Overloading stream insertion and Extraction(< >) operators in C++

In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.

We must know following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object istream class
2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Why these operators must be overloaded as global?
In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator.

For example, consider the statement “ob1 + ob2” (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload ‘+’ in class of ‘ob1’ or make ‘+’ a global function.
The operators ‘<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'.

So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

```
//Declaration
friend ostream& operator << (ostream&, const X&);
//Definition
ostream& operator <<(ostream& stream, const X& obj)
{
   //output fields of the object using obj and the dot operator.


   ……..
   return stream;
}
```

```
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in,  Complex &c);
};
ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}
```

```
istream & operator >> (istream &in,  Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imagenory Part ";
    in >> c.imag;
    return in;
}
int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
   return 0;
}
```

Output:

```
Enter Real Part 10
Enter Imagenory Part 20
The complex object is 10+i20
```

String Manipulation Using Operator Overloading

Manipulating of strings in C++ by operator overloading using character arrays, pointers and string functions. There are no operators for manipulating the strings. There are no direct operator that could act upon the strings or manipulate the strings.

Although there are these limitations exist in C++, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal number. We can manipulate strings by operator overloading as this is not achieved by operators only.

For example :
We should be able to use statement like this in manipulating strings using operator overloading -
string3 = string1 + string2;

C++ provides following two types of string representations −

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

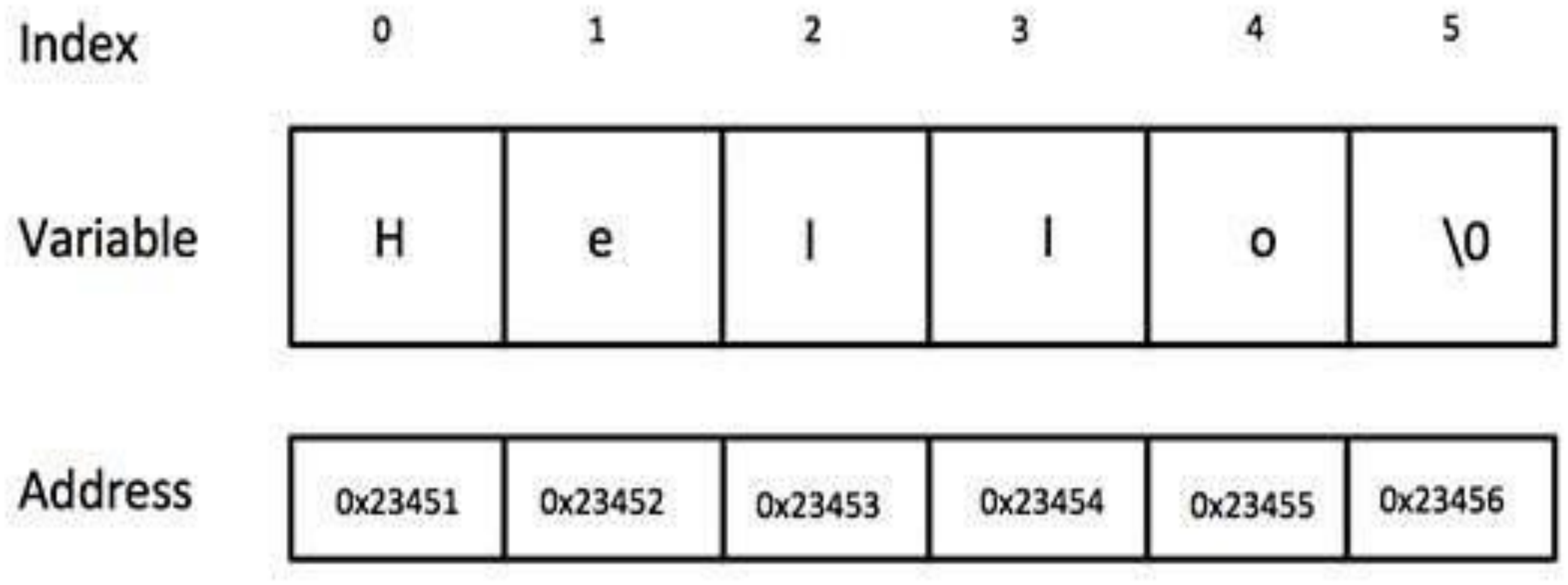
The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed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

