CS-15
C++ and Object Oriented Programming

Unit – 4

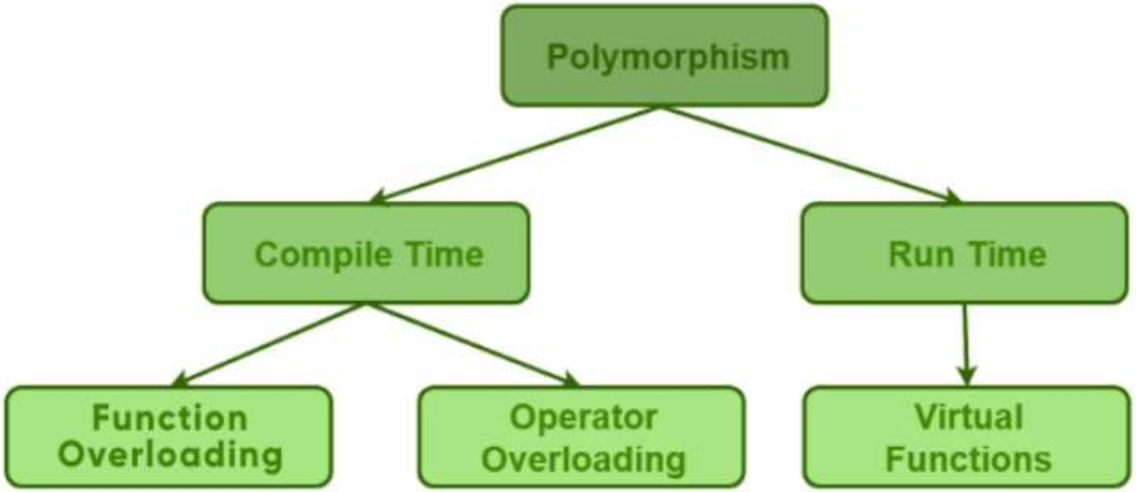
Presented By : Dhruvita Savaliya

Pointer, Virtual Functions and Polymorphism, RTTI Console I/O Operations

Topics :

- Pointer to Object
- Pointer to derived class
- this Pointer
- Rules for virtual function
- Virtual function and pure virtual function
- Run Time Type Identification (RTTI)
- C++ Streams
- C++ Stream Classes
- Unformatted and formatted I/O operations
- Use of Manipulators.

- **Introduction to polymorphism :**
- The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.



- Till now we have used polymorphism using function overloading and operator overloading.
- This type of polymorphism is known as static or compile-time polymorphism.
- In this chapter we will learn about pointers, virtual functions and we will see how the pointers and virtual functions are used to achieve dynamic or runtime polymorphism.

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.
There are 2 type of overriding :

1. Compile time
2. Run time

1. Compile Time overriding :

```
class Parent {
public:
   void Print()
   {
      cout << "Base Function" << endl;
   }
};

class Child : public Parent {
public:
   void Print()
   {
      cout << "Derived Function" << endl;
   }
};

void main()
{
   Child  c_obj;
   c_obj.Print();
}
```

Output :
Derived Function

2. Run Time overriding :

```
class Parent {
public:
   virtual void Print()
   {
      cout << "Base Function" << endl;
   }
};

class Child : public Parent {
public:
   void Print()
   {
      cout << "Derived Function" << endl;
   }
};

void main()
{
   Child  c_obj;
   c_obj.Print();
}
```

Output :
Derived Function

Pointer to Object (object pointer):

- "A ***pointer*** is a variable that holds the memory address of another variable, allowing indirect access and manipulation of values."
- Pointers are very useful in handling complex programs.
- Those programs contained pointer to normal variables, pointers to arrays, pointers to member variables and pointer to function.
- But in case of virtual functions and runtime polymorphism we need to create pointers to objects.
- **Syntax :**

  ClassName *pointer_object;

  pointer_object=&normal_object;

There are to way's to access data of normal object.

