Dr. D. Y. Patil Pratishthan's
Institute for Advanced Computing and Software Development

Sub-c++
Day7

Late binding-Virtual Function

- To implement late binding, the function is declared with the keyword `virtual` in the base class.
- Points to note:
  - Virtual function is a member function of a class.
  - Virtual functions can be redefined in the derived class as per the design of the class.
    - Also considered virtual by the compiler

Virtual Function

- It is used to tell the compiler to perform dynamic linkage or late binding on the 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.

Generic Pointers
Base Class pointer can point at a derived object

```
int main ()
{

    cEmployee e1(…), *pemp;
  cTrainer sp1(…);

  pemp = &e1;
  pemp = &sp1;

  pemp ->calSalary();
  return 0;
}
```

`pemp` is a pointer to base class

Base class or generic pointer can point to objects of derived class.

Method Overloading Vs Overriding

	Overloading	Overriding
Scope	In the same class	In the inherited classes
Purpose	Handy for program design as different method names need not be remembered	Message is same but its implementation needs to be specific to the derived class
Signature of methods	Different for each method overloaded	Has to be same in derived class as in base class
Return Type	Can be same or different as it is not considered	Return type also needs to be same

Virtual Functions

- Some points to note:
  - Should be non-static member function of the base class
  - Generally functions that are overridden in the derived class are declared as virtual functions in the base class.
  - Constructors cannot be declared as `virtual`
  - If a function is declared as `virtual` in the base class then, it will be treated as virtual in the derived class even if the keyword `virtual` is not used.

Pure Virtual Function

- A virtual function without any executable code
- Declared by using a pure specifier `(= 0)` in the declaration of a virtual member function in the class declaration.
- For example, in class `cEmployee`

```
virtual float computeSalary() = 0;
```

- A class containing at least one pure virtual function is termed as abstract class.

How virtual functions works internally using vTable and vPointer?

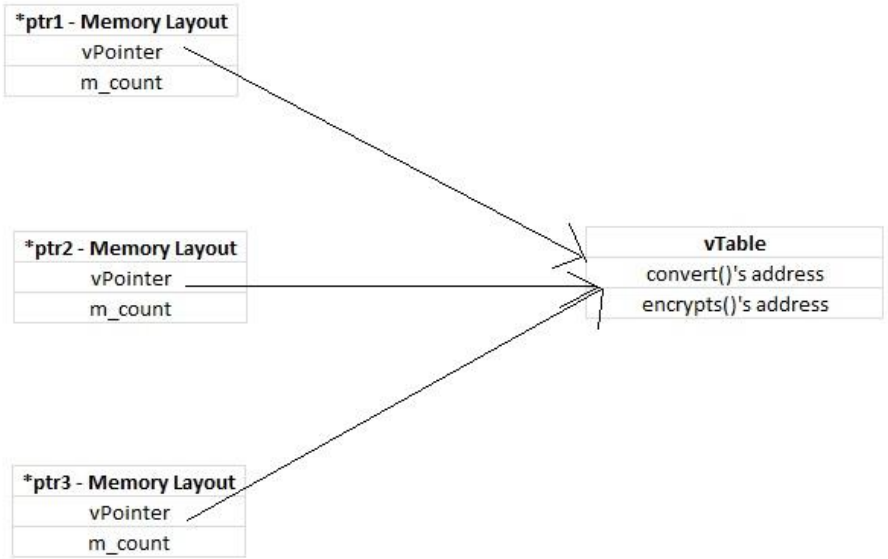
-Binding is a kind of mapping of a function call with the function’s definition i.e. function’s address. For example,When we make a function call like,
obj.display();
then before its execution, it gets bonded to display() function definition i.e. function’s address, so that while code execution, correct function should be called.
When we make a member function virtual then compiler performs run time binding for that function i.e. any call to that virtual function will not be linked to any function’s address during compile time. Actual function’s address to this call will be calculated at run time. To resolve the actual function’s address or definition at run time, C++ compiler adds some additional data structure around virtual functions i.e.
vTable
vPointers

vTable:
Every class that has one or more virtual member functions in it has a vTable associated with it.
vTable is a kind of function pointer array that contains the addresses all virtual functions of this class. Compiler builds this vTable at compile time.
vPointer:
Now for every object of a class that has a vTable associated with it, contains a vPointer in first 4 bytes. This vPointer points to the vTable of that class.
This vPointer will be used to find the actual function address from vTable at run rime.

```
class MessageCoverter
{
int m_count;
public:
virtual string convert(string msg)
{
msg = "[START]" + msg + "[END]";
return msg;
}
virtual string encrypt(string msg)
{
return msg;
}
void displayAboutInfo()
{
cout<<"MessageCoverter Class"<<endl;
}
};
```

vTable
convert()'s address
encrypts()'s address

Now suppose we created three different objects for this class,
MessageCoverter * ptr1 = new MessageCoverter();
MessageCoverter * ptr2 = new MessageCoverter();
MessageCoverter * ptr3 = new MessageCoverter();



vPointer and vTable for MessageConverter class

In Every object first 4 bytes will a pointer that points to the **vTable** of that class. This pointer is called **vPointer**.
Because for virtual functions linking was not done at compile time. So, what happens when a call to virtual function is executed ,i.e.
ptr1->convert("hello");
Steps are as follows,
vpointer hidden in first 4 bytes of the object will be fetched
vTable of this class is accessed through the fetched vPointer
Now from the vTable corresponding function’s address will be fetched
Function will be executed from that function pointer

