# C/C++ Programming Language

CS205 Spring

Feng Zheng

2019.05.23


南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY


# Content

- Review
- Classes with Object Members
- Private Inheritance
- Multiple Inheritance
- Class Templates

# Brief Review

# Review

- Class Inheritance
- Static and Dynamic Binding
- Access Control: protected
- Inheritance and DMA


# Reuse of Code

- Do you remember public inheritance?
- More choices?
  - Class members
    - ✓ Referred to as containment or composition or layering
  - Private or protected inheritance
    - ✓ *has-a* relationships
- Do you remember function templates?
- Class templates for reuse of code
  - A class template lets you define a class in generic terms
  - Then use the template to create specific classes defined for specific types

# Classes with Object Members


# Two Classes for Defining Student Class

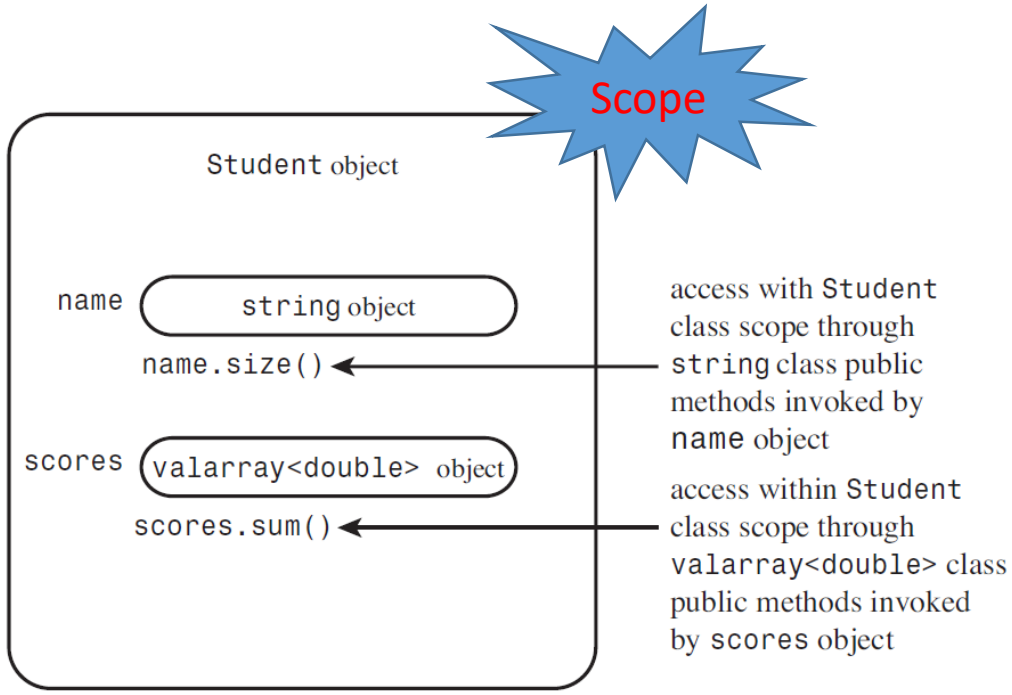
- What is a student?
  - Someone with an identifying name and a set of quiz scores?
  - Two members: one for the name and one for the scores
- C++ string class
- The valarray Class
  - Examples of constructor

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;          // an array of double, size 0
valarray<int> v2(8);          // an array of 8 int elements
valarray<int> v3(10,8);       // an array of 8 int elements,
                              // each set to 10
valarray<double> v4(gpa, 4); // an array of 4 elements
                  // initialized to the first 4 elements of gpa
```

  - A few of the methods
    - ✓ size(): return the number of elements
    - ✓ sum(): return the sum of the elements
    - ✓ max(): return the largest element
    - ✓ min(): return the smallest element
    - ✓ operator[](): provide access to individual elements

# Define The Student Class

- has-a relationship
  - A student has a name,
  - A student has an array of scores
- Interfaces and implementations
  - For public inheritance (is-a), a class can inherit an implementation
  - Not inheriting the interface is part of the has-a relationship
    - ✓ Example 1: string overloads the + operator, but it doesn't make sense to concatenate two Student objects
    - ✓ Example 2: can use the operator<() method from the string interface to sort Student objects by name