1. pointer_object->member_name;
2. *(pointer_object).member_name;

- Pointers to objects are very useful for creating objects at runtime.
- Using this pointer to object, we can access the public members of the class but using **arrow(->) operator instead of dot(.) operator.**

Example :

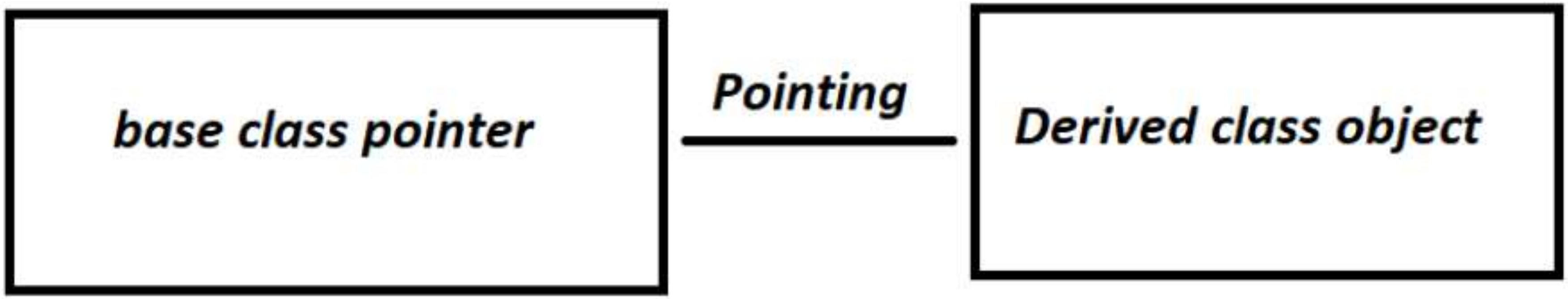
```
class Student{
        public :
        int rollno , houseno;
        void show()
        {
                cout << "\nRoll no of the Student is Using Pointer = " <<rollno ;
                cout << "\nHouse of the Student is = " << houseno;
        }
};
void main ()
{
        Student  s_obj;
        Student  *ptr_s_obj;
         ptr_s_obj=&s_obj;
         ptr_s_obj ->rollno = 26 ;                    //*(ptr_s_obj).rollno=26;
         ptr_s_obj -> houseno = 24 ;                  //*(ptr_s_obj).houseno=26;
         ptr_s_obj->show();                           //*(ptr_s_ibj).show();
        getch();
}
```

Output:

```
Roll no of the Student is Using Pointer = 26
House of the Student is = 24
```

Pointer to derived class :

- A **pointer** is a data type that stores the address of other data types.
- Pointers can be used for base objects as well as objects of derived classes.
- A pointer to the object of the derived class and a pointer to the object of the base class are ***type-compatible*** *(may be used in different ways)*.



base class pointer

Pointing

Derived class object

- A derived class is a class that takes some properties from its base class.
- It is true that a pointer of one class can point to another class, but classes must be a base and derived class, then it is possible.
- To access the **variable of the base class**, a base class pointer will be used.
- So, a pointer is a type of base class, and it can access all, public function and variables of the base class since the pointer is of the base class, this is known as a binding pointer.
- In this pointer base class is owned by the base class but points to the derived class object.
- The same works with derived class pointer, values are changed.

- **Example of base class pointer & object of derived class :**

```
class base
{
   public:
   void fun()
   {
         cout<<"\nbase";
   }
};
class derived : public base
{
   public:
   void fun()
   {
         cout<<"\nderived1";
   }
};

void main()
{
   clrscr();
   derived d;
   base *bptr=&d;
                         //we can't do : //derived d=&base_obj;
   bptr->fun();

   //bptr->fun2();
   //we cannot call function of derived class
   getch();
}
```

Output :

```
base
```

- **Example of base class pointer & object of derived class *using virtual function*:**

```
class base
{
    public:
    virtual void fun()
    {
            cout<<"\nbase";
    }
};

class derived1 : public base
{
    public:
    void fun()
    {
            cout<<"\nderived1";
    }
};
```

```
class derived2 : public base
{
    public:
    void fun()
    {
            cout<<"\nderived2";
    }
};
void main()
{
    clrscr();
    derived1 d1;
    base *bptr=&d1;
    bptr->fun();

    derived2 d2;
    base *bptr=&d2;
    bptr->fun();
    getch();
}
```

