Object Oriented Programming with C++(BCS306B)

Module – 4

Chapter 1 :

Virtual Functions and Polymorphism:

SYLLABUS:

Virtual Functions and Polymorphism: Virtual Functions, The Virtual Attribute is Inherited, Virtual Functions are Hierarchical, Pure Virtual Functions, Using Virtual Functions, Early vs Late Binding. Templates: Generic Functions, Applying Generic Functions, Generic Classes. The type name and export Keywords. The Power of Templates.

4.1 Virtual Functions

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

```
class Base {
   public:
    void print() {
        // code
    }
};
```

```
class Derived : public
Base {
   public:
    void print() {
        // code
    }
};
```

if we create a pointer of Base type to point to an object of Derived class and call the print() function, it calls the print() function of the Base class.

In other words, the member function of Base is not overridden.

```
int main() {
    Derived derived1;
    Base* base1 = &derived1;

    // calls function of Base class
    base1->print();

    return 0;
}
```

In order to avoid this, we declare the print() function of the Base class as virtual by using the virtual keyword.

```
class Base {
   public:
    virtual void print() {
        // code
    }
};
```

Here, we have declared the print() function of Base as virtual.

So, this function is overridden even when we use a pointer of Base type that points to the Derived object derived1.

```
// 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;
}
```

Output :
print derived class
show base class

Explanation:

- Runtime polymorphism is achieved only through a pointer (or reference) of the base class type.
- Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class.
- In the above code, the base class pointer 'bptr' contains the address of object 'd' of the derived class.

4.2 The Virtual Attribute is Inherited

- When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.
- Put differently, no matter how many times a virtual function is inherited, it remains virtual.

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};
```

```
/* derived2 inherits virtual function vfunc()
   from derived1. */
class derived2 : public derived1
{
public:
  // vfunc() is still virtual
  void vfunc()
  {
    cout << "This is derived2's vfunc().\n";
  }
};
```

```
int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()

  return 0;
}
```

As expected, the preceding program displays this output:

This is base's vfunc().

This is derived1's vfunc().

This is derived2's vfunc().

In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc()** is still virtual.

4.3 Virtual Functions are Hierarchical

- when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden.
- When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used.
- For example, consider this program in which **derived2** does not override **vfunc()** :

```
#include <iostream>
using namespace std;
class base
{public:
 virtual void vfunc() {
   cout << "This is base's vfunc().\n";
} };
class derived1 : public base {
public:
 void vfunc() {
 cout << "This is derived1's vfunc().\n";
 }};
class derived2 : public base {
public:
// vfunc() not overridden by derived2, base's is used
};
```

```
int main()
{ base *p, b;
  derived1 d1;
  derived2 d2;
  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()
  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
  // point to derived2
  p = &d2;
  p->vfunc(); // use base's vfunc()
 return 0;
}
```

The program produces this output:

This is base's vfunc().

This is derived1's vfunc().

This is base's vfunc().

- Because **derived2** does not override **vfunc()** , the function defined by **base** is used when **vfunc()** is referenced relative to objects of type **derived2**.
- The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical.
- This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used.

- For example, in the following program, **derived2** is derived from **derived1**, which is derived from **base.**
- However, **derived2** does not override **vfunc() .** This means that, relative to **derived2**, the closest version of **vfunc()** is in **derived1.**
- Therefore, it is **derived1::vfunc()** that is used when an object of **derived2** attempts to call **vfunc() .**

4.4 Pure Virtual Functions

- 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.

A ***pure virtual function*** is a virtual function that has no definition within the base class.

To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list)* = 0;

- When a virtual function is made pure, any derived class must provide its own definition.
- If the derived class fails to override the pure virtual function, a compile-time error will result.

Example of Pure Virtual Functions

```
// An abstract class
class Test {
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Complete Example

A pure virtual function is implemented by classes that are derived from an

Abstract class.

```
// An abstract class
class Test {
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};

// C++ Program to illustrate the abstract class and
virtual
// functions
#include <iostream>
using namespace std;
```

```
class Base {
    // private member variable
    int x;

public:
    // pure virtual function
    virtual void fun() = 0;

    // getter function to access x
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived : public Base {
    // private member variable
    int y;

public:
    // implementation of the pure virtual function
    void fun() { cout << "fun() called"; }
};
```

```
int main(void)
{
    // creating an object of Derived class
    Derived d;

    // calling the fun() function of Derived class
    d.fun();

    return 0;
}
```

4.5 Using Virtual Functions

- One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism.
- Using these features, you create a class hierarchy that moves from general to specific (base to derived).

