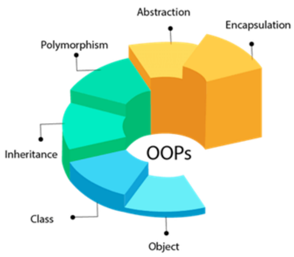
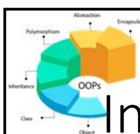


Virtual Functions and Dynamic Binding

Chapter 6

Object Oriented Programming

By DSBaral



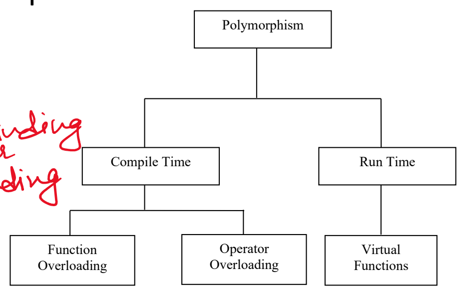
Introduction

- Polymorphism is one of the important features of C++.
- Function overloading and operator overloading are the examples of polymorphism feature in C++.
- Along with function overloading and operator overloading C++ also has other type of polymorphism called virtual function.
- In function overloading and operator overloading the selection of the function takes place at compile time while selection of the function for the virtual functions takes place at runtime.
- The classes that have virtual function are called polymorphic classes.

Introduction (Contd…)

- The objects of the polymorphic class when accessed by pointers respond differently for the same message (function call).
- Such mechanism postpones the binding of function call till runtime.
- The types of polymorphism in C++ are as follows

early binding or static binding



late binding or dynamic binding

Figure: Classification of polymorphism

Introduction (Contd…)

- Resolving the function call at compile time is called *static binding* or *early binding*.
- Resolving the function call at runtime is called *dynamic binding* or *late binding*.
- Runtime polymorphism is achieved through virtual functions
- Now let us see a condition where a derived class has overridden a function of the base as

```
class shape{
private:
    int type;
public:
    void draw(){cout<<"shape class draw";}
};
```

Introduction (Contd…)

```
class circle:public shape
{
private:
   float radius;
public:
   void draw() {cout<<"circle class draw";}
};
int main()
{
    shape sp;
    circle cir;
    shape *psp;
```

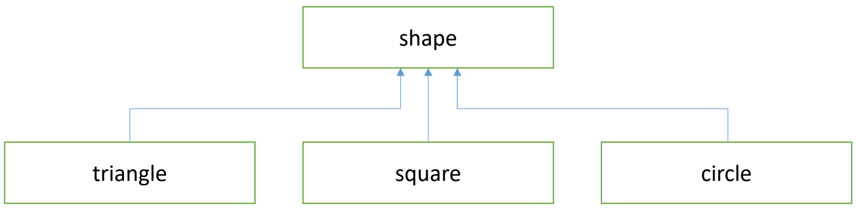

Introduction (Contd…)

```
    psp=&sp;
    psp->draw();
    psp=&cir;  //derived class is a type of base class
    psp->draw();
    return 0;
}
```

- The output of both the `draw()` function call is same, that is both the function calls invoke the `shape` class `draw()`.
- After adding `virtual` keyword in front of member function `draw()` in base class then respective draw of base and derived are called.
  - This is because the function binding is left until runtime and during runtime it call specific function depending upon which object it is pointing to.

Introduction (Contd…)

- Suppose, we create different derived classes like `triangle`, `square`, and `circle` from the base class `shape` as



Introduction (Contd…)

- If classes `triangle`, `square`, and `circle` override the virtual function `draw()` we can use array of base class shape pointer to invoke the respective member function `draw()` of derived classes.

```
shape *sptr[3];
triangele tr;
square sq;
circle cr;
sptr[0]=&tr;
sptr[1]=&sq;
sptr[2]=&cr;
for(int i=0;i<3;i++)
    sptr[i]->draw();
```

Or,

```
shape *sptr[3];
sptr[0]=new triangle;
sptr[1]=new square();
sptr[2]=new circle;
for(int i=0;i<3;i++)
    sptr[i]->draw();
```

Introduction (Contd…)

- Meaning of virtual function becomes obvious if the virtual function is overridden and the base class pointer points (holds the addresses of) the derived class object.
- Virtual functions should be defined in the public section of a class to realize its full potential benefits.
- When virtual functions are made, it allows deciding which function to be used at runtime, based on the type of object, pointed to by the base pointer, rather than the type of the pointer.
- ***Virtual functions are needed to react to the same message (function call) to an object of a class hierarchy to react differently (call functions of different class)***