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

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

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

```
char greeting[] = "Hello";
```



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

```
#include <iostream>
using namespace std;
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
 cout << "Greeting message: ";
cout << greeting << endl;
return 0;
}
```

The String Class in C++

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

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
        string str1 = "Hello";
        string str2 = "World";
        string str3;
        int len ;
// copy str1 into str3
```

```
        str3 = str1;
        cout << "str3 : " << str3 << endl;
// concatenates str1 and str2

        str3 = str1 + str2;
        cout << "str1 + str2 : " << str3 << endl;
// total length of str3 after concatenation

        len = str3.size();
        cout << "str3.size() : " << len << endl;
        return 0;
}
```

OUTPUT:

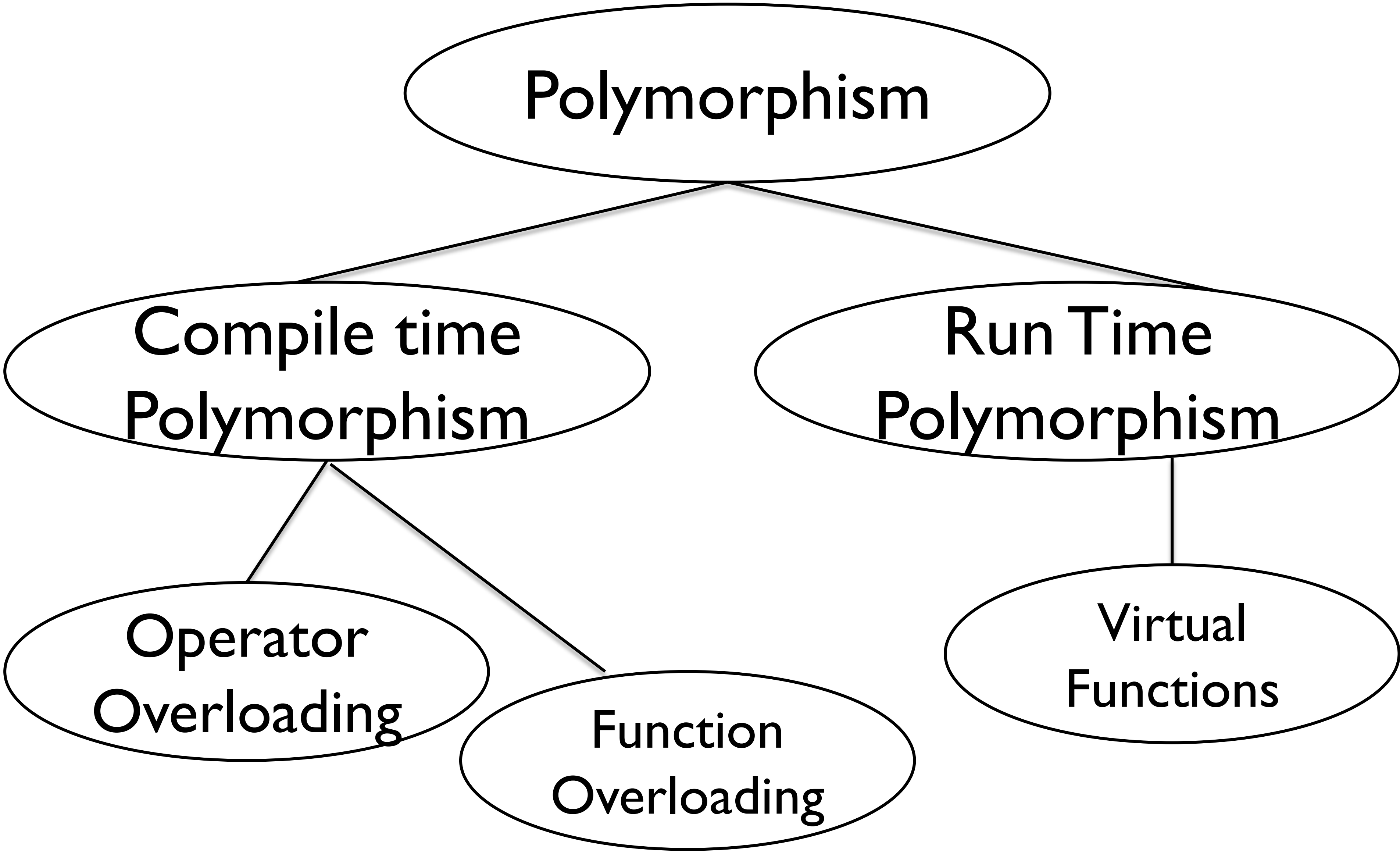
```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10
```

Runtime Polymorphism

The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Syntax: class_name *this;


Polymorphism
Compile time Polymorphism
Run Time Polymorphism
Operator Overloading
Function Overloading
Virtual Functions

