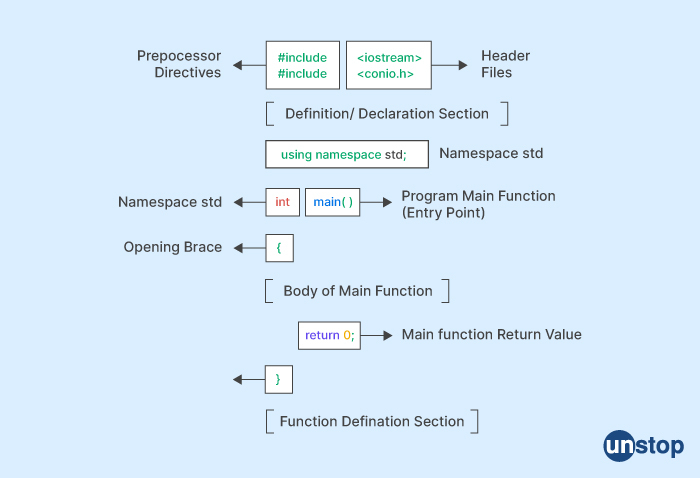
**Unit-1**

**Structure Of C++ Program**

The structure of a C++ program is made up of multiple source code files that cater to different components such as main function, member functions, class definition, headers/ standard headers, comments, variables, data types, namespaces, input/ output statements, etc.



### Pre-processor Section

Preprocessor directives are invoked to perform various pre-processing tasks, such as importing header files

### Namespace Declarations

Namespace declarations are used in defining named scopes which help in organizing and grouping related code

### Global variables

* Global variables are variables that are declared outside of any function or class and are accessible from any part of the program.

### Main Function

The main function is the entry point or the startup function for every C++ program,



**Data Types in C++ are Mainly Divided into 3 Types:**

**1. Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc. Primitive data types available in C++ are:

* Integer
* Character
* Boolean
* Floating Point
* Double Floating Point
* Valueless or Void
* Wide Character

**2. Derived Data Types:Derived Data Types** that are derived from the primitive or built-in data types are referred to as Derived Data Types. These can be of four types namely:

* Function
* Array
* Pointer
* Reference

**3. Abstract or User-Defined Data Types**  These are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

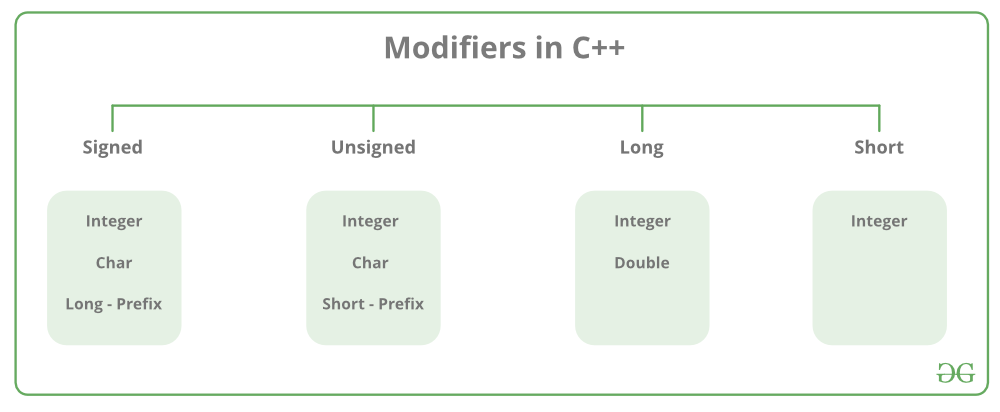
* Class
* Structure
* Union
* Enumeration
* Typedef defined Datatype

**Primitive Data Types**

**I**

* **nteger**: The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.

