Institute for Advanced Computing And Software Development (IACSD) Akurdi, Pune

Cpp set2

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

11: Virtual Functions and Abstract Class.

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a **virtual** keyword in a base class.
- The resolving of a function call is done at runtime.

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

Consider the following simple program showing the runtime behavior of virtual functions.

```
// C++ program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
virtual void print() { cout << "print base class\n"; }

void show() { cout << "show base class\n"; }
};

class derived : public base {
public:
void print() { cout << "print derived class\n"; }

void show() { cout << "show derived class\n"; }
};
```

```
int main()
{
base* bptr;
derived d;
bptr = &d;

// Virtual function, binded at runtime
bptr->print();

// Non-virtual function, binded at compile time
bptr->show();

return 0;
}
```

Pure Virtual Functions and Abstract Classes in C++

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**.For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw(). Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A **pure virtual function** (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declaration

Run Time Polymorphism

Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.

In contrast, to compile time or static polymorphism, the compiler deduces the object at run time and then decides which function call to bind to the object. In C++, runtime polymorphism is implemented using method overriding.

In this tutorial, we will explore all about runtime polymorphism in detail.

Function Overriding

Function overriding is the mechanism using which a function defined in the base class is once again defined in the derived class. In this case, we say the function is overridden in the derived class.

We should remember that function overriding cannot be done within a class. The function is overridden in the derived class only. Hence inheritance should be present for function overriding.

The second thing is that the function from a base class that we are overriding should have the same signature or prototype i.e. it should have the same name, same return type and same argument list.

```
#include <iostream>
```



```
using namespace std;
class Base
{
public:
void show_val()
{
cout << "Class::Base"<<endl;
}
};
class Derived:public Base
{
public:
void show_val() //function overridden from base
{
cout << "Class::Derived"<<endl;
}
};
int main()
{
Base b;
Derived d;
b.show_val();
d.show_val();
}
```

Output:

Class::Base
Class::Derived

In the above program, we have a base class and a derived class. In the base class, we have a function show_val which is overridden in the derived class. In the main function, we create an object each of Base and Derived class and call the show_val function with each object. It produces the desired output.

The above binding of functions using objects of each class is an example of static binding.

Now let us see what happens when we use the base class pointer and assign derived class objects as its contents.

Virtual Function

For the overridden function should be bound dynamically to the function body, we make the base class function virtual using the "virtual" keyword. This virtual function is a function that is overridden in the derived class and the compiler carries out late or dynamic binding for this function.

```
#include <iostream>
using namespace std;.
class Base
{
public:
virtual void show_val()
{
```

```
cout << "Class::Base";
}
};
class Derived:public Base
{
public:
void show_val()
{
cout << "Class::Derived";
}
};
int main() {
Base* b; //Base class pointer Derived d; //Derived class object b = &d; b->show_val();
//late Binding
}
```

Output:
Class::Derived

So in the above class definition of Base, we made show_val function as "virtual". As the base class function is made virtual, when we assign derived class object to base class pointer and call show function, the binding happens at runtime.

Thus, as the base class pointer contains derived class object, the show_val function body in the derived class is bound to function show_val and hence the output.

In C++, the overridden function in derived class can also be private. The compiler only checks the type of the object at compile time and binds the function at run time, hence it doesn't make any difference even if the function is public or private.

Note that if a function is declared virtual in the base class, then it will be virtual in all of the derived classes.

But till now, we haven't discussed how exactly virtual functions play a part in identifying correct function to be bound or in other words, how late binding actually happens.

The virtual function is bound to the function body accurately at runtime by using the concept of the **virtual table (VTABLE)** and a hidden pointer called **_vptr.**
Both these concepts are internal implementation and cannot be used directly by the program.

Working Of Virtual Table And _vptr

First, let us understand what a virtual table (VTABLE) is.

The compiler at compile time sets up one VTABLE each for a class having virtual functions as well

the classes that are derived from classes having virtual functions.

A VTABLE contains entries that are function pointers to the virtual functions that can be called by the objects of the class. There is one function pointer entry for each virtual function.

In the case of pure virtual functions, this entry is NULL. (This the reason why we cannot instantiate the abstract class).

