C/C++ Programming Language

CS205 Spring

Feng Zheng

Lecture 12

南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Content

- Review
- Class Inheritance
- Static and Dynamic Binding
- Access Control: protected
- Inheritance and Dynamic Memory Allocation
- Class Design Review

Brief Review

Review

- An example of causing problems (auto generated functions)
  - Default constructors
  - Copy constructors
  - Assignment operators
- Improved class example of string
  - Comparison members
  - Static class member functions
  - Assignment operator overloading
- Using pointers to objects

Class Inheritance

The Reasons for Inheritance

- Traditional C
  - Function libraries: the number of functions is limited
  - Many vendors furnish specialized C libraries
    - ✓ Problem 1: can't extend or modify the functions to meet particular needs
    - ✓ Problem 2: run the risk of unintentionally modifying or altering the relationships
- One of the main goals of object-oriented programming is to provide reusable code
  - Often class libraries are available in source code, which means you can modify them to meet your needs
  - Class inheritance: lets you derive new classes from old ones, with the derived class inheriting the properties of the base class

How to Inheriting a Fortune: Add and Modify

- What is the fortune of a class?
- Add functionality to an existing class
  - For example, given a basic array class, you could add arithmetic operations
- Add to the data that a class represents
  - For example, given a basic string class, you could derive a class that adds a data member representing a color to be used when displaying the string
- Modify how a class method behaves
  - For example, given a Passenger class that represents the services provided to an airline passenger, you can derive a FirstClassPassenger class that provides a higher level of services

Deriving a Class

- Beginning with a simple base class
  - Run tblt.h, tabtenn0.cpp, usett0.cpp
- How to include a new member (point rating)?

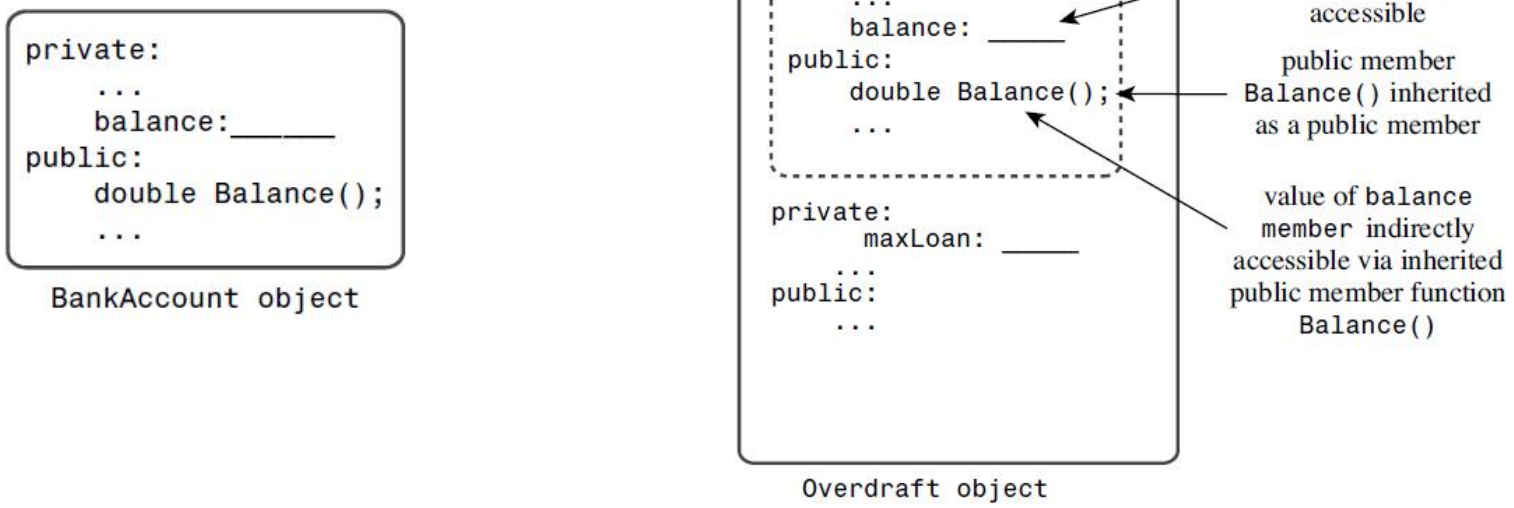
```
// RatedPlayer derives from the TableTennisPlayer base class
class RatedPlayer : public TableTennisPlayer
{
...
};
```