* **Character**: Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
* **Boolean**: Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true*or *false*. The keyword used for the Boolean data type is **bool**.
* **Floating Point**: Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
* **Double Floating Point**: Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.
* **void**: Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
* **Wide Character**: [Wide character](https://www.geeksforgeeks.org/wide-char-and-library-functions-in-c/) data type is also a character data type but this data type has a size greater than the normal 8-bit data type. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.
* **sizeof() operator:** [sizeof() operator](https://www.geeksforgeeks.org/sizeof-operator-c) is used to find the number of bytes occupied by a variable/data type in computer memory.



| **Data Type** | **Size (in bytes)** | **Range** |
| --- | --- | --- |
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |
| long long int | 8 | -(2^63) to (2^63)-1 |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| float | 4 | -3.4×10^38 to 3.4×10^38 |
| double | 8 | -1.7×10^308 to1.7×10^308 |
| long double | 12 | -1.1×10^4932 to1.1×10^4932 |
| wchar\_t | 2 or 4 | 1 wide character |

# C++ Expression

C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language.

It can also contain function calls which return values.

An expression can consist of one or more operands, zero or more operators to compute a value.

Every expression produces some value which is assigned to the variable with the help of an assignment operator.

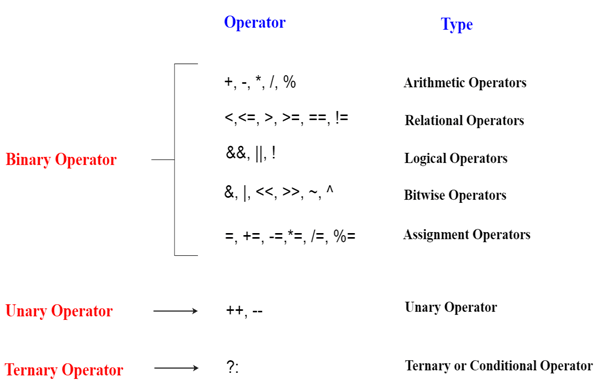
**Examples of C++ expression:**

1. (a+b) - c
2. (x/y) -z
3. 4a2 - 5b +c
4. (a+b) \* (x+y)

# C++ Operators

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operator
* Unary operator
* Ternary or Conditional Operator

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



## Precedence of Operators in C++

## The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Right to left |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == !=/td> | Right to left |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Right to left |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# Expression Evaluation

Evaluate an expression represented by a String. The expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, \*, and /. Arithmetic Expressions can be written in one of three forms:

* ***Infix Notation:*** Operators are written between the operands they operate on, e.g. 3 + 4.
* ***Prefix Notation:***Operators are written before the operands, e.g + 3 4
* ***Postfix Notation:*** Operators are written after operands.

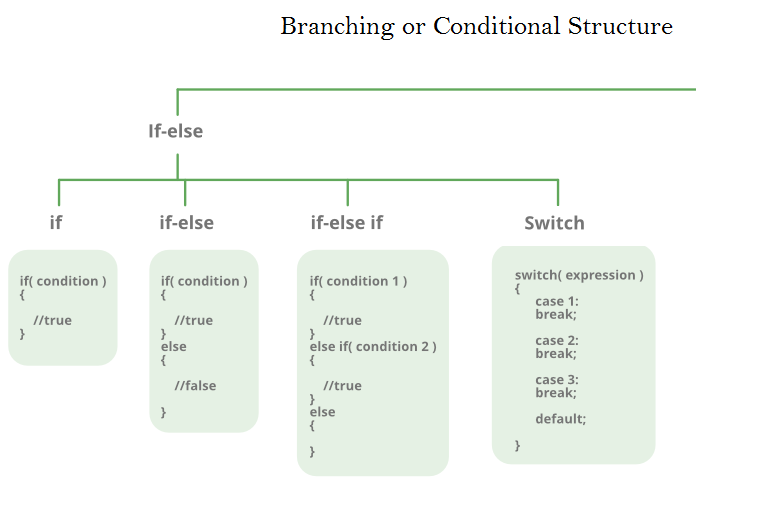
# Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as ‘automatic type conversion’.
   * Done by the compiler on its own, without any external trigger from the user.
   * Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   * All the data types of the variables are upgraded to the data type of the variable with largest data type.
   * bool -> char -> short int -> int ->
   * unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

**C ++ consists of three Conditional Statements –**



#### 1. if statement –

The if statement is a control statement that is used to test a particular condition. In this, the condition is executed only once when the condition is true.If the condition is true in the statement then the statement is Execute.