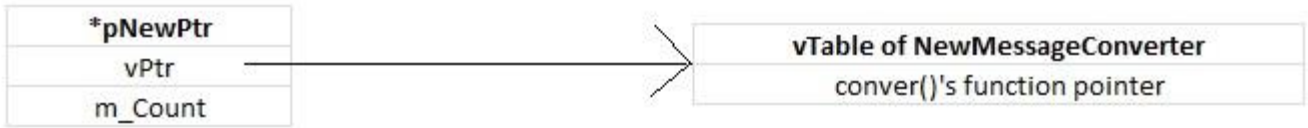
Create a Derived class from MessageCoverter i.e. NewMessageCoverter,

```
class NewMessageCoverter : public MessageCoverter
{
public:
string convert(string msg)
{
msg = "<START> " + msg + " <END>";
return msg;
}
};
```

Once a function is declared virtual in a class then for all its derived classes that function will remain virtual.

vTable
convert()'s address

Now when we create a object of NewMessageConverter then first 4 bytes of this object i.e. vpointer points to the vTable of this new class.
i.e.
MessageConverter * pNewPtr = new NewMessageConverter();



vTable of the NewMessageConverter contains the function address of Derived class’s (NewMessageConverter class) convert() function.

So, if we call the function convert from pNewPtr then convert() function of Derived class is called i.e.

pNewPtr->convert("hello"); // This calls the NewMessageConverter's convert() function.

This is how everything works behind the curtain for every virtual function call.

RTTI(Run time type Identification)

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.

Run Time Type Identification

RTTI is a process that exposes the information of an object data type at run time during program execution.To implement RTTI there are certain operators that gives the runtime type of an object, performs dynamic casting of an object.

1. Typeid
2. dynamic_cast
1. reinterpret_cast

1. typeid operator: this operator is used for fetching the runtime type/dynamic type of a polymorphic object and it can also get the static type.

2. dynamic_cast: It is an operator that is used specifically for down casting. i.e. when the base class pointer need to be type casted to derived class pointer. Down casting is required when we need to invoke special functions of derived class using base class pointer.

3. reinterpret_cast: this operator is used for converting one pointer to another pointer of any type. It doesn’t check/match the type of pointer. The use of this casting is avoided.

Lets see example of RTTI operator dynamic_cast

```
// C++ program to demonstrate
// Run Time Type Identification successfully
// With virtual function

#include <iostream>
using namespace std;

// Initialization of base class
class B {
    virtual void fun() {}
};

// 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();
    return 0;
}
```

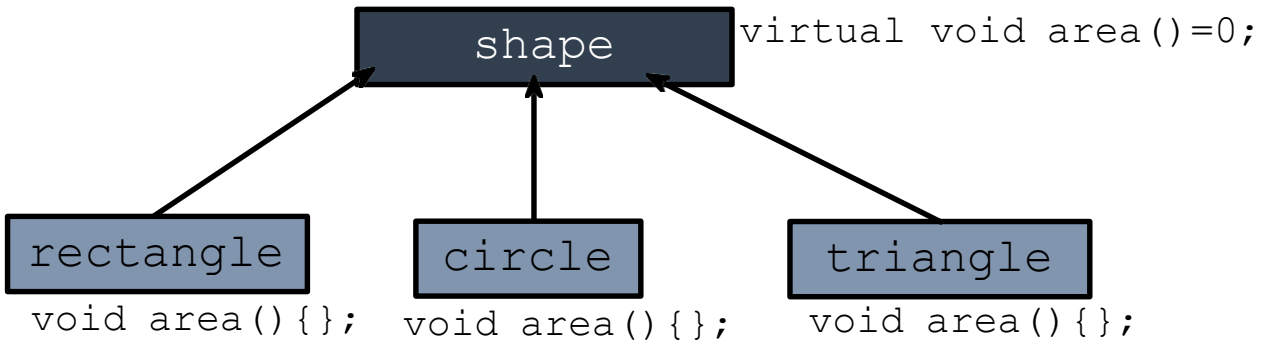
Adding a **virtual function** to the base class B makes it work.

Types of Classes

- Concrete class
  - A class which describes the functionality of the objects.
  - Class which is used to create object.
- Abstract class
  - A class which contains generic or common features that multiple derived classes can share.
  - Cannot be instantiated.(can’t create object)
- Pure abstract class
  - All the member functions of a class are pure virtual functions.
  - It is just an interface and cannot be instantiated.

Abstract Class

- An object of an abstract class cannot be created.
  - However, pointer or reference to abstract class can be created.
  - Therefore, abstract classes support run-time polymorphism.
- Pure virtual functions must be overridden in derived classes; otherwise derived classes are treated as also abstract.



Points to Remember . . .

- Only public inheritance is ‘is a’ kind of relationship.
- Private inheritance is the last stage of inheritance.
- It is not true inheritance.
  - Only advantage of private inheritance is reusability of code.
  - Do not use private inheritance. Use containment in that case.
  - Private inheritance is not ‘is a’ kind of inheritance.

Dynamic object in c++

In C++, the objects can be created at run-time. C++ supports two operators **new** and **delete** to perform memory allocation and de-allocation. These types of objects are called dynamic objects. The new operator is used to create objects dynamically and the delete operator is used to delete objects dynamically. The dynamic objects can be created with the help of pointers.

Syntax:

```
ClassName *ptr_obj;             // pointer to object

ptr_obj = new ClassName    // Dynamic object creation
delete ptr_obj;                  // Delete object dynamically
```

Interfaces in C++ (Abstract Classes)-

Abstract classes are the way to achieve abstraction in C++.
Abstraction in C++ is the process to hide the internal details
and showing functionality only. Abstraction can be achieved by two ways:

Abstract class

Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.