Output :

```
derived1
derived2
```

This Pointer :

- **this** is a keyword that refers to the current instance of the class.
- Main usage of this keyword in C++:

1. When local variable’s name is same as member’s name.
2. To return reference to the calling object.

- **Example When local variable's name is same as member's name :**

```
class Test
{
    int x;
    public:
     void setX (int x){ this->x = x;  }
    void print( )
    {
            cout << "x = " << x << endl;
    }
};
void main()
{
  Test obj;
  obj.setX(20);
  obj.print();
}
```

Output:

X=20

- **Example of To return reference to the calling object :**

```
class Test
{
    int x;
    public:
    Test &setX(int a)
    {
            x = a;
            return *this;
    }
    void print() { cout << "x = " << x ; }
};
void main()
{
  Test obj1;
  obj1.setX(10);
  obj1.print();
}
```

Output:

X=10

Virtual Function :

- 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 of Virtual Functions :

1. Virtual functions cannot be static.
2. Virtual functions must be members of some class.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
6. The prototype of virtual functions should be the same in the base as well as the derived class.
7. 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.
8. A class may have a virtual destructor but it cannot have a virtual constructor.

Example :

```
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;
    }
};
void main()
{
   clrscr();
    A* a,a_obj;    //pointer of base class
    B b;     //object of derived class
    a = &b;
    a->display();   //Late Binding occurs
    a_obj.display();
    getch();
}
```

Output:

```
Derived Class is invoked
Base Class is invoked
```

- **Why virtual function :**

  ✓ If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context.

  ✓ This feature in C++ programming is known as polymorphism which is one of the important features of OOP.

  ✓ If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.

Default Argument to Virtual Function :

- As normal functions, we can also have default arguments in virtual functions.
- We can pass default arguments to the virtual functions as well as pure virtual functions also.
- The default arguments passed in virtual function will be valid for its derived class version also.
- All the rules for default argument function apply to the virtual function with default argument.
- **Syntax :**

```
Virtual returnType Function_name(
arg1, arg2..., arg=value)
{
      //Function Body
}
```

Pure Virtual Function :

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**.
- A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- If we have declared a pure virtual function in a base class, then all the derived class of that base class must implement that function.
- If one or more derived class is not defining this function, then they must declare the pure virtual function as in the base class.

- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.
- **Pure virtual function can be defined as:**

  virtual void display( ) = 0;

- **Example of pure virtual function :**

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
   virtual void show()=0;
    //it is a pure virtual function
    //do nothing function
};
class B:public A
{
   public:
   void show()
   {
   cout<<“This is child class”;
}
};
void main()
{
   clrscr();
   B b;
   b.show();//override method
   getch();
}

      //A a;
      //we can’t create an object of
      abstract class
      //because it has a pure virtual
      function
      //a.show();
```

Output:

```
This is child class
```

Run Time Type Identification (RTTI) :

- 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.

- The **dynamic_cast operator.**
- It is used for conversion between base class and derived class objects.
  - **Syntax :**

    dynamic_cast <new_data_type>(expression)

```
Base_cls o1;
Derived_cls o2;
Base_cls *ptr_b = dynamic_cast< Base_cls *>(&o2);
Derived_cls *ptr_d = dynamic_cast< Derived _cls*>(&o1);
```

- The **typeid operator.**
- It is used for identifying the actual type of an object.
  - **Syntax :**

    typeid(type); OR typeid(expression);