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

```
void Display()
{
cout << "Display Derived class" << endl;
}
};
int main(){
Base* bpointr;
Derived dpointr;
bpointr = &dpointr;
// virtual function binding
bpointr->Output();
// Non-virtual function binding
bpointr->Display();
}
```

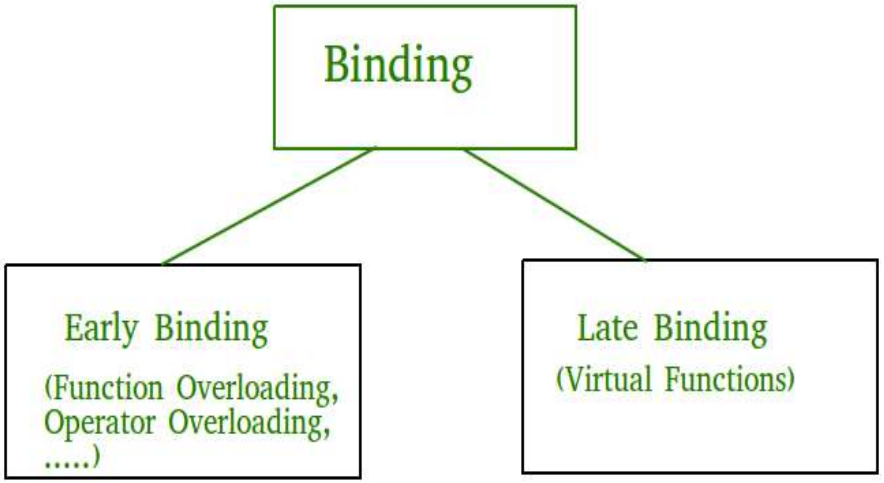
```
Output:
Output Derived class
Display Derived class
```

- In the below example, you will create a base class and a derived class. In both classes, you will create two functions: Output and Display.
- To see the difference between the virtual function and a regular function, you will only declare the Output function of the base class as virtual, and keep the display function as it is.

4.6 Early vs Late Binding

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses.

Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

```
// CPP Program to illustrate early binding.
// Any normal function call (without virtual)
// is binded early. Here we have taken base
// and derived class example so that readers
// can easily compare and see difference in
// outputs.
#include<iostream>
using namespace std;

class Base
{
public:
    void show() { cout<<" In Base \n"; }
};
```

```
class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived;

    // The function call decided at
    // compile time (compiler sees type
    // of pointer and calls base class
    // function.
    bp->show();

    return 0;
}
```

Late Binding : (Run time polymorphism) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition .This can be achieved by declaring a virtual function.

```
// CPP Program to illustrate late binding
#include<iostream>
using namespace std;
 class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};
class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};
```

```
int main(void)
{
    Base *bp = new Derived;
    bp->show();  // RUN-TIME POLYMORPHISM
    return 0;
}
```

Output:

```
In Derived
```

Module – 4
Chapter 2:
Templates:

4.7 Generic Functions

- Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- Instead of specifying the actual data type, we supply a placeholder in templates, and that placeholder is substituted by the data type that was utilized during compilation.
- Functions having generic types that can change their behavior depending on the data type provided during the function call are written using function templates.
- This makes it simpler for us to carry out the same action without code duplication on several data kinds.

How to declare a function template?

A function template starts with the keyword template followed by template parameter/s inside < >which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
    ... .. ...
}
```

```
# include < iostream >
// Template Function with a Type T
// This T will be changed to the data type of the argument
 during instantiation.
template < class T >
T maxNum ( T x , T y ) {
 return ( a > y ? a : y ) ;
}
int main ( )
{    int a = 5 , b = 2 ;
     float i = 4.5 , j = 1.3 ;
     std :: cout < < maxNum < int > ( a , b ) < < " \ n ";
     std :: cout < < maxNum < float > ( c , d ) ;
     return 0 ;
}
```

```
OUTPUT:
5
4.5
???..
Process executed in 0.11 seconds
Press any key to continue
```

Explanation

The data type was supplied with the function call itself in the example above, but this step might be avoided because the compiler would figure it out based on the values we pass to the function.

- A function template works in a similar to a normal function, with one key difference.
- A single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.
- However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

4.8 Applying Generic Functions

Generic Functions using Template:

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

```
#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;
}
```

Output:

