

605.404 Object-Oriented Programming with C++
Summer 2007
Lectures 09

09	Derived Classes and Templates	S: Chaps 12-13
----	-------------------------------	----------------

Class is a design concept (defines new types)

Object is a run-time concept (specific instances of a type)

Chapter 12 Derived Classes

- I. 12.1 Introduction
 - A. need to represent concepts and the relationships between concepts
 - B. need to express kind-of relationships
- II. 12.2 Derived Classes
 - A. first example page 303
 - 1. a Manager is an Employee with a few additional pieces of information
 - 2. arrow in derivation diagram points to parent class
 - 3. derived class inherits properties from base class
 - 4. data in derived class object is a superset of the data in the base class
 - 5. derived class holds more data and provides more functions
 - 6. derived class can be used wherever the base class is acceptable
 - 7. an object of a derived class can be treated as an object of its base class when manipulated through pointers and references
 - 8. a class must be defined in order to be used as a base
 - B. 12.2.1 Member Functions
 - 1. a member of a derived class can use the public and protected members of its base class as if they were declared in the derived class itself
 - 2. derived class cannot access the private members of its base class
 - 3. a protected member is like a public member to a member of a derived class, yet it is like a private member to other functions
 - 4. cleanest solution is for the derived class to use ONLY public members of its base class
 - C. 12.2.2 Constructors and Destructors
 - 1. if a base class has constructors, then a constructor must be invoked
 - 2. if all constructors for a base require arguments, then a constructor for that base must be explicitly called
 - 3. default constructors can be invoked implicitly otherwise
 - 4. arguments for the base class' constructor are specified in the definition of a derived class' constructor (the base class acts exactly like a member of the derived class)
 - 5. a derived class constructor can specify initializers for its own members and immediate bases only; it cannot directly initialize members of a base
 - D. 12.2.3 Copying
 - 1. constructors are not inherited
 - 2. watch out for supplying copy constructor and assignments only for part of class hierarchy
 - E. 12.2.4 Class Hierarchies
 - 1. C++ can express a directed acyclic graph of classes

- F. 12.2.5 Type Fields
 - 1. Four fundamental resolutions to problems (page 308) for pointer to type *Base**
 - a) Should avoid simple type field solution (as in E for employee and M for manager)
- G. 12.2.6 Virtual Functions
 - 1. overcome the problems with type field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class
 - 2. *virtual* indicates that the function name acts as an interface to the function defined in this class and the functions defined in classes derived from it
 - 3. to allow a virtual function declaration to act as an interface to functions defined in derived classes, the argument types specified for a function in a derived class cannot differ from the argument types declared in the base (and only slight changes are allowed for return type)
 - 4. a virtual function must be defined for the class in which it is first declared (unless it is declared to be a pure virtual function)
 - 5. a virtual function can be used even if no class is derived from its class
 - 6. a derived class that does not need its own version of a virtual function need not provide one
 - 7. a function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to override the base class version of the virtual function
 - 8. getting the “right” behavior from the class’ functions independently of exactly what kind of class is used is called polymorphism
 - 9. a type with virtual functions is called a polymorphic type
 - a) to get polymorphic behavior in C++ the member functions called must be virtual and objects must be manipulated through pointers or references
 - 10. calling a function using the scope resolution operator ensures that the virtual mechanism is not used
 - a) if a virtual function is also inline, then inline substitution can be used for calls specified using the scope resolution operator
- III. 12.3 Abstract Classes
 - A. some classes represent abstract concepts for which objects cannot exist
 - B. pure virtual functions [use the initializer =0]
 - C. a class with one or more pure virtual functions is an abstract class; no objects of that class can be created
 - D. abstract class can only be used as an interface and as a base class for other classes
 - E. a pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is an abstract class
 - F. important use of abstract classes is to provide an interface without exposing any implementation details
- IV. 12.4 Design of Class Hierarchies
 - A. Problem: Retrieve an integer value from a user interface
 - B. *Ival_box* knows what range of input values it will accept
 - C. 12.4.1 A Traditional Class Hierarchy
 - 1. (model of Simula, Smalltalk and older C++ programming styles)
 - 2. *Ival_box* defines the basic interface to all *Ival_boxes* and specifies a default

implementation that more specific kinds of *Ival_boxes* can override with their own versions; also includes data needed to implement the basic *Ival_box*