```
typeid(object_name).name()
typeid(*object_name).name()
```

- The **type_info class.**
- This class stores the type information returned by the **typeid operator.**

Example of Dynamic Cast operator :

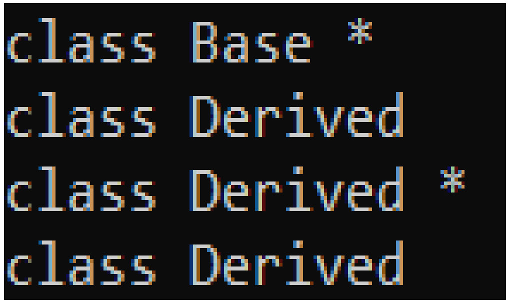
```
#include <iostream>
using namespace std;

class B {
public:
    virtual void fun() {}
};

 class D : public B {
public:
    void fun()
    {
        cout << "i am Derived class";
    }
};

int main()
{
    B* b = new D; // Base class pointer
    D* d = dynamic_cast<D*>(b); // Derived class pointer
    d->fun();
        if (d != NULL)
            cout << “\nworks";
        else
            cout << “\ncannot cast B* to D*";
        getch();
    return 0;
}
```

Output:

```
i am Derived class
works
```

Example of typeid operatr :

```
#include <typeinfo>
#include <iostream>
#include<conio.h>
using namespace std;
class Base {
public:
        virtual void vvfunc( ) {  }
};
class Derived : public Base {  };
void main( ) {
   Derived* pd = new Derived;
   Base* pb = pd;
   cout << typeid(pb).name( ) << endl;   //prints "class Base *"
   cout << typeid(*pb).name( ) << endl;   //prints "class Derived"
   cout << typeid(pd).name( ) << endl;   //prints "class Derived *"
   cout << typeid(*pd).name( ) << endl;   //prints "class Derived"
   getch( );
}
```

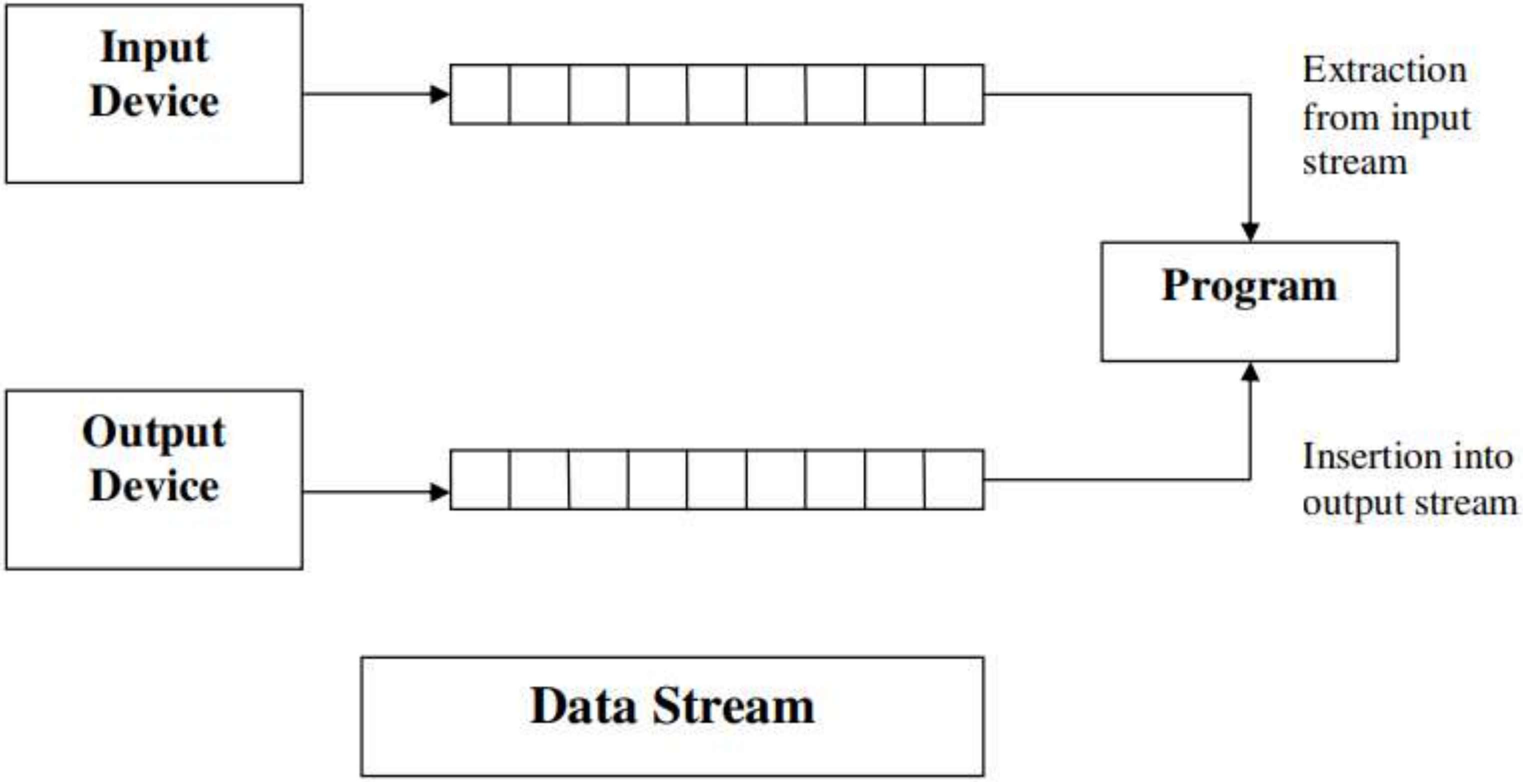


