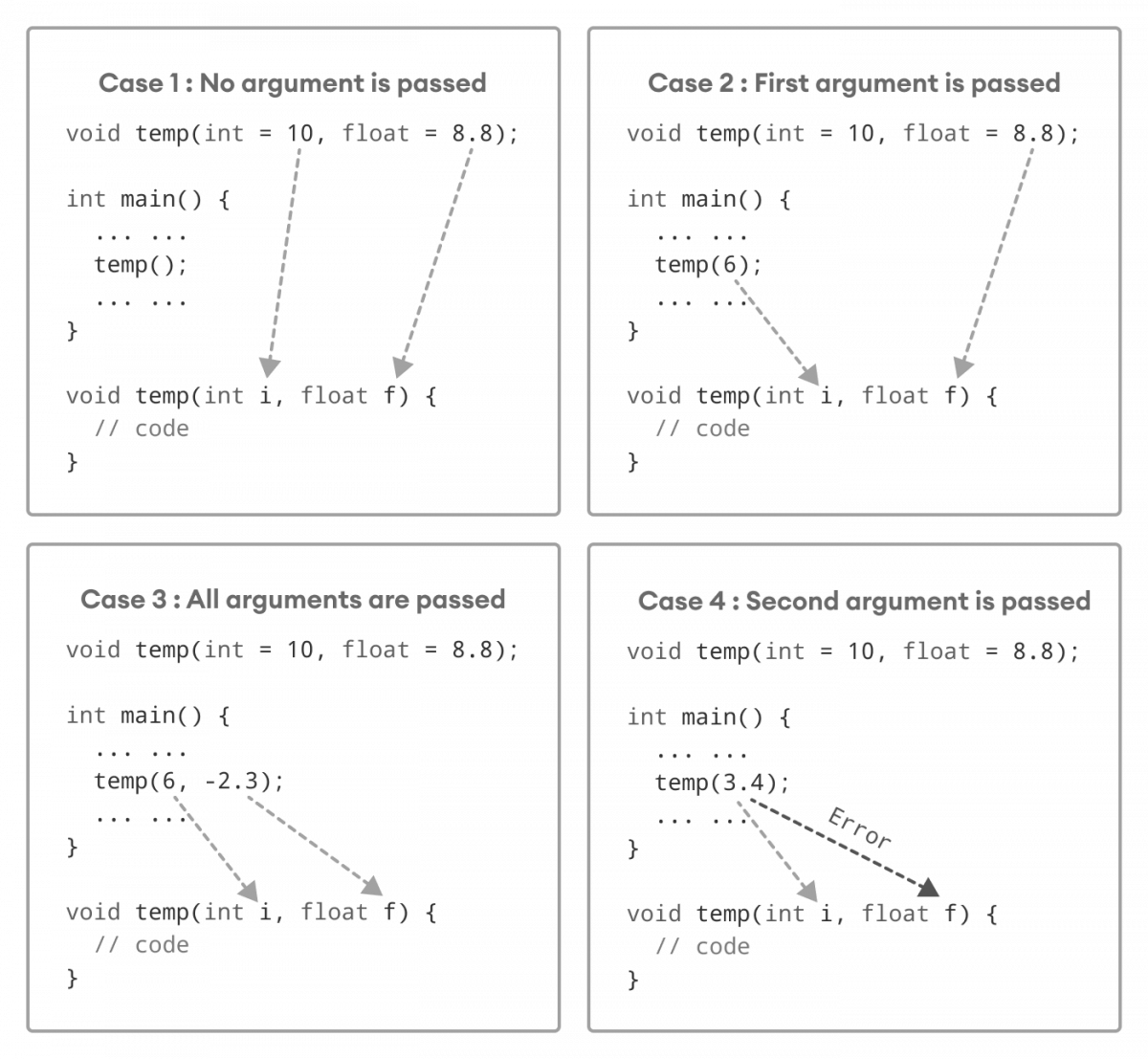
Working of default arguments

In C++ programming, we can provide default values for [function](https://www.programiz.com/cpp-programming/function) parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the

function, the default arguments are ignored.



**How default arguments work in C++:**

* We can understand the working of default arguments from the image above:
* When temp() is called, both the default parameters are used by the function.
* When temp(6) is called, the first argument becomes 6 while the default value is used for the second parameter.
* When temp(6, -2.3) is called, both the default parameters are overridden, resulting in i = 6 and f = -2.3.
* When temp(3.4) is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.

