**Module - 1 (Computer Basics)**

**4 . OOPs and C++**

**What is C++ (programming language)?**

“C++ is a statically-typed, free-form, (usually) compiled, multi-paradigm, intermediate-level general-purpose middle-level programming language.”

In simple terms, C++ is a sophisticated, efficient and a general-purpose programming language based on C. It was developed by [Bjarne Stroustrup](http://www.stroustrup.com/) in 1979.

Many of today’s operating systems, system drivers, browsers and games use C++ as their core language. This makes C++ one of the most popular languages today.

Since it is an enhanced/extended version of [C programming](https://www.programiz.com/c-programming) language, C and C++ are often denoted together as C/C++.

**History of C++**

While Bjarne Stroustrup was working in AT&T Bell Labs in 1979, he faced difficulties in analyzing UNIX kernel for distributed systems. The current languages were either too slow or too low level. So, he set forward to create a new language.

For building this language, he chose C. Why C? Because it is a general purpose language and is very efficient as well as fast in its operations.

He used his knowledge of object-oriented model from SIMULA and began working on class extensions to C. His aim was to create a language with far higher level of abstraction while retaining the efficiency of C.

This new programming language was named C withClasses, but was later renamed to C++ (++ refers to the increment operator in C).

Being a general-purpose language, C++ is undoubtedly feature-rich. Going through all the features will take you some time but, as a beginner, below are the most important features you should know.

1. **C++ is fast**  
     
   Since, C++ is an extended version of C, the C part of it is very low level.  
     
   This offers a huge boost in speed that high level languages like Python, Java don’t give you.
2. **C++  is statically typed**  
     
   C++ is a statically typed programming language.  
     
   In simple terms, C++ doesn’t allow the compiler to make assumptions about the type of data e.g. 10 is different from “10” and you have to let C++ know which one you are talking about.  
     
   This helps the compiler catch errors and bugs before execution of the program.
3. **C++ is a multi-paradigm programming language**  
     
   C++ supports at least 7 different styles of programming and gives developers the freedom to choose one at their will.  
     
   Unlike Java and Python, you don’t need to use objects to solve every task (if it’s not necessary).  
     
   You can choose the programming style that fits your use case.
4. **Object oriented programming with C++**  
     
   Object oriented programming helps you solve a complex problem intuitively.  
     
   With its use in C++, you are able to divide these complex problems into smaller sets by creating objects.
5. **Power of standard library (Standard template library - STL)**  
     
   The power of C++ extends with the use of standard libraries contained in it.  
     
   These libraries contain efficient algorithms that you use extensively while coding.  
     
   This saves ample amount of programming effort, which otherwise would have been wasted **reinventing the wheel**.

**Your first C++ program**



Now you have installed the compiler based on your OS, it’s time to write your first C++ program.

**“Hello World!”**

Your first C++ program will be a “Hello World!” program.

You might have noticed “Hello World!” being the first program while starting out with any programming language. This is because:

* It is a standard check to see whether everything is working fine or not.
* There will be very less code to start with.
* The less code makes it intuitive for the beginners to get acquainted with the language.
* The code is enough to learn the basic syntax and semantics of the language.

So, let’s get coding.

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. cout<<"Hello World!";
6. return 0;
7. }

The program prints Hello World! in the output screen.

**How the program works?**

Now, let’s dissect the above code. The code is divided into six major parts:

* #include <iostream>
* using namespace std
* ;
* int main() { }
* cout << “Hello World!”;
* return 0;

1. **What is #include <iostream>?**  
     
   If you’ve already written code in C language before, you might seen this line of code before. If you haven’t, don’t worry we’ll cover it now.  
     
   This statement includes the header file into the application so that you are able to use the operations included in them. Also, you can create your own header files and include them in your program using the #include.  
     
   **What is iostream?**

iostream is what you call the header file. It is a standard C++ input/output library file.  
It comes packaged with the compiler/IDE and contain mechanisms to get the information from the user and print same or added information to a file, screen or any other media.

**What is #include?**

The #include iostream file, into the program. This ensures that now you’re able to use the operations, iostream operations (like: taking input from user, displaying output on the screen), in the program.

1. **What is using namespace std;”?**  
     
   The statement is intuitive in itself, you are “using” the “namespace” “std” in your file.  
   We use the namespace std to make it easier to reference operations included in that namespace.  
   If we hadn’t used the namespace, we’d have written **std::cout** instead of **cout**. This tells the compiler that every **cout** is actually **std::cout**.  
     
   **What’s a namespace?**  
     
   It’s a region where your code resides. It limits or expands the scope of your code to one or more files.  
     
   **Why do you use namespace?**  
     
   Like two persons can have the same name, variables and functions in C++ can have same names as well. The use of namespace is to avoid the confusion of which variables/functions you are referencing to.  
     
   **What is std?**  
     
   std is a standard namespace used in C++.
2. **Semicolon ”;”**  
     
   Ask any C++ programmer and they will tell you at least one horror story related to the semicolon ; .  
     