Next entity, _vptr which is called the vtable pointer is a hidden pointer that the compiler adds to the base class. This _vptr points to the vtable of the class. All the classes derived from this base class inherit the _vptr.

Every object of a class containing the virtual functions internally stores this _vptr and is transparent to the user. Every call to virtual function using an object is then resolved using this _vptr.

Let us take an example to demonstrate the working of vtable and _vtr.

```
#include<iostream>
using namespace std;
class Base_virtual
{
public:
virtual void function1_virtual() {cout<<"Base :: function1_virtual()\n";};
virtual void function2_virtual() {cout<<"Base :: function2_virtual()\n";};
virtual ~Base_virtual(){};
};

class Derived1_virtual: public Base_virtual
{
public:
~Derived1_virtual(){};
virtual void function1_virtual() { cout<<"Derived1_virtual :: function1_virtual()\n";}; };
 int main() { Derived1_virtual *d = new Derived1_virtual; Base_virtual *b = d;
b->function1_virtual();
b->function2_virtual();
delete (b);

return (0);
}
```

Output:
Derived1_virtual :: function1_virtual()
Base :: function2_virtual()

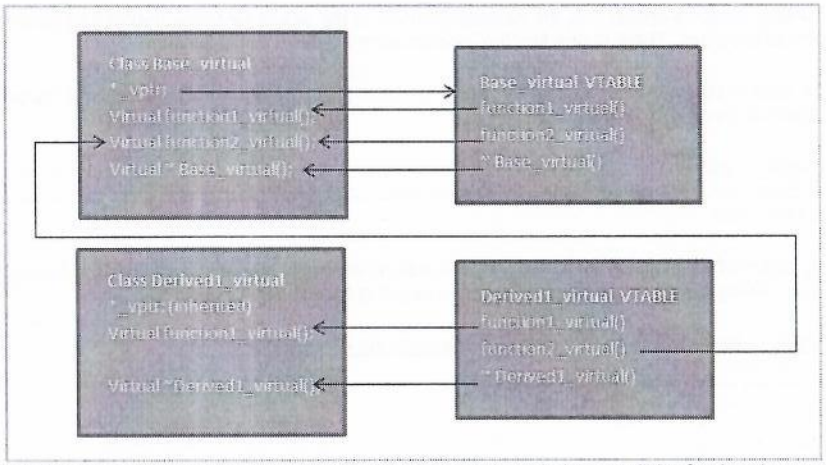
In the above program, we have a base class with two virtual functions and a virtual destructor. We have also derived a class from the base class and in that; we have overridden only one virtual function. In the main function, the derived class pointer is assigned to the base pointer.

Then we call both the virtual functions using a base class pointer. We see that the overridden function is called when it is called and not the base function. Whereas in the second case, as the function is not overridden, the base class function is called.

Now let us see how the above program is represented internally using vtable and _vptr.

As per the earlier explanation, as there are two classes with virtual functions, we will have two

vtables – one for each class. Also, _vptr will be present for the base class.



Above shown is the pictorial representation of how the vtable layout will be for the above program. The vtable for the base class is straightforward. In the case of the derived class, only

function1_virtual is overridden.

Hence we see that in the derived class vtable, function pointer for function1_virtual points to the overridden function in the derived class. On the other hand function pointer for function2_virtual

points to a function in the base class.

Thus in the above program when the base pointer is assigned a derived class object, the base pointer points to _vptr of the derived class.

So when the call b->function1_virtual() is made, the function1_virtual from the derived class is called and when the function call b->function2_virtual() is made, as this function pointer points to the base class function, the base class function is called.

Pure Virtual Functions And Abstract Class

We have seen details about virtual functions in C++ in our previous section. In C++, we can also define a "**pure virtual function**" that is usually equated to zero.
The pure virtual function is declared as shown below.

virtual return_type function_name(arg list) = 0;

The class which has at least one pure virtual function that is called an "**abstract class**". We can never instantiate the abstract class i.e. we cannot create an object of the abstract class.
This is because we know that an entry is made for every virtual function in the VTABLE (virtual table). But in case of a pure virtual function, this entry is without any address thus rendering it incomplete.