Therefore, 3.4 is passed as the first argument. Since the first argument has been defined as int, the value that is actually passed is 3.

## Inline functions:

* The inline function is powerful concept that is commonly used with classes.
* If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
* Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.
* To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function.
* The compiler can ignore the inline qualifier in case defined function is more than a line.
* A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

## Example:

#include <iostream>

using namespace std;

inline int Max(int x, int y) {

return (x > y)? x : y;

}

// Main function for the program int main() {

cout << "Max (20,10): " << Max(20,10) << endl;

cout << "Max (0,200): " << Max(0,200) << endl;

cout << "Max (100,1010): " << Max(100,1010) << endl;

return 0;

}

## Output:

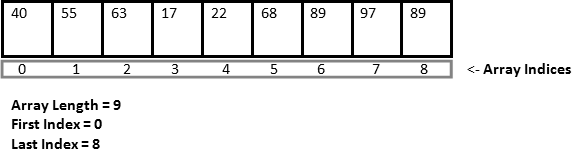
Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

# Array:

* It is a group of variables of similar data types referred to by a single element.
* Its elements are stored in a contiguous memory location.
* The size of the array should be mentioned while declaring it.
* Array elements are always counted from zero (0) onward.
* Array elements can be accessed using the position of the element in the array.
* The array can have one or more dimensions.



**Array Initialization:**

In C++, it's possible to initialize an array during declaration.

**For example,**

// declare and initialize and array int x[6] = {19, 10, 8, 17, 9, 15};

Here,

int - type of element to be stored x - name of the array

6 - size of the array

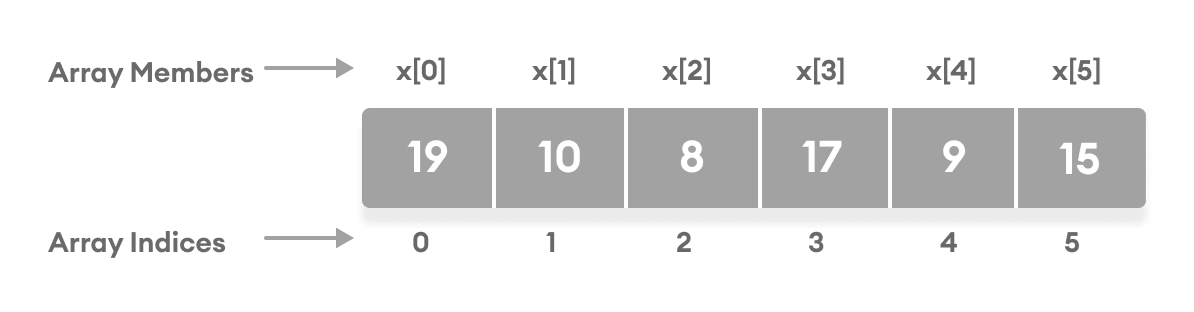
**Array Declaration:**

dataType arrayName[arraySize];

**For example**,

int x[6];

C++ Array elements and their data



Another method to initialize array during declaration:

// declare and initialize an array int x[] = {19, 10, 8, 17, 9, 15};

Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.

## Array Types:

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array 2. Multidimensional Array

## Single Dimensional Array:

We are going to create, initialize and traverse array.

#include <iostream>

using namespace std;

int main()

{

int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array

//traversing array

for (int i = 0; i < 5; i++)

{

cout<<arr[i]<<"\n";

}

}

## Output:

10

0

20

0

30

## Multidimensional Arrays:

* The multid

own as rectangular arrays in C++.

imensional array is also kn

* It can be two dimensional or three dimensional.
* The data is stored in tabular form (row ∗ column) which is also known as matrix.

Multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

## Example:

#include <iostream> using namespace std; int main()

{

int test[3][3]; //declaration of 2D array

test[0][0]=5; //initialization

test[0][1]=10;

test[1][1]=15;

test[1][2]=20;

test[2][0]=30;

test[2][2]=10;

//traversal

for(int i = 0; i < 3; ++i)

{

for(int j = 0; j < 3; ++j)

{

cout<< test[i][j]<<" ";

}

cout<<"\n"; //new line at each row

}

return 0;

}

## Output:

5 10 0

0 15 20

30 0 10

# Storage Classes:

* Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.
* Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

**Types:**

1. Automatic
2. Register
3. Static
4. External

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Storage Class** | **Keyword** | **Lifetime** | **Visibility** | **Initial Value** |
| Automatic | auto | Function Block | Local | Garbage |
| Register | register | Function Block | Local | Garbage |
| External | extern | Whole Program | Global | Zero |
| Static | static | Whole Program | Local | Zero |

**Automatic Storage Class:**

* It is the default storage class for all local variables.
* The auto keyword is applied to all local variables automatically.

