哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

高级语言程序设计
High-level Language Programming

Lecture 11 Polymorphism

Yitian Shao (shaoyitian@hit.edu.cn)
School of Computer Science and Technology

Polymorphism

Course Overview

- Polymorphism: basic idea
- Templates
- Virtual function
- Abstract base classes

11.1 What is polymorphism

The word ***polymorphism*** is derived from a Greek word meaning “many forms”.

Polymorphism is one of the most important features in object-oriented programming and refers to the **ability of different objects to respond differently to the same command** (or ‘message’ in object-oriented programming terminology).

11.1 What is polymorphism

- For example, the command ‘**open**’ means different things when applied to different objects.
  - Opening a bank account is very different from opening a window, which is different from opening a window on a computer screen.

  - The command (‘**open**’) is the **same**, but depending on what it is applied to (the object) the **resultant actions are different**.

11.1 What is polymorphism

```
class advanced_computer
{
public:
void hello()
{
  cout << "Hello from the Advanced Computer" << endl ;
}
} ;

class simple computer
{
public:
void hello()
{
  cout << "Hello from the Simple Computer" << endl ;
}
} ;

```

```
main()
{
  advanced_computer HAL ;
  simple_computer PC ;
  HAL.hello() ;
  PC.hello() ;
}
```

Polymorphism:
The same command to different objects invokes different actions.

11.1 What is polymorphism

- Polymorphism can be divided into two broad categories: ***static*** (or compile-time) and ***dynamic*** (or run-time) *polymorphism*.
- Static polymorphism occurs when the program is being compiled, dynamic polymorphism when the program is actually running.
  - *static* or *early binding:* normally, which function a call refers to is made at compile time. The compiler knows which function to call based on the object that calls it.
  - *dynamic* or *late binding*: it is left until run-time to determine which function should be called. This decision is based on the object making the call.

11.2 Static and dynamic polymorphism

- C++ has three **static polymorphism** mechanisms:
  - function overloading （Lecture 8 Functions）
  - operator overloading
  - Templates

- **Dynamic polymorphism** of C++: **virtual function**

function overloading

```
int sum_array ( const: int array [] , int no_of_elements )
{
    int total = 0;

    for(int index = 0 ; index < no_of_elements ; index ++ )
        total += array[index] ;
    return total;
}

int sum_array ( const int array[][2] int no_of_rows )
{
    int total = 0;

    for(int row = 0 ; index < no_of_rows ; row ++ )
    {
        for(int col = 0 ; col < 2 ; col++ )
            total += array[row][col] ;
    }
    return total;
}
```

Functions with the same names and different parameter lists mean function overloading.

11.2 Static and dynamic polymorphism

- Programmer can use some operator symbols to define special **member functions** of a class, providing convenient notations for object behaviors

- The + operator, for example, is appropriate for strings as well and is taken to mean concatenation. This means that **the operator + has a different meaning for numeric data types than for string data types**.

- When you create a new class you can redefine or overload existing operators such as +, - , *, /, %, etc. (Existing operators can be overloaded for specific use in a class by writing operator functions for the class)

11.3 Template

- A ***template*** is a **framework** for generating a **class** or a **function**.

  - *Instead* of explicitly specifying the **data types** used in a class or a function, **parameters are used**.

  - When actual data types are assigned to the parameters, the class or function is generated by the compiler.

11.3 Template

- A function to find the maximum of two **integer** values

```
int maximum( const int n1, const int n2 )
{
  if ( n1 > n2 )
    return n1 ;
  else
    return n2 ;
}
```

- A function to find the maximum of two **floating-point** values a nearly identical

```
float maximum( const float n1, const float n2 )
{
  if ( n1 > n2 )
    return n1 ;
  else
    return n2 ;
}
```

The reason two different functions are required is simply that the function header is different in the two functions.

11.3 Template

- **Function templates** allow both of these functions to be replaced by a single function in which the parameter data types are replaced by a parameter T (for type) giving a generic or type-less function that can be used for all data types.