So the compiler doesn't allow creating an object for the class with incomplete VTABLE entry.

This is the reason for which we cannot instantiate an abstract class.

The below example will demonstrate Pure virtual function as well as Abstract class.

```
#include <iostream>
using namespace std;
class Base_abstract
{
public:
virtual void print() = 0;     // Pure Virtual Function
};
class Derived_class:public Base_abstract
{
public:
void print()
{
cout << "Overriding pure virtual function in derived class\n"; } }; int main()
{ // Base obj; //Compile Time Error Base_abstract *b; Derived_class d; b = &d;
b->print();
}
```

Output:
Overriding pure virtual function in the derived class

In the above program, we have a class defined as Base_abstract which contains a pure virtual function which makes it an abstract class. Then we derive a class "Derived_class" from

Base_abstract and override the pure virtual function print in it.

In the main function, not that first line is commented. This is because if we uncomment it, the compiler will give an error as we cannot create an object for an abstract class.

But the second line onwards the code works. We can successfully create a base class pointer and then we assign derived class object to it. Next, we call a print function which outputs the contents of the print function overridden in the derived class.

Let **us list some characteristics of abstract class in brief:**

- We cannot instantiate an abstract class.
- An abstract class contains at least one pure virtual function.
- Although we cannot instantiate abstract class, we can always create pointers or references to this class.
- An abstract class can have some implementation like properties and methods along with pure virtual functions.
- When we derive a class from the abstract class, the derived class should override all the pure virtual functions in the abstract class. If it failed to do so, then the derived class will also be an abstract class.

Virtual Destructors

Destructors of the class can be declared as virtual. Whenever we do upcast i.e. assigning the

derived class object to a base class pointer, the ordinary destructors can produce unacceptable results.

For Example, consider the following upcasting of the ordinary destructor.

```
#include <iostream>
using namespace std;
```

```
class Base
{
public:
~Base()
{
cout << "Base Class:: Destructor\n";
}
};
class Derived:public Base
{
public:
~Derived()
{
cout<< "Derived class:: Destructor\n";
}
};
int main()
{
Base* b = new Derived;      // Upcasting
delete b;
}
```

Output:
Base Class:: Destructor

In the above program, we have an inherited derived class from the base class. In the main, we assign an object of the derived class to a base class pointer.

Ideally, the destructor that is called when "delete b" is called should have been that of derived class but we can see from the output that destructor of the base class is called as base class pointer

points to that.

Due to this, the derived class destructor is not called and the derived class object remains intact thereby resulting in a memory leak. The solution to this is to make base class constructor virtual so that the object pointer points to correct destructor and proper destruction of objects is carried out.

The use of virtual destructor is shown in the below example.

```
#include <iostream>
using namespace std;

class Base
{
public:
virtual ~Base()
{
cout << "Base Class:: Destructor\n";
}
};
class Derived:public Base
{
public:
~Derived()
{
cout<< "Derived class:: Destructor\n";
}
};
int main()
{
Base* b = new Derived;      // Upcasting
```



```
delete b;
}
```

Output:
Derived class:: Destructor
Base Class:: Destructor

This is the same program as the previous program except that we have added a virtual keyword in front of the base class destructor. By making base class destructor virtual, we have achieved the desired output.

We can see that when we assign derived class object to base class pointer and then delete the base class pointer, destructors are called in the reverse order of object creation. This means that first the derived class destructor is called and the object is destroyed and then the base class object is destroyed.

Note: In C++, constructors can never be virtual, as constructors are involved in constructing and initializing the objects. Hence we need all the constructors to be executed completely.

Conclusion

Runtime polymorphism is implemented using method overriding. This works fine when we call the methods with their respective objects. But when we have a base class pointer and we call

overridden methods using the base class pointer pointing to the derived class objects, unexpected results occur because of static linking.

To overcome this, we use the concept of virtual functions. With the internal representation of vtables and _vptr, virtual functions help us accurately call the desired functions. In this tutorial, we have

seen in detail about runtime polymorphism used in C++.

With this, we conclude our tutorials on object-oriented programming in C++. We hope this tutorial will be helpful to gain a better and thorough understanding of object-oriented programming concepts in C++.

The casting operator ***dynamic_cast*** in C++ is used to change a pointer or reference from one type

to another type. A polymorphic type can be safely downcast at runtime using the ***dynamic_cast operator***. The class hierarchy of polymorphic types includes at least one virtual function.

Syntax:

The syntax for using dynamic_cast is as follows:

1. **dynamic_cast**<new_type>(expression)

   Where ***"expression"*** is the expression that is being cast and ***"new_type"*** denotes the type to which the ***expression*** is being cast. A ***pointer*** or reference to a polymorphic type must be present in the expression.

To confirm that the object being addressed to or referenced is true of the specified ***new_type***, the ***dynamic_cast*** operator will perform a runtime type check. The dynamic_cast operator will return a ***null pointer*** (for a pointer cast) or throw a ***std::bad_cast exception*** (for a reference cast) if the conversion is impossible.

Example:

A code snippet that demonstrates how to use **"dynamic_cast"** to perform a downcast:

```
1. class Animal {
2. public:
3. virtual ~Animal() {}
4. };
5.
6. class Dog : public Animal {
7. public:
8. void bark() {
9. std::cout<< "Woof woof!" <<std::endl;
10.}
11.};
12.
13.class Cat : public Animal {
14.public:
15.void meow() {
16.std::cout<< "Meow meow!" <<std::endl;
17.}
18.};
19.
20.int main() {
21.Animal* animalPtr = new Dog;
22.Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);
23.if (dogPtr != nullptr) {
24.dogPtr->bark();
25.}
26.delete animalPtr;
27.return 0;
28.}
```

Output:

```
Woof woof!
```


Static Cast

This is the simplest type of cast that can be used. It is a **compile-time cast**. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

Syntax of static_cast

`static_cast <dest_type> (source);`

The return value of static_cast will be of **dest_type.**

Example of static_cast

Below is the C++ program to implement static_cast:

- C++

```
// C++ Program to demonstrate

// static_cast

#include <iostream>

using namespace std;



// Driver code

int main()

{

float f = 3.5;



// Implicit type case

// float to int

int a = f;

cout << "The Value of a: " << a;
```

```
// using static_cast for float to int

int b = static_cast<int>(f);

cout << "\nThe Value of b: " << b;

}
```

Output

```
The Value of a: 3
The Value of b: 3
```

static_cast for Inheritance in C++

```
// C++ Program to demonstrate
// static_cast in inheritance
#include <iostream>
using namespace std;
class Base
{};
class Derived : public Base
{};

// Driver code
int main()
{
Derived d1;

// Implicit cast allowed
Base* b1 = (Base*)(&d1);

// upcasting using static_cast
Base* b2 = static_cast<Base*>(&d1);
return 0;
}
```

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student* const this'.

```
#include <iostream>
```



```
using namespace std;

class student
{
private:
int roll;
public:
// constructor
student(int r):roll(r) {}

// A const function that changes roll with the help of const_cast
void fun() const
{
( const_cast <student*> (this) )->roll = 5;
}

int getRoll() { return roll; }
};

int main(void)
{
student s(3);
cout << "Old roll number: " << s.getRoll() << endl;

s.fun();

cout << "New roll number: " << s.getRoll() << endl;

return 0;
}
```

```
Old roll number: 3
New roll number: 5
```

reinterpret_cast in C++ | Type Casting operators

reinterpret_cast is a type of casting operator used in C++.

- It is used to convert a pointer of some data type into a pointer of another data type,
- even if the data types before and after conversion are different.
- It does not check if the pointer type and data pointed by the pointer is same or not.

- **Syntax :**

```
data_type *var_name =
reinterpret_cast <data_type *>(pointer_variable);
```

```
// CPP program to demonstrate working of
// reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
int* p = new int(65);
char* ch = reinterpret_cast<char*>(p);
cout << *p << endl;
cout << *ch << endl;
cout << p << endl;
cout << ch << endl;
return 0;
}
```

Purpose for using reinterpret_cast

1. reinterpret_cast is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).
2. It can typecast any pointer to any other data type.
3. It is used when we want to work with bits.
4. If we use this type of cast then it becomes a non-portable product. So, it is suggested not to use this concept unless required.
5. It is only used to typecast any pointer to its original type.
6. Boolean value will be converted into integer value i.e., 0 for false and 1 for true.