{

auto int y; float y = 3.45;

}

The above example defines two variables with a same storage class,

auto can only be used within functions.

## Register Storage Class:

* The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.
* It is recommended to use register variable only for quick access such as in counter.
* Note: We can't get the address of register variable. register int counter=0;

## Static Storage Class:

* The static variable is initialized only once and exists till the end of a program.
* It retains its value between multiple functions call.
* The static variable has the default value 0 which is provided by compiler. Eg:

#include <iostream> using namespace std; void func() {

static int i=0; //static variable int j=0; //local variable

i++; j++;

cout<<"i=" << i<<" and j=" <<j<<endl;

}

int main()

{

func();

func();

func();

}

## Output:

i= 1 and j= 1

i= 2 and j= 1

i= 3 and j= 1

## External Storage Class:

**Symbols used in pointer:**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Name** | **Description** |
| & (ampersand sign) | Address operator | Determine the address of a variable. |
| ∗ (asterisk sign) | Indirection operator | Access the value of an address. |

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

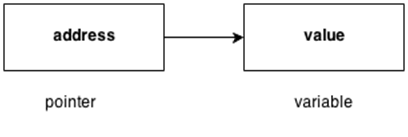
extern int counter=0;

**Pointer:**

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

## Declaring a pointer:

3) It makes you able to access any memory location in the computer's memory.



**Advantage of pointer:**

1. Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
2. We can return multiple values from function using pointer.

The pointer in C++ language can be declared using ∗ (asterisk symbol).

int ∗ a; //pointer to int char ∗ c; //pointer to char

**Pointer Example**:

//pointers printing the address and value. #include <iostream>

using namespace std;

int main()

{

int number=30; int ∗ p;

p=&number;//stores the address of number variable cout<<"Address of number variable is:"<<&number<<endl; cout<<"Address of p variable is:"<<p<<endl; cout<<"Value of p variable is:"<<\*p<<endl;

return 0;

}

## Output:

Address of number variable is:0x7ffccc8724c4 Address of p variable is:0x7ffccc8724c4 Value of p variable is:30

Pointer Program to swap 2 numbers without using 3rd variable:

#include <iostream> using namespace std; int main()

{

int a=20,b=10,∗p1=&a,∗p2=&b;

cout<<"Before swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;

∗p1=∗p1+∗p2;

∗p2=∗p1-∗p2;

∗p1=∗p1-∗p2;

cout<<"After swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;

return 0;

}

## Output:

Before swap: ∗p1=20 ∗p2=10 After swap: ∗p1=10 ∗p2=20

## Array of pointers:

Array and pointers are closely related to each other. In C++, the name

of an array is considered às a pointer, i.e., the name of an array contains the

address of an element. C++ considers the array name as the address of the first

element.

For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

**Example:**

#include <iostream> using namespace std; int main()

{

int \*ptr; // integer pointer declaration

int marks[10]; // marks array declaration

std::cout << "Enter the elements of an array :" << std::endl;

for(int i=0;i<10;i++)

{

cin>>marks[i];

}

ptr=marks; // both marks and ptr pointing to the same element..

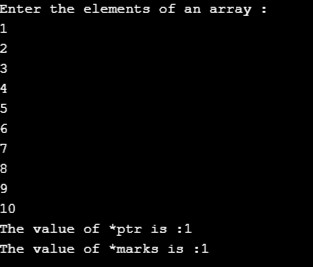
std::cout << "The value of \*ptr is :" <<\*ptr<< std::endl;

std::cout << "The value of \*marks is :" <<\*marks<<std::endl;

}

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0].

When we try to print the values of \*ptr and \*marks, then it comes out to be same.



## 

## Function Pointer in C++:

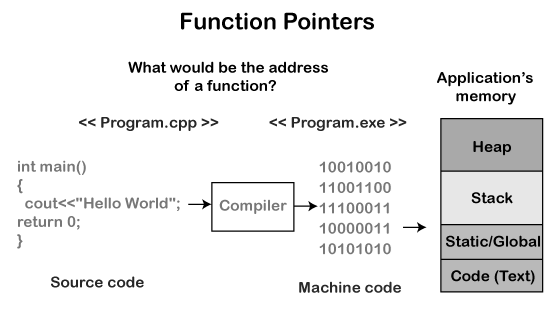
As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

Hence, it is proved that the array name stores the address of the first element of an array.