**Syntax –**  
if(condition)  
{  
statements  
}

**Example –**

|  |
| --- |
| **#include <iostream>** **using namespace std;**  **int main(){**  **int a=100, b=200;** **if (a < b)** **{** **cout<<“a is less than b”;** **}** **return 0;** **}** |

#### 2. if else statements –

The if-else statement is used to test a particular condition. If the condition is true then the if statement is executed if the condition is false then the else statement is executed.

**Syntax –**  
if(condition)  
{statement  
}else  
{statement  
}**#include <iostream>**  
**using namespace std;**  
  
**int main(){**  
  
**int a=10, b=10;**  
  
**if( a == b )**  
**{**  
  
**cout<<“a is equal to b”;**  
**}**  
**else**  
**{**  
**cout<<“a is not equal to b”;**  
**}**  
**return 0;**

Nested if statements in C++

It involve placing one if statement inside another if statement.

This allows for more complex conditions and branching logic.

Here's example:

#include <iostream>

using namespace std;

int main() {

int x = 10;

int y = 20;

if (x == 10) {

cout << "x is 10" << endl;

if (y == 20) {

cout << "Nested if: y is 20" << endl;

} else {

cout << "Nested if: y is not 20" << endl;

}

} else {

cout << "x is not 10" << endl;

}

return 0;

}

#### Switch and Break statements

#### C++ break statement

#### The break statement enables a program to skip over part of the code. A break statement terminates the smallest enclosing [for](https://codescracker.com/cpp/cpp-iteration-statements.htm#a), [while](https://codescracker.com/cpp/cpp-iteration-statements.htm#b), [do-while](https://codescracker.com/cpp/cpp-iteration-statements.htm#c), or [switch](https://codescracker.com/cpp/cpp-switch-statement.htm) statement. Execution resumes at the statement immediately following the body of the terminated statement

#include<iostream>

using namespace std;

int main()

{

int a, b, c, i;

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

{

cout<<"Enter two numbers: ";

cin>>a>>b;

if(b==0)

break;

else

c = a/b;

cout<<a<<"/"<<b<<" = "<<c<<"\n\n";

}

#### switch statements –

* This statement is also a selection statement that defines various paths for the execution of a program.
* Switch case statement has expression and some cases related to it.
* The case which matches that expression or declares variable is printed in the output.
* If no case matches the expression then it will print the default statement in the output.
* We can break after every statement, that means it will print only the statement before it.
* If you do not break, then it will print both the first and the second statement. Do not break after the default case.

**syntax:-**  
switch(variable)  
{  
case constant 1;  
statements(s);  
break:  
case constant 2;  
statement(s);  
break;  
case constant 3;  
statement(s);  
break;  
default  
statement(s);  
}

example

// Program to build a simple calculator using switch Statement

#include <iostream>

using namespace std;

int main() {

char oper;

float num1, num2;

cout << "Enter an operator (+, -, \*, /): ";

cin >> oper;

cout << "Enter two numbers: " << endl;

cin >> num1 >> num2;

switch (oper) {

case '+':

cout << num1 << " + " << num2 << " = " << num1 + num2;

break;

case '-':

cout << num1 << " - " << num2 << " = " << num1 - num2;

break;

case '\*':

cout << num1 << " \* " << num2 << " = " << num1 \* num2;

break;

case '/':

cout << num1 << " / " << num2 << " = " << num1 / num2;

break;

default:

// operator is doesn't match any case constant (+, -, \*, /)

cout << "Error! The operator is not correct";

break;

}

return 0;

}

## C++ continue statement

## Unlike the "break" statement, the "continue" statement is used when we need to skip the remaining statements of the loop's body and continue for the next iteration of the loop. For example:

## #include<iostream>

## using namespace std;

## int main()

## {

## int a, b, c, i;

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

## {

## cout<<"Enter two numbers: ";

## cin>>a>>b;

## if(b==0)

## continue;

## else

## c = a/b;

## cout<<a<<"/"<<b<<" = "<<c<<"\n\n";

## }

## return 0;

## }

## C++ goto statement

## The goto statement can transfer program control anywhere in the program. The target destination of a goto statement is marked by a LABEL.

## The target LABEL and goto must appear in the same function.

## The general form of the "goto" statement is as follows:

## LABEL:

.

.