```
class Base *
class Derived
class Derived *
class Derived
```

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 stream.
- A stream is a sequence of bytes.
- It acts either as source from which the input data can be obtained or as a destination to which the output data can be sent.
- The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the **output stream. In other words a program**

- extracts the bytes from an input stream and inserts bytes into an output stream as illustrated in following figure :



C++ Streams Classes :

- **What is #include in C++?**
- iostream stands for standard input-output stream.
- #include iostream declares objects that control reading from and
- writing to the standard streams.
- In other words, the iostream library is an object-oriented library that
- provides input and output functionality using streams.
- A stream is a sequence of bytes. You can think of it as an abstraction
- representing a device.
- You can perform I/O operations on the device via this abstraction.
- You must include iostream header file to input and output from a C++
- program.
- #include iostream provides the most used standard input and output
- streams, cin and cout.

- **1. Standard Output Stream -- cout :**

- It is an instance of the ostream class.
- It produces output on the standard output device, i.e., the display screen.
- We need to use the stream insertion operator << to insert data into
- the standard output stream cout, which has to be displayed on the screen.
- **Syntax:**

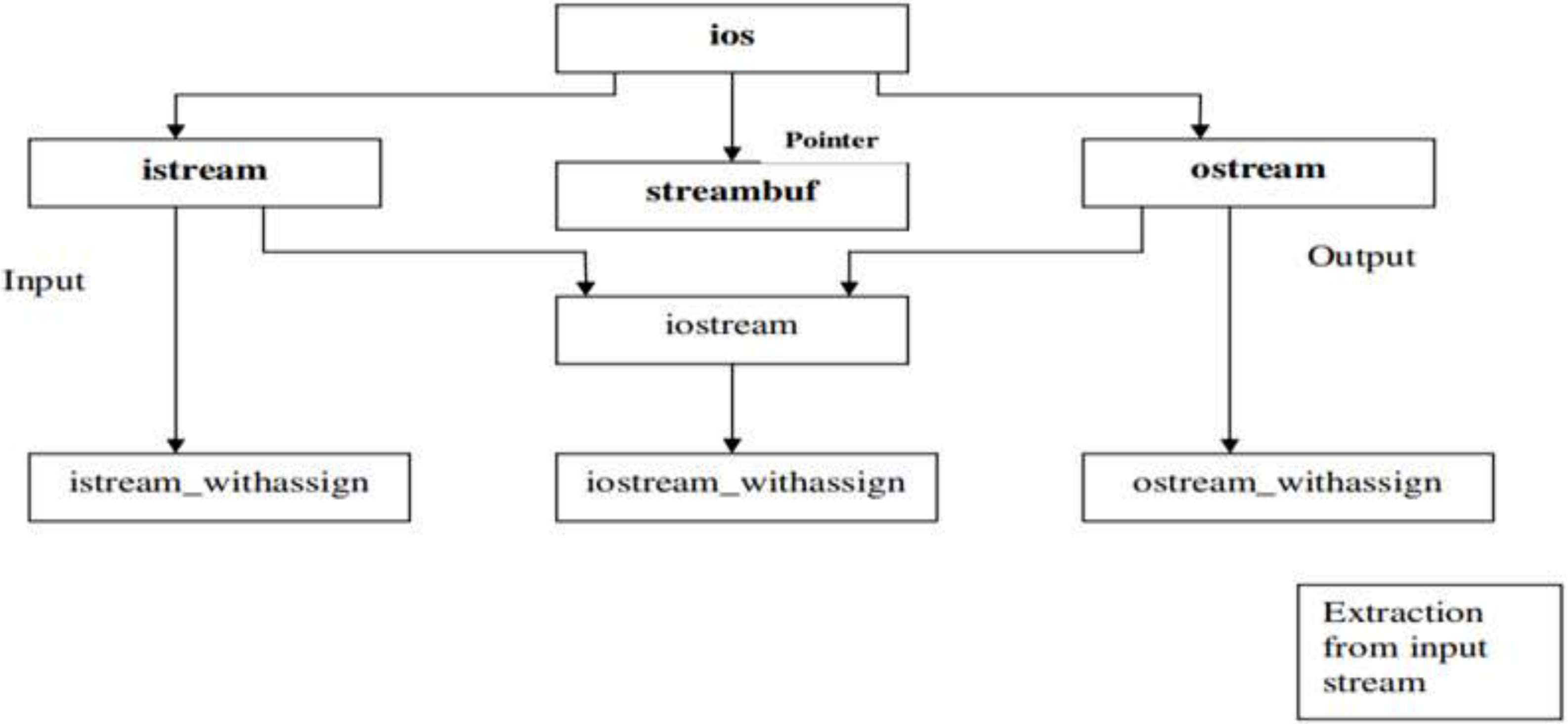
cout << variable_name; OR cout << variable1 << variable2 << … ;

- **2. Standard Input Stream -- cin :**

- It is an instance of the istream class.
- It reads input from the standard input device, i.e., the keyboard.
- We need to use the stream extraction operator >> to extract data entered using the keyboard.
- **Syntax:**

cin >> variable_name; OR cin >> variable1 >> variable2 >> … ;

- 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. Theses are called stream classes.



- These classes are declared in the header file **iostream.h. This file should be included in**
- all the programs that communicate with the console unit.
