Module-4

Virtual Functions and Runtime Polymorphism: Virtual function, Calling a Virtual function through a base class reference, Virtual attribute is inherited, Virtual functions are hierarchical, Pure virtual functions, abstract classes, Using Virtual functions, Early & late binding.

Standard C++ I/O Classes: Old vs. Modern C++ I/O, C++ Streams, The C++ Stream Classes, C++'s, Predefined Streams, Formatted I/O, Formatting Using the ios Members, Setting the Format Flags, Clearing Format Flags, Overloading << and >>, Manipulators

4.1 Virtual functions and runtime polymorphism:

- ***Definition***: virtual function can be defined as a member function that is declared in a base class and redefined by a derived class.
- Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A '*virtual*' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

```
class A
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B : public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;       //pointer of base class
 B b;        //object of derived class
 a = &b;
 a->display();     //Late Binding occurs
}
```

Output

```
Derived Class is invoked
```

4.1.1 Calling a virtual function through a base class reference

- Instead of pointers, the references can be used for having the effect of virtual functions.
- References are used for accessing a virtual function when the reference is a function parameter.
- Consider the program that illustrates virtual function is called using base reference.

```
class B{
  public:
        virtual void display()
          {
              cout<<"Iam in B"<<endl;
          }
};
class D1: public B {
      public:
         void display(){
              cout<<"Iam in D1"<<endl;
          }
};
void fun(B &ref){
      ref.dispaly();
}
int main(){
     B ob;
     D1 ob1;

     fun(ob);
     fun(ob1)

}
```

Output:
Iam in B
Iam in D1

- Here**,** the fun( ) is used for defining a reference parameter of base. In main( ) function, we call this function with the parameter of B, D1 respectively. Inside the function fun(), the type of object determines which of the version of display() function must be called.

4.1.2 Virtual attribute is inherited

- Suppose that we have class `D1` derived from the class `B` containing a virtual function. Now, if a class `D2` is inherited from `D1`, then virtual nature of the base class function remains same in `D2` also.
- Consider the program

```
class B {
  Public:
  virtual void display()
          {
              cout<<"Iam in B"<<endl;
          }
};
class D1: public B {
      public:
         void display(){
              cout<<"Iam in D1"<<endl;
          }
};
class D2: public D1 {
      public:
         void display(){
              cout<<"Iam in D2"<<endl;
          }
};
int main(){
  B *p;
  D1 ob1;
  D2 ob2;

  P=&ob1;
  p->display();

  p=&ob2;
  p->display();
}
```

Output
I am in D1
I am in D2

4.1.3 Virtual functions are hierarchical

- Virtual functions are hierarchical in nature. That is,
    - When a derived class does not override a virtual function of the base, the base class function itself will be invoked, even if it is accessed through derived class object.
    - In case of multilevel inheritance, if any class does't override the virtual function of the topmost base class, then the nearest possible overridden function is called.
    - Example

```
class B {
  Public:
  virtual void display()
          {
```

```
              cout<<"Iam in B"<<endl;
          }
};
class D1: public B {
      public:
         void display(){
              cout<<"Iam in D1"<<endl;
          }
};
class D2: public B {
      public:    //not overriding the virtual function
};
class D3: public D1 {
      public:    //not overriding the virtual function
};
class D4: public D3 {
      public:    //not overriding the virtual function
};
int main()
{
  B *p;
  D1 ob1;
  D2 ob2;
  D3 ob3;
  D4 ob4;

  p=&ob1;
  p->display();

  p=&ob2;
  p->display();

  p=&ob3;
  p->display();

  p=&ob4;
  p->display();

}
```

```
Output:
Iam in D1
Iam in B
Iam in D1
Iam in D1
```

- Here, invoking display() function using pointer to D4 object will call the display() defined inside D1. That is, the function of nearest parent.

4.1.4 Pure virtual functions

- A pure virtual function that has no definition in the base class and hence it must be overridden in the derived class.
- The general form

```
virtual ret_type fun_name(para_list) = 0;
```

- Here, the symbol= 0 is just a notation used for pure virtual function, and there is no such meaning that the value of a function is zero.
- Example:

```
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
   {
        cout <<"Derived class is derived from the base class.";

    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Output:

```
Derived class is derived from the base class
```

4.1.5 abstract classes

- Abstract Class is a class which contains at least one Pure Virtual function in it.
- Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.
- Characteristics of Abstract Class
    - Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
    - Abstract class can have normal functions and variables along with a pure virtual function.
    - Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

- Example

