Chapter 10

NAME CONTROL

***Listing 10-1. Static Variables in Functions***

**//: C10:StaticVariablesInfunctions.cpp**

**#include "../require.h" // To be INCLUDED from Header FILE in *Chapter 9***

**#include <iostream>**

**using namespace std;**

**char oneChar(const char\* charArray = 0) {**

**static const char\* s;**

**if(charArray) {**

**s = charArray;**

**return \*s;**

**}**

**else**

**require(s, "un-initialized s");**

**if(\*s == '\0')**

**return 0;**

**return \*s++;**

**}**

**char\* a = "abcdefghijklmnopqrstuvwxyz";**

**int main() {**

**// oneChar(); // require() fails**

**oneChar(a); // Initializes s to a**

**char c;**

**while((c = oneChar()) != 0)**

**cout << c << endl;**

**} ///:~**

***Listing 10-2. Static Class Objects Inside Functions***

**//: C10:StaticObjectsInFunctions.cpp**

**#include <iostream>**

**using namespace std;**

**class X {**

**int i;**

**public:**

**X(int ii = 0) : i(ii) {} // Default**

**~X() { cout << "X::~X()" << endl; }**

**};**

**void f() {**

**static X x1(47);**

**static X x2; // Default constructor required**

**}**

**int main() {**

**f();**

**} ///:~**

***Listing 10-3. Static Object Destructors***

**//: C10:StaticDestructors.cpp**

**/// Static object destructors**

**#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() not called**

**out << "leaving main()" << endl;**

**} ///:~*****Listing 10-4. Local extern***

**//: C10:LocalExtern.cpp**

**//{L} LocalExtern2**

**#include<iostream>**

**int main() {**

**extern int i;**

**std::cout << i;**

**} ///:~**

***Listing 10-5. Another Local extern***

**//: C10:LocalExtern2.cpp {O}**

**int i = 5;**

**///:~**

***Listing 10-6. Creating a Namespace***

**//: C10:MyLib.cpp**

**namespace MyLib {**

**// Declarations**

**}**

**int main() {} ///:~**

***Listing 10-7. Illustrating Continuation of a Namespace Definition***

**//: C10:Header1.h**

**#ifndef HEADER1\_H**

**#define HEADER1\_H**

**namespace MyLib {**

**extern int x;**

**void f();**

**// ...**

**}**

**#endif // HEADER1\_H ///:~**

**//: C10:Header2.h**

**#ifndef HEADER2\_H**

**#define HEADER2\_H**

**#include "Header1.h" // To be INCLUDED from Header FILE above**

**// Add more names to MyLib**

**namespace MyLib { // NOT a redefinition!**

**extern int y;**

**void g();**

**// ...**

**}**

**#endif // HEADER2\_H ///:~**

**//: C10:Continuation.cpp**

**#include "Header2.h" // To be INCLUDED from Header FILE above**

**int main() {} ///:~**

***Listing 10-8. Illustrating Continuation of a Namespace Definition (over Multiple Header Files)***

**//: C10:BobsSuperDuperLibrary.cpp**

**namespace BobsSuperDuperLibrary {**

**class Widget { /\* ... \*/ };**

**class Poppit { /\* ... \*/ };**

**// ...**

**}**

**// Too much to type! I’ll alias it:**

**namespace Bob = BobsSuperDuperLibrary;**

**int main() {} ///:~*****Listing 10-9. Unnamed Namespaces***

**//: C10:UnnamedNamespaces.cpp**

**namespace {**

**class Arm { /\* ... \*/ };**

**class Leg { /\* ... \*/ };**

**class Head { /\* ... \*/ };**

**class Robot {**

**Arm arm[4];**

**Leg leg[16];**

**Head head[3];**

**// ...**

**} xanthan;**

**int i, j, k;**

**}**

**int main() {} ///:~**

***Listing 10-10. Injecting a friend into a Namespace***

**//: C10:FriendInjection.cpp**

**namespace Me {**

**class Us {**

**//...**

**friend void you();**

**};**

**}**

**int main() {} ///:~**

***Listing 10-11. Explicitly Specifying a Name in a Namespace (using the Scope Resolution Operator)***

**//: C10:ScopeResolution.cpp**

**namespace X {**

**class Y {**

**static int i;**

**public:**

**void f();**

**};**

**class Z;**

**void func();**

**}**

**int X::Y::i = 9;**

**class X::Z {**

**int u, v, w;**

**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() {**

**X::Z a(1);**

**a.g();**

**}**

**int main(){} ///:~*****Listing 10-12. Illustrating A Simple Namespace***

**//: C10:NamespaceInt.h**

**#ifndef NAMESPACEINT\_H**

**#define NAMESPACEINT\_H**

**namespace Int {**

**enum sign { positive, negative };**

**class Integer {**

**int i;**

**sign s;**

**public:**

**Integer(int ii = 0)**

**: i(ii),**

**s(i>= 0 ? positive : negative)**

**{}**

**sign getSign() const { return s; }**

**void setSign(sign sgn) { s = sgn; }**

**// ...