CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 16 – Static Classes and Name Spaces
Announcement

- **Lab #4**
  - Programs and Mini project due: 3/12 (Thursday) at 11:59PM

- **Lab #5**
  - Programs and Mini project due 3/19 (Thursday) at 11:59PM

- **Mid-term exam on 3/31 (Tuesday) during lecture**
  - Review on 3/19

- **Final project out this Friday (in UCMCROPS)**
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- **Reading assignment**
  - Ch. 8 and 9
Static Variables

- Recall what a static variable is
  - a variable in a static storage space that exists for the entire duration of the program
  - the visibility of the variable can be local, for example only inside a function or class

- Static variables inside functions
  - initialized only when the function is called for the first time

- Static classes inside functions
  - same thing: constructors are only called when the function is called for the first time
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X ( int ii=0 ) : i(ii) { cout << "X::X("<<i<<")" << endl; } // Default
    ~X() { cout << "X::~X("<<i<<")" << endl; }
};

void f() {
    static X x1(47);
    static X x2; // Default constructor required
}

int main() {
    cout<<"before f:\n";
    f();
    cout<<"after f:\n";
}
Destructors

- Are destructors of static objects called?
  - Yes!
  - when main() exits
    - when exit() is called as well

- Destruction of static objects
  - occurs in the reverse order of initialization
  - only objects that have been constructed are destroyed.
    - C++ keeps track of initialization order and the objects that have been constructed.
Destructors

- Global objects
  - always constructed before entering `main()`.  
  - destroyed as `main()` exits
  - if a function containing a local static object is never called:
    - the constructor for that object is never executed,
    - so the destructor is also not executed.

- Global objects declared static in a file
  - have file scope
    - not accessible from other files
//: C10:StaticDestructors.cpp
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
  char c; // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Global (static storage): Constructor & destructor always called
void f() { static Obj b('b'); }
void g() { static Obj c('c'); }

int main() {
  out << "inside main()" << endl;
f(); // Calls static constructor for b
g(); // Calls static constructor for c
  out << "leaving main()" << endl;
}
Global Static Classes

- Global variables can be used by all objects of a class.
  - Not too safe: everyone can modify the data.
  - Their names can clash with other identical names in a large project.

- How to safely define global variables?
  - One option: put all of them as private static members of a class.
    - Create members (static or not) to provide safe access
Global Static Classes

//: C10::Statinit.cpp Scope of static initializer
int x = 100;
class WithStatic {
    static int x;
    static int y;
public:
    void print() const {
        cout << "WithStatic::*x = " << x << endl;
        cout << "WithStatic::*y = " << y << endl;
    }
};

int WithStatic::*x = 1;  // Define storage x
int WithStatic::*y = x + 1;  // Define storage y
    // WithStatic::*x, NOT ::x

int main() {
    WithStatic ws;
    ws.print();
}
How to safely ensure only one instance?

```cpp
//: C10:Singleton.cpp
// Static member of same type, ensures that only one object of this type exists.
// Also referred to as the "singleton" pattern.
#include <iostream>
using namespace std;

class Egg {
    static Egg e;
    int i;
    Egg(int ii) : i(ii) {} // Constructor is private (no default constructor)
    Egg(const Egg&); // Prevent copy-construction, to be safe
public:
    static Egg* instance() { return &e; }
    int val() const { return i; }
};

Egg Egg::e(47);

int main() {
    // Egg x(1); // Error -- can't create an Egg
    // You can access the single instance:
    cout << Egg::instance()->val() << endl;
}
```
How to safely ensure the order of static variables initialization?

- Order will follow declaration order in a given file
- But order is unknown across multiple files
  - problem if there is dependency of static/global variables!

Ex: the following case is unpredictable:

```cpp
// file1.cpp has:
extern int y;
int x = y + 1;

// file2.cpp has:
extern int x;
int y = x + 1;
```
How to solve the order problem?

- Solution 1: add a class to initialize your variables in one file, but have it depend on a variable declared in another file:

```cpp
Initializer::Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialize first time only
    if(initCount++ == 0) {
        // InitCount is a static declared in another file (= 0)
        x = 100;
        y = 200;
    }
}
```

- Solution 2:
  - Place every static variable inside a function that gives access to it.
  - In that way, when a value is needed, the function access will guarantee initialization.
Names can easily conflict with each other
- Names of functions, variables, classes, etc.
- When needed, use namespaces!
- A namespace puts the names of its members in a distinct space.

Creating a name space:

```cpp
namespace MyLib {
    // Put any declarations here
    // Put any more declarations here
    // ...
}
```
Name Spaces

Name spaces can continue across different files:

header1.h

```c
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
   extern int x;
   void f();
   // ...
}
#endif // HEADER1_H
```

header2.h

```c
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
   extern int y;
   void g();
   // ...
}
#endif // HEADER2_H //://:~
```
Name Spaces

- A namespace name can be aliased to another name:

```cpp
//: C10:BobsSuperDuperLibrary.cpp
namespace VeryLongLibraryName {
    class Widget { /* ... */ }; 
    class Poppit { /* ... */ }; 
    // ...
}

// Too much to type! I’ll alias it:
namespace Lib = VeryLongLibraryName;
int main() {}
Name Spaces

“unnamed namespaces” can be used instead of using static:
(“static functions” are already deprecated)

```cpp
//: C10:UnnamedNamespaces.cpp
namespace { // declarations only available in the current file:
    class Arm { /* ... */ };
    class Leg { /* ... */ };
    int i, j, k;
    // etc
}

int main() { /* ... */ };
```
friend declarations include external members into a namespace:

```cpp
namespace Me {
    class Us {
        //...
        friend void you();
    }
    void you();
}
```
The same scope resolution operator that is used for classes is used for namespaces.

```cpp
//: C10:ScopeResolution.cpp
namespace X {
  class Y { public:
    static int i;
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9; // Y is a class inside namespace X

class X::Z { // Z is a class inside namespace X
  int u, v, w; // ( Z is being defined outside namespace X {} )
  public:
    Z(int i);
    int g();
};

X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() { // f is a function inside namespace X
  X::Z a(1); // (but being defined outside)
  a.g();
}
The “using” Directive

- Allows to import an entire namespace to the existing namespace
  - We do this with \texttt{std} all the time.
  - The directive is in effect until the end of the file.

```cpp
//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
using namespace Int;
Integer a, b; // From Int namespace
Integer divide(Integer, Integer);
// ...
}
#endif // NAMESPACEMATH_H
```
The “using” Directive

- We can also
  - limit the directive to the scope of a function
  - always use the namespace to disambiguate.

```cpp
#include "NamespaceMath.h"
int main() {
    using namespace Math;
    Integer a;
    a.setSign(negative);
    // Now scope resolution is necessary
    // to select Math::a :
    Math::a.setSign(positive);
}
```
The “using” Directive

- We can also
  - import only one name from a namespace

```cpp
//: C10:UsingDeclaration.h
namespace U {
    inline void f() {}
    inline void g() {}
}
namespace V {
    inline void f() {}
    inline void g() {}
}

void h() {
    using namespace U;  // Using directive
    using V::f;        // Using declaration
    f();              // Calls V::f();
    U::f();           // Must fully qualify to call
}
```