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

template <typename T>
T maximum( const T n1, const T n2 )
{
  if ( n1 > n2 )
    return n1 ;
  else
    return n2 ;
}
```

T is called the **type parameters**. T is normally used, but any valid name can be used.

11.3 Template

- A **function template** declaration consists of the keyword template and a list of one or more data type parameters.

- The data **type parameter** is preceded by either the keyword class or the more meaningful keyword **typename**.

- **Type parameters** are enclosed within angle brackets < and >. When multiple data type parameters are used, they are separated by commas.

11.3 Template

```
6   template <typename T>

15 main()
16 {
17    char c1 = 'a', c2 = 'b' ;
18    int i1 = 1, i2 = 2 ;
19    float f1 = 2.5, f2 = 3.5 ;
20
21    cout << maximum( c1, c2 ) << endl ;  //  a maximum() for chars
22    cout << maximum( i1, i2 ) << endl ;  //  a maximum() for ints
23    cout << maximum( f1, f2 ) << endl ;  //  a maximum() for floats
24 }
```

This process is called *instantiation of the template* and the result is *a* conventional function generated by the compiler.

```
template <typename T>
T maximum( const T n1, const T n2
{
  if ( n1 > n2 )
    return n1 ;
  else
    return n2 ;
}
```

Line 21 has the effect of replacing the type parameter *T* in line 6 with the data type of the arguments c1 and c2, i.e. char. The compiler generates a function with the prototype.

11.3 Template

- Instead of using three separate functions in the program, **a single function template is used.**
  - The program **source file is smaller**, but since the compiler generates three separate functions from the template, **the executable file size remains the same.**
- Template definitions can be placed in a header file

```
// Demonstration of including a function
#include<iostream>
#include "maximum.h"
using namespace std ;

main()
{
    char c1 = 'a', c2 = 'b' ;
    int i1 = 1, i2 = 2 ;
    float f1 = 2.5, f2 = 3.5 ;

    cout << maximum( c1, c2 ) << endl ;
    cout << maximum( i1, i2 ) << endl ;
    cout << maximum( f1, f2 ) << endl ;
}
```

11.3 Template

- Templates can be also used for generating entire C++ classes.
  - templates allow one class to be written for **all data types**

- Class templates are also known as *parameterized types*. A **parameterized type** is a data type defined in terms of other data types, some of which are unspecified.

- To create templates: 1) **write a non-template data type** specific version of the function or class. 2) when the data type specific version is working satisfactorily, **replace the specific data types with template parameters**.

11.3 Template

Class Templates Example

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

template <class T>
class Number {
   private:
    // Variable of type T
    T num;

   public:
    // constructor
    Number(T n) : num(n) {}

    T getNum() {
        return num;
    }
};
```

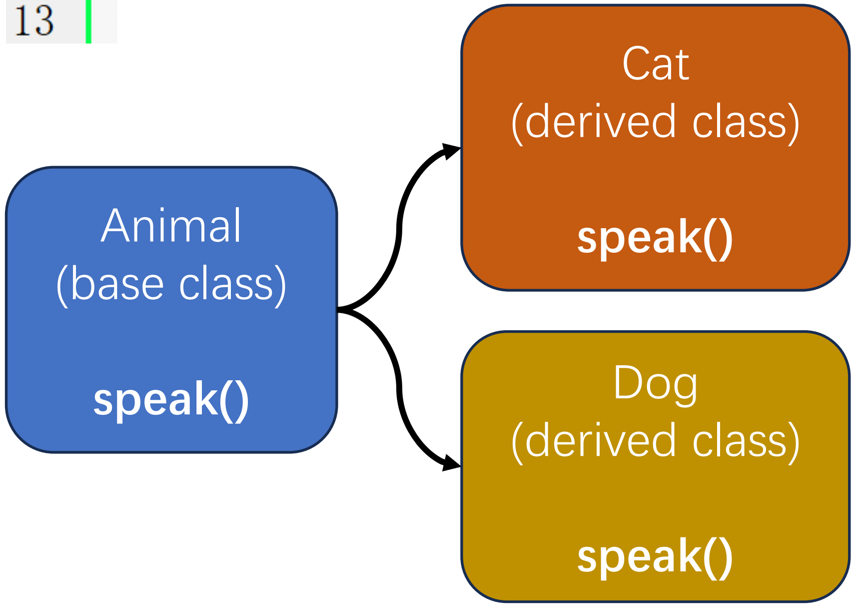
```cpp
int main() {

    // create object with int type
    Number<int> numberInt(7);

    // create object with double type
    Number<double> numberDouble(7.7);

    cout << "int Number = " << numberInt.getNum() << endl;
    cout << "double Number = " << numberDouble.getNum() << endl;

    return 0;
}
```