- 3. 12.4.1.1 Critique
 - a) lacks flexibility for use of screen tools (in this case) and limits environment applications
- D. 12.4.2 Abstract Classes
 - 1. goals page 318
 - 2. example of multiple inheritance (*Ival_box* and *Bbwindow*)
- E. 12.4.3 Alternative Implementations
 - 1. need to solve the versioning issues in both prior examples
 - 2. 12.4.3.1 Critique
- F. 12.4.4 Localizing Object Creation
 - 1. centralizing object creation through an indirection (an intermediate class in this case)
- V. 12.5 Class Hierarchies and Abstract Classes

Chapter 13 Templates

- VI. 13.1 Introduction
 - A. “independent concepts should be independent represented and should be combined only when needed”
 - B. templates provide direct support for generic programming
 - 1. a type parameter is allowed in the definition of a class or function
 - 2. template depends only on the properties that it actually uses from its parameter types and does not require different types used as arguments to be explicitly related
 - 3. the argument types used for a template need not be from a single inheritance hierarchy
 - C. refer back to intro in section 2.7.1 and syntax in Appendix C.13
- VII. 13.2 A Simple String Template
 - A. consider a string of characters
 - B. what if you were to provide standard string behaviors (subscripting, concatenation, and comparison) to many different kinds of characters
 - C. want to minimize dependency of definition of string on the specific kind of character
 - D. start with design of concrete class for string in chapter 11
 - E. parameterize the char type
 - 1. template type argument is used like other type names after its introduction
 - 2. it is a type name – it need not be a class
 - F. creating instances of the template class page 329
 - G. (standard library *map* introduced section 3.7.4 page 55)
 - 1. contains a key type and a value or mapped type
 - 2. more detail section 17.4.1 page 480
 - H. (string in the standard library)
 - 1. typedef basic_string <char> string;
 - I. 13.2.1 Defining a Template
 - 1. the use of a template does not imply any run-time mechanism beyond what is used for an equivalent “hand-written” class
 - 2. start with a non-template version of a class and debug it before conversion to the template version of the class recommendation

3. a template member need not be defined within the template class itself; examples of definition outside the class page 330
 4. template parameters are parameters rather than names of types defined externally to the template; within the scope of the template class, qualification with <type> is redundant for the name of the template itself
 5. It is not possible to overload a class template name, so if a class template is declared in a scope, no other entity can be declared there with the same name
 6. a type used as a template argument must provide the interface expected by the template
 7. there is no requirement that different arguments for the same template parameters should be related by inheritance
- J. 13.2.2 Template Instantiation
1. a version of a template for a particular template argument is called a specialization
 2. in general it is the implementation's job – not the programmer's – to ensure that versions of a template function are generated for each set of template arguments used
- K. 13.2.3 Template Parameters
1. can take type parameters, parameters of ordinary types, and template parameters
 2. passing in size arguments as a strategy to limit free store use
 3. constant expression, address of an object or function with external linkage, or a non-overloaded pointer to a member arguments are allowed
 4. string literal is NOT acceptable as a template argument
 5. integer template argument MUST be a constant
 6. a non-type template parameter is constant within the template so that an attempt to change the value of a parameter is an error
- L. 13.2.4 Type Equivalence
1. when using the same set of template arguments for a template, we always refer to the same generated type
- M. 13.2.5 Type Checking
1. when the template is defined, the definition is checked for syntax errors and possibly also for other errors that can be detected in isolation from a particular set of template arguments
 2. a name used in a template definition must either be in scope or in some reasonably obvious way depend on a template parameter
 3. errors that relate to the use of template parameters cannot be detected until the template is used
- VIII. 13.3 Function Templates
- A. when a template function is called, the types of the function arguments determine which version of the template is used
 - B. see page 158 for initial definition in text of shell short function
 - C. note need for overloaded < operator (see section 13.4 to come) for the example presented in the text
 - D. 13.3.1 Template Arguments
 1. a compiler can deduce type and non-type arguments from a call, providing the function argument list uniquely identifies the set of template arguments
 2. note that class template parameters are NEVER deduced