```
class Student
{
private
    string name;
    valarray<double> scores;
    …
};
```


# Using the New Student Class

- See program example 1
- Some points
  - typedef enables the remaining code to use the more convenient notation
  - explicit constructors
    - ✓ With one argument, constructors serve as an implicit conversion function
    - ✓ Using explicit turns off implicit conversions
  - Initializing contained objects
    - ✓ For inherited objects, constructors use the class name in the member initializer list to invoke a specific base-class constructor
    - ✓ For member objects, constructors use the member name
  - Using an interface for a contained object
    - ✓ The interface for a contained object isn't public, but it can be used within the class
    - ✓ A friend function (access private) uses the string version of the << operator

# Private Inheritance


# Implementing a New has-a Relationship

- Private inheritance
  - public and protected members of the base class become private members of the derived class
    - ✓ Methods of base class don't become the public interface of the derived object
    - ✓ They can be used inside the member functions of the derived class
    - ✓ The derived class does not inherit the base-class interface
  - What is the difference to the public inheritance?
    - ✓ Public inheritance: inherit the base-class interface
  - What is the relationship with the containment?
    - ✓ Acquire the implementation: the same
    - ✓ Don't acquire the interface: the same

```
class Student : private std::string, private std::valarray<double>
{
public:
    ...
};
```


# Initializing

- What is the difference to the containment?
  - Containment provides explicitly named objects as members
  - Private inheritance provides nameless subobjects as inherited members
- Initializing base-class components
  - Use the class name instead of a member name (containment) to identify a constructor
  - The same to public inheritance

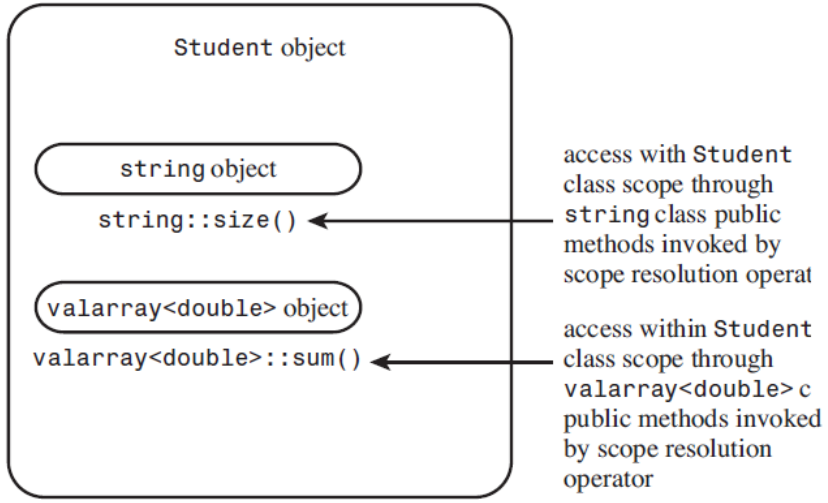
```
Student(const char * str, const double * pd, int n)
    : std::string(str), ArrayDb(pd, n) {}  // use class names for inheritance

Student(const char * str, const double * pd, int n)
    : name(str), scores(pd, n) {}      // use object names for containment
```

# Accessing Base-Class Methods

- Private inheritance limits the use of base-class methods to within derived-class methods
- Inheritance lets you use the class name and the scope-resolution operator to invoke base-class methods

typedef std::valarray<double> ArrayDb;

```
double Student::Average() const
{
    if (ArrayDb::size() > 0)
        return ArrayDb::sum()/ArrayDb::size();
    else
        return 0;
}
```




# More Accessing

- What if you need the base-class object itself?
- Accessing base-class objects

```
const string & Student::Name() const
{
    return (const string &) *this;
}
```

  - Use the type cast to create a reference
  - Return a reference to the inherited string object residing in the invoking Student object
- Accessing base-class friends
  - Use an explicit type cast
  - Have a name for object

```
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << (const String &) stu << ":\n";
...