- The **ios is the base class for istream (input stream) and ostream (output stream) The class ios is declared as virtual base class so that only one copy of its members are inherited by the iostream.**

Unformatted and formatted I/O operations

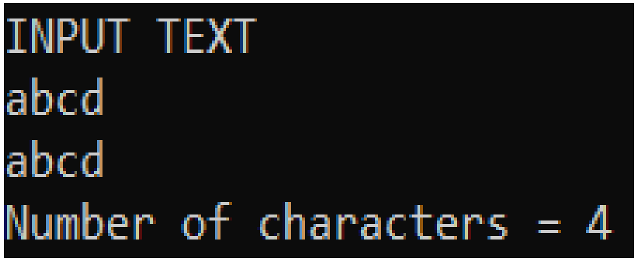
1.Unformatted I/O operation:

- The printed data with default setting by the I/O function of the language is known as **unformatted data.**
- It is the basic form of input/output and transfers the internal binary representation of the data directly between memory and the file.
- **For example, in cin statement it asks for a number while executing. If the user enters a decimal number, the entered number is displayed using cout statement.**
- There is no need to apply any external setting, by default the I/O function represents the number in decimal format.
- in C++, we can *read the input entered by a user at console using an* object **cin of *istream class and through this object we can access the***
- functions of *istream class, such as -* ***get(char *), get(void) and getline().***
- In C++, we can *write the output at console using an object* ***cout of ostream class and through this object we can access the functions of ostream class, such as - put(), write().***
- Some of the most important formatted console input/output functions are -

Functions	Description
get(char *)	Reads a single character from the user at the console and assigns it to the char array in its argument, but needs an Enter key to be pressed at the end..
get()	Reads a single character from the user at the console, and returns it.
getline(char* arr, int size)	Reads a line of characters, entered by the user at the console which ends with a newline character or until the size of .
put(char ch)	Writes a single character at the console.
write(char *arr, int num)	Writes a number of characters in a char array to the console.

Example of get( ) & put( ) :

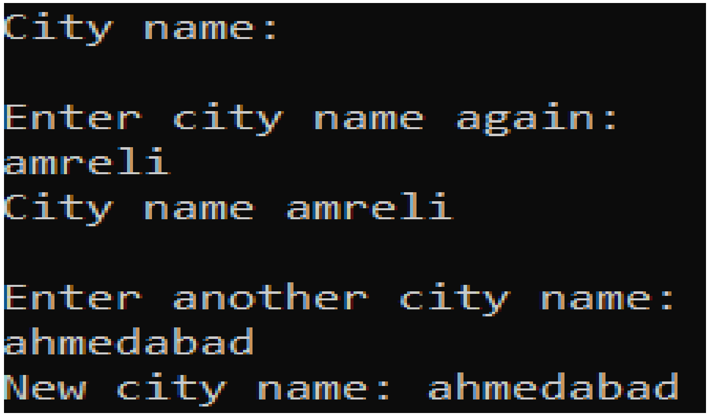
```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
   int count = 0;
   char c;
   cout << "INPUT TEXT\n";
   cin.get(c);  //to get first char
   while (c != '\n')
   {
          cout.put(c);
          count++;
          cin.get(c);
   }
   cout << "\nNumber of characters = " << count << "\n";
getch();
return 0;
}
```