Output:

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.

What is the address of a function?



Computer only understands the low-level language, i.e., binary form. The program we write in C++ is always in high-level language, so to convert the program into binary form, we use compiler. Compiler is a program that converts source code into an executable file. This executable file gets stored in RAM. The CPU starts the execution from the main() method, and it reads the copy in RAM but not the original file.

All the functions and machine code instructions are data. This data is a bunch of bytes, and all these bytes have some address in RAM. The function pointer contains RAM address of the first instruction of a function.

## Syntax for Declarationof a function pointer:

int (\*FuncPtr) (int,int);

The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list.

In the above syntax, we first supply the return type, and then the name of the pointer, i.e., FuncPtr which is surrounded by the brackets and preceded by the pointer symbol, i.e., (\*). After this, we have supplied the parameter list (int,int).

The above function pointer can point to any function which takes two integer parameters and returns integer type value.

## Address of a function:

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.

Example**:**

#include <iostream> using namespace std; int main()

{

std::cout << "Address of a main() function is : " <<&main<< std::endl;

return 0;

}

In the above program, we are displaying the address of a main() function. To print

the address of a main() function, we have just mentioned the name of the function,

there is no bracket not parameters. Therefore, the name of the function by itself without any brackets or parameters means the address of a function.

We can use the alternate way to print the address of a function, i.e., &main.

**Another Example of Function Pointer.**

#include <iostream> using namespace std;

void printname(char \*name)

{

std::cout << "Name is :" <<name<< std::endl;

}

int main()

{

char s[20]; // array declaration

void (\*ptr)(char\*); // function pointer declaration

ptr=printname; // storing the address of printname in ptr.

std::cout << "Enter the name of the person: " << std::endl;

cin>>s;

cout<<s;

ptr(s); // calling printname() function

return 0;

}

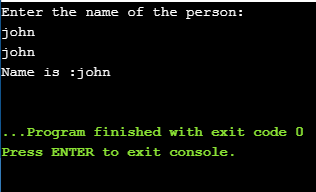
In the above program, we define the function printname() which contains the char pointer as a parameter.

We declare the function pointer, i.e., void (\*ptr)(char\*). The statement

ptr=printname means that we are assigning the address of printname() function to ptr.

Now, we can call the printname() function by using the statement ptr(s).

**Output:**



**C++ classes:**

* C++ is an object-oriented programming language.
* Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object.
* The car has attributes, such as weight and color, and methods, such as drive and brake.
* Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".
* A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

## Create a Class:

To create a class, use the class keyword:

## Example

Create a class called "MyClass":

class MyClass { // The class

public: // Access specifier

int myNum; // Attribute (int variable)

string myString; // Attribute (string variable)

};

## Example explained:

* The class keyword is used to create a class called MyClass.
* The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about [access specifiers](https://www.w3schools.com/cpp/cpp_access_specifiers.asp) later.
* Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.
* At last, end the class definition with a semicolon ;.

## Object:

* In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.
* To create an object of MyClass, specify the class name, followed by the object name.
* To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

## Example

Create an object called "myObj" and access the attributes: class MyClass { // The class

public: // Access specifier

int myNum; // Attribute (int variable)

string myString; // Attribute (string variable)

};

int main() {

MyClass myObj; // Create an object of MyClass

// Access attributes and set values

myObj.myNum = 15;

myObj.myString = "Some text";

// Print attribute values

cout << myObj.myNum << "\n";

cout << myObj.myString;

return 0;

}

**Data members and member functions:**

* "Data Member" and "Member Functions" are the new names/terms for the members of a class, which are introduced in C++ programming language.
* The variables which are declared in any class by using any [fundamental data](https://www.includehelp.com/c/basic-data-types-their-sizes.aspx) [types](https://www.includehelp.com/c/basic-data-types-their-sizes.aspx) (like int, char, float etc) or derived data type (like class, structure, pointer etc.) are known as Data Members. And the functions which are declared either in private section of public section are known as Member functions.

## There are two types of data members/member functions in C++:

* Private members
* Public members

## Private members:

* + The members which are declared in private section of the class (using private access modifier) are known as private members.
  + Private members can also be accessible within the same class in which they are declared.