   The semicolon is a terminal. It terminates a statement. When missed or incorrectly used, it will cause a lot of issues.
3. **int main() { }**  
     
   As the name suggests, it is the main function of the program. The code inside { } is called the body and is executed first when you run your C++ program.  
     
   It is one code that is mandatory in a C++ program. If you just have this line of code alone, your program will be valid.
4. **cout << “Hello World!”;**  
     
   This statement prints “Hello World!” onto the output screen.  
     
   The cout is an object of standard output stream. What this means is, it outputs/prints the data after *<<* , i.e. Hello World! into a stream (in this case, the output screen).  
     
   **What is a stream?**  
     
   Stream is basically a sequence of objects, usually bytes. It can describe files, input/output terminal, sockets, etc.  
   **What is <<?**  
     
   << is the insertion operator used to write formatted data into the stream.
5. **What is return 0;?**  
     
   This statement returns 0 ‘zero’.  
     
   This is called a return statement. It isn’t mandatory to return anything from the main() function but is rather a convention. If not return, the compiler returns a status automatically.  
     
   **Why zero in return statement?**  
     
   It denotes Exit status of the application that basically the tells system “The program worked fine.”

C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear −

[Live Demo](http://tpcg.io/JeEWd6)

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

# C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

* Default constructor
* Parameterized constructor

## C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Employee
4. {
5. **public**:
6. Employee()
7. {
8. cout<<"Default Constructor Invoked"<<endl;
9. }
10. };
11. **int** main(**void**)
12. {
13. Employee e1; //creating an object of Employee
14. Employee e2;
15. **return** 0;
16. }

Output:

Default Constructor Invoked

Default Constructor Invoked

## C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

#include <iostream>

using namespace std;

class Employee {

public:

int id;//data member (also instance variable)

string name;//data member(also instance variable)

float salary;

Employee(int i, string n, float s)

{

id = i;

name = n;

salary = s;

}

void display()

{

cout<<id<<" "<<name<<" "<<salary<<endl;

}

};

int main(void) {

Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee

Employee e2=Employee(102, "Nakul", 59000);

e1.display();

e2.display();

return 0;

}

Output:

101 Sonoo 890000

102 Nakul 59000

# C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

#### Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

## C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Employee
4. {
5. **public**:
6. Employee()
7. {
8. cout<<"Constructor Invoked"<<endl;
9. }
10. ~Employee()
11. {
12. cout<<"Destructor Invoked"<<endl;
13. }
14. };
15. **int** main(**void**)
16. {
17. Employee e1; //creating an object of Employee
18. Employee e2; //creating an object of Employee
19. **return** 0;
20. }

Output:Constructor Invoked

Constructor Invoked

Destructor Invoked

Destructor Invoked

**Polymorphism**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

[Live Demo](http://tpcg.io/LSyBTc)

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

When the above code is compiled and executed, it produces the following result −

Parent class area :

Parent class area :

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

virtual int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

Rectangle class area

Triangle class area

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

**Encapsulation**

All C++ programs are composed of the following two fundamental elements −

* **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
* **Program data** − The data is the information of the program which gets affected by the program functions.

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**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected**and **public** members. By default, all items defined in a class are private. For example −

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

};

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

[Live Demo](http://tpcg.io/lZz6R0)

#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;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

**Inheritance**

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

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

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

Base and Derived Classes

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

class derived-class: access-specifier base-class

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

Consider a base class **Shape** and its derived class **Rectangle** as follows −

[Live Demo](http://tpcg.io/ykk4hq)

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total area: 35

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

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

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

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

* **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
* **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
* **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

[Live Demo](http://tpcg.io/NPe3oQ)

#include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost {

public:

int getCost(int area) {

return area \* 70;

}

};

// Derived class

class Rectangle: public Shape, public PaintCost {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total area: 35

Total paint cost: $2450

**Data Abstraction**

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

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

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

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

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

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

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

[Live Demo](http://tpcg.io/iaAqF8)

#include <iostream>

using namespace std;

int main() {

cout << "Hello C++" <<endl;

return 0;

}

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

* Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
* Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

* Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
* The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

[Live Demo](http://tpcg.io/Dm4Yt3)

#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;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

**Template**

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work −

Function Template

The general form of a template function definition is shown here −

template <class type> ret-type func-name(parameter list) {

// body of function

}

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values −

[Live Demo](http://tpcg.io/6cvVGA)

#include <iostream>

#include <string>

using namespace std;

template <typename T>

inline T const& Max (T const& a, T const& b) {

return a < b ? b:a;

}

int main () {

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

If we compile and run above code, this would produce the following result −

Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World

Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −

template <class type> class class-name {

.

.

.

