ch10: Name Control

2 Feb 2009
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Review: Polymorphism

- Upcasting
- Virtual methods (ch15)
- Abstract superclasses and pure virtual methods
- References, pass-by-reference, const refs
- The copy constructor (ch11)
- Operator overloading (ch12)

What's on for today:

- The static keyword: two uses
  - For local vars: persistent storage
  - For global names: file scope
- Namespaces and using
- Class variables (static member variables)
  - Application: shared data tables
- Class methods (static member functions)
  - Application: singleton classes

Managing Names

- As your projects get more complex, the number of names to manage increases:
  - Variables, functions, classes, libraries
- The keyword static has two basic uses:
  - Static storage: persistent data, allocated once and reused
  - Static visibility: file scope
- Namespaces are a C++ way to manage names

Static local variables

- Local variables inside functions flagged as static are persistent across calls to the function:
  - **int tick() {**
    - **static int counter = 0;**
    - **return ++counter; }**
  - Static vars last until main() finishes or exit() is called
  - Objects may be static vars, too:
    - Destructor is called when main() finishes
    - Destructors not called if use abort() instead of exit()

Static (internal) linkage

- When applied to global names, static indicates internal linkage
  - By default, global names (not inside a class or function) are visible everywhere
    - Even to other files / translation units
    - This is called external linkage
  - static limits visibility to just this file
- Useful for variables declared in headers
  - What if header gets #included into several translation units?

Creating namespaces

- A C++ namespace is a container for names, in order to help manage names
  - **namespace AppleLib {**
    - **extern int numApples;**
    - **class Apple { ...**
  - **}**
- Namespace definitions can be spread across multiple files:
  - Use the same namespace container
- Namespaces may be aliased:
  - **namespace AL = AppleLib;**

Default file namespace

- Each translation unit (compiled file) has its own anonymous namespace:
  - **namespace {**
    - **class Student { ...**
- Items in this namespace can be used in that file without qualifier, but not outside that file
  - It's the C++ way of doing file static (internal linkage)

Accessing namespaces (::)

- An item inside a namespace can be specified using its namespace:

```
namespace AppleLib {
    class Apple {
        string name;
        void setName(string n);
    }
}
void AppleLib::Apple::setName(string n)
    { name = n; }
AppleLib::Apple myAp;
myAp.setName("Fuji");
```

using

- The using directive permits you to use items from the specified namespace without qualifier:
  - **using namespace AppleLib;**
  - **Apple myAp; // don't need AppleLib::**
- You can also use just one item from a namespace:
  - **using AppleLib::Apple;**
- Rule of thumb: global using directives go only in *.cpp files, not in header files! (Why?)

Class variables (static members)

- Class variables are shared by all instances of the class
  - e.g., many connections to shared database
  - e.g., track # of instances: enforce singleton
- In C++: declare as static in header file:
  - **class Apple {**
    - **static int numApples; ....**
- Initialize it outside class declaration (in .cpp file):
  - **int Apple::numApples = 1;**

Application: shared tables

- static const arrays can be used for shared data tables used by all instances
- e.g., a class to break up words into syllables:
  - **class Syllababble {**
    - **static const char vowels[];**
  - **}**
  - **const char Syllababble::vowels[] =**
    { 'a', 'e', 'i', 'o', 'u', 'y' };
- Only integral built-in types may be initialized inline; all other static consts must be initialized outside the class (typically, in the *.cpp file)

Class methods

- Just like class variables, methods may also be made static: associated with the whole class, not with individual instances:
  - **class Counter {**
    - **static int count = 0;**
    - **public:**
    - **static int inc() { return ++count; }**
  - **}**
  - **Counter::inc(); // no instance needed!**
- May only access other static methods/variables

Aside: Java's main()

- In Java, everything must go inside a class
  - Every file has one public class
  - Same name as the file
- main() is just a method within that class
  - Must be declared public static:
  - **public class HelloWorld {**
    - **public static void main(String args[]) {**
      - **System.out.println("Hi!");**
    - **}**
  - **}**
- Not instantiated; just run HelloWorld::main()

Application: singleton

- A singleton is a class that only allows one instance to be created
  - e.g., domain controller, DHCP server, etc.
- Use static members and a private constructor:
  - **class Egg {**
    - **static Egg e;** **// the one instance**
    - **int chick;** **// payload**
    - **Egg(int c) : chick(c) {}** **// constructor**
    - **Egg(const Egg &);** **// disallow copy**
  - **public:**
    - **static Egg* theEgg() { return e; }**

Singleton: usage

- Use the static class method to access the singleton instance:
  - **Egg e = Egg::theEgg();**
- But we can't create a new Egg because the constructor is private:
  - **Egg myEgg(5); // doesn't work!**