## Public members:

* + The members which are declared in public section of the class (using public access modifier) are known as public members.
  + Public members can access within the class and outside of the class by using the object name of the class in which they are declared.

**Consider the example:**

class Test

{

private:

int a;

float b;

char \*name;

void getA() { a=10; }

...;

public:

int count;

void getB() { b=20; }

...;

};

Here, a, b, and name are the private data members and count is a public data member. While, getA() is a private member function and getB() is public member functions.

**C++ program that will demonstrate, how to declare, define and access data members an member functions in a class?**

#include <iostream> #include <string.h> using namespace std;

#define MAX\_CHAR 30

//class definition class person

{

//private data members

private:

char name [MAX\_CHAR];

int age;

//public member functions

public:

//function to get name and age

void get(char n[], int a)

{

strcpy(name , n);

age = a;

}

//function to print name and age

void put()

{

cout<< "Name: " << name <<endl;

cout<< "Age: " <<age <<endl;

}

};

//main function int main()

{

//creating an object of person class

person PER;

//calling member functions

PER.get("Manju Tomar", 23);

PER.put();

return 0;

}

## Output

Name: Manju Tomar

Age: 23

As we can see in the program, that private members are directly accessible within the member functions and member functions are accessible within in main() function (outside of the class) by using period (dot) operator like object\_name.member\_name;

## Inheritance:

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

**derived class (child)** - the class that inherits from another class

**base class (parent)** - the class being inherited from To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

## Example

// Base class class Vehicle {

public:

string brand = "Ford";

void honk() {

cout << "Tuut, tuut! \n" ;

}

};

// Derived class

class Car: public Vehicle {

public:

string model = "Mustang";

};

int main() {

Car myCar;

myCar.honk();

cout << myCar.brand + " " + myCar.model;

return 0;

}

## Types of inheritance:

* Single inheritance
* Multiple inheritance
* Hierarchical inheritance
* Multilevel inheritance
* Hybrid inheritance

## Single Inheritance:

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



## Eg:

#include <iostream> using namespace std;

class Account {

public:

float salary = 60000;

};

class Programmer: public Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

cout<<"Salary: "<<p1.salary<<endl;

cout<<"Bonus: "<<p1.bonus<<endl;

return 0;

}

## Output:

Salary: 60000

Bonus: 5000

## Multilevel Inheritance:

Multilevel inheritance is a process of deriving a class from another derived class.



Eg:

#include <iostream> using namespace std;

class Animal {

public:

void eat() {

cout<<"Eating..."<<endl;

}

};

class Dog: public Animal

{

public:

void bark(){

cout<<"Barking..."<<endl;

}

};

class BabyDog: public Dog

{

public:

void weep() {

cout<<"Weeping...";

}

};

int main(void) {

BabyDog d1;

d1.eat();

d1.bark();

d1.weep();

return 0;

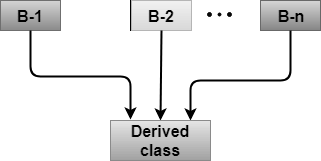
}

## Output:

Eating... Barking... Weeping...

## Multiple Inheritance:

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

class D : visibility B-1, visibility B-2, ?

{

// Body of the class;

}

## Eg:

#include <iostream> using namespace std; class A

{

protected:

int a;

public:

void get\_a(int n)

{

a = n;

}

};

class B

{

protected:

int b;

public:

void get\_b(int n)

{

b = n;

}

};

class C : public A,public B

{

public:

void display()

{

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

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

cout<<"Addition of a and b is : "<<a+b;

}

};

int main()

{

C c;

c.get\_a(10);

c.get\_b(20);

c.display();

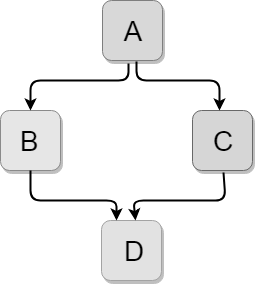
return 0;

}

## Output:

The value of a is : 10 The value of b is : 20 Addition of a and b is : 30 **Hybrid Inheritance:**

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example: #include <iostream> using namespace std;

class A

{

protected:

int a;

public:

void get\_a()

{

std::cout << "Enter the value of 'a' : " << std::endl;

cin>>a;

}

};

class B : public A

{

protected:

int b;

public:

void get\_b()

{

std::cout << "Enter the value of 'b' : " << std::endl;

cin>>b;