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows –

```
class Box {
public:
// pure virtual function
virtual double getVolume() = 0;

private:
double length;     // Length of a box
double breadth;    // Breadth of a box
```



```
double height;     // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface.** Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC.

Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes.**

Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()** –

Live Demo

```
#include <iostream>

using namespace std;

// Base class
class Shape {
public:
// pure virtual function providing interface framework.
virtual int getArea() = 0;
void setWidth(int w) {
width = w;
}

void setHeight(int h) {
height = h;
}

protected:
int width;
int height;
};

// Derived classes
class Rectangle: public Shape {
public:
```

```
int getArea() {
return (width * height);
}
};

class Triangle: public Shape {
public:
int getArea() {
return (width * height)/2;
}
};

int main(void) {
Rectangle Rect;
Triangle  Tri;

Rect.setWidth(5);
Rect.setHeight(7);

// Print the area of the object.
cout << "Total Rectangle area: " << Rect.getArea() << endl;

Tri.setWidth(5);
Tri.setHeight(7);

// Print the area of the object.
cout << "Total Triangle area: " << Tri.getArea() << endl;

return 0;
}
```

Exception Handling

Exception handling is a mechanism that separates code that detects and handles exceptional circumstances

from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions
- Exception specifications

C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw** – A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** – A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

```
try {
// protected code
} catch( ExceptionName e1 ) {
// catch block
} catch( ExceptionName e2 ) {
// catch block
} catch( ExceptionName eN ) {
// catch block
}
```

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs –

```
double division(int a, int b) {
if( b == 0 ) {
throw "Division by zero condition!";
}
return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword

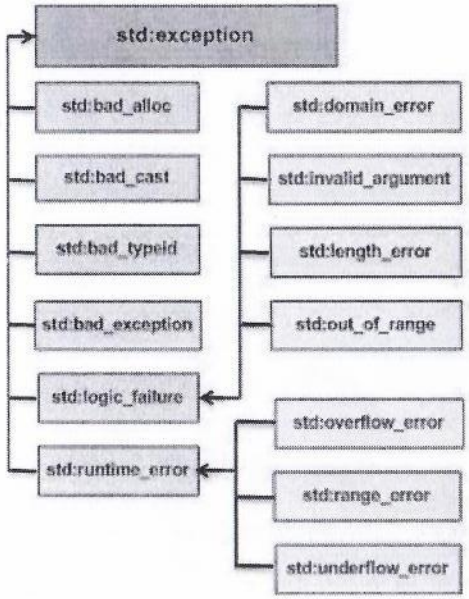
catch.

```
try {
// protected code
} catch( ExceptionName e ) {
// code to handle ExceptionName exception
}
```

```
#include <iostream>
using namespace std;

double division(int a, int b) {
if( b == 0 ) {
throw "Division by zero condition!";
}
return (a/b);
}

int main () {
int x = 50;
int y = 0;
double z = 0;

try {
z = division(x, y);
cout << z << endl;
} catch (const char* msg) {
cerr << msg << endl;
}

return 0;
}
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –



Managing Console I/O Operations

Every program takes some data as input and generates processed data as an output following the familiar input process output cycle. It is essential to know how to provide the input data and present the results in the desired form. The use of the **cin** and **cout** is already known with the operator **>>** and **<<** for the input and output operations. In this article, we will discuss how to control the way the output is printed.

C++ supports a rich set of I/O functions and operations. These functions use the advanced features of C++ such as classes, derived classes, and virtual functions. It also supports all of C's set of I/O functions and therefore can be used in C++ programs, but their use should be restrained due to two reasons.

1. I/O methods in C++ support the concept of OOPs.
2. I/O methods in C cannot handle the user-define data type such as class and object.

C++ Stream

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the **stream.**

- A stream is a sequence of bytes.
- The source stream that provides the data to the program is called the input stream.
- The destination stream that receives output from the program is called the output stream.

- The data in the input stream can come from the keyboard or any other input device.
- The data in the output stream can go to the screen or any other output device.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include **cin** and **cout**. It is known that **cin** represents the input stream connected to the standard input device (usually the keyboard) and **cout** represents the output stream connected to the standard output device (usually the screen).

C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. The hierarchy of stream classes used for input and output operations is with the console unit. These classes are declared in the **header file iostream**. This file should be included in all the programs that communicate with the console unit.