.

goto LABEL;

where LABEL is a user-supplied [identifier](https://codescracker.com/cpp/cpp-identifiers.htm) and can appear either before or after **goto**. For example:

#include<iostream>

using namespace std;

int main()

{

int a=10, b=20;

CODESCRACKER:

if(a%2 == 0)

cout<<a<<endl;

a++;

if(a<=b)

goto CODESCRACKER;

return 0;

}

**control structure**.

Sometimes you require a set of statements to be executed a number of times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called **looping**.

C++ provides the following constructs:

* while loop
* do - while loop
* for loop

**While loop**

Syntax of while loop:

while(condition)  
{  
statement(s);  
}

A condition (loop control variable) is first evaluated.

If the condition is true, the loop body is executed and the condition is re-evaluated. Hence, the loop body is executed repeatedly as long as the condition remains true.

 Normally, the three operations listed below must be performed on the loop control variable.

1. Initialize the loop control variable ii Test the loop control variable iii Update the loop control variable
2. // C++ Program to print numbers from 1 to 5
3. #include <iostream>
4. using namespace std;
5. int main() {
6. int i = 1;
7. // while loop from 1 to 5
8. while (i <= 5) {
9. cout << i << " ";
10. ++i;
11. }
12. return 0;}

**do-while loop**

Syntax of do-while loop:

do  
{  
} while (condition);

After each execution of the loop body, the condition is true, the loop body is executed again. If the condition evaluates to false, loop exit occurs and the next program statement is executed. The loop body is always executed at least once.

One important difference between the while loop and the do-while loop is the relative ordering of the conditional test and loop body execution.

In the while loop, the loop repetition test is performed before each execution of the loop body; the loop body is not executed at all if the initial test fails.

In the do-while loop, the loop termination test is Performed after each execution of the loop body; hence, the loop body is always executed at least once.

// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {

int i = 1;

// do...while loop from 1 to 5

do {

cout << i << " ";

++i;

}

while (i <= 5);

return 0;

}

**For loop**

It is a count controlled loop in the sense that the program knows in advance how many times the loop is to be executed.

Syntax of for loop:

for(initialization;decision;increment/decrement)  
{  
statement(s);  
}

In for loop, three operations take place:

Initialization of loop control variable

Testing of loop control variable

Update the loop control variable either by incrementing or decrementing

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 5; ++i) {

cout << i << " ";

}

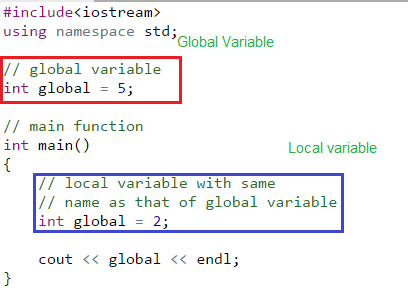
return 0;

}

scope of a variable

is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables



**Functions in c++**

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

we 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.

Create a Function

To create (often referred to as declare) a function, specify the name of the function, followed by parentheses **()**:

Syntax

void myFunction()//myFunction() is the name of the function

// void means that the function does not have a return value.

{  
  // code to be executed  
}

## Call a Function

## Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

## To call a function, write the function's name followed by two parentheses () and a semicolon ;

## Example

## Inside main, call myFunction():

## // Create a function void myFunction() {   cout << "I just got executed!"; } int main() {   **myFunction();** // call the function   return 0; }

## A C++ function consist of two parts:

## Declaration: the return type, the name of the function, and parameters (if any)

## Definition: the body of the function (code to be executed)

void **myFunction()**

 { // **declaration**  
  // the body of the function (**definition**)  
}

Example

// **Function declaration**  
void myFunction();  
// The main method  
int main() {  
  myFunction();  // **call** the function  
  return 0;  
}  
// **Function definition**  
void myFunction()

{  
  cout << "I just got executed!";  
}

**A default argument**

Is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument

#include <iostream>

using namespace std;

// A function with default arguments,

// it can be called with

// 2 arguments or 3 arguments or 4 arguments.

int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0

{

    return (x + y + z + w);

}

// Driver Code

int main()

{

    // Statement 1

    cout << sum(10, 15) << endl;

    // Statement 2

    cout << sum(10, 15, 25) << endl;

    // Statement 3

    cout << sum(10, 15, 25, 30) << endl;

    return 0;