}

};

class C

{

protected:

int c;

public:

void get\_c()

{

std::cout << "Enter the value of c is : " << std::endl;

cin>>c;

}

};

class D : public B, public C

{

protected:

int d;

public:

void mul()

{

get\_a();

get\_b();

get\_c();

std::cout << "Multiplication of a,b,c is : " <<a\*b\*c<< std::endl;

}

};

int main()

{

D d;

d.mul();

return 0;

}

## Output:

Enter the value of 'a' :

10

Enter the value of 'b' :

20

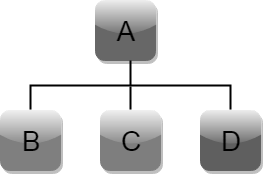
Enter the value of c is :

30

Multiplication of a,b,c is : 6000

## Hierarchical Inheritance:

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

class A

{

// body of the class A.

}

class B : public A

{

// body of class B.

}

class C : public A

{

// body of class C.

}

class D : public A

{

// body of class D.

}

Let's see a simple example:

#include <iostream> using namespace std;

class Shape // Declaration of base class.

{

public:

int a;

int b;

void get\_data(int n,int m)

{

a= n;

b = m;

}

};

class Rectangle : public Shape // inheriting Shape class

{

public:

int rect\_area()

{

int result = a\*b;

return result;

}

};

class Triangle : public Shape // inheriting Shape class

{

public:

int triangle\_area()

{

float result = 0.5\*a\*b;

return result;

}

};

int main()

{

Rectangle r;

Triangle t;

int length,breadth,base,height;

std::cout << "Enter the length and breadth of a rectangle: " << std::endl;

cin>>length>>breadth;

r.get\_data(length,breadth);

int m = r.rect\_area();

std::cout << "Area of the rectangle is : " <<m<< std::endl;

std::cout << "Enter the base and height of the triangle: " << std::endl;

cin>>base>>height;

t.get\_data(base,height);

float n = t.triangle\_area();

std::cout <<"Area of the triangle is : " << n<<std::endl;

return 0;

}

## Output:

Enter the length and breadth of a rectangle:

23

20

Area of the rectangle is : 460 Enter the base and height of the triangle:

2

5

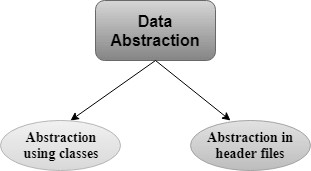
Area of the triangle is : 5

# Abstraction :

* + 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.

## 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.

## Eg:

#include <iostream> #include<math.h> using namespace std; int main()

{

int n = 4;

int power = 3;

int result = pow(n,power); // pow(n,power) is the power function

std::cout << "Cube of n is : " <<result<< std::endl;

return 0;

}

## Output:

Cube of n is : 64

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

# Encapsulation:

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Syntax:

class Box { public:

double getVolume(void) { return length \* breadth \* height;

}

private:

double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box

};

## Eg:

#include <iostream> using namespace std;

class Adder {

public:

// constructor

Adder(int i = 0) {

total = i;

}

// interface to outside world

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

// hidden data from outside world

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

## Output:

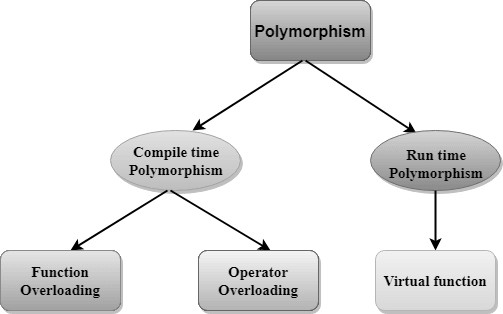
Total 60

# Polymorphism:

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

## 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.



## Compile time polymorphism:

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

## Run time polymorphism:

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

## Function Overloading:

With function overloading, multiple functions can have the same name with different parameters:

## Example

int myFunction(int x) float myFunction(float x)

double myFunction(double x, double y)

Consider the following example, which have two functions that add numbers of different type:

## Example

int plusFuncInt(int x, int y) {

return x + y;

}

double plusFuncDouble(double x, double y) {

return x + y;

}

int main() {

int myNum1 = plusFuncInt(8, 5);

double myNum2 = plusFuncDouble(4.3, 6.26);

cout << "Int: " << myNum1 << "\n";

cout << "Double: " << myNum2;

return 0;

}

## Operators Overloading:

* + Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
  + Operator overloading is used to overload or redefines most of the operators available in C++.
  + It is used to perform the operation on the user-defined data type.
  + For example, C++ provides the ability to add the variables of the user- defined data type that is applied to the built-in data types.
  + The advantage of Operators overloading is to perform different operations on the same operand.