**

**};**

**}**

**#endif // NAMESPACEINT\_H ///:~**

***Listing 10-13. Illustrating the*** *using* ***Directive***

**//: C10:NamespaceMath.h**

**#ifndef NAMESPACEMATH\_H**

**#define NAMESPACEMATH\_H**

**#include "NamespaceInt.h" // To be INCLUDED from Header FILE above**

**namespace Math {**

**using namespace Int;**

**Integer a, b;**

**Integer divide(Integer, Integer);**

**// ...**

**}**

**#endif // NAMESPACEMATH\_H ///:~**

***Listing 10-14. Illustrating the*** *using* ***Directive (albeit in a different way)***

**//: C10:Arithmetic.cpp**

**#include "NamespaceInt.h"**

**void arithmetic() {**

**using namespace Int;**

**Integer x;**

**x.setSign(positive);**

**}**

**int main(){} ///:~**

***Listing 10-15. Illustrating Namespace Overriding***

**//: C10:NamespaceOverriding1.cpp**

**#include "NamespaceMath.h" // To be INCLUDED from Header FILE**

**// above**

**int main() {**

**using namespace Math;**

**Integer a;**

**// Hides Math::a;**

**a.setSign(negative);**

**// Now scope resolution is necessary**

**// to select Math::a :**

**Math::a.setSign(positive);**

**} ///:~**

***Listing 10-16. Illustrating Namespace Overriding (again, albeit in a different way)***

**//: C10:NamespaceOverriding2.h**

**#ifndef NAMESPACEOVERRIDING2\_H**

**#define NAMESPACEOVERRIDING2\_H**

**#include "NamespaceInt.h"**

**namespace Calculation {**

**using namespace Int;**

**Integer divide(Integer, Integer);**

**// ...**

**}**

**#endif // NAMESPACEOVERRIDING2\_H ///:~**

***Listing 10-17. Illustrating Overriding Ambiguity***

**//: C10:OverridingAmbiguity.cpp**

**#include "NamespaceMath.h"**

**#include "NamespaceOverriding2.h" // To be INCLUDED from Header**

**// FILE above**

**void s() {**

**using namespace Math;**

**using namespace Calculation;**

**// Everything's ok until:**

**//! divide(1, 2); // Ambiguity**

**}**

**int main() {} ///:~**

***Listing 10-18. Illustrating a using Declaration***

**//: C10:UsingDeclaration.h**

**#ifndef USINGDECLARATION\_H**

**#define USINGDECLARATION\_H**

**namespace U {**

**inline void f() {}**

**inline void g() {}**

**}**

**namespace V {**

**inline void f() {}**

**inline void g() {}**

**}**

**#endif // USINGDECLARATION\_H ///:~**

**//: C10:UsingDeclaration1.cpp**

**#include "UsingDeclaration.h" // To be INCLUDED from Header // FILE above**

**void h() {**

**using namespace U; // Using directive**

**using V::f; // Using declaration**

**f(); // Calls V::f();**

**U::f(); // Must fully qualify to call**

**}**

**int main() {} ///:~**

***Listing 10-19. Illustrating a using Declaration within a Namespace***

**//: C10:UsingDeclaration2.cpp**

**#include "UsingDeclaration.h"**

**namespace Q {**

**using U::f;**

**using V::g;**

**// ...**

**}**

**void m() {**

**using namespace Q;**

**f(); // Calls U::f();**

**g(); // Calls V::g();**

**}**

**int main() {} ///:~**

***Listing 10-20. Illustrating the Scope of Static Initializer***

**//: C10:Statinit.cpp**

**// Scope of static initializer**

**#include <iostream>**

**using namespace std;**

**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;**

**int WithStatic::y = x + 1;**

**// WithStatic::x NOT ::x**

**int main() {**

**WithStatic ws;**

**ws.print();**

**} ///:~**

***Listing 10-21. Syntax of Static Arrays***

**//: C10:StaticArray.cpp**

**// Initializing static arrays in classes**

**class Values {**

**// static consts are initialized in-place:**

**static const int scSize = 100;**

**static const long scLong = 100;**

**// Automatic counting works with static arrays.