Definition of Virtual Functions

- With virtual functions and function overriding, the the exact member function belonging to class is selected at runtime.
- The syntax of defining a virtual function is as follows:

```
class class_name {
private:
   //......
public:
   virtual return_type function_name(parameters…)
   {
       //body of virtual function
   }
   //......
};
```

Pure Virtual functions and Abstract Class

- Normally, when creating class hierarchy with virtual functions, in most of the cases it seems that the base class pointers are used but the base class objects are rarely created.
- Also, the virtual functions are overridden and are rarely accessed.
- Since the base class objects are rarely created and the base class virtual functions are not called, the base class virtual function can be defined with a null body as

```
class class_name{
public:
   virtual return_type function_name() = 0;
    //pure virtual function
};
```

Null Body

Pure Virtual functions and Abstract Class (Contd...)

- A virtual function with null body is called a *pure virtual function*.
- When a class has at least one pure virtual function, then the object of the class cannot be created but they can serve as base class for further derivation, however, pointer to abstract class can be created.
- The base class which has at least one pure virtual function is called *abstract base class* or just *abstract class*.
- The class whose objects can be created is called *concrete class*.

Pure Virtual functions and Abstract Class (Contd...)

- An abstract class can be used as a base for other classes. For example if we have

```
class shape
{
public:
   virtual void draw()=0;
};
```

So, when the object of this class `shape` is created as

```
shape sh; //error
```

it causes an error but pointer type of this class can be created as

```
shape *shp;//okay
```


Pure Virtual functions and Abstract Class (Contd...)

- This shape can serve as base for other classes as

```
class triangle:public shape
{
public:
   void draw()
   {cout<<"Triangle"<<endl;}
};
```

- When creating derived class from the base class having pure virtual function we must override the virtual function other wise derived class becomes abstract again.

Pure Virtual functions and Abstract Class (Contd…)

- So the pure virtual functions must be overridden if the derived class has to be a concrete class.
- The pure virtual function and the abstract class provides a framework for derived class so that they either provides the function implementation (define function) or leave their derived classes to provide implementation.
- Because of the pure virtual function the abstract class impose compulsion to the derived class to override the pure virtual function if it needs to instantiate an object or leave the other derived class in the class hierarchy to override the pure virtual function.

Virtual Destructor

- When the base class and derived class have destructors then to clean up the object both the base class and derived class destructors must be called.
- But when a derived class object pointed by the base class pointer is deleted using the `delete` operator, it calls the destructor of the base class only like other normal functions but not the destructor of the derived class.
- To clean up the object, both the destructors from base class and derived class must be called.
- To call destructor of base class and derived class when deleting derived class object pointed by base class pointer, the base class destructor must be made virtual.
- The constructors cannot be virtual.

Virtual Destructor

- The syntax for defining virtual destructor is

```
class class_name
{
    //......
public:
    //......
    virtual ~classname()
    {
          //body
    }
};
```


The `reinterpret_cast` Operator

- The `reinterpret_cast` is one among different cast operators available in C++.
- In C and older C++, single cast operators are used for conversion but it is a bad idea in C++.
- The `static_cast` operator is used for any standard type conversion and no runtime checks are performed which adds less overhead for the compiler.
- When we need to convert from one type to fundamentally different type for example from one pointer type to other, pointer to integer or integer to pointer then `static_cast` cannot be used.

The reinterpret_cast Operator (Contd…)

- The `reinterpret_cast` handles conversions between unrelated types.
- The `reinterpret_cast` operator has the following form
  ```
  reinterpret_cast<target_type>(expr);
  ```
- If the target has at least as many bits as the original value, we can convert the result back to its original type by using `reinterpret_cast` again.
- The `reinterpret_cast` is one of the crudest and potentially dangerous among the type conversion operators.
- The `reinterpret_cast` operator changes the type of section of memory without caring if it makes sense, so we should be careful.

The reinterpret_cast Operator (Contd…)

- The `reinterpret_cast` is used as follows:
  ```
  int ivar;
  double dvar;
  int *pivar;
  double *pdvar;
  pivar=reinterpret_cast<int*>(dvar);
  pdvar=reinterpret_cast<double*>(ivar);
  ```
- In this case, the source type and the target type may be represented with different number of bits which may be destructive.
- The use of `reinterpret_cast` in this way is not recommended, but in very few cases it may be the only solution.

