Polymorphism

Overview

- Pointers to Base Class
- Which Function Through Base Reference
- Polymorphism
- Creating Abstract Base Classes
- What is Inherited
- Final Correct Canonical Header File

2

Pointers to Base Class

- A base class pointer can point to a derived object
- A derived class pointer cannot point to a base object

```
Account acnt;
Credit crt;

Account* ac = &crt;    // OK
Credit* cr = &acnt;    // NOT POSSIBLE
```

3

POINTERS TO BASE CLASS

In a previous module we mentioned the important IS A relationship between a derived and base class. Whenever we derive a class from another class that derived class behaves as a base class. The latter is true because the derived class guarantees that it contains the same functionality as the base class and we can not remove some functionality of the base class in the derived class.

Let us take the example of the Account and Credit classes. When we declare a pointer to the Account class we can assign Credit instances to this pointer. The pointer is declared as pointing to a Account object.

Pointers to Base Class

```
Account* pacn;
Account acnt;

pacn = &acnt;
pacn->display();    // Account::display()
```

Using the pointer we can call the member functions of the Account object.
A derived object behaves the same as a base object; a Credit object therefore would behave the same as an Account object thus we can assign the Account pointer also to a Credit object because the Credit object always guarantees the same functionality as an Account object.

Base Class Functions

```
Credit crt;

pacn = &crt;
pacn->display();    // Account::display()
```

Function Visibility

- When using pointers to base classes
  - Can only call functions of base class
  - Derived member functions not accessible any more

```
Credit crt;
Account* acn = &crt;

acn->Display();         // OK, Account function
acn->SetCredit(10.0);   // NOT OK Credit function
```

4

Function Visibility

We already know how to assign derived objects to base pointers. Through this pointer we can call the member functions that are declared in the base class only.

Base Class Functions

```
Account* pacn;
Credit crt;

pacn = &crt;
pacn->display();    // Account::display()
```

The pointer in the above example points to a derived object which has more functionality than a base object but this extra functionality (e.g. set_credit()) can not be used because the pointer points to the Account(base) part of the Credit(derived) object.

Pointers to Base Class

```
class B
{
      void f1();
      void f2();
};

class D : public B
{
      void f2();
      void f3();
};

void main()
{
      B* basep = new D;
      basep->f1(); // B::f1() called
      basep->f2(); // B::f2() called
      basep->f3(); // Not allowed
      delete basep;
}
```

Polymorphism

Similar objects respond differently to the same message

- List or array of same kind of objects
- All objects receive same message (member function call)
- Different actions
  `Credit::Display()`
  `Savings::Display()`

5

POLYMORPHISM

The definition states exactly what polymorphism is. Suppose that we have a list of different accounts which contains a list of Credit and Savings objects. If we call the withdraw() for each of those objects we would get different results. The withdraw() function of a Credit object would check to see if there is enough balance using the credit_limit as well. The other object type (a savings object) would check if there is enough balance without using a credit limit.

In C++ we would create this list as an array of pointers to the base class. In this case the Account class and assign different objects to the pointers.

Array of Accounts

```
void main()
{
       Account* arr[4];    // Array of pointers to Accounts

       arr[0] = new Credit(...);
       arr[1] = new Credit(...);
       arr[2] = new Savings(...);
       arr[3] = new Credit(...);

       for(int i=0; i < 4; i++)
              cout << arr[i]->display();

       for(i=0; i < 4; i++)
              delete arr[i];

}
```

The example first creates an array of 4 Account pointers. Using the knowledge that we can assign derived objects to base pointers we can assign Credit, Savings or Account objects to any of those pointers. This example creates the objects dynamically.

In the for loop we iterate through the array and call the display() function for each instance. In the previous example we saw that this will result in 4 calls to Account::display() because a base class pointer will result in call to the base class function.
Polymorphism is that not the Account::display() function is called but the display() function of each derived class.

Polymorphism and C++

- Create member function in base class
- Make function ‘virtual’
  - Keyword in header file only

```
class Account
{
public:
  virtual void Display();
};
```

6

POLYMORPHISM AND C++

As we saw in the previous example the Account::display() function was called for all objects in the array of Account pointers.

Virtual Function

```
class Account
{
        ...
public:
        virtual void display();
};

class Credit
{
        ...
public:
        void display ();
};
```

We really would like that the display() function of all the corresponding classes would be called. This can be implemented using polymorphism.
Implementing polymorphism in C++ entails creating a so-called virtual function in the base class which than can be overridden in the derived classes.

When we use the same example again the appropriate display() function will be called (which is what we really want).

The virtual keyword does not have to be repeated in all the derived classes. It is placed only in the header file and not in the source file because it is a specifier.