}

# Parameter Passing Techniques in C/C++

## Lightbox

#include <stdio.h>

void func(int a, int b)

{

    a += b;

    printf("In func, a = %d b = %d\n", a, b);

}

int main(void)

{

    int x = 5, y = 7;

    // Passing parameters

    func(x, y);

    printf("In main, x = %d y = %d\n", x, y);

    return 0;

}

## Lightbox

// **call by reference**

**#include <stdio.h>**

**void swapnum(int\* i, int\* j)**

**{**

**int temp = \*i;**

**\*i = \*j;**

**\*j = temp;**

**}**

**int main(void)**

**{**

**int a = 10, b = 20;**

**return(0);**

**}**

# Templates in C++ with Examples

A **template** is a simple yet very powerful tool in C++.

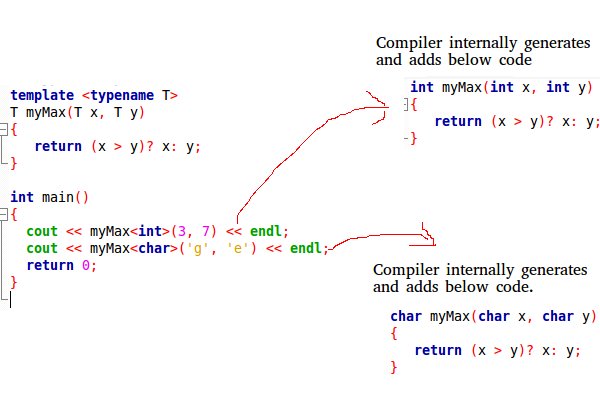
The simple idea is to pass the data type as a parameter so that we don’t need to write the same code for different data types.

Templates are expanded at compiler time.

This is like macros.

The difference is, that the compiler does type-checking before template expansion

source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



There are 3 types of template

Function template

Class template and

Variable template

Examole of **Function template**

#include <iostream>

using namespace std;

// One function works for all data types.  This would work

// even for user defined types if operator '>' is overloaded

template <typename T> T myMax(T x, T y)

{

    return (x > y) ? x : y;

}

int main()

{

    // Call myMax for int

    cout << myMax<int>(3, 7) << endl;

    // call myMax for double

    cout << myMax<double>(3.0, 7.0) << endl;

    // call myMax for char

    cout << myMax<char>('g', 'e') << endl;

    return 0;

}

## Class Templates

Class templates like function templates, class templates are useful when a class defines something that is independent of the data type

#include <iostream>

using namespace std;

 template <typename T> class Array {

private:

    T\* ptr;

    int size;

public:

    Array(T arr[], int s);

    void print();

};

template <typename T> Array<T>::Array(T arr[], int s)

{

    ptr = new T[s];

    size = s;

    for (int i = 0; i < size; i++)

        ptr[i] = arr[i];

}

template <typename T> void Array<T>::print()

{

    for (int i = 0; i < size; i++)

        cout << " " << \*(ptr + i);

    cout << endl;

}

int main()

{

    int arr[5] = { 1, 2, 3, 4, 5 };

    Array<int> a(arr, 5);

    a.print();

    return 0;

}

# C++ Variable

# A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

## Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Ex: **int** a;

**Class in C++**

Is the building block that leads to Object-Oriented programming.

It is a user-defined data type,

## A C++ class is like a blueprint for an object.

## For Example: Consider the Class of **Cars**.

## ****Defining Class and Declaring Objects****

## Lightbox

## Create an Object

In C++, an object is created from a 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;}

## What is Constructor in C++?

## These are special class members which are called by the compiler every time an object of that class is instantiated

## Constructors have the same name as the class and may be defined inside or outside the class definition

## Constructors don’t have any return type because their work is to just create and initialize an object

## The basic syntax of the constructor is given below:

**class** **class\_name**

{

**private**:

*// private members*

**public**:

*// declaring constructor*

class\_name({parameters})

{

*// constructor body*

}

};

There are four types of constructors in c++

* Default constructor
* Parameterized constructor
* Copy Constructor
* Dynamic Constructor

### 1 Default Constructor

Default constructor is also known as a zero-argument constructor

It doesn’t take any parameter.