**

**// Arrays, Non-integral and non-const statics**

**// must be initialized externally:**

**static const int scInts[];**

**static const long scLongs[];**

**static const float scTable[];**

**static const char scLetters[];**

**static int size;**

**static const float scFloat;**

**static float table[];**

**static char letters[];**

**};**

**int Values::size = 100;**

**const float Values::scFloat = 1.1;**

**const int Values::scInts[] = {**

**99, 47, 33, 11, 7**

**};**

**const long Values::scLongs[] = {**

**99, 47, 33, 11, 7**

**};**

**const float Values::scTable[] = {**

**1.1, 2.2, 3.3, 4.4**

**};**

**const char Values::scLetters[] = {**

**'a', 'b', 'c', 'd', 'e',**

**'f', 'g', 'h', 'i', 'j'**

**};**

**float Values::table[4] = {**

**1.1, 2.2, 3.3, 4.4**

**};**

**char Values::letters[10] = {**

**'a', 'b', 'c', 'd', 'e',**

**'f', 'g', 'h', 'i', 'j'**

**};**

**int main() { Values v; } ///:~**

***Listing 10-22. Illustrating Static Arrays of Class Objects***

**//: C10:StaticObjectArrays.cpp**

**// Static arrays of class objects**

**class X {**

**int i;**

**public:**

**X(int ii) : i(ii) {}**

**};**

**class Stat {**

**// This doesn't work, although**

**// you might want it to:**

**//! static const X x(100);**

**// Both const and non-const static class**

**// objects must be initialized externally:**

**static X x2;**

**static X xTable2[];**

**static const X x3;**

**static const X xTable3[];**

**};**

**X Stat::x2(100);**

**X Stat::xTable2[] = {**

**X(1), X(2), X(3), X(4)**

**};**

**const X Stat::x3(100);**

**const X Stat::xTable3[] = {**

**X(1), X(2), X(3), X(4)**

**};**

**int main() { Stat v; } ///:~**

***Listing 10-23. Illustrating Static Members and Local Classes***

**//: C10:Local.cpp**

**// Static members & local classes**

**#include <iostream>**

**using namespace std;**

**// Nested class CAN have static data members:**

**class Outer {**

**class Inner {**

**static int i; // OK**

**};**

**};**

**int Outer::Inner::i = 47;**

**// Local class cannot have static data members:**

**void f() {**

**class Local {**

**public:**

**//! Static int i; // Error**

**// (How would you define i?)**

**} x;**

**}**

**int main() { Outer x; f(); } ///:~**

***Listing 10-24. Illustrating a Simple static Member Function***

**//: C10:SimpleStaticMemberFunction.cpp**

**class X {**

**public:**

**static void f(){};**

**};**

**int main() {**

**X::f();**

**} ///:~**

***Listing 10-25. Illustrating static Data Members and static Member Functions (used in Combination)***

**//: C10:StaticMemberFunctions.cpp**

**class X {**

**int i;**

**static int j;**

**public:**

**X(int ii = 0) : i(ii) {**

**// Non-static member function can access**

**// static member function or data:**

**j = i;**

**}**

**int val() const { return i; }**

**static int incr() {**

**//! i++; // Error: static member function**

**// cannot access non-static member data**

**return ++j;**

**}**

**static int f() {**

**//! val(); // Error: static member function**

**// cannot access non-static member function**

**return incr(); // OK -- calls static**

**}**

**};**

**int X::j = 0;**

**int main() {**

**X x;**

**X\* xp = &x;**

**x.f();**

**xp->f();**

**X::f(); // Only works with static members**

**} ///:~*****Listing 10-26. Illustrating the “Singleton” Pattern***

**//: 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 E {**

**static E e;**

**int i;**

**E(int ii) : i(ii) {}**

**E(const E&); // Prevent copy-construction**

**public:**

**static E\* instance() { return &e; }**

**int val() const { return i; }**

**};**

**E E::e(47);**

**int main() {**

**//! E x(1); // Error -- can't create an E**

**// You can access the single instance:**

**cout << E::instance()->val() << endl;**

**} ///:~*****Listing 10-27. Illustrating “Technique One”***

**//: C10:Initializer.h**

**// Static initialization technique**

**#ifndef INITIALIZER\_H**

**#define INITIALIZER\_H**

**#include <iostream>**

**extern int x; // Declarations, not definitions**

**extern int y;**

**class Initializer {**

**static int initCount;**

**public:**