```
7
7
g
```

4.9 Generic Classes

Class Template

- Like function templates, we can also use templates with the class to make it compatible with more than one data type.
- In C++ programming, you might have used the vector to create a dynamic array, and you can notice that it works fine with every data type you pass inside the <>, ex- vector<int>. This is just because of the class template.

Syntax:

```
template <class T>
class className {
 // Class Definition.
 // We can use T as a type inside this class.
};
```

Class Template and Friend Functions

- A non-member function that is defined outside the class and can access the private and protected members of the class is called a friend function.
- These are used to create a link between classes and functions. We can define our friend function as a template or a normal function in the class template.

Class Templates and Static Variables

- The classes in C++ can contain two types of variables, static and non-static(instance).
- Each object of the class consists of non-static variables. But the static variable remains the same for each object means it is shared among all the created objects.
- As we are discussing the templates in C++, a point worth noticing is that the static variable in template classes remains shared among all objects of the **same type**.

```
#include <iostream>
using namespace std;
template < class T >
  class Container {
    private:
      T data;
    public:
      static int count;
    Container() {
      count++;}
    static void displayStaticVariable() {
      cout << count << endl;
    }
  };
```

```
template < class T >
  int Container < T > ::count = 0;


int main() {
  Container < int > obj1;
  Container < float > obj2;
  Container < int > obj3;
  Container < int > ::displayStaticVariable();
  Container < float > ::displayStaticVariable();

  return 0;
}
```

Output:

2

1

Explanation:

- We have created a template class named Container which contains static member count.
- We could see from the above example that for different data types, static variables have different values, which shows every type has a separate copy of the static variable.

4.10 The type name and export Keywords

- To use templates in C++, we need to use the two keywords - **template** and **typename.**
- We should first write the keyword template which tells the compiler that the current function or class is a blueprint or template.
- After writing the template, we mention the keyword **typename** and a placeholder name (T) for a data type used by the function or class.
- Templates in C++ can be implemented in two ways, i.e., Function Templates and Class Templates. Refer to the next section for a detailed explanation and implementation.

Class Templates

- Just like the function templates in C++, we can also use class templates to create a single class that can work with the various data types.
- Just like function templates, class templates in C++ can make our code shorter and more manageable.

Syntax of the template function:

```
template <class T> class class-name
{
    // class body
}
```

- In the syntax above for the class template:

  T is a placeholder template argument for the data type. T or type of argument will be specified when a class is instantiated.
- **class** is a keyword used to specify a generic type in a template declaration.

  Note: When a class uses the concept of template in C++, then the class is known as a generic class.
- Some pre-defined examples of class templates in C++ are LinkedList, Stack, Queue, Array, etc. Let's take an example to understand the working and syntax of class templates in C++.

Functional Templates

- Function templates are similar to normal functions. Normal functions work with only one data type, but a function template code can work on multiple data types.
- Hence, we can define function templates in C++ as a single generic function that can work with multiple data types.

```
template <class T> T function-name(T args)
{
   // body of function
}
```

In the syntax above:

- **T** is the type of argument or placeholder that can accept various data types.

class is a keyword used to specify a generic type in a template declaration.

- As we have seen earlier, we can always write typename in the place of class.

Some of the pre-defined examples of function templates in C++ are sort(), max(), min(), etc. Let's take an example to understand the working and syntax of function templates in C++

4.11 The Power of Templates

- The **templates** are one of the most powerful and widely used methods added to C++, allowing us to write generic programs.
- It allow us to define generic functions and classes. It promote generic programming, meaning the programmer does not need to write the same function or method for different parameters.
- We can define a template as a blueprint for creating generic classes and functions.
- The idea behind the templates in C++ is straightforward. We pass the data type as a parameter, so we don't need to write the same code for different data types. Refer to the image below for better visualization.

- The templates are one of the most powerful and widely used methods added to C++, which allows us to write generic programs. It will enable us to define generic functions and classes.
- To use templates in C++, we use the two keywords - template and typename. We can also use the class keyword instead of typename.
- It removes code duplication and helps us to make generic callbacks.
- It helps us to write very efficient and powerful libraries. Example: STL in C++. It gets expanded at compiler time, just like any macros.
- Function templates are similar to normal functions. Function templates in C++ are single generic functions that can work with multiple data types.
- Just like the function templates in C++, we can also use class templates to create a single class that can work with the various data types.
- As templates are calculated at compile time rather than run time when template functions or classes are large and complicated, they can slow the compile time.