## Operator that cannot be overloaded are as follows:

* + Scope operator (::)
  + Sizeof
  + member selector(.)
  + member pointer selector(\*)
  + ternary operator(?:)

## Syntax of Operator Overloading:

return\_type class\_name : : operator op(argument\_list)

{

// body of the function.

}

* + Where the return type is the type of value returned by the function.
  + class\_name is the name of the class.
  + operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

## Rules for Operator Overloading:

* + Existing operators can only be overloaded, but the new operators cannot be overloaded.
  + The overloaded operator contains atleast one operand of the user-defined data type.
  + We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
  + When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
  + When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

## C++ Operators Overloading Example:

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++. #include <iostream.h>

using namespace std; class Test

{

private: int num; public:

Test(): num(8){}

void operator ++() { num = num+2;

}

void Print() {

cout<<"The Count is: "<<num;

}

};

int main()

{

Test tt;

++tt; // calling of a function "void operator ++()" tt.Print();

return 0;

}

## Output:

The Count is: 10

Let's see a simple example of overloading the binary operators.

## 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.

## 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.

## Eg:

#include <iostream> using namespace std; class A

{

int x=5; public:

void display()

{

std::cout << "Value of x is : " << x<<std::endl;

}

};

class B: public A

{

int y = 10; public:

void display()

{

std::cout << "Value of y is : " <<y<< std::endl;

}

};

int main()

{

A \*a; B b;

a = &b;

a->display(); return 0;

}

## Output:

Value of x is : 5

## Virtual destructors:

Deleting a derived class object using a pointer to a base class, the base class should be defined with a virtual destructor.

**Example Code** #include<iostream> using namespace std; class b {

public: b() {

cout<<"Constructing base \n";

}

virtual ~b() { cout<<"Destructing base \n";

}

};

class d: public b { public:

d() {

cout<<"Constructing derived \n";

}

~d() {

cout<<"Destructing derived \n";

}

};

int main(void) {

d \*derived = new d(); b \*bptr = derived; delete bptr;

return 0;

}

**Output** Constructing base Constructing derived Destructing derived

Destructing base

## Virtual base classes:

Virtual classes are primarily used during multiple inheritance. To avoid, multiple instances of the same class being taken to the same class which later causes ambiguity, virtual classes are used.

## Example

#include <iostream> using namespace std; class A {

public: int a; A(){

a = 10;

}

};

class B : public virtual A {

};

class C : public virtual A {

};

class D : public B, public C {

};

int main(){

//creating class D object D object;

cout << "a = " << object.a << endl; return 0;

}

## Output

a = 10

## The New and Delete Operator:

**New Operator:**

* The new operator is used to allocate memory for a variable or any other entity like objects or arrays on a heap memory area.
* If a sufficient amount of memory is available on the heap, the new operator will initialize the memory and return the address of newly allocated memory and you can use pointers to store the address of that memory location.

## Syntax for new operator:

<data\_type pointer\_name> = new <data\_type>

Here in above syntax,

data\_type can be any inbuilt data type of C++ or any user defined data-type. pointer\_name is a pointer variable of the type ‘data\_type’.

## Eg:

int\* scaler = new int;

In the given example, scaler is a pointer of type int that points to a new memory block created at run time.

## Delete Operator:

* Since the programmer has allocated memory at runtime, it’s the responsibility of the programmer to delete that memory when not required.
* So at any point, when programmers feel a variable that has been dynamically allocated is not anymore required, they can free up the memory that it occupies in the free store or heap with the “delete” operator.
* It returns the memory to the operating system. This is also known

as memory deallocation. Also, memory will be deallocated automatically once the program ends.

## Syntax for delete operator:

delete pointer\_variable;

Here in above syntax memory has been released which was pointed by pointer\_variable

## Eg:

delete scaler

In the given example, the memory that is pointed by the variable ‘scaler’ will be deallocated.

## Friend function:

* If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
* By using the keyword friend compiler knows the given function is a friend function.
* For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

class class\_name

{

friend data\_type function\_name(argument/s); // syntax of friend function.

};

In the above declaration, the friend function is preceded by the keyword friend.

* The function can be defined anywhere in the program like a normal C++ function.
* The function definition does not use either the keyword friend or scope resolution operator.