}
```

- See program example 2


# Containment or Private Inheritance?

- Containment
  - Easier to follow
  - Explicitly named objects representing the contained classes
- Inheritance
  - Appear more abstract
  - Have problems:
    - ✓ Separate base classes: have methods with the same name
    - ✓ Separate base classes: share a common ancestor
    - ✓ Limit to a single object
- Use private inheritance
  - If new class needs to access protected members in the original class
  - If new class needs to redefine virtual functions
  - Redefined functions would be usable just within the class, not publicly


# Protected Inheritance

```
class Student : protected std::string,
                protected std::valarray<double>
{...};
```

- A variation on private inheritance
  - Public and protected members become protected members of the derived class
  - The interface is available to the derived class but not to the outside world (Inherit the interface?)
- Difference between private and protected inheritance
  - When derive another class from the derived class
    - ✓ With private inheritance, this third-generation class doesn't get the internal use of the base-class interface
      - Public base-class methods become private in the derived class, and private members and methods can't be directly accessed by the next level of derivation
    - ✓ With protected inheritance, public base-class methods become protected in the second generation and so are available internally to the next level of derivation


# Varieties of Inheritance

- Implicit upcasting means that a base-class pointer or reference can be used to refer to a derived class object without using an explicit type cast

| Property | Public Inheritance | Protected Inheritance | Private Inheritance |
|---|---|---|---|
| Public members become | Public members of the derived class | Protected members of the derived class | Private members of the derived class |
| Protected mem-bers become | Protected members of the derived class | Protected members of the derived class | Private members of the derived class |
| Private members become | Accessible only through the base-class interface | Accessible only through the base-class interface | Accessible only through the base-class interface |
| Implicit upcasting | Yes | Yes (but only the derived class) within | No |


# Redefining Access with Using

- One option
  - Define a derived-class method that uses the base-class method

```
double Student::sum() const    // public Student method
{
    return std::valarray<double>::sum(); // use privately-inherited method
}
```

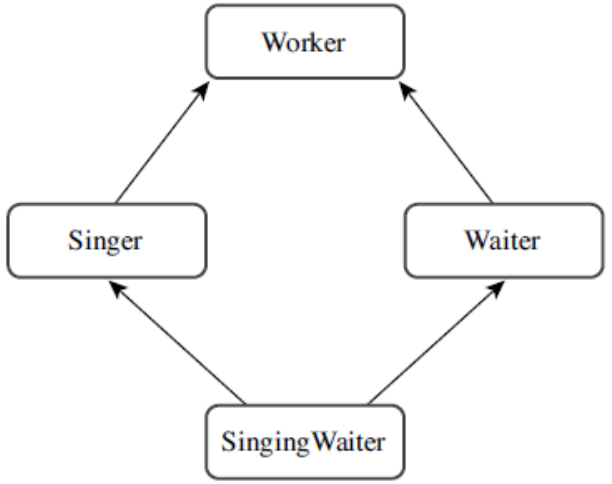
- Another: wrapping one function call in another
  - Use a using declaration
    - ✓ Announce that a particular base-class member can be used by the derived class even though the derivation is private
    - ✓ No parentheses, no function signatures, no return types

```
class Student : private std::string, private std::valarray<double>
{
...
public:
    using std::valarray<double>::min;
    using std::valarray<double>::max;
    using std::valarray<double>::operator[];
};
```

# Multiple Inheritance

# Multiple Inheritance (Public MI)

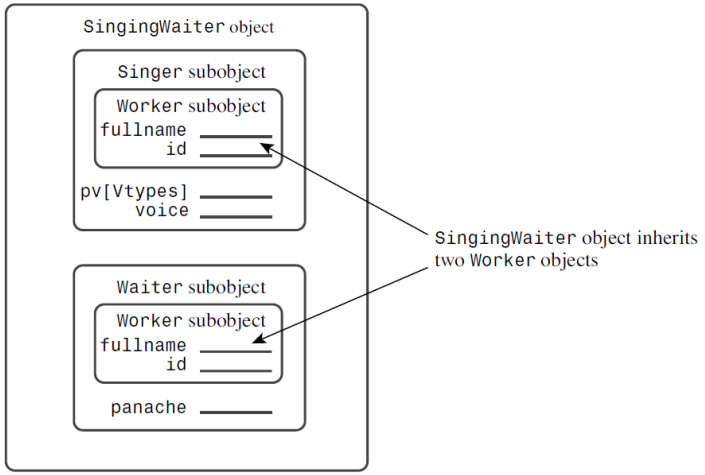
- MI describes a class that has more than one immediate base class
- An example:
  - If you have a Waiter class and a Singer class, you could derive a SingingWaiter class from the two