```
#include <iostream>
using namespace std;

int main()
{

cout << " This is my first"
" Blog on the gfg";
return 0;
}
```

Output:

```
This is my first Blog on the gfg
```

The **ios** are the base class for **istream** (input stream) and **ostream** (output stream) which are in turn base classes for **iostream** (input/output stream). The class **ios** are declared as the virtual base class so that only one copy of its members is inherited by the iostream.
The class **ios** provide the basics support for formatted and unformatted I/O operations. The class istream provides the facilities formatted and unformatted input while the class ostream (through inheritance) provides the facilities for formatted output.
The class iostream provides the facilities for handling both input and output streams. Three classes add an assignment to these classes:

- **istream_withassign**
- **ostream_withassign**
- **iostream_withassign**

Unformatted Input/Output functions

Unformatted I/O functions are used only for character data type or character array/string and cannot be used for any other datatype. These functions are used to read single input from the user at the console and it allows to display the value at the console.

Why they are called unformatted I/O?

These functions are called unformatted I/O functions because we cannot use format specifiers in these functions and hence, cannot format these functions according to our

needs.

The following unformatted I/O functions will be discussed in this section-

1. getch()
2. getche()
3. getchar()
4. putchar()
5. gets()
6. puts()
7. putch()

getch():

getch() function reads a single character from the keyboard by the user but doesn't display that character on the console screen and immediately returned without pressing enter key. This function is declared in conio.h(header file). getch() is also used for hold the screen.

Syntax:

getch();
or
variable-name = getch();

```
/ C program to implement
// getch() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
printf("Enter any character: ");

// Reads a character but
// not displays
getch();

return 0;
}
```

getche():

getche() function reads a single character from the keyboard by the user and displays it on the console screen and immediately returns without pressing the enter key. This function is declared in conio.h(header file).

Syntax:

getche();
or
variable_name = getche();

```
// C program to implement
// the getche() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
```

```
{
printf("Enter any character: ");

// Reads a character and
// displays immediately
getche();
return 0;
}
```

getchar():
The getchar() function is used to read only a first single character from the keyboard whether multiple characters is typed by the user and this function reads one character at one time until and unless the enter key is pressed. This function is declared in stdio.h(header file)

Variable-name = getchar();

```
// C program to implement
// the getchar() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
// Declaring a char type variable
char ch;

printf("Enter the character: ");

// Taking a character from keyboard
ch = getchar();

// Displays the value of ch
printf("%c", ch);
return 0;
}
```

gets():
gets() function reads a group of characters or strings from the keyboard by the user and these characters get stored in a character array. This function allows us to write space-separated texts or strings. This function is declared in stdio.h(header file).

Syntax:
char str[length of string in number]; //Declare a char type variable of any length
gets(str);

```
// C program to implement
// the gets() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
// Declaring a char type array
// of length 50 characters
char name[50];
```



```
printf("Please enter some texts: ");

// Reading a line of character or
// a string
gets(name);

// Displaying this line of character
// or a string
printf("You have entered: %s",
name);
return 0;
}
```

C++ helps you to format the I/O operations like determining the number of digits to be displayed after the decimal point, specifying number base etc.

Example:

- If we want to add + sign as the prefix of out output, we can use the formatting to do so:
- `stream.setf(ios::showpos)`

  `If input=100, output will be +100`

- If we want to add trailing zeros in out output to be shown when needed using the formatting:
- `stream.setf(ios::showpoint)`

  `If input=100.0, output will be 100.000`

Note: Here, stream is referred to the streams defined in c++ like cin, cout, cerr, clog. There are two ways to do so:

1. Using the ios class or various ios member functions.
2. Using manipulators(special functions)

Formatting using the ios members:
The stream has the format flags that control the way of formatting it means Using this setf function, we can set the flags, which allow us to display a value in a particular format. The ios class declares a bitmask enumeration called fmtflags in which the values(showbase, showpoint, oct, hex etc) are defined. These values are used to set or clear the format flags.

Few standard ios class functions are:

1. **width():** The width method is used to set the required field width. The output will be displayed in the given width
2. **precision():** The precision method is used to set the number of the decimal point to a float value
3. **fill():** The fill method is used to set a character to fill in the blank space of a field
4. **setf():** The setf method is used to set various flags for formatting output
5. **unsetf():** The unsetf method is used To remove the flag setting

