

**605.404 Object-Oriented Programming with C++**  
**Summer 2007**  
**Lectures 11**

11	Class Hierarchies	S: Chap 15
----	-------------------	------------

Chapter 15 Class Hierarchies

- I. 15.1 Introduction and Overview
- II. 15.2 Multiple Inheritance
  - A. C++ supports multiple inheritance
  - B. (sections 2.5.4 and 12.3 do not have multiple inheritance examples)
  - C. allows the union of operations from the base classes and what is defined in the derived class to be applied
  - D. 15.2.1 Ambiguity Resolution
    - 1. two base classes may have member functions with the same name
    - 2. must disambiguate in the derived class which member function is to be used
      - a) (via use of base class name and scope resolution operator)
      - b) defining a new function in the derived class to do the resolution
  - E. 15.2.2 Inheritance and Using-Declarations
    - 1. overload resolution is not applied across different class scopes
    - 2. (see section 8.2.2, page 169, for using-declaration information)
    - 3. using-declarations allow a programmer to compose a set of overloaded functions from base classes and the derived class
    - 4. functions in the derived class hide functions that would otherwise be available from a base
    - 5. virtual functions from bases can be overridden
    - 6. using-declaration must refer to members of a base class; it may not be used for a member of a class from outside that class, its derived classes, and their member functions; it may not appear in a class definition and may not be used for a class
  - F. 15.2.3 Replicated Base Classes
    - 1. see figure and example page 394
    - 2. 15.2.3.1 Overriding
      - a) a virtual function of a replicated base class can be overridden by a (single) function in a derived class
  - G. 15.2.4 Virtual Base Classes
    - 1. every *virtual* base of a derived class is represented by the same (shared) object
    - 2. 15.2.4.1 Programming Virtual Bases
      - a) the constructor of the virtual base is invoked (implicitly or explicitly) from the constructor for the complete object (the constructor of the most derived class)
      - b) the constructor for a virtual base is called before the constructors for its derived classes
      - c) can simulate this by calling a virtual base class function only from the most derived class
  - H. 15.2.5 Using Multiple Inheritance

1. using multiple inheritance to provide implementations for abstract classes is more fundamental than gluing together unrelated classes through multiple inheritance in the way that it affects the way a program is designed
  - a) one base class may be an interface and another base class the implementation for that interface
2. multiple inheritance allow sibling classes to share information without introducing a dependence on a unique common base class in a program (diamond-shaped inheritance)
  - a) most manageable if either the virtual base classes or the classes directly derived from it are abstract classes
3. 15.2.5.1 Overriding Virtual Base Functions
  - a) a derived class can override a virtual function of its direct or indirect virtual base class
  - b) two different classes might override different virtual functions from the virtual base
  - c) different classes can override the same function if and only if some overriding class is derived from every other class that overrides the function (one function must override all others)
  - d) a class that provides some – but not all – of the implementation for a virtual base class is often called a ***mixin***

### III. 15.3 Access Control

- A. page 402 definitions of *public*, *protected*, and *private*
  1. functions implementing the class (its friends and members),
  2. functions implementing a derived class (the derived class' friends and members), and
  3. other functions
- B. 15.3.1 Protected Members
  1. a derived class can access a base class' protected members only for objects of its own type
  2. 15.3.1.1 Use of Protected Members
    - a) declaring data members protected is usually a design error
    - b) placing significant amounts of data in a common class for all derived classes to use leaves that data open to corruption
    - c) good use, though for functions
- C. 15.3.2 Access to Base Classes
  1. **Access Specifiers for Base Classes** ARM Section 11.2 page 242
    - a) If a class is declared to be a base class for another class using the *public* access specifier, the *public* members of the base class are *public* members of the derived class and the *protected* members of the base class are *protected* members of the derived class
    - b) If a class is declared to be a base class for another class using the *private* access specifier, the *public* members of the base class are *private* members of the derived class and the *protected* members of the base class are *private* members of the derived class
    - c) *Private* members of a base class remain inaccessible even to derived classes unless *friend* declarations within the base class declaration are used to grant access explicitly
    - d) In the absence of an access specifier for a base class, *public* is

assumed when the derived class is declared *struct* and *private* is assumed when the derived class is declared *class*

2. 15.3.2.1 Multiple Inheritance and Access Control

- a) if a name or base class can be reached through multiple paths in a multiple inheritance lattice, it is accessible if it is accessible through any path
- b) if a single entity is reachable through several paths, we can still refer to it without ambiguity

3. 15.3.2.2 Using-Declarations and Access Control

- a) a *using-declaration* cannot be used to gain access to additional information
- b) once access is granted, though, it can be granted to other users