11.4 Virtual functions

- C++ uses **virtual functions** to implement dynamic binding.
- Virtual functions allow polymorphism to work in all situations.
- The keyword **virtual** is optional in the derived classes, although it is probably a good idea to include it for the sake of clarity.

11.4 Virtual functions

```cpp
#include <iostream>
#include <string>

using namespace std;

class Animal
{
public:
    Animal() {};

    string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat() {};

    string speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog() {};

    string speak() { return "Woof"; }
};

void Says(Animal& animal)
{
    cout << animal.speak() << '\n';
}
```



11.4 Virtual functions

```cpp
void Says(Animal& animal)
{
    cout << animal.speak() << '\n';
}

int main()
{
    Animal myAnimal;
    Cat myCat;
    Dog myDog;

    cout << "My animal (base class) says ";
    Says(myAnimal);

    cout << "My cat says ";
    Says(myCat);

    cout << "My dog says ";
    Says(myDog);
}
```

Code execution result:

```
My animal (base class) says ???
My cat says ???
My dog says ???
```

11.4 Virtual functions

```cpp
class Animal
{
public:
    Animal() {};

    virtual string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat() {};

    string speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog() {};

    string speak() { return "Woof"; }
};

void Says(Animal& animal)
{
    cout << animal.speak() << '\n';
}

int main()
{
    Animal myAnimal;
    Cat myCat;
    Dog myDog;

    cout << "My animal (base class) says ";
    Says(myAnimal);

    cout << "My cat says ";
    Says(myCat);

    cout << "My dog says ";
    Says(myDog);
}
```

Code execution result:

```
My animal (base class) says ???
My cat says Meow
My dog says Woof
```

11.4 Virtual functions

- **When to use virtual functions**
  - There are memory and execution time overheads associated with virtual functions, so be judicious in the choice of whether a base class function is virtual or not.
  - In general, declare a base class member function as **virtual** if it **may be overridden in a derived class.**
- **Overriding and overloading**
  - Overloading is a compiler technique that distinguishes between **functions with the same name** but with different parameter lists. Function overloading is covered in lecture 8.
  - Overriding occurs in inheritance when **a member function of a derived class has the same name and the same parameters as a member function of its base class.**

11.5 Abstract base classes

- A base class that contains a pure virtual function is called an **abstract base class**.
  - The base class member function has no code and is assigned to 0

virtual void display_details() = 0 ;

  - A class member function defined in this way is called a ***pure virtual function***.
  - A pure virtual function is required by the compiler but doesn't actually do anything.
  - A pure virtual function is overridden in all the derived classes and hence there is no need to implement it.

11.5 Abstract base classes

- **Abstract base classes** are not used to create objects, but exist **solely as a base for deriving other classes**.

- It is **not** possible to define an object of an **abstract base class**, i.e. objects of an abstract base class **cannot be instantiated**.

```
class Animal    //Abstract base class
{
  public :

  virtual void speak() = 0; //Pure virtual Function
};
```

```
Animal myAnimal; // error : variable of an abstract class
```