It can be defined by the user if not then the compiler creates it on his own. Default constructor always initializes data members of the class with the same value they were defined

**#include <iostream>**

**using** **namespace** std;

**class** **Person**

{

**private**: *// declaring private class data members*

string name;

**int** age;

**public**:

*// declaring constructor*

Person()

{

cout<<"Default constructor is called"<<endl;

name = "student";

age = 12;

}

*// display function to print the class data members value*

**void** **display**()

{

cout<<"Name of current object: "<<name<<endl;

cout<<"Age of current object: "<<age<<endl;

}

};

**int** **main**()

{

*// creating object of class using default constructor*

Person obj;

*// printing class data members*

obj.display();

**return** 0;

}

### 2 Parameterized Constructor

Parameterized constructor is used to initialize data members with the values provided by the user.

This constructor is basically the upgraded version of the default constructor.

**#include <iostream>**

**using** **namespace** std;

**class** **Person**{

*// declaring private class data members*

**private**:

string name;

**int** age;

**public**:

*// declaring parameterized constructor of three different types*

Person(string person\_name)

{

cout<<"Constructor to set name is called"<<endl;

name = person\_name;

age = 12;

}

Person(**int** person\_age)

{

cout<<"Constructor to set age is called"<<endl;

name = "Student";

age = person\_age;

}

Person(string person\_name, **int** person\_age)

{

cout<<"Constructor for both name and age is called"<<endl;

name = person\_name;

age = person\_age;

}

*// display function to print the class data members value*

**void** **display**()

{

cout<<"Name of current object: "<<name<<endl;

cout<<"Age of current object: "<<age<<endl;

cout<<endl;

}

};

**int** **main**()

{

*// creating objects of class using parameterized constructor*

Person **obj1**("First person");

*// printing class data members for first object*

obj1.display();

Person **obj2**(25);

*// printing class data members for second object*

obj2.display();

Person **obj3**("Second person",15);

*// printing class data members for third object*

obj3.display();

**return** 0;

}

**3 Copy Constructor**

If we have an object of a class and we want to create its copy in a new declared object of the same class, then a copy constructor is used.

The compiler provides each class a default copy constructor and users can define it also.

It takes a single argument which is an object of the same class.

**#include <iostream>**

**using** **namespace** std;

**class** **Person**{

*// declaring private class data members*

**private**:

string name;

**int** age;

**public**:

Person(string person\_name, **int** person\_age)

{

cout<<"Constructor for both name and age is called"<<endl;

name = person\_name;

age = person\_age;

}

Person(**const** Person& obj)

{

cout<<"Copy constructor is called"<<endl;

name = obj.name;

age = obj.age;

}

*// display function to print the class data members value*

**void** **display**()

{

cout<<"Name of current object: "<<name<<endl;

cout<<"Age of current object: "<<age<<endl;

cout<<endl;

}

};

**int** **main**()

{

*// creating objects of class using parameterized constructor*

Person **obj1**("First person",25);

*// printing class data members for first object*

obj1.display();

*// creating copy of the obj1*

Person **obj2**(obj1);

*// printing class data members for second object*

obj2.display();

**return** 0;}

4 Dynamic Constructor

When memory is allocated dynamically to the data members at the runtime using a new operator, the constructor is known as the dynamic constructor.

This constructor is similar to the default or parameterized constructor; the only difference is it uses a new operator to allocate the memory.

**#include <iostream>**

**using** **namespace** std;

**class** **Person**{

*// declaring private class data members*

**private**:

**int**\* age;

**public**:

Person(**int**\* person\_age)

{

cout<<"Constructor for age is called"<<endl;

*// allocating memory*

age = **new** **int**;

age = person\_age;

}

*// display function to print the class data members value*

**void** **display**()

{

cout<<"Age of current object: "<<\*age<<endl;

cout<<endl;

}

};

**int** **main**()

{

*// creating objects of class using parameterized constructor*

**int** age = 25;

Person **obj1**(&age);

*// printing class data members for first object*

obj1.display();

**return** 0;

}

# ‘this’ pointer in C++

 ‘this’ could be the reference than the pointer

The ‘this’ pointer is passed as a hidden argument to all non static member function calls

‘this’ pointer is not available in static member functions as static member functions can be called without any object