- Derive a class
  - Header: the colon ":"
  - Header: public derivation:
    - ✓ The public members become public members of the derived class
    - ✓ The private portions become part of the derived class, but they can be accessed only through public and **protected** methods of the **base** class

More Operations for Derivation

- What needs to be added to the inherited features?
  - A derived class needs its own constructors
  - A derived class can add additional data members and member functions as needed

```
class Overdraft : public BankAccount {...};
```



Constructors: Access Considerations

- A derived class
  - Don't have direct access to the private members of the base class
  - Has to work through the base-class methods
    - ✓ The derived-class constructors have to use the base-class constructors
    - ✓ A program constructs a derived-class object, it first constructs the base-class object
- Key points
  - The base-class object is constructed first
  - The derived-class constructor should
    - ✓ pass information to a base-class constructor via a member initializer list
    - ✓ initialize the data members that were added to the derived class
- Run tblt1.h, tabtenn1.cpp, usett1.cpp

Special Relationships Between Derived and Base Classes

- Relationships
  - A derived-class object
    - ✓ Can use base-class methods, provided that the methods are not private (question?)
  - A base-class
    - ✓ Pointer can point to a derived class object without an explicit type cast
    - ✓ Reference can refer to a derived-class object without an explicit type cast
    - ✓ Pointer or reference can invoke **just** base-class methods but couldn't use the derived-class method

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
rplayer1.Name();  // derived object uses base method
```

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();    // invoke Name() with reference
pt->Name();   // invoke Name() with pointer
```

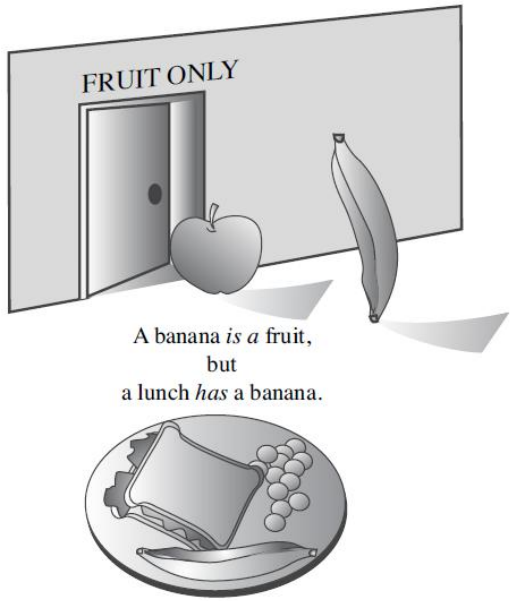
More Relationships Between Two Classes

- Generally, references and pointer types should match but this rule is relaxed for inheritance
  - The rule relaxation is just in one direction
  - Functions defined with base-class reference or pointer arguments can be used with either base-class or derived-class objects
  - **Why**???

```
TableTennisPlayer player("Betsy", "Bloop", true);
RatedPlayer & rr = player;      // NOT ALLOWED
RatedPlayer * pr = player;      // NOT ALLOWED
```

Inheritance: An *Is-a* Relationship

- Public inheritance is the most common inheritance form
  - Model an *is-a* relationship
  - An object of a derived class should be an object of the base class
  - Anything you do with a base-class object, you should be able to do with a derived-class object
- Five relationships
  - is-a-kind-of (is-a): apple->fruit
  - has-a: meal->apple
  - is-like-a: lawyer->shark
  - is-implemented-as-a: stack->array
  - uses-a: computer->printer



A banana *is a* fruit,
but
a lunch *has* a banana.