```
//Abstract base class
class Base
{
    public:
    virtual void show() = 0;      // Pure Virtual Function
};

class Derived:public Base
{
    public:
    void show()
    {
        cout<<"Implementation of Virtual Function in Derived
class\n";
    }
};

int main()
{
    Base obj;   //Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

Output:

```
Implementation of Virtual Function in Derived class
```

4.1.6 using virtual functions

- One of the important mechanism of oops that is, polymorphism with the philosophy of "one interface, multiple methods" can be achieved by using virtual functions, abstract classes and run-time polymorphism.
- Using all these features, one can achieve the class hierarchy that moves from general to specific version of one's requirement. Using these techniques, it is possible to define very common features and interfaces inside a base class. Then in the derived classes, the specific version that makes the class as an individual class can be included. Thus the usage of virtual functions makes the creation of new situation from the existing situation very easy.

4.1.7 Early & late binding

Binding:

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

Early Binding

- In early binding, the compiler matches the function call with the correct function definition at compile time. It is also known as **Static Binding** or **Compile-time Binding**.
- By default, the compiler goes to the function definition which has been called during compile time. So, all the function calls you have studied till now are due to early binding.

```
#include<iostream>
using namespace std;
class Base {
   public:
    void display() {
      cout<<"In Base class" << endl;
   }
};
class Derived: public Base {
   public:
   void display() {
      cout<<"In Derived class" <<endl;
   }
};
int main() {
   Base *base_pointer;
   Derived d;
   base_pointer=&d;
   base_pointer->display();
   return 0;
}
```

Output

```
In Base class
```

Late Binding

- The compiler matches the function call with the correct function definition at runtime. It is also known as **Dynamic Binding** or **Runtime Binding**.
- In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.
- This can be achieved by declaring a **virtual function**

```
#include<iostream>
using namespace std;
class Base {
   public:
   virtual void display() {
      cout<<"In Base class" << endl;
   }
```

```
};
class Derived: public Base {
   public:
   void display() {
      cout<<"In Derived class" <<endl;
   }
};
int main() {
   Base *base_pointer;
   Derived d;
   base_pointer=&d;
   base_pointer->display();
   return 0;
}
```

Output
In Derived class

5.1 Standard C++ I/O Classes

- **Stream:** A stream is a logical device that either produces or consumes information.
- A stream is linked to a physical device by I/O system.
- All streams behave same way even though the actual devices they are connected to may differ.
- For example: one can use the same function for writing data into a file or into a printer or into a screen.
- A stream basically be represented as an indefinite source or destination of characters. The number of characters to be inputted into stream to be outputted from the stream may be unknown.
- A stream can be better understood with the help of diagram

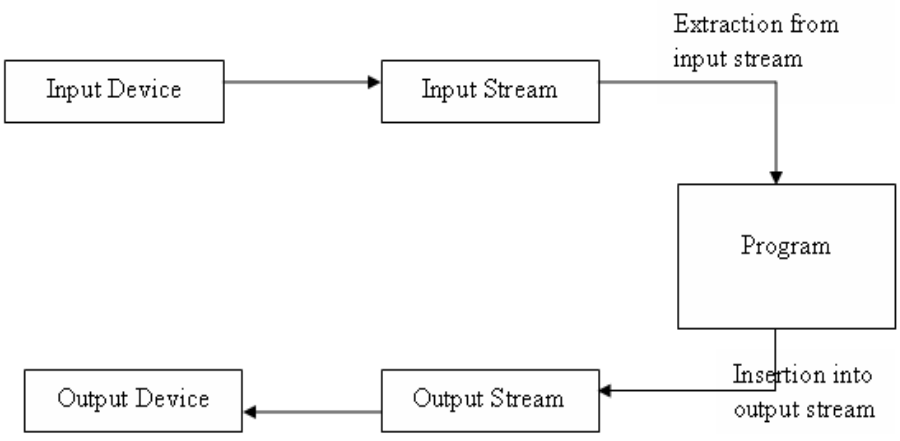

Figure 4.1 Data stream

- Thus, it can be observed that input stream is the flow or data from a keyboard/file to program variables whereas the output stream is the flow of data program variables to monitor/file.

5.1.1 old vs Modern C++ I/O

- There are currently two versions of the C++ object-oriented I/O library in use:

- ***Old I/O library***: it is based on the original specifications for C++
  - I/O library is supported by the header file **<iostream.h>**.
  - the old-style I/O library was in the global namespace.

- ***New I/O library***: defined by Standard C++
  - The new I/O library is supported by the header **<iostream>**.
  - contains a few additional features and defines some new data types
  - the new-style library is in the **std** namespace.

5.1.2 C++ streams class hierarchy

- C++ provides a library of classes that have functionality to implement various aspects of stream handling. This library is known as iostream library.
- The iostream library is an object-oriented library providing input and output functionality using streams.
- In c++, a stream is represented by an object of a particular class. The iostream hierarchy of few important classes are shown below.

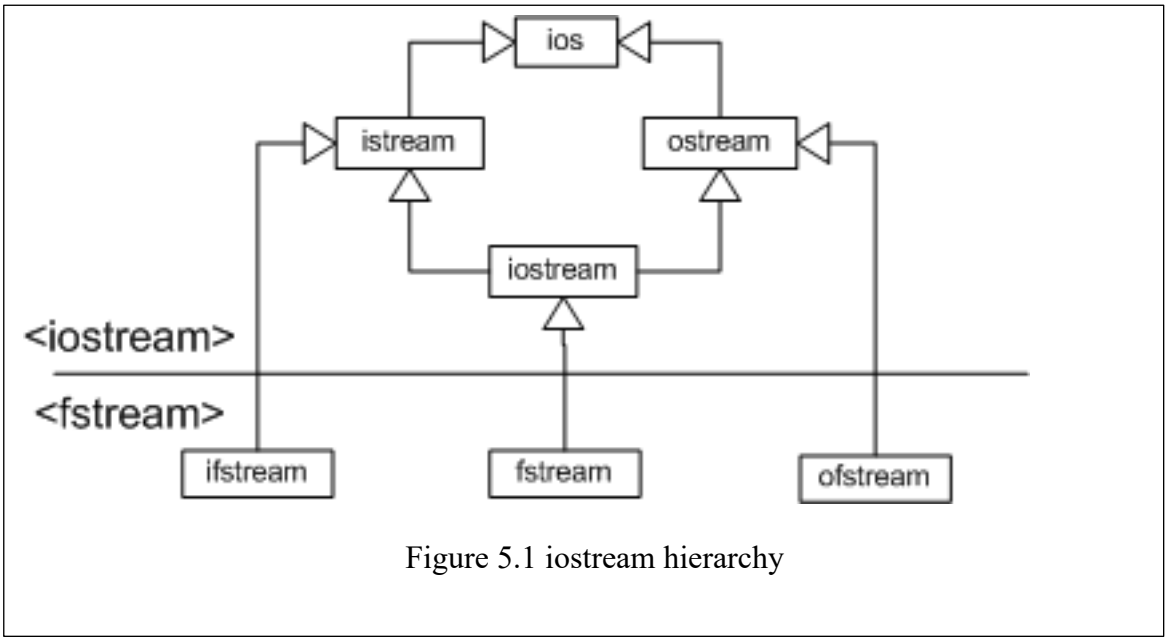

Figure 5.1 iostream hierarchy

The above components are explained below

- **ios**: this class is a base class for all stream classes. But, it describes the members that are template-dependent. It contains data members that help formatting of stream data, the error-status flags and the file operation mode.

- **istream**: this is derived class of ios class. It provides the member functions for reading and interpreting input from a stream buffer. The most commonly used member function of this class is the overloaded >> operator(extraction) function. We can use the standard object cin that is an instantiation of this class, to call overloaded operator member function >>.

- **ostream**: this is also has been derived from ios class to provide the member functions for performing output operations with a stream buffer. The most commonly used member function of this class is the overloaded << operator(insertion) function. the standard object cout of ostream class is used to call the operator function <<.

- **iostream**: it is class derived from both **istream** and **ostream** classes using multiple inheritance concept. The classes inherited from iostream class can perform both input and output operations.

- **ifstream**: this class is derived from both **istream** and **fstreambase** classes. This class provides a stream interface to read the data from a file. The file to be processed can be opened using a member function of this class viz. `open`() and the file can be closed using another member function `close`().

- **ofstream**: this class is derived from both **ostream** and **fstreambase** classes. The object of this class can be used to write the data into file. This class contains the member functions `open`() and `close`() are the most frequently used functions of this class.

- **fstream**: this class is derived from both **iostream** and **fstreambase** classes. This provides a stream interface to read the data from a file and to write the data into a file.