## Characteristics of a Friend function:

* The function is not in the scope of the class to which it has been declared as a friend.
* It cannot be called using the object as it is not in the scope of that class.
* It can be invoked like a normal function without using the object.
* It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
* It can be declared either in the private or the public part.

## Example

Let's see the simple example of C++ friend function used to print the length of a box.

#include <iostream> using namespace std; class Box

{

private:

int length; public:

Box(): length(0) { }

friend int printLength(Box); //friend function

};

int printLength(Box b)

{

b.length += 10; return b.length;

}

int main()

{

Box b;

cout<<"Length of box: "<< printLength(b)<<endl; return 0;

}

## Output:

Length of box: 10

## Friend class:

A friend class can access both private and protected members of

the class in which it has been declared as friend.

## Example:

#include <iostream> using namespace std;

class A

{

int x =5;

friend class B; // friend class.

};

class B

{

public:

void display(A &a)

{

cout<<"value of x is : "<<a.x;

}

};

int main()

{

A a;

B b; b.display(a); return 0;

}

## Output:

value of x is : 5

In the above example, class B is declared as a friend inside the

class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

## C++ Copy Constructor:

A Copy constructor is an overloaded constructor used to

declare and initialize an object from another object.

## Copy Constructor is of two types:

1. **Default Copy constructor**:

The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.

## User Defined constructor:

The programmer defines the user-defined constructor.

## Templates:

* + A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming.
  + Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

## Templates can be represented in two ways:

* + Function templates
  + Class templates

## Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

## Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

## Static class members:

* + We can define class members static using static keyword.
  + When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
  + A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.

#include <iostream> using namespace std;

class Box { public:

static int objectCount;

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) { cout <<"Constructor called." << endl;

length = l; breadth = b; height = h;

// Increase every time object is created objectCount++;

}

double Volume() {

return length \* breadth \* height;

}

private:

double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box

};

// Initialize static member of class Box int Box::objectCount = 0;

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1 Box Box2(8.5, 6.0, 2.0); // Declare box2

// Print total number of objects.

cout << "Total objects: " << Box::objectCount << endl;

return 0;

}

## Output:

Constructor called. Constructor called. Total objects: 2i

## Files and Streams:

* + we are using the iostream standard library, it provides cin and cout methods for reading from input and writing to output respectively.
  + To read and write from a file we are using the standard C++ library called fstream. Let us see the data types define in fstream library is:

|  |  |
| --- | --- |
| Data Type | Description |
| fstream | It is used to create files, write information to files, and read information from files. |
| ifstream | It is used to read information from files. |
| ofstream | It is used to create files and write information to the files. |

**Example: writing to a file** #include <iostream> #include <fstream>

using namespace std; int main () {

ofstream filestream("testout.txt"); if (filestream.is\_open())

{

filestream << "Welcome to java.\n"; filestream << "C++ Tutorial.\n"; filestream.close();

}

else cout <<"File opening is fail."; return 0;

}

## Output:

The content of a text file testout.txt is set with the data: Welcome to java.

C++ Tutorial.

## Example: reading from a file:

#include <iostream> #include <fstream> using namespace std; int main () {

string srg;

ifstream filestream("testout.txt"); if (filestream.is\_open())

{

while ( getline (filestream,srg) )

{

cout << srg <<endl;

}

filestream.close();

}

else {

cout << "File opening is fail."<<endl;

}

return 0;

}

## Output:

Welcome to java.

C++ Tutorial.

## Exceptions:

* + When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
  + When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

## Try and Catch:

Exception handling in C++ consist of three keywords: try, throw and catch:

* + The try statement allows you to define a block of code to be tested for errors while it is being executed.
  + The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
  + The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
  + The try and catch keywords come in pairs:

Syntax:

try {

// Block of code to try

throw exception; // Throw an exception when a problem arise

}

catch () {

// Block of code to handle errors

}

## Eg:

#include <iostream> using namespace std;

int main() {

try {

int age = 15;

if (age >= 18) {

cout << "Access granted - you are old enough.";

} else {

throw (age);

}

}

catch (int myNum) {

cout << "Access denied - You must be at least 18 years old.\n";

cout << "Age is: " << myNum;

}

return 0;

}

## Output:

Access denied - You must be at least 18 years old. Age is: 15

\*\*\*\*\*\*\*\*\*\* THANK YOU \*\*\*\*\*\*\*\*\*\*