Example

#include<iostream>

using namespace std;

/\* local variable is same as a member's name \*/

class Test

{

private:

   int x;

public:

   void setX (int x)

   {

       // The 'this' pointer is used to retrieve the object's x

       // hidden by the local variable 'x'

       this->x = x;

   }

   void print() { cout << "x = " << x << endl; }

};

int main()

{

   Test obj;

   int x = 20;

   obj.setX(x);

   obj.print();

   return 0;

}

# The C++ Standard Template Library (STL)

## The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc

## The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs

STL has 4 components:

* Algorithms
* Containers
* Functors
* Iterators

The name of a class member has *class scope* and can only be used in the following cases:

* In a member function of that class
* In a member function of a class derived from that class
* After the . (dot) operator applied to an instance of that class
* After the . (dot) operator applied to an instance of a class derived from that class, as long as the derived class does not hide the name
* After the -> (arrow) operator applied to a pointer to an instance of that class
* After the -> (arrow) operator applied to a pointer to an instance of a class derived from that class, as long as the derived class does not hide the name
* After the :: (scope resolution) operator applied to the name of a class
* After the :: (scope resolution) operator applied to a class derived from that class

# Destructors in C++

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed

Destructor destroys the class objects created by the constructor.

Destructor has the same name as their class name preceded by a tilde (~) symbol.

It is not possible to define more than one destructor.

Destructor can-not be overloaded.

Destructor neither requires any argument nor returns any value.

It is automatically called when an object goes out of scope.

Destructor release memory space occupied by the objects created by the constructor.

In destructor, objects are destroyed in the reverse of an object creation.

The syntax for defining the destructor within the class:

~ <class-name>() {

*// some instructions*

}

Example

using namespace std;

class Test {

public:

    // User-Defined Constructor

    Test() { cout << "\n Constructor executed"; }

     // User-Defined Destructor

    ~Test() { cout << "\nDestructor executed"; }

};

main()

{

    Test t;

     Return(0);

}

**Explicit type conversion** in C++

It is the process of manually converting one data type to another data type using casting operators.

It is also called as typecasting.

In some cases, it may be necessary to explicitly convert a variable from one data type to another data type to perform certain operations or assignments

The word “explicit” means ‘open’ or ‘clear’.

In explicit C++ type casting, the data type in which the value is to be converted is clearly specified in the program.

It is done by cast operator. The cast operator is unary operator. It converts the value of an expression into a value of the type specified.

Example to convert an integer to float value

|  |
| --- |
| using namespace std;    int main()  {      int a, b;      a=15;      b=2;      cout<<a/b<<endl;      cout<<a/float(b)<<endl;  } Defining a Function Template A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.  template <typename T>  T functionName(T parameter1, T parameter2, ...) {  // code  }  Adding 2 numbers using function templates  #include <iostream>  using namespace std;  template <typename T>  T add(T num1, T num2) {  return (num1 + num2);  }  int main() {  int result1;  double result2;  // calling with int parameters  result1 = add<int>(2, 3);  cout << "2 + 3 = " << result1 << endl;  // calling with double parameters  result2 = add<double>(2.2, 3.3);  cout << "2.2 + 3.3 = " << result2 << endl;  return 0;  } o/p 5, 5.5  In C++, [Pointers](https://www.programiz.com/cpp-programming/pointers) are variables that hold addresses of other variables.  Not only can a pointer store the address of a single variable,  it can also store the address of cells of an [array](https://www.programiz.com/cpp-programming/arrays).  Consider this example:  // C++ Program to display address of each element of an array  #include <iostream>  using namespace std;  int main()  {  float arr[3];  // declare pointer variable  float \*ptr;  cout << "Displaying address using arrays: " << endl;  // use for loop to print addresses of all array elements  for (int i = 0; i < 3; ++i)  {  cout << "&arr[" << i << "] = " << &arr[i] << endl;  }  // ptr = &arr[0]  ptr = arr;  cout<<"\nDisplaying address using pointers: "<< endl;  // use for loop to print addresses of all array elements  // using pointer notation  for (int i = 0; i < 3; ++i)  {  cout << "ptr + " << i << " = "<< ptr + i << endl;  }  return 0;} |