```
INPUT TEXT
abcd
abcd
Number of characters = 4
```

Example of getline with char[ ] & string:

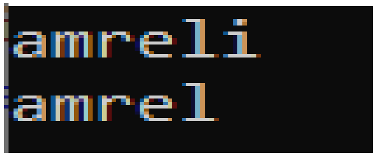
```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
int main()
{
    int size = 20;
    char city[20];

    string city1;

    cout << "City name:" << "\n\n";
    cout << "Enter city name again: \n";
            cin.getline(city, size);  //no need to add string.h
    cout << "City name "<< city << "\n\n";

    cout << "Enter another city name: \n";
            getline(cin,city1);  //must add string.h
    cout << "New city name: " << city1 << "\n\n";

    getch();
    return 0;
}
```



```
City name:

Enter city name again:
amreli
City name amreli

Enter another city name:
ahmedabad
New city name: ahmedabad
```

Example of getline() and write() function :

```
#include <iostream>
#include <string>
#include<conio.h>
using namespace std;
int main()
{
   char line[100];
   cin.getline(line, 10);
// Print the data
   cout.write(line, 5);
getch();
return 0;
}
```



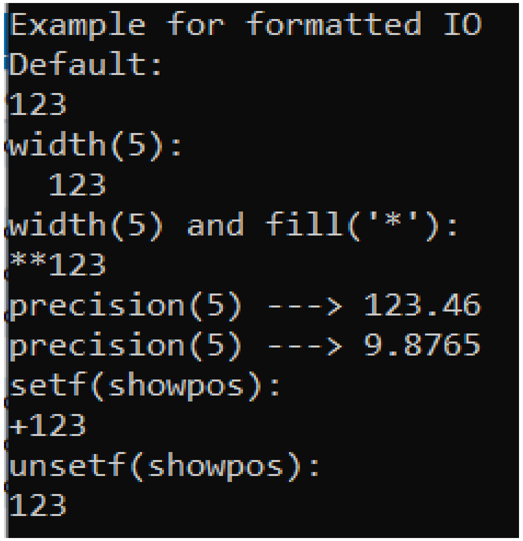
```
amreli
amrel
```

2.Formatted data I/O operations:

- If the user needs to display a number in hexadecimal format, the data is represented with the manipulators are known as **formatted data.**
- It converts the internal binary representation of the data to ASCII characters which are written to the output file.
- It reads characters from the input file and coverts them to internal form.

Function	Description
width(int)	Used to set the width in number of character spaces for the immediate output data.
fill(char)	Used to fill the blank spaces in output with given character.
precision(int)	Used to set the number of the decimal point to a float value.
setf(format flags)	Used to set various flags for formatting output like showbase, showpos, oct, hex, etc.
unsetf(format flags)	Used to clear the format flag setting.