```
class SingingWaiter : public Waiter, public Singer {...};

class SingingWaiter : public Waiter, Singer {...}; // Singer is a private base
```

```
class Singer : public Worker { ...};
class Waiter : public Worker { ...};
class SingingWaiter : public Singer, public Waiter { ...};
```

- New problems
  - Inheriting different methods with the same name from two different base classes
  - Inheriting multiple instances of a class via two or more related immediate base classes



# Class Templates


# Defining a Class Template

- Reuse code
  - Inheritance (public, private or protected)
  - Containment
  - Function template
- C++'s class templates provide a better way to generate class declarations
  - Preface a template
  - Change the class qualifier

```
template <class Type>
template <typename Type>  // newer choice
Stack<Type>::
```

- See program example 3

```
typedef unsigned long Item;


class Stack
{
private:
    enum {MAX = 10};     // constant specific to class
    Item items[MAX];     // holds stack items
    int top;             // index for top stack item
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);   // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);          // pop top into item
};
```


# An Array Template Example and Non-Type Arguments

- Use a template argument to provide the size for a array

```
template <class T, int n>
```

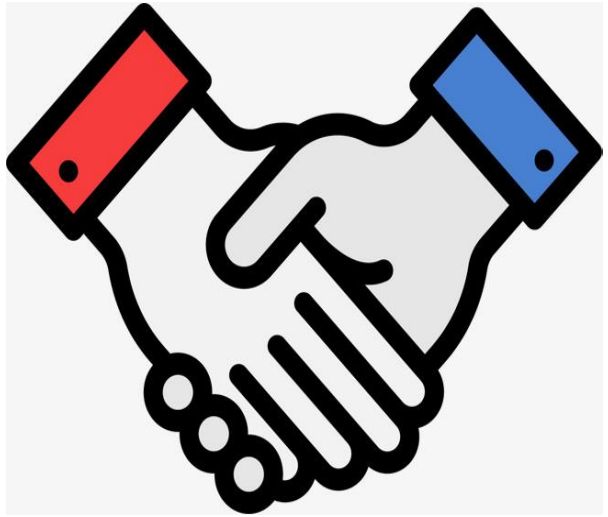
  - class (typename) identifies T as a type parameter, or type argument
  - int identifies n as being an int type
  - This second parameter specifies a type instead of acting as a generic name for a type, is called a non-type, or expression, argument
  - Expression arguments have some restrictions
    - ✓ An expression can be an integer, an enumeration, a reference, or a pointer
    - ✓ double is ruled out, but double & and double * are allowed
    - ✓ Also the template code can't alter the value of the argument or take its address
- See program example 4


# More About the Array Template

- Drawbacks: compared to constructor approach
  - Generate multiple separate class declaration
  - Constructor approach is more versatile
    - ✓ The array size is stored as a class member rather than being hard-coded into the definition
- Template versatility
  - Serve as base classes
  - Can be component classes
  - Can be type arguments
  - Using a template recursively

```
ArrayTP< ArrayTP<int,5>, 10> twodee;
```

```
template <typename T>   // or <class T>
class Array
{
private:
    T entry;
    ...
};

template <typename Type>
class GrowArray : public Array<Type> {...};  // inheritance

template <typename Tp>
class Stack
{
       Array<Tp> ar;       // use an Array<> as a component
      ...
};
...
Array < Stack<int> > asi;  // an array of stacks of int
```



# Thanks

zhengf@sustech.edu.cn