}

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack −

[Live Demo](http://tpcg.io/ixzLqn)

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace std;

template <class T>

class Stack {

private:

vector<T> elems; // elements

public:

void push(T const&); // push element

void pop(); // pop element

T top() const; // return top element

bool empty() const { // return true if empty.

return elems.empty();

}

};

template <class T>

void Stack<T>::push (T const& elem) {

// append copy of passed element

elems.push\_back(elem);

}

template <class T>

void Stack<T>::pop () {

if (elems.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

}

// remove last element

elems.pop\_back();

}

template <class T>

T Stack<T>::top () const {

if (elems.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

// return copy of last element

return elems.back();

}

int main() {

try {

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate int stack

intStack.push(7);

cout << intStack.top() <<endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

} catch (exception const& ex) {

cerr << "Exception: " << ex.what() <<endl;

return -1;

}

}

If we compile and run above code, this would produce the following result −

7

hello

Exception: Stack<>::pop(): empty stack

**Strings**

C++ provides following two types of string representations −

* The C-style character string.
* The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

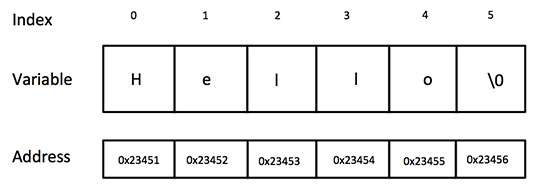
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization, then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

[Live Demo](http://tpcg.io/OzjL2L)

#include <iostream>

using namespace std;

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

cout << "Greeting message: ";

cout << greeting << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings −

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

Following example makes use of few of the above-mentioned functions −

[Live Demo](http://tpcg.io/oA6mP3)

#include <iostream>

#include <cstring>

using namespace std;

int main () {

char str1[10] = "Hello";

char str2[10] = "World";

char str3[10];

int len ;

// copy str1 into str3

strcpy( str3, str1);

cout << "strcpy( str3, str1) : " << str3 << endl;

// concatenates str1 and str2

strcat( str1, str2);

cout << "strcat( str1, str2): " << str1 << endl;

// total lenghth of str1 after concatenation

len = strlen(str1);

cout << "strlen(str1) : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

[Live Demo](http://tpcg.io/upuWFC)

#include <iostream>

#include <string>

using namespace std;

int main () {

string str1 = "Hello";

string str2 = "World";

string str3;

int len ;

// copy str1 into str3

str3 = str1;

cout << "str3 : " << str3 << endl;

// concatenates str1 and str2

str3 = str1 + str2;

cout << "str1 + str2 : " << str3 << endl;

// total length of str3 after concatenation

len = str3.size();

cout << "str3.size() : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

str3 : Hello

str1 + str2 : HelloWorld

str3.size() : 10

**I/O Streams**

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

|  |  |
| --- | --- |
| **Sr.No** | **Data Type & Description** |
| 1 | **ofstream**  This data type represents the output file stream and is used to create files and to write information to files. |
| 2 | **ifstream**  This data type represents the input file stream and is used to read information from files. |
| 3 | **fstream**  This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. |

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char \*filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

|  |  |
| --- | --- |
| **Sr.No** | **Mode Flag & Description** |
| 1 | **ios::app**  Append mode. All output to that file to be appended to the end. |
| 2 | **ios::ate**  Open a file for output and move the read/write control to the end of the file. |
| 3 | **ios::in**  Open a file for reading. |
| 4 | **ios::out**  Open a file for writing. |
| 5 | **ios::trunc**  If the file already exists, its contents will be truncated before opening the file. |

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

Similar way, you can open a file for reading and writing purpose as follows −

fstream afile;

afile.open("file.dat", ios::out | ios::in );

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void close();

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

[Live Demo](http://tpcg.io/MLhc7C)

#include <fstream>

#include <iostream>

using namespace std;

int main () {

char data[100];

// open a file in write mode.

ofstream outfile;

outfile.open("afile.dat");

cout << "Writing to the file" << endl;

cout << "Enter your name: ";

cin.getline(data, 100);

// write inputted data into the file.

outfile << data << endl;

cout << "Enter your age: ";

cin >> data;

cin.ignore();

// again write inputted data into the file.

outfile << data << endl;

// close the opened file.

outfile.close();

// open a file in read mode.

ifstream infile;

infile.open("afile.dat");

cout << "Reading from the file" << endl;

infile >> data;

// write the data at the screen.

cout << data << endl;

// again read the data from the file and display it.

infile >> data;

cout << data << endl;

// close the opened file.

infile.close();

return 0;

}

When the above code is compiled and executed, it produces the following sample input and output −

$./a.out

Writing to the file

Enter your name: Zara

Enter your age: 9

Reading from the file

Zara

9

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg( n );

// position n bytes forward in fileObject

fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject

fileObject.seekg( n, ios::end );

// position at end of fileObject

fileObject.seekg( 0, ios::end );

**Operator Overloading**

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

[Live Demo](http://tpcg.io/cR3W4M)

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

When the above code is compiled and executed, it produces the following result −

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

[Live Demo](http://tpcg.io/4Alfgt)

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | delete | delete [] |

Following is the list of operators, which can not be overloaded −

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |