

---

# CS11 – Advanced C++

---

Spring 2008-2009

Lecture 9 (!!!)

---

# The **static** Keyword

- C++ provides the **static** keyword
    - Also in C, with slightly different usage
  - Used in two main contexts:
    - Declaring static members of a class
    - Declaring static local variables within functions
  - Both uses are very helpful in certain situations
-

---

# Class Members

- Class-members are usually associated with objects

```
class Matrix4F {  
    float values[16];  
public:  
    ...  
    bool isOrthogonal() { ... }  
};
```

- Each matrix-object has its own copy of **values** member
- Calling **isOrthogonal()** requires a specific object

```
Matrix4F m = ... ;  
if (m.isOrthogonal())  
    m.transpose(); // can invert by transposing
```

---

---

# Static Members

- Might want to provide constants for our class
    - Only want one copy of the constant for the entire class, not one per object!
  - Also might want to provide general utility functions for our class
    - Functions that help with the class, but don't need to be called on a specific object
  - Static class-members aren't associated with a specific object
    - The member is associated with the class itself, not with individual objects
-

---

# Static Constants

- Add an **Identity** constant to our matrix class
- Updated declaration, in our **matrix.hh** file:

```
class Matrix4F {  
    ...  
public:  
    ...  
    // Identity matrix constant  
    static const Matrix4F Identity;  
};
```

- Static member is not defined or initialized in the declaration!
    - *Separately* initialized in corresponding **matrix.cc** file
    - (reasons are gross and involve compilation/linking issues)
-

---

## Static Constants (2)

- Within the corresponding `matrix.cc` file:

```
const Matrix4F Matrix4F::Identity =  
    Matrix4F().setIdentity();
```

- Only use `static` keyword in declaration, not in definition
- Refer to static member by its qualified name:  
`Matrix4F::Identity`

- To use default initialization for static member:

```
const Matrix4F Matrix4F::Identity;
```

- Still must appear in the `.cc` file! Otherwise, linker errors...

- Other code can refer to the static constant:

```
#include "matrix.hh"
```

```
if (m == Matrix4F::Identity)
```

```
...
```

---

---

# Static Functions

- Add a static function to convert Euler angles into corresponding transform matrix

```
class Matrix4F {  
    ...  
public:  
    ...  
    static Matrix4F getEulerTransform(  
        const EulerAngles &a);  
};
```

- Function definition can appear in class declaration or in definition
  - (only static member-variables are finicky)
-

---

## Static Functions (2)

- Can call static function without requiring an object

```
#include "matrix.hh"
```

```
Matrix4F m;
```

```
EulerAngles a;
```

```
...
```

```
m = Matrix4F::getEulerTransform(a);
```

- Use qualified name to refer to static member-function
  - Within the **Matrix4F** class definition, don't even need fully qualified name to refer to static member
-

---

# Local Variables

- Within functions, local variables are initialized when execution reaches each variable's declaration

```
float compute(float a, float b) {  
    float result = 0; // result is initialized  
    if (a < b) {  
        result = a * b;  
    } else {  
        float c = 0.1 * (a - b); // c is initialized  
        result = c * compute(a - c, b + c);  
    }  
    return result;  
}
```

- Every time execution reaches a variable declaration, that variable is initialized
-

---

# Static Local Variables

- Can also declare static local variables
  - Initialized once, when execution reaches the variable's declaration for the first time
  - Variable's value is *remembered* between calls!
- Example: function to get a unique ID

```
int GetNextID() {  
    static int nextID = 10000;  
    int id = nextID;  
    nextID++;  
    return id;  
}
```

- At first call, `nextID` is initialized to 10000
  - Variable remembers its value across multiple function calls
-

---

# Static Local Variables and Concurrency

- Static local variables break concurrent execution
  - Reentrant functions – functions that can be called from multiple concurrent threads of execution
  - Static state must be guarded with mutexes, or function will not generate correct results
  - (For reentrant functions, better to just avoid static state!)
- Our unique ID function:

```
int GetNextID() {  
    static int nextID = 10000;  
    int id = nextID;  
    nextID++;  
    return id;  
}
```

} All these operations need to be guarded for correct concurrent execution.

- Concurrent calls from different threads can easily produce the same ID, or not update `nextID` properly.
-

---

# Singletons

- A very common design pattern: Singleton
    - “Ensure a class has only one instance, and provide a global point of access to it.”
      - Design Patterns, Gamma et al.
  - Some examples:
    - Only one window manager in an operating system
    - Only one system clock
    - Only one filesystem manager
  - Frequently run into singletons in practice
    - “Manager” or “factory” classes often singletons
-

---

# Implementing Singletons

- Two main problems to solve:
    - Ensure a class has only one instance.
    - Provide a global point of access to it.
  - Not terribly hard problems to solve, but some ways are better than others
    - Might want to avoid creating the singleton object until it's actually needed
    - If singleton object manages system resources then destructor must be called at proper time!
    - Will singleton be used from multiple threads?
-

---

# First Task: Only One Instance

- Create our singleton class:

```
class Singleton {  
    ...  
public:  
    Singleton() { ... }  
    ~Singleton() { ... }  
    int getImportantValue();  
    ...  
};
```

- How can we ensure only one copy?
    - Need to hide the constructor – make it private
    - Also need to hide the copy-constructor
    - Prevent callers from getting the instance, then copying it!
-

---

# Private Constructors

- Updated singleton class:

```
class Singleton {  
    ...  
    // Make constructor private.  
    Singleton() { ... }  
  
    // Make copy-constructor private; disallow copying.  
    Singleton(const Singleton &) { assert(false); }  
public:  
    ~Singleton() { ... }  
    int getImportantValue();  
    ...  
};
```

- What about the destructor?

- ❑ Don't want a caller to get the instance and then destroy it!
  - ❑ Make the destructor private too.
-

---

# Private Destructor

- Updated singleton class:

```
class Singleton {  
    ...  
    // Make constructor and destructor private.  
    Singleton() { ... }  
    ~Singleton() { ... }  
  
    // Make copy-constructor private; disallow copying.  
    Singleton(const Singleton &) { assert(false); }  
public:  
    int getImportantValue();  
    ...  
};
```

- Now the `Singleton` class has total control over its own lifecycle
-

---

# Singletons and Assignment

- What about assignment?

```
class Singleton {  
    ...  
    // Make constructor and destructor private.  
    Singleton() { ... }  
    ~Singleton() { ... }  
  
    // Make copy-constructor private; disallow copying.  
    Singleton(const Singleton &) { assert(false); }  
public:  
    int getImportantValue();  
    ...  
};
```

- If there is only one instance of `Singleton` then all assignment is self-assignment.
  - Don't really need assignment, so make that private too.
-

---

# Singletons and Assignment (2)

- Updated singleton class:

```
class Singleton {  
    ...  
    // Make constructor and destructor private.  
    Singleton();  
    ~Singleton();  
  
    // Make copy-constructor private; disallow copying.  
    Singleton(const Singleton &) { assert(false); }  
  
    // Make assignment-operator private, and disallow.  
    void operator=(const Singleton &) { assert(false); }  
  
    ...  
};
```

- All these are in the private section of singleton class
-

---

# Storing the Single Instance

- Singleton class has strict control over its lifecycle now
  - Still need to store the single instance somewhere
  - Two obvious options:
    - Make a static **Singleton** object
    - Make a static **Singleton** pointer
-

---

# Static Singleton Object

- A static singleton object:

```
class Singleton {
    static Singleton instance;
    ...
public:
    ...
    // Provide access to the singleton.
    static Singleton & Instance() {
        return instance;
    }
};
```

- This is in the header file `singleton.hh...`
  - Source file `singleton.cc` initializes the instance:  

```
// Initialize singleton with default constructor.
Singleton Singleton::instance;
```
-

---

# Static Singleton Object (2)

- This technique relies on dynamic initialization

```
// Initialize singleton with default constructor.
Singleton Singleton::instance;
```

    - `instance` is initialized with a constructor-call, not with a compile-time constant
  - When does this initialization occur?
  - High-level, hand-wavey answer:
    - Within the code generated from `singleton.cc`, dynamic initialization of static objects is done first.
      - Called a “translation unit” by the specification
    - When multiple translation units are involved, situation becomes ambiguous!
      - Can’t guarantee order that translation units are initialized
-

---

## Static Singleton Object (3)

- In another source-file **fubar.cc**, you have a global variable:

```
int ImportantValue =  
    Singleton::Instance().getImportantValue();
```

- Separate translation unit with its own dynamic initialization
  - No guarantee that **instance** has been initialized when accessed from the other module!
    - Compiler may have chosen to initialize **fubar.cc**'s translation unit before **singleton.cc**'s translation unit
  - Moral: Don't use non-local static singleton objects, or you will be sad. ☹
-

---

# Static Singleton Pointer

- Instead of storing an object, store a static pointer:

```
class Singleton {
    static Singleton *pInstance;
    ...
public:
    ...
    // Provide access to the singleton.
    static Singleton & Instance() {
        if (!pInstance) // Need to construct!
            pInstance = new Singleton();
        return *pInstance;
    }
};
```

- Again, in `singleton.cc` we have:

```
Singleton *Singleton::pInstance = 0;
```

---

---

## Static Singleton Pointer (2)

- Two benefits over static singleton-object approach!
- Singleton instance is only constructed when it's actually needed
  - Especially useful if singleton construction is expensive
- Doesn't suffer from dynamic-initialization issues
  - Singleton-pointer is statically initialized to 0 right when the module is loaded into memory
  - No dynamic operations needed
  - No issues with calls from other translation units

```
int ImportantValue =  
    Singleton::Instance().getImportantValue();
```

---

---

## Static Singleton Pointer (3)

- Still one problem though...
    - How do we call the destructor on our singleton object?
  - Memory management is not the issue:
    - When the process terminates, OS can reclaim all memory allocated by the process
  - Many other resources cannot be reclaimed by OS!
    - Shared memory is notorious: requires a reboot.
    - Semaphores, other inter-process communication resources
    - Low-level networking resources
  - Hmm, we need to call that destructor.
    - Static singleton pointers aren't so cool after all. ☹
-

---

# Static Local Singletons

- One other way to create a singleton object:

```
class Singleton {  
    ... // No static singleton members!  
public:  
    ...  
    static Singleton & Instance() {  
        static Singleton instance;  
        return instance;  
    }  
};
```

- **instance** is constructed the first time execution passes through the variable declaration...
  - Value of **instance** is remembered between calls...
  - A very simple and elegant way of creating a singleton!
-

---

# Static Local Singletons (2)

- C++ standard is wonderfully clear about static local objects!

```
class Singleton {  
    ...  
    static Singleton & Instance() {  
        static Singleton instance; // static local object  
        return instance;  
    }  
};
```

- Static local objects are constructed the first time execution passes through the variable declaration
    - Exactly what we need for lazy initialization
  - C++ also registers a destructor call for static local objects
    - Destructor is always called at process termination!
-

---

# Static Local Singletons (3)

- This technique still has issues with multithreading:

```
class Singleton {  
    ...  
    static Singleton & Instance() {  
        static Singleton instance;  
        return instance;  
    }  
};
```

- Compiler effectively manages a Boolean flag that tracks whether **instance** has been initialized
  - Access from multiple threads can cause multiple constructor calls on **instance** – not good.
  - Easier to solve *efficiently* when using the more explicit “singleton-pointer” approach, using a mutex.
    - Look up “double-checked locking”
-

---

# Singletons: A Summary

- Provide a static accessor – the “global access point”
  - These standard operations are made private:
    - ❑ Constructor
    - ❑ Destructor
    - ❑ Copy-constructor (don't support)
    - ❑ Assignment operator (don't support)
  - Several ways to create singleton objects
    - ❑ For single-threaded use, static local object approach is simple and clean
    - ❑ For multithreaded use, static pointer approach is easier to make correct
    - ❑ Avoid static object approach – prone to nasty failures!
-

---

# Singleton References

- More Effective C++, Item 26
    - Scott Meyers
    - The “static local object” approach
  
  - Modern C++ Design, Chapter 6
    - Andrei Alexandrescu
    - A broad and deep coverage of all singleton patterns and major design considerations
    - A great book for advanced C++ techniques in general! Buy it!
-