**CLASSES AND OBJECT**

* C++ is an object-oriented programming language.
* Everything in C++ is associated with classes and objects, along with its attributes and methods.
* Attributes and methods are basically **variables** and **functions** that belong 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**

* A class is a blueprint for the object.
* A class is defined in C++ using keyword **class** followed by the name of class.
* A Class is a user defined data-type which has data members and member functions.

**Example**

Create a class called "MyClass":

|  |
| --- |
| **class MyClass // The class**  **{            public:             // Access specifier     int myNum;        // Attribute (int variable)     string myString;  // Attribute (string variable) };** |

**Create an Object**

* An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
* For example: in real life, a car is an **object**. The car has **properties**, such as weight and color, and **methods**, such as drive and brake.

**Example**

* Create an object called "myObj" and access the attributes:
* 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:

|  |
| --- |
| 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; } |

**CONSTRUCTOR**

* A constructor is a member function of a class which initializes objects of a class.
* In C++, Constructor is automatically called when object (instance of class) create. It is special member function of the class.
* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).
* There are 3 types in constructors. They are,
  + [**Default Constructors**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/)
  + **Parameterized Constructors**
  + **Copy Constructor**
* **Default Constructors:** Default constructor is the constructor which doesn’t take any argument. It has no parameters.
* **Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.
* **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class.

**EXAMPLE FOR CONSTRUCTOR**

**#include<iostream.h>**

**Class const**

**{**

**Public:**

**Const()**

**{**

**Cout<<”DEFAULT CONSTRUCTOR”;**

**}**

**Const(int x)**

**{**

**Cout<<”PARAMETERIZED CONSTRUCTORS”;**

**}**

**Void display()**

**{**

**Cout<<”NORMAL CONSTRUCTOR”;**

**}**

**};**

**Void main()**

**{**

**Const c1;**

**Const c(10);**

**c.display();**

**}**

**DESTRUCTOR**

* **C++ destructor** is a special member function that is executed automatically when an object is destroyed that has been created by the [constructor](http://www.trytoprogram.com/cplusplus-programming/constructors/).
* Destructor is a member function which destructs or deletes an object.
* C++ destructors are used to de-allocate the memory that has been allocated for the object by the constructor.
* Destructors have same name as the class preceded by a tilde (~)
* Destructors don’t take any argument and don’t return anything
* Its syntax is same as constructor except the fact that it is preceded by the tilde sign.

**Syntax for Destructors**

**~class\_name() { }; //syntax of destructor**

**When is destructor called?**

* A destructor function is called automatically when the object goes out of scope:

(1) The function ends

(2) The program ends

(3) A block containing local variables ends

(4) A delete operator is called

**Example for Destructor**

**#include<iostream.h>**

**Class cn**

**{**

**Public:**

**cn()**

**{**

**Cout<<”constructor”;**

**}**

**~cn()**

**{**

**Cout<<”destructor”;**

**}**

**};**

**Void main()**

**{**

**cn c;**

**cout<<”hai students”;**

**}**

# NEW OPERATOR IN C++

* When you create a new object, memory is allocated using operator new function and then the constructor is invoked to initialize the memory.
* Here, the new operator does both the allocation and the initialization, whereas the operator new only does the allocation.
* The new operator is an **operator** which denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable. When you create an object of class using new keyword (normal new).

Operator new is a **function**that allocates raw memory and conceptually a bit similar to [malloc()](https://www.geeksforgeeks.org/malloc-vs-new/).

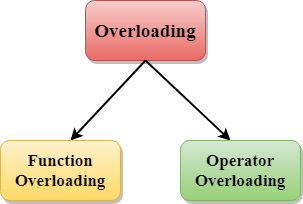
* It is the mechanism of overriding the default heap allocation logic.
* It doesn’t initializes the memory i.e constructor is not called. However, after our overloaded new returns, the compiler then automatically calls the constructor also as applicable.
* It’s also possible to [overload operator new](https://www.geeksforgeeks.org/overloading-new-delete-operator-c/) either globally, or for a specific class

**OPERATOR OVERLOADING**

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

* methods,
* constructors, and
* indexed properties

It is because these members have parameters only.



## Types of overloading in C++ are:

* Function overloading
* Operator overloading

## Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

**EXAMPLE FOR FUNCTION OVERLOADING**

**#include <iostream>**

**using namespace std;**

**class Cal {**

**public:**

**static int add(int a,int b){**

**return a + b;**

**}**

**static int add(int a, int b, int c)**

**{**

**return a + b + c;**

**}**

**};**

**int main(void) {**

**Cal C; // class object declaration.**

**cout<<C.add(10, 20)<<endl;**

**cout<<C.add(12, 20, 23);**

**return 0;**

**}**

**OUTPUT**

**30**

**55**

## 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.

**Operators 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.**  **}** |

**Example for operator Overloading**

|  |
| --- |
| **#include <iostream>**  **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** |

**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 Conversion
2. Explicit Conversion

**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.
* It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

|  |
| --- |
| **#include <iostream>**  **using namespace std;**  **int main()**  **{**  **int x = 10; // integer x**  **char y = 'a'; // character c**    **// y implicitly converted to int. ASCII**  **// value of 'a' is 97**  **x = x + y;**  **// x is implicitly converted to float**  **float z = x + 1.0;**    **cout << "x = " << x << endl**  **<< "y = " << y << endl**  **<< "z = " << z << endl;**  **return 0;**  **}** |

**OUTPUT**

|  |
| --- |
| **x = 107**  **y = a**  **z = 108** |

**Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

* **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

|  |
| --- |
| **#include <iostream>**  **using namespace std;**    **int main()**  **{**  **double x = 1.2;**    **// Explicit conversion from double to int**  **int sum = (int)x + 1;**    **cout << "Sum = " << sum;**    **return 0;**  **}** |

OUTPUT

|  |
| --- |
| **Sum = 2** |

**Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type.  
C++ supports four types of casting:

1. [Static Cast](https://www.geeksforgeeks.org/static_cast-in-c-type-casting-operators/)
2. Dynamic Cast
3. [Const Cast](https://www.geeksforgeeks.org/casting-operators-in-c-set-1-const_cast/)
4. [Reinterpret Cast](https://www.geeksforgeeks.org/reinterpret_cast-in-cpp/)

|  |
| --- |
| **#include <iostream>**  **using namespace std;**  **int main()**  **{**  **float f = 3.5;**    **// using cast operator**  **int b = static\_cast<int>(f);**    **cout << b;**  **}** |

**OUTPUT**

|  |
| --- |
| **3** |