5.1.3 C++'s, Predefined Streams

- When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

- Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**.
- By default, the standard streams are used to communicate with the console.
- Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

5.1.4 Formatted I/O

- The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed.
- There are two related but conceptually different ways that you can format data
  - *access members of the ios class:*
  - *manipulators*

5.1.4.1 Formatting using the ios members

- Each stream has associated with it a set of format flags that control the way information is formatted
- The **ios** class declares a bitmask enumeration called ***fmtflags*** in which the following values are defined.

flags	description
skipws	leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream.
left	output is left justified
right	output is right justified
internal	numeric value is padded to fill a field by inserting spaces between any sign or base character
oct	output to be displayed in octal.
hex	output to be displayed in hexadecimal.
dec	output to decimal
showbase	base of numeric values to be shown.
uppercase	characters are displayed in uppercase
showpos	causes a leading plus sign to be displayed before positive values
showpoint	causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.
scientific	floating-point numeric values are displayed using scientific notation.
fixed	floating-point values are displayed using normal notation.
unitbuf	the buffer is flushed after each insertion operation.
boolalpha	Booleans can be input or output using the keywords **true** and **false.**

- Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the format flags

- To set a flag, use the setf( ) function. This function is a member of ios. Its most common form is shown here:

```
fmtflags setf(fmtflags flags);
```

- This function returns the previous settings of the format flags and turns on those flags specified by flags. For example, to turn on the showpos flag, you can use this statement:

```
stream.setf(ios::showpos);
```

- Here, stream is the stream you wish to affect. Notice the use of ios:: to qualify showpos. Since showpos is an enumerated constant defined by the ios class, it must be qualified by ios when it is used.

- The following program displays the value 100 with the showpos and showpoint flags turned on.

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

int main()
{
cout.setf(ios::showpoint);
cout.setf(ios::showpos);

cout << 100.0; // displays +100.000
return 0;
}
```

- It is important to understand that **setf( )** is a member function of the ios class and affects streams created by that class.

Clearing format flags

- The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags. Its general form is:

```
void unsetf(fmtflags flags);
```

- The flags specified by flags are cleared. (All other flags are unaffected.)
- The following program illustrates ***unsetf().*** It first sets both the uppercase and scientific flags. It then outputs 100.12 in scientific notation. In this case, the "E" used in the scientific notation is in uppercase. Next, it clears the uppercase flag and again outputs 100.12 in scientific notation, using a lowercase "e."

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

int main()
{
   cout.setf(ios::uppercase | ios::scientific);

   cout << 100.12; // displays 1.001200E+02
   cout.unsetf(ios::uppercase); // clear uppercase
   cout << " \n" << 100.12; // displays 1.001200e+02
return 0;
}
```

5.1.4.2 Manipulators

- The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression.
- The standard manipulators are shown in below table. As you can see by examining the table, many of the I/O manipulators parallel member functions of the **ios** class.
- To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

Manipulator	Function
setw(int n)	To set the field width to *n*
setbase	To set the base of the number system
setprecision(int p)	The precision is fixed to *p*
setfill(char f)	To set the character to be filled
setiosflags(long l)	Format flag is set to *l*
resetiosflags(long l)	Removes the flags indicated by *l*
endl	Gives a new line
skipws	Omits white space in input
noskipws	Does not omit white space in the input
ends	Adds null character to close an output string
flush	Flushes the buffer stream
lock	Locks the file associated with the file handle
ws	Omits the leading white spaces present before the first field
hex, oct, dec	Displays the number in hexadecimal or octal or in decimal format

```
// example for manipultors
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
cout << hex << 100 << endl;

cout << setfill('?') << setw(10) << 2343.0;
return 0;
}

This displays

64
??????2343
```

5.5 Overloading << and >>

- The general form for overloading << (output) is given as

```
ostream & operator << (ostream &x, class_type &obj){
     //body of insertion
     return x;
}
```

- here, operator function takes two parameters. One parameter is a reference to the output stream i.e **ostream**. Another parameter is the object to be inserted. This function returns a reference to the output stream.
- The overloaded << operator function must be declared as friend function

- The general form for overloading >>(input) is given as

```
istream & operator >> (istream &x, class_type &obj){
     //body of >>
     return x;
}
```

- here, operator function takes two parameters. One parameter is a reference to the input stream i.e **istream**. Another parameter is the object to be extracted . This function returns a reference to the input stream

- following example explains how extraction operator >> and insertion operator <<.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output, const
Distance &D ) {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
```

```
      }

      friend istream &operator>>( istream  &input, Distance &D
) {
         input >> D.feet >> D.inches;
         return input;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11), D3;

   cout << "Enter the value of object : " << endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;

   return 0;
}
```

Output:

```
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```