Polymorphic Public Inheritance

- How to do when you want a method to behave differently for the derived class than it does for the base class?
- Polymorphic: have multiple behaviors for a method
  - Redefining base-class methods in a derived class
  - Using virtual methods: keyword- **virtual**
- Run usebrass1.cpp, brass.cpp, brass.h
- An array of pointers: run usebrass2.cpp
  - Showing virtual method behavior

Static and Dynamic Binding

Binding

- What is binding?
  - Interpreting a function call in the source code as executing a particular block of function code
  - Compiler has to look at the function arguments as well as the function name to figure out which function to use
    - ✓ Static binding (or early binding): take place during compilation
  - Dynamic binding (or late binding): the correct virtual method is selected as the program runs
    - ✓ Virtual functions
    - ✓ Take place during program running

Pointer and Reference Type Compatibility

- General rules
  - Doesn't assign an address of one type to a pointer of another type
  - Nor does it let a reference to one type refer to another type
- Public inheritance
  - Upcasting: converting a derived-class reference or pointer to a base-class reference or pointer
    - ✓ This rule is part of expressing the *is-a* relationship
    - ✓ Using an implicit type cast
  - Downcasting: converting a base-class pointer or reference to a derived-class pointer or reference
    - ✓ It is not a is-a relationship
    - ✓ Using an explicit type cast

Virtual Member Functions and Dynamic Binding

- An example: invoking a method with a reference or pointer

```
BrassPlus ophelia;     // derived-class object
Brass * bp;            // base-class pointer
bp = &ophelia;         // Brass pointer to BrassPlus object
bp->ViewAcct();        // which version?
```

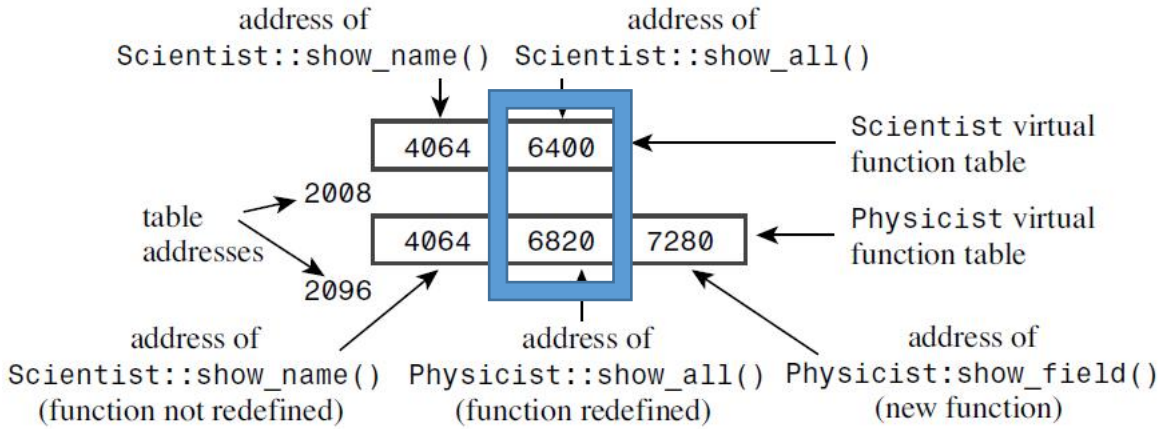
  - Nonvirtual function: the compiler uses static binding
  - Virtual function: the compiler uses dynamic binding
- Questions:
  - Why have two kinds of binding?
  - If dynamic binding is so good, why isn't it the default?
  - How does dynamic binding work?

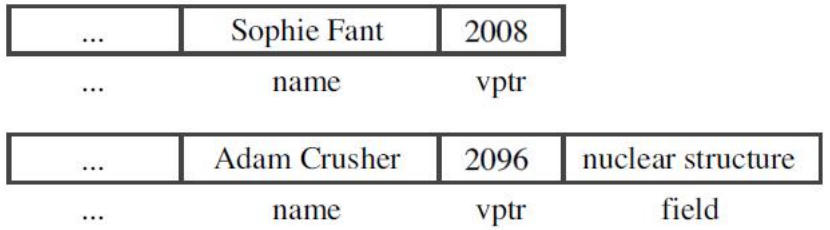
Why and How it works

- Why two kinds of binding and why static is the default
  - Efficiency: static binding is more efficient
  - A conceptual model
    - ✓ Making this function nonvirtual in base class, if no need to redefine it
    - ✓ Reserving the virtual label just for methods expected to be redefined
- How virtual functions work
  - Leave the implementation up to the compiler writer
  - Add a hidden pointer member to each object
  - Hold a pointer to an array of function addresses: virtual function table (vtbl)
  - Contain the addresses of the virtual functions of that class
  - An object of a derived class contains a pointer to a separate table of addresses
    - ✓ Provide a new definition of a virtual function, the vtbl holds the address of the new function
    - ✓ Doesn't redefine the virtual function, the vtbl holds the address of the original version of the function
    - ✓ Define a new function and makes it virtual, its address is added to the vtbl