Not Overriding Virtual Function

- Virtual function can be overridden (not mandatory)
- When not overridden base function specifies default behaviour.

7

NOT OVERRIDING VIRTUAL FUNCTION

If a function is declared in the base class as virtual it is not mandatory to be overridden in a derived class. When the virtual function is not overridden in the derived class the function of the base class is called. It defines the default behaviour.

Using Polymorphism

- Call function through base pointer or reference

```
void main()
{
  Account* acn;
  Credit crt;

  acn = &crt;
  acn->Display(); // Credit::Display() and not Account::Display()
}
```

8

Defining an Interface with Polymorphism

- Create a virtual function with no implementation
  - Pure Virtual member Function (PVMF)

```
class Account
{
public:
  virtual void Withdraw(double amount) = 0;
};
```

- Base class with PVMF called Abstract Base Class (ABC)

9

DEFINING AN INTERFACE WITH POLYMORPHISM

If we want to use polymorphism for our Account classes we need to create the function we want polymorphic in the base class Account. Take for example the function withdraw() this function should react differently in the different Account classes.
If we want to use polymorphism the function should be part of the base class interface and make it virtual.
The only problem is that this function has no meaning in this base class, we cannot withdraw from a Account. So we need this function for the polymorphism but we do not want to give it a body. This is possible by a technique that is called defining an interface.
The base class specifies the function withdraw but does not give it a body. We want to force the derived classes to implement this function. This will result in errors if the derived classes do not implement it, they cannot be instantiated (cannot create objects). This interface is created by adding a so-called Pure Virtual Member Function. This function describes a prototype of a function but it is not implemented. In the header file we say the function is 0, it has no corresponding code in the source file. This results in creating a so-called Abstract Base Class (ABC). This class cannot be instantiated because one of its member functions has no implementation. The derived classes have to implement this function otherwise they will become an ABC as well (via the normal inheritance scenario).

Abstract Base Class

```
class Account
{
public:
      virtual void withdraw() = 0;
};
```

Account has become an abstract base class. It is not possible to create instances of an abstract base class but we can declare pointers to an abstract base class.

Usage Abstract Base Class

```
Account acnt;       // NOT POSSIBLE
Account* pacn;      // Possible
```

Abstract Base Classes

- Describes an interface
- An ABC is a base class with at least one PVMF
- ABCs have no instances, they are place holders for concrete classes
- Can create a pointer to ABC
- Derived concrete classes MUST implement the PVMF or they will be an ABC as too

10

ABSTRACT BASE CLASSES

It is not possible to create an instance of an abstract base class; an abstract base class is just a place holder or container. In practice concrete classes are derived from the abstract class. Furthermore, an abstract base class can be used to define an interface for which derived classes provide a variety of implementations.

Examples of abstract classes are:

- Shape class for forms in two-dimensional geometry.
- WINDOW class for classes that occur in software development in GUIs (graphical user interfaces).
- Device class as abstract base class for network devices (e.g. hosts, routers, bridges, ...).
- MATRIX class which serves as a base class for special types of matrices (e.g. full, sparse, block diagonal etc.).

We note that abstract base classes are not meant to have instances but they serve as containers for existing classes and new derived classes. It is however, possible to create a pointer or a reference to an abstract base class. It is also not possible to use an instance of an abstract class as an argument in a function argument list.

A given class is deemed to be abstract if it contains at least one pure virtual function.

Virtual Destructors (1/2)

- Problem with inheritance and pointers is that the default destructor of the base class is called

```
void main()
{
  Account* acn;
  acn = new Credit();
  delete acn; // By default destructor of Account called
}
```

11

Virtual Destructors

Remember from the beginning of this module that if we use pointers to base classes we always use the functionality of the base class the member functions of the base class are called unless they are specified as virtual.

Virtual Destructor

```
class Account
{
public:
       Account();                  // Default constructor
       ~Account();                 // Destructor
};

class Credit : public Account
{
public:
       Credit();                   // Default constructor
       ~Credit();                  // Destructor
};

void main()
{
       Account* pacnt;

       pacnt = new Credit  ;       // Default Credit constructor called

       delete pacnt;               // Destructor of base class Account called
}
```

The example shows that if we create a derived object dynamically which is assigned to a base pointer the constructor of the derived class is called. But when we delete the object the destructor of the base class is called. This is because the pointer uses the functionality of the base class unless that specific function is made virtual. The destructor was not virtual so it will call the destructor of the class Account and not that of Credit. Specifying that the destructor should be virtual solves this problem.

Virtual Destructors (2/2)

- Solution declare destructor virtual

```
class Account
{
public:
  virtual ~Account();
};
```

- Always use virtual destructors in all classes!
- Not necessary to declare virtual in derived classes, but advisable

12