The reinterpret_cast Operator (Contd…)

- The `reinterpret_cast` is normally used when the conversion is to be made from one pointer type to another pointer type as

```
pivar=reinterpret_cast<int*>(&dvar);
pdvar=reinterpret_cast<double*>(&ivar);
```

- Here, the number of bits required to represent address for any pointer type will be same so it is not destructive.
- The conversion from any pointer type to void pointer do not require type cast so the following statements are valid

```
void *pvoid;
pvoid=&ivar;//ok, int* to void*
pvoid=&dvar;//ok, double* to void*
```


Run-Time Type Information

- When the base class pointer is holding the address of the objects of its derived classes then the type of the object the base class pointer points is known at the runtime only.
- Since base class pointer may be used to point various objects of its derived class, it is not always possible to know in advance the type of the object the base class pointer is pointing to.
- With polymorphic base class (class with virtual function) it is possible to find out the type of the object at runtime.
- The ability of finding the object’s type at runtime is called run-time type information (RTTI).

Run-Time Type Information (Contd…)

- The operators `dynamic_cast` and `typeid` provides us the run-time type information feature.
- Usually the run-time type information is used in situation where a variety of classes are derived from the base class.
- The `dynamic_cast` operator is used to convert types between objects of derived class and base class.
- The `typeid` operator is useful for polymorphic classes, however, it can be used for determining type for any data.

The `dynamic_cast` Operator

- The `dynamic_cast` operator performs a runtime cast along with verifying the validity of the cast.
- The syntax of the dynamic cast is as follows

```
dynamic_cast<target-type>(expr);
```

- The `dynamic_cast` operator is actually used to perform cast on polymorphic types.
- When conversion is to be done from base class type to derived class type `dynamic_cast` operator is used.
  - The conversion from derived class to base class is not necessary
- The `dynamic_cast` operator returns address if the object that is pointed by the base class pointer is the derived class object.

The `dynamic_cast` Operator (Contd…)

- If the cast fails, then the `dynamic_cast` return `NULL` pointer for pointer types and it throws `bad_cast` exception.
- The `dynamic_cast` is used as follows
- If `Cow` is derived from polymorphic class `Animal` then

```
Animal *panm,anm;
Cow *pcw,cw;
panm=&cw; //ok, base pointer points to derived object
pcw=dynamic_cast<Cow *>(panm); //cast to derived type
```

Here cast is successful

The `dynamic_cast` Operator (Contd…)

- But if

```
panm=&anm; //ok, base pointer points to base object
pcw=dynamic_cast<Cow *>(panm); //cast to derived type
```

Here the cast is unsuccessful and NULL pointer is returned

- The success of the conversion can be checked as

```
if(pcw)
   cout<<"Cast succeeded"<<endl;
else
   cout<<"Cast failed"<<endl;
```

The `typeid` Operator

- The `dynamic_cast` operator is used to test that the object pointed by source type matches with the destination type.
- Instead of just checking the type sometime we may need to know the exact type of the object during runtime.
- The operator `typeid` is used to get the object's type pointed by pointer during runtime.
- The `typeid` operator returns reference to a standard library class type `type_info` defined in `<typeinfo>`.
  - The header `<typeinfo>` must be included to use `typeid`.
- The syntax of using the `typeid` operator is as follows

```
typeid(expr);
```


The `typeid` Operator (Contd...)

- The copy and assignment operators of `type_info` class are private, that is, objects of this type cannot be copied.
- The `type_info` class has overloaded == and != operators for comparison of types and provides a member function `name()` that returns the pointer to the name of the type (mangled in gcc and clang).
- The `typeid` operator can be used as follows

```
class Animal{
    public:
        virtual void display(){ }
};
class Cow:public Animal{ };
```

The `typeid` Operator (Contd…)

- The value returned by typeid is of type type_info and is used as

```
const type_info& ti1 = typeid(*anm1); //const is required
```

- Here `Animal` is polymorphic class and `Cow` is a derived class of `Animal` class, then

```
Animal *panm,anm;
Cow cw;
panm=&anm;
cout<<typeid(*panm).name(); //displays class Animal
panm=&cw;
cout<<typeid(*panm).name(); //displays class Cow
```

- But if `Animal` is not polymorphic then the above statements displays `class Animal` for both the cases.

The `typeid` Operator (Contd…)

- The comparison of `type_info` reference returned by `typeid` can also be done as follows.

```
if(typeid(*panm1)==typeid(*panm2))
     cout<<"type are same"<<endl;
else
     cout<<"type are not same"<<endl;
```

or

```
if(typeid(*panm1)!=typeid(*panm2))
     cout<<"type are not same"<<endl;
else
     cout<<"type are same"<<endl;
```