A Virtual Function Mechanism

```
class Scientist{
{
    ...
    char name[40];
public:
    virtual void show_name();
    virtual void show_all();
    ...
};
class Physicist : public Scientist
{
    ...
    char field[40];
public;
    void show_all(); // redefined
    virtual void show_field(); // new
    ...
};
```





`Scientist` object with hidden pointer member `vptr`, which points to `Scientist` virtual function table

`Physicist` object with hidden pointer member `vptr`, which points to `Physicist` virtual function table

```
Physicist adam("Adam Crusher", "nuclear structure");
Scientist * psc = &adam;
psc->show_all();
```

1. Find value of `psc->vptr`, which is 2096.
2. Go to table at `2096`.
3. Find address of second function in table, which is `6820`.
4. Go to that address `(6820)` and execute the function found there.

Things to Know About Virtual Methods

- Main points
  - virtual makes the function virtual for the base and all derived classes
  - If a virtual method is invoked by a reference or a pointer, the program uses the method defined for the ***object*** type
  - If a class will be used as a base class for inheritance and the methods which may have to be redefined in derived classes should be virtual
  - Constructors can't be virtual
  - Destructors should be virtual unless a class isn't as a base class
    - ✓ Provide a base class with a virtual destructor even if the class doesn't need a destructor
  - Friends can't be virtual functions
  - Fail to redefine a function
    - ✓ Use the base class version of the function
    - ✓ Use the most recently defined version of the function, if there is a long chain

Access Control: protected

Reasons for Protected Members

- Two introduced keywords
  - public and private: control access to class members
- The protected keyword like private
  - The outside world can access class members in a protected section only by using public class members
- The difference between private and protected
  - Members of a derived class can access protected members directly, but cannot directly access private members of the base class
- Protected access control can be quite useful for member functions, giving derived classes access to internal functions that are not available publicly

Inheritance and Dynamic Memory Allocation

Questions for Inheritance and Dynamic Memory Allocation

- How does inheritance interact with dynamic memory allocation?
  - More specifically, if a base class uses dynamic memory allocation and redefines assignment and a copy constructor, how does that affect the implementation of the derived class?
- The answer depends on the nature of the derived class
  - If the derived class does not itself use dynamic memory allocation, you needn't take any special steps.
  - If the derived class does also use dynamic memory allocation, then there are a couple new tricks to learn

Case 1: Derived Class Doesn't Use new

- An example of no new used in derived
  - Base class has included constructors use new, a destructor, a copy constructor, and an overloaded assignment operator
  - You needn't to define an **explicit** destructor, copy constructor, and assignment operator for the derived class
    - ✓ Default destructor for a derived class always does something
    - ✓ Default copy constructor does member-wise copying
    - ✓ The same to overloaded assignment operators

```
//  Base Class Using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
...
};
```

```
// derived class without DMA
class lacksDMA :public baseDMA
{
private:
    char color[40];
public:
...
};
```

Case 2: Derived Class Does Use new

- An example of new used in derived class
  - Have to define an explicit destructor, copy constructor, and assignment operator for the derived class
    - ✓ A derived class destructor automatically calls the base-class destructor, so its own responsibility is to clean up after what the derived-class constructors do
    - ✓ Copy constructors and assignment operator follows the usual patterns
- An Inheritance Example with Dynamic Memory Allocation and Friends
  - Run dma.h, dma.cpp, usedma.cpp

```
// derived class with DMA
class hasDMA :public baseDMA
{
private:
    char * style;  // use new in constructors
public:
...
};
```

```
hasDMA::hasDMA(const hasDMA & hs)
        : baseDMA(hs)
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}
```