```
#include<bits/stdc++.h>
using namespace std;

// The width() function defines width
// of the next value to be displayed
```

```
// in the output at the console.
void IOS_width()
{
cout << "-------------------------\n";
cout << "Implementing ios::width\n\n";

char c = 'A';

// Adjusting width will be 5.
cout.width(5);

cout << c <<"\n";


int temp = 10;

// Width of the next value to be
// displayed in the output will
// not be adjusted to 5 columns.
cout<<temp;
cout << "\n-------------------------\n";
}

void IOS_precision()
{
cout << "\n-------------------------\n";
cout << "Implementing ios::precision\n\n";
cout << "Implementing ios::width";
cout.setf(ios::fixed, ios::floatfield);
cout.precision(2);
cout<<3.1422;
cout << "\n-------------------------\n";
}

// The fill() function fills the unused
// white spaces in a value (to be printed
// at the console), with a character of choice.
void IOS_fill()
{
cout << "\n-------------------------\n";
cout << "Implementing ios::fill\n\n";
char ch = 'a';

// Calling the fill function to fill
// the white spaces in a value with a
```



```
// character our of choice.
cout.fill('*');

cout.width(10);
cout<<ch <<"\n";


int i = 1;

// Once you call the fill() function,
// you don't have to call it again to
// fill the white space in a value with
// the same character.
cout.width(5);
cout<<i;
cout << "\n-------------------------\n";
}

void IOS_setf()
{
cout << "\n-------------------------\n";
cout << "Implementing ios::setf\n\n";
int val1=100,val2=200;
cout.setf(ios::showpos);
cout<<val1<<" "<<val2;
cout << "\n-------------------------\n";
}

void IOS_unsetf()
{
cout << "\n-------------------------\n";
cout << "Implementing ios::unsetf\n\n";
cout.setf(ios::showpos|ios::showpoint);
// Clear the showflag flag without
// affecting the showpoint flag
cout.unsetf(ios::showpos);
cout<<200.0;
cout << "\n-------------------------\n";
}

// Driver Method
int main()
{
IOS_width();
IOS_precision;
```

```
IOS_fill();
IOS_setf();
IOS_unsetf();
return 0;
}
```

File Handling in C++

File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).

How to achieve the File Handling

For achieving file handling we need to follow the following steps:-

STEP 1-Naming a file
STEP 2-Opening a file
STEP 3-Writing data into the file
STEP 4-Reading data from the file
STEP 5-Closing a file

Sr.No	
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

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.

C++ FileStream example: writing to a file

Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5. ofstream filestream("testout.txt");
6. if (filestream.is_open())
7. {
```

```
8. filestream << "Welcome to javaTpoint.\n";
9. filestream << "C++ Tutorial.\n";
10. filestream.close();
11. }
12. else cout <<"File opening is fail.";
13. return 0;
14. }
```

Output:

```
The content of a text file testout.txt is set with the data:
Welcome to javaTpoint.
C++ Tutorial.
```

C++ FileStream example: reading from a file

Let's see the simple example of reading from a text file **testout.txt** using C++ FileStream programming.

