Abstract Classes and Class Design

Topics Covered
- Reading: 603 - 632 (Rest of Chapter 13)
- Inheritance review and class relationships
- Pure virtual functions
- Abstract base classes
- Inheritance and dynamic memory
- Class design review
- Homework 13 & 14
- Midterm Exam 2 - 8:00 - 10:15 pm

Inheritance and Generalization
- Inheritance allows "class generalization"
  - Suppose you have several functions that all contain the same code sequence
  - You "factor" the sequence into a function
  - This is "code generalization"
- Inheritance does this with classes
  - Common attributes and methods can be shared in a related "family" of classes
  - Called a "class hierarchy" [MFC as an example]
Class Relationships I

- Inheritance is a technique for implementing particular kinds of class relationships
- Different kinds of class relationships
  - The uses-a relationship [no inheritance]
  - The has-a relationship [no inheritance]
  - The is-a relationship [public inheritance]
  - The implemented-as relationship [private inheritance]

Class Relationships II

- The is-a relationship
  - Occurs when one class is a "kind of" another
    - Ostrich is a Bird is an Animal
      - A Label is a Component, so is a Button
    - In C++, IS A is implemented by public inheritance like this:
      ```
      class Bird : public Animal {
        // Code & data for Bird class goes here
      };
      ```

Basic Public Inheritance I

- The syntax of public inheritance
  ```
  class Base { ... };
  class Derived : public Base { ... };
  ```
- Accessibility
  - Derived has no direct access to Base private members
    - All private members are inherited, however
  - Each Derived object "contains" a Base object
  - Derived class has direct access to public members
Basic Public Inheritance II

- Derived methods hide or replace base methods
  - Call hidden base methods using scope resolution
    ```cpp
def void Derived::print() { Base::print(); cout << z_; }
```
- Overloading only works in single scope
  - Using name hides all inherited methods of that name
    ```cpp
class Base { ... void sing(); void sing(char *); };  
class Derived : public Base { ... void sing(); };  
```

Basic Public Inheritance III

- Protected access is between public and private
  - Derived classes are allowed direct access
  - Advice: use for methods, not fields
- Constructors are not inherited
  - Call base constructor in initialization list
    ```cpp
    Derived::Derived(int x, int y, int z) : Base(x,y), z_(x) {}  
    ```
  - Runs before first line in derived constructor
  - Default constructor called if you don't

Basic Public Inheritance IV

- Construction order
  - Arguments used to find requested constructor
  - Initialization list searched for base constructor
    » If not found, then default base constructor called
  - Initialization list searched for member constructors
    » Initialized in member declaration order
  - Body of constructor is executed
- Destruction order: most derived to base class

Basic Polymorphism I

- Objects use declared type to call methods:
  ```
  Pet p; Dog d; Cat c;
p.speak(); d.speak(); c.speak();
  Dog d; *dp = &d;
d.speak(); dp->speak();
  ```
  - Type known at compile-time: static or early binding

- Derived objects assigned to base variables
  - Derived portion is "sliced" off (next slide)

Basic Polymorphism II

- Here's what happens when you pass a Dog object to the function `show(Pet p)`

  ![Diagram: Pet object passed to show(Pet p), Dog object "sliced" into Dog and Pet parts]

  ```
  Pet p; Dog d; show(Pet p);
  ```
  - Pet part
  - Dog part

Basic Polymorphism III

- With pointers or reference, no slicing occurs
  - Function calls are still statically bound, though
    ```
    Cat c;
    Pet * p = &c;
c.show(); // Calls Cat::show()
p->show(); // Calls Pet::show()
    ```

- Use `virtual` in base class methods: late binding
  ```
  virtual void show() ...
  virtual void license() ...
  ```
Basic Polymorphism IV

- Use a virtual function through a base pointer
  ```cpp
dog d; cat c;
pet *p = &d;
p->speak();
p = &c;
c->speak();
```
- Compiler uses pointer's dynamic type to decide exactly which function to call

Basic Polymorphism V

- Rules for using the keyword virtual
  - Optionally may be repeated in derived class
  - Cannot make a non-virtual function virtual
  - Cannot make a virtual function "unvirtual"
  - Virtual functions are only virtual when called through a pointer or reference
    ```cpp
cat c; c.speak(); // Static binding
pet p = c; p.speak(); // Pet::speak()
pet *p = &c; p->speak(); // Cat::speak()
```

Pure Virtual Functions

- Design of Pet hierarchy has one problem
  - "Generic Pets" can't really speak
  - Can't remove the speak() method
  - Lose ability to call Cat::speak() through Pet*
  - Solution is to create a pure virtual function

- Pure virtual function syntax: [pets03.h]
  ```cpp
  virtual void speak() = 0;
  ```
  - May also supply an optional body
Abstract Classes

- A class that contains a pure virtual function is called an abstract base class
- An abstract class:
  - Cannot be instantiated
  - You can create pointers to abstract classes
- Derived classes must override each pure virtual function, or derived class is also abstract [pets07.cpp]
- A general rule: all base classes should be ABC

Virtual Destructors

- Constructors cannot be virtual
  - Overridden functions must have same name
- Destructors should always be virtual
  - Exception to previous rule
  - What happens if destructor is not virtual?
    » Child destructor not called [pets08.cpp, pets04.h]
  - Add virtual destructor to base, even if empty
  - If destructor is pure virtual, must supply a body

Inheritance & Dynamic Memory I

- If a derived class does not allocate memory
  - You don't have to write any extra functions
- If a derived class allocates memory, you must:
  - Write a copy constructor
    » Call base constructor in initialization list
    » Argument rhs is acceptable, because it is a Vegie
    » Corn(const Corn& rhs) : Vegie(rhs) ...
Inheritance & Dynamic Memory II

- Make sure base class destructor is virtual
  - In derived destructor, free memory in derived class
- You must write an assignment operator
  - Not virtual because argument must be derived type
  - Must manually call base class assignment operator
    » Argument will be converted to base type automatically

```cpp
Corn& Corn::operator=(const Corn& rhs) {
  Vegi& Vegi::operator=(rhs); // assign to base part
  // allocate Vegi memory here
}
```

Semantics of Polymorphism I

- Each feature in C++ allows you to express a particular idea. You must understand what each feature means before using it.
- Your basic design choices:
  - Public inheritance says "is-a"
  - Composition says "has-a"
  - Private, protected inheritance says "made-of"
- Don’t confuse composition and inheritance

Semantics of Polymorphism II

- Regular, virtual, or pure-virtual?
  - Only applicable in cases of public inheritance
  - Regular member function: do not override
    » Children inherit interface & implementation
  - Simple virtual function: you may override
    » Children get interface & optional implementation
  - Pure virtual function: you must override
    » Children inherit mandatory interface only
Uses of Polymorphism I

- The MiniSketch MFC example
  - An SDI drawing program that draws shapes
  - Shape class hierarchy [Shapes.h]

Uses of Polymorphism II

- Document contains array of Shape pointers
- Shape functions are all virtual

Uses of Polymorphism III

- When saved to disk [serialized], objects must be "flattened"
Homework

- Homework 13: Virtual Functions
  - Due Sunday April 4 [Grade Monday AM]
  - Modify classes created for Homework 12
    » Add char * field to SalariedEmployee
    » Add virtual getInfo() (input) methods to each class
    » Add virtual destructors
    » Add a virtual print() method
  - Add overloaded insertion and extraction operators
    » Take a reference to a Person
    » Call the virtual getInfo() and print() functions

- Homework 14 : A little Spring Break