IV. 15.4 Run-Time Type Information

- A. may lose information about the type of objects passes to a subsystem and then later returned to the original subsystem
- B. recovering the “lost” type information of an object requires asking the object to reveal its type
- C. most obvious and useful operation for inspecting the type of an object at run time is a type conversion operation that returns a valid pointer if the object is of the expected type and a null pointer if it is not of the expected type
- D. the *dynamic\_cast* operator does this
  1. *dynamic\_cast* translates from the implementation-oriented language of one entity to the language of another entity
- E. casting from a base class to a derived class is often called **downcast**
- F. cast from a derived class to a base is called an **upcast**
- G. cast from a base to a sibling class is called a **crosscast**
- H. 15.4.1 Dynamic\_cast

1. takes two operands

- a) a type bracketed by <> and
- b) a pointer or reference bracketed by ()

2. purpose of dynamic cast is to deal with the case in which the correctness of the conversion cannot be determined by the compiler

3. requires a pointer or a reference to a polymorphic type in order to do a downcast or a crosscast

4. use of a “type information object” example bottom page 409/top page 410

5. 15.4.1.1 Dynamic\_cast of References

a) the behavior of dynamic cast in returning a 0 when failure occurs is OK for pointer types but not reference types

b) need to ask “is the object pointed to by *p* of type *T*?”

c) may assume that *dynamic\_cast*<*T*&>(r) refers to an object

(1) it is not a question but an assertion: “The object referred to by r is of type T”

(2) a *bad\_cast* exception is thrown if the reference is not valid

I. 15.4.2 Navigating Class Hierarchies

1. sometimes we want to explicitly name an object of a base class or a member of a base class
2. sometimes we want to get a pointer to the object representing a base or derived class of an object given a pointer to a complete object or some

sub-object

3. 15.4.2.1 Static and Dynamic Casts
  - a) a `static_cast` does not examine the object it casts from so it cannot do some operations (examples page 413)
  - b) dynamic cast requires a polymorphic operand because there is no information stored in a non-polymorphic object that can be used to find the objects for which it represents a base
- J. 15.4.3 Class Object Construction and Destruction
  1. class object is built from “raw memory” by its constructors and it reverts to “raw memory” as its destructors are executed
  2. virtual functions can be called during intermediate construction and destruction – urged not to do this
- K. 15.4.4 Typeid and Extended Type Information
  1. sometimes need to know the exact type of an object
  2. `typeid` operator yields an object representing the type of its operand
  3. `typeid()` most commonly used to find the type of an object referred to by a reference or a pointer
    - a) if the value of a pointer or a reference operand of a polymorphic type is 0, `typeid()` throws a `bad_typeid` exception
    - b) if the operand has non-polymorphic type or is not an lvalue, the result is determined at compile time without evaluating the operand expression
    - c) use `==` on `type_info` objects to test equality rather than `==` on pointers to such objects
  4. 15.4.4.1 Extended Type Information
- L. 15.4.5 Uses and Misuses of RTTI
  1. use virtual functions rather than RTTI to handle most cases when run-time discrimination based on type is needed
  2. proper to use RTTI when some service code is expressed in terms of one class and a user wants to add functionality through derivation
- V. 15.5 Pointers to Members
  - A. C++ offers a facility for indirectly referring to a member of a class
  - B. A pointer to a member is a value that identifies a member of a class
    1. (can think of it as the position of the member in an object of the class)
  - C. a pointer to member can be obtained by applying the address-of operator `&` to a fully qualified class member name
  - D. a pointer to a virtual member is a kind of offset, therefore it does not depend on an object’s location in memory
  - E. 15.5.1 Base and Derived Classes
    1. a derived class has at least the members that it inherits from its base classes; often it has more
    2. we can safely assign a pointer to a member of a base class to a pointer of a member of a derived class, but not the other way around (*contravariance*)
- VI. 15.6 Free Store
  - A. you may need to define operator `new()` and operator `delete()` for a specific class
  - B. member operator `new()` and operator `delete()` are implicitly static members (they do not have a `this` pointer and do not modify an object. They provide storage that a constructor can initialize and a destructor can clean up

- C. may need a virtual destructor in the base class to properly deallocate memory, even if it is an empty destructor (see bottom page 422)
- D. if you want to supply an allocator/deallocator pair that works correctly for derived classes, you must either supply a virtual destructor in the base class or refrain from using the *size\_t* argument in the deallocator
- E. 15.6.1 Array Allocation
- F. 15.6.1 "Virtual Constructors"
  1. constructors cannot be virtual
  2. a constructor needs an exact type of the object it is to create
  3. can, though, define a function that calls a constructor and returns a constructed object