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;
    baseDMA::operator=(hs);  // copy base portion
    delete [] style;         // prepare for new style
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
    return *this;
}
```

Class Design Review

Class Design Review

- Member functions that the compiler generates for you (5)
  - Default constructors
  - Copy constructors: a constant reference for argument
  - Assignment operators (Don't confuse assignment with initialization)
- Constructor considerations (default, copy, conversion)
  - ✓Different from other class methods
  - ✓Constructors aren't inherited
- Destructor considerations
  - ✓Define an explicit destructor if new is used in the constructors
  - ✓Provide a virtual destructor for base class

Class Design Review

- Conversion considerations (type cast)
  - ✓ Any constructor that can be invoked with exactly one argument defines conversion from the argument type to the class type
  - ✓ A conversion function is a class member function with no arguments or declared return type that has the name of the type to be converted to
- Passing an object by value versus passing a reference
  - ✓ For efficiency
  - ✓ A function defined as accepting a base-class reference argument can also be used successfully with derived object
- Returning an object versus returning a reference
  - ✓ Shouldn't return a reference to a temporary object

Class Design Review

- Using const (variable, argument, member function)
  - Use it to guarantee that a method doesn't modify an argument
  - Use const to guarantee that a method won't modify the object
  - Use const to ensure that a reference or pointer return value can't be used to modify data in an object
- Public inheritance considerations
  - *Is-a* relationship
  - What's not inherited:
    - ✓ Constructors
    - ✓ Destructors
    - ✓ Assignment operators

Class Design Review

- Assignment operator considerations
  - ✓ Need to provide an explicit assignment operator if class constructors use new to initialize pointers
- Private versus protected members
- Virtual method considerations
  - ✓ Define the method as virtual in the base class, which could be redefined
  - ✓ Enable late, or dynamic, binding (a table used)
- Destructor considerations
  - ✓ A base class destructor should be virtual
- Friend considerations
  - ✓ It's not inherited
  - ✓ To use: Type cast a derived-class reference or pointer to the base-class equivalent and then invoke the base-class friend

Class Design Review

- Observations on using base-class methods
  - ✓A derived-class destructor automatically invokes the base-class destructor
  - ✓A derived-class constructor (in a member-initialization list)
    - automatically invokes the base-class default constructor if don't specify in list
    - explicitly invokes the base-class constructor specified in list
  - ✓A derived object automatically uses inherited base-class methods if the derived class hasn't redefined the method
  - ✓Derived-class methods can use the **scope-resolution operator** to invoke public and protected base-class methods
  - ✓Friend functions

Class Function Summary

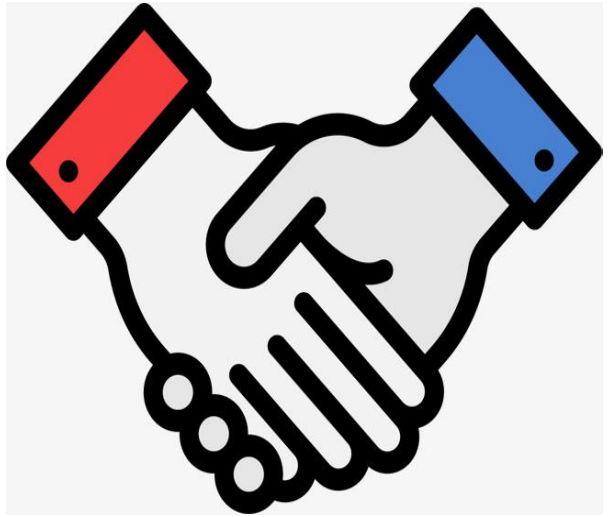
- Member function properties

Function	Inherited	Member or Friend	Generated by Default	Can Be Virtual	Can Have a Return Type
Constructor	No	Member	Yes	No	No
Destructor	No	Member	Yes	Yes	No
=	No	Member	Yes	Yes	Yes
&	Yes	Either	Yes	Yes	Yes
Conversion	Yes	Member	No	Yes	No
()	Yes	Member	No	Yes	Yes
[]	Yes	Member	No	Yes	Yes
->	Yes	Member	No	Yes	Yes
op=	Yes	Either	No	Yes	Yes
new	Yes	Static member	No	No	void *
delete	Yes	Static member	No	No	void
Other operators	Yes	Either	No	Yes	Yes
Other members	Yes	Member	No	Yes	Yes
Friends	No	Friend	No	No	Yes

copy

address

No return type but has return value



Thanks

zhengf@sustech.edu.cn