- 3. if the template argument cannot be deduced from the template function arguments (See appendix C13.4) we must specify it explicitly
- E. 13.3.2 Function Template Overloading
 - 1. one can declare several function templates with the same name and even declare a combination of function templates and ordinary functions with the same name
 - 2. rules for resolution in the presence of function templates are generalizations of the function overload resolution rules pages 336-337
- IX. 13.4 Using Template Arguments to Specify Policy
 - A. consider the sorting of string problem
 - 1. can't hardwire the sort criteria into the container because the container can't in general impose its needs on the element types
 - 2. can't hardwire the sorting criteria into the element types because there are many different ways of sorting elements
 - 3. criteria must be supplied when a specific operation needs to be performed
 - B. solution is to pass comparison operations as template parameter
 - 1. several operations can be passed as a single argument with no run-time cost
 - 2. comparison operators `eq()` and `lt()` are trivial to inline; whereas inlining a call through a pointer to a function requires exceptional attention from a compiler
 - C. 13.4.1 Default Template Parameters
 - 1. pick a default so that only the uncommon criteria have to be explicitly specified
 - 2. the default parameters should specify the most common policy
- X. 13.5 Specialization
 - A. user-defines specializations or user specializations provide a way for the programmer to choose certain specializations of the class
 - B. in `Vector` example, recognize that there will often be `Vectors` of pointer types (to facilitate type-safe containers)
 - 1. containers of pointers can share a single implementation
 - 2. `template<>` prefix notes that this is a specialization that can be specified without a template parameter
 - 3. the template arguments for which the specialization is to be used are those specified in the `<>` brackets after the name (in this case `<void*>`)
 - a) means that this definition is to be used as the implementation for every vector for which T is `void*`
 - C. `Vector<void*>` is a complete specialization (there is no template parameter to specify or deduce when we use this specialization)
 - D. Need partial specialization to define a specialization that is used for every `Vector` of pointers and only for `Vectors` of pointers (see page 324)
 - 1. (see section 13.6 for more on derivation and templates)
 - 2. **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*
 - 3. in partial specialization a template parameter is deduced from the specialization pattern ; the template parameter is not simply the actual template argument
 - 4. the general template must be declared before any specializations
 - 5. is a user specializes a template somewhere, that specialization must be in scope for every use of the template with type for which it was specialized
 - E. 13.5.1 Order of Specializations
 - 1. one specialization is more specialized than another if every argument list matches its specialization pattern also matches the other but not vice versa
 - F. 13.5.2 Template Function Specialization
 - 1. useful when there is a more efficient alternative to a general algorithm for a set of template arguments
 - 2. also handy when an irregularity of an argument type causes the general algorithm to give an undesired result (often built-in pointer and array types)
- XI. 13.6 Derivation and Templates
- A. template can be used to provide an elegant and type-safe interface to an otherwise unsafe and inconvenient-to-use facility
 - B. having the same template parameter for the base class and the derived class is the most common case but not a requirement; may pass the derived type itself to the base class
 - C. a class generated from a class template is a perfectly ordinary class
 - D. 13.6.1 Parameterization and Inheritance
 - 1. virtual functions provide run-time polymorphism
 - 2. templates provide compile-time polymorphism or parametric polymorphism
 - E. 13.6.2 Member Templates
 - 1. a member template cannot be *virtual*
 - F. 13.6.3 Inheritance Relationships
 - 1. as far as the C++ language rules are concerned, there is no relationship between two classes generated from a single class template
 - 2. 13.6.3.1 Template Conversions
 - a) there cannot be any default relationship between classes generated from the same templates
 - b) member templates allow the specification of such relationships when needed
- XII. 13.7 Source Code Organization
- A. organizing code using templates
 - 1. include template definitions before their use in a translation unit
 - 2. include template declarations (only) before their use in a translation unit, and compile their definitions separately
 - B. recommends separate compilation
 - 1. explicit use of *export*