```
#include <iostream>
using namespace std;
class Box
{
        public:
// Constructor definition
        Box(double l = 2.0, double b = 2.0, double h = 2.0)
        {
                cout <<"Constructor called." << endl;
                length = l;
                breadth = b;
                height = h;
        }
        double Volume()
        {
                return length * breadth * height;
        }
```

```
          int compare(Box box)
          {
                    return this->Volume() -> box.Volume();
          }
           private: double length; // Length of a box
          double breadth; // Breadth of a box
          double height; // Height of a box
};
int main(void)
{
          Box Box1(3.3, 1.2, 1.5); // Declare box1
          Box Box2(8.5, 6.0, 2.0); // Declare box2
          if(Box1.compare(Box2))
          {
                    cout << "Box2 is smaller than Box1" <<endl;
          }
```

```
else
        {
                cout << "Box2 is equal to or larger than Box1" <<endl;
        }
        return 0;
}
```

OUTPUT:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Pointer to objects:

Pointers do access to class members. A pointer can point to an object created by class.

e.g. employee x;

Defining it with pointer would be,

employee x;

employee *ptr = &x;

We can refer in two ways:

i) By using the dot operator and object

ii) By using the arrow i=operator and the object pointer

(*ptr).show()

employee *ptr = new employee;

ptr -> show();

```
#include <iostram.h>
Class employee
{
            int code;
            float salary;
            public:
            void getdata(int a, float b)
            {
                        code=a;
                        salary=b;
            }
            void show(void)
            {
                        cout<<“Code:”<<code<<“\n”;
                        cout<<“Salary:”<<salary<<“\n”;
            }
};
```

```
int main()
{
        employee *p = new employee[2];
        employee *d =p;
        int  x,i;
        float y;
        for (i=0;i<2;i++)
        {
                cout<<“Input code and salary for employee”<<i+1;
                cin>>x>>y;
                p ->getdata(x,y);
                p++;
        }
        for(i=0;i<2;i++)
        {
                cout<<“Employee:”<<i+1<<“\n”;
                d ->show();  d++;
        } return 0;  }
```

OUTPUT:

```
Input code and salary for employee for employee1   30 8000
Input code and salary for employee for employee2   80 18000

Employee: 1
Code:30
Salary: 8000

Employee: 2
Code:80
Salary: 18000
```

Pointers to derived classes:

C++ allows a pointer in base class to point to either a base class object or to any derived class object.

e.g.

```
class baseA
{
        …..
        ……
};
class derived : public baseA
{
        …….
        ….
};
void main()
{
        baseA *ptr;     //pointer to baseA
}
```

```
#include<iostream.h>
class baseA
{
        public:
        int b;
        void show()
        {
                cout<<"b="<<b<<"\n";
        }
};
class derivedD:public baseA
{
        public:
        int d;
        void show()
        {
                cout<<"b="<<b<<"\n"<<"d="<<d<<"\n";
        }
};
```

```
int main()
{
		baseA *bptr;
		baseA base;
		bptr=&base;
		bptr->b=100;
		cout<<"bptr points to base object\n";
		bptr->show();
//derived class
		derivedD derived;
		bptr=&derived;
		bptr->b=200;
		cout<<"bptr now points to derived object\n";
		bptr->show();
```

```
//accessing d using a pointer of type derived class
derivedD
        derivedD *dptr;
        dptr=&derived;
        dptr->d=300;
        cout<<"dptr is derived pointer\n";
        dptr->show();
        cout<<"using ((derivedD *) bptr\n";
        ((derivedD *)bptr)->d=400;
        ((derivedD *)bptr)->show();
        return 0;
}
```