```
#include <iostream>
#include <fstream>
#include<conio.h>
using namespace std;
int main()
{
cout << "Default: " << endl;
cout << 123 << endl;

cout << "width(5): " << endl;
cout.width(5);
cout << 123 << endl;

cout << "width(5) and fill('*'): " << endl;
cout.width(5);
cout.fill('*');
cout << 123 << endl;
cout.precision(5);
cout << "precision(5) ---> " <<
    123.4567890 << endl;
cout << "precision(5) ---> " <<
    9.876543210 << endl;

cout << "setf(showpos): " << endl;
cout.setf(ios::showpos);
cout << 123 << endl;

cout << "unsetf(showpos): " << endl;
cout.unsetf(ios::showpos);
cout << 123 << endl;
getch();
return 0;
}
```



```
Example for formatted IO
Default:
123
width(5):
  123
width(5) and fill('*'):
**123
precision(5) ---> 123.46
precision(5) ---> 9.8765
setf(showpos):
+123
unsetf(showpos):
123
```

Use of Manipulators :

- Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Following table shows some important manipulators functions that are frequently used. To access manipulators, the file iomanip.h should be included in the program.
- **Advantages and Purpose of Manipulators**

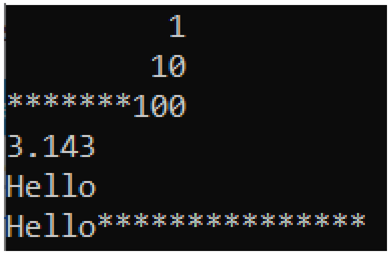
1. It is mainly used to make up the program structure.
2. Manipulators functions are special stream function that changes
3. Certain format and characteristics of the input and output.
4. To carry out the operations of the manipulators <iomanip.h> must be included.
5. Manipulators functions are specially designed to be used in conjunction with insertion (<<) and extraction (>>) operator on stream objects.
6. Manipulators are used to changing the format of parameters on streams and to insert or extract certain special characters.

- **Standard input/output Manipulators in C++ :**

1. **setw (int n)** – To set field width to n The setw() function is an output manipulator that inserts whitespace between two variables. You must enter an integer value equal to the needed space.
2. **setprecision (int p)** – It is an output manipulator that controls the number of digits to display after the decimal for a floating point integer. Make careful to include the ipmanip header in your program because the function is defined there.
3. **setfill (Char f)** – To set the character to be filled The precision is fixed to p It replaces setw(whitespaces)'s with a different character. It's similar to setw() in that it manipulates output, but the only parameter required is a single character. It's worth noting that a character is contained in single quotes.
4. **setiosflags (long l)** – Format flag is set to l Generally it is used to set different types of the flag in our program. That means when we use setw it print the matter from Right to left but names are always print from left to right. So for that, we use setiosflag. Setiosflag also used to represent the sign, to represent the number scientifically and also use it in the number system.
5. **resetiosflags (long l)** – Removes the flags indicated by setiosflags.
6. **endl –** Gives a new line The endl character introduces a new line or a line feed.

   It is analogous to the "n" character in the C computer language, and C++ supports the old line feed.

```
#include<iomanip>
#include<iostream>
using namespace std;
int main()
{
cout<< setw(10) << 1 << endl;
cout<< setw(10) << 10 << endl;
cout<< setw(10) << setfill('*')<< 100 << endl;
cout<< setprecision(4) << 22/7.0 << endl;
cout<< setw(5) << setiosflags(ios::left)<<"Hello"<< endl;
cout<< setw(20) << resetiosflags(ios::right)<<"Hello"<< endl;
}
```



```
         1
        10
*******100
3.143
Hello
Hello***************
```