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5. string srg;
6. ifstream filestream("testout.txt");
7. if (filestream.is_open())
8. {
9. while ( getline (filestream,srg) )
10. {
11. cout << srg <<endl;
12. }
13. filestream.close();
14. }
15. else {
16. cout << "File opening is fail."<<endl;
17. }
18. return 0;
19. }
```

Output:

```
20. Welcome to javaTpoint.
```



```
21. C++ Tutorial.
```

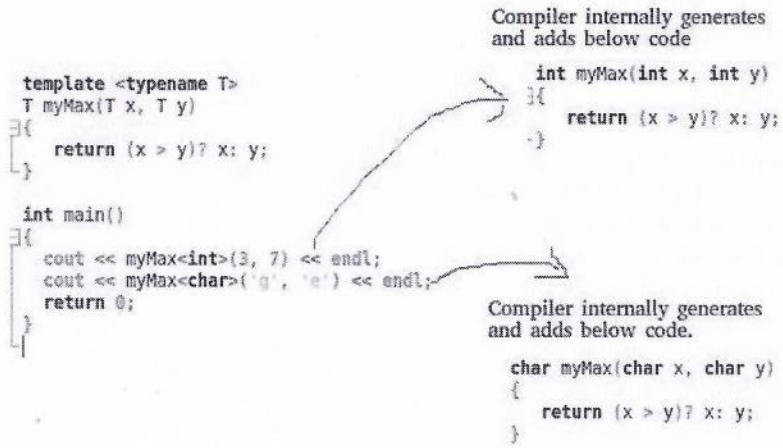
C++ Read and Write Example

```
1. #include <fstream>
2. #include <iostream>
3. using namespace std;
4. int main () {
5. char input[75];
6. ofstream os;
7. os.open("testout.txt");
8. cout <<"Writing to a text file:" << endl;
9. cout << "Please Enter your name: ";
10. cin.getline(input, 100);
11. os << input << endl;
12. cout << "Please Enter your age: ";
13. cin >> input;
14. cin.ignore();
15. os << input << endl;
16. os.close();
17. ifstream is;
18. string line;
19. is.open("testout.txt");
20. cout << "Reading from a text file:" << endl;
21. while (getline (is,line))
22. {
23. cout << line << endl;
24. }
25. is.close();
26. return 0;
27. }
```

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. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.
C++ adds two new keywords to support templates: ***'template'*** and ***'type name'***.
The second keyword can always be replaced by the keyword '**class**'.

How Do Templates Work?

Templates are expanded at compiler time. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



Function Templates

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

```
// C++ Program to demonstrate
// Use of 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. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
// C++ Program to implement
// template Array class
#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;
}
```

Can there be more than one argument for templates?
Yes, like normal parameters, we can pass more than one data type as arguments to templates. The following example demonstrates the same.

```
// C++ Program to implement
// Use of template
#include <iostream>
using namespace std;

template <class T, class U> class A {
T x;
U y;

public:
A() { cout << "Constructor Called" << endl; }
};

int main()
{
A<char, char> a;
A<int, double> b;
return 0;
}
```

What is the difference between function overloading and templates?
Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

Introduction to C++ Standard Library

The C++ Standard Library can be categorized into two parts –

- **The Standard Function Library** – This library consists of general-purpose,stand-alone functions that are not part of any class. The function library is inherited from C.
- **The Object Oriented Class Library** – This is a collection of classes and associated functions.

he Standard Function Library

The standard function library is divided into the following categories –

- I/O,
- String and character handling,
- Mathematical,
- Time, date, and localization,

- Dynamic allocation,
- Miscellaneous,
- Wide-character functions,

The Object Oriented Class Library

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following –

- The Standard C++ I/O Classes
- The String Class
- The Numeric Classes
- The STL Container Classes
- The STL Algorithms
- The STL Function Objects
- The STL Iterators
- The STL Allocators
- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library

Introduction to RTTI (Run-Time Type Information) in C++
In C++, **RTTI (Run-time type information)** is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.
Runtime Casts
The runtime cast, which checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:

- **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
- **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.

Using *'dynamic_cast'*: In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.

```
// C++ program to demonstrate
// Run Time Type Identification(RTTI)
// but without virtual function
```

```
#include <iostream>
using namespace std;

// initialization of base class
class B {};

// initialization of derived class
class D : public B {};

// Driver Code
int main()
{
B* b = new D; // Base class pointer
D* d = dynamic_cast<D*>(b); // Derived class pointer
if (d != NULL)
cout << "works";
else
cout << "cannot cast B* to D*";
getchar(); // to get the next character
return 0;
}
```