**Initializer() {**

**std::cout << "Initializer()" << std::endl;**

**// Initialize first time only**

**if(initCount++ == 0) {**

**std::cout << "performing initialization"**

**<< std::endl;**

**x = 100;**

**y = 200;**

**}**

**}**

**~Initializer() {**

**std::cout << "~Initializer()" << std::endl;**

**// Clean up last time only**

**if(--initCount == 0) {**

**std::cout << "performing cleanup"**

**<< std::endl;**

**// Any necessary cleanup here**

**}**

**}**

**};**

**// The following creates one object in each**

**// file where Initializer.h is included, but that**

**// object is only visible within that file:**

**static Initializer init;**

**#endif // INITIALIZER\_H ///:~**

***Listing 10-28. Illustrating Definitions for the Header File in Listing 10-27***

**//: C10:InitializerDefs.cpp {O}**

**// Definitions for Initializer.h**

**#include "Initializer.h" // To be INCLUDED from Header FILE**

**// above**

**// Static initialization will force**

**// all these values to zero:**

**int x;**

**int y;**

**int Initializer::initCount;**

**///:~**

***Listing 10-29. Illustrating Static Initialization (For First File)***

**//: C10:Initializer.cpp {O}**

**// Static initialization**

**#include "Initializer.h"**

**///:~**

***Listing 10-30. Illustrating More*** ***Static Initialization (for Second File)***

**//: C10:Initializer2.cpp**

**//{L} InitializerDefs Initializer**

**// Static initialization**

**#include "Initializer.h"**

**using namespace std;**

**int main() {**

**cout << "inside main()" << endl;**

**cout << "leaving main()" << endl;**

**} ///:~**

***Listing 10-31. Illustrating the First Dependency Class***

**//: C10:Dependency1.h**

**#ifndef DEPENDENCY1\_H**

**#define DEPENDENCY1\_H**

**#include <iostream>**

**class Dependency1 {**

**bool init;**

**public:**

**Dependency1() : init(true) {**

**std::cout << "Dependency1 construction"**

**< <std::endl;**

**}**

**void print() const {**

**std::cout << "Dependency1 init: "**

**<< init << std::endl;**

**}**

**};**

**#endif // DEPENDENCY1\_H ///:~*Listing 10-32. Illustrating the Second Dependency Class***

**//: C10:Dependency2.h**

**#ifndef DEPENDENCY2\_H**

**#define DEPENDENCY2\_H**

**#include "Dependency1.h" // To be INCLUDED from Header FILE**

**// above**

**class Dependency2 {**

**Dependency1 d1;**

**public:**

**Dependency2(const Dependency1& dep1): d1(dep1){**

**std::cout << "Dependency2 construction ";**

**print();**

**}**

**void print() const { d1.print(); }**

**};**

**#endif // DEPENDENCY2\_H ///:~**

***Listing 10-33. Illustrating Technique Two***

**//: C10:Technique2.cpp**

**#include "Dependency2.h" // To be INCLUDED from Header FILE**

**// above**

**using namespace std;**

**// Returns a value so it can be called as**

**// a global initializer:**

**int separator() {**

**cout << "---------------------" << endl;**

**return 1;**

**}**

**// Simulate the dependency problem:**

**extern Dependency1 dep1;**

**Dependency2 dep2(dep1);**

**Dependency1 dep1;**

**int x1 = separator();**

**// But if it happens in this order it works OK:**

**Dependency1 dep1b;**

**Dependency2 dep2b(dep1b);**

**int x2 = separator();**

**// Wrapping static objects in functions succeeds**

**Dependency1&d1() {**

**static Dependency1 dep1;**

**return dep1;**

**}**

**Dependency2&d2() {**

**static Dependency2 dep2(d1());**

**return dep2;**

**}**

**int main() {**

**Dependency2& dep2 = d2();**

**} ///:~**

***Listing 10-34. Illustrating the First Header File***

**//: C10:Dependency1StatFun.h**

**#ifndef DEPENDENCY1STATFUN\_H**

**#define DEPENDENCY1STATFUN\_H**

**#include "Dependency1.h"**

**extern Dependency1& d1();**

**#endif // DEPENDENCY1STATFUN\_H ///:~**

***Listing 10-35. Illustrating the Second Header File***

**//: C10:Dependency2StatFun.h**

**#ifndef DEPENDENCY2STATFUN\_H**

**#define DEPENDENCY2STATFUN\_H**

**#include "Dependency2.h"**

**extern Dependency2& d2();**

**#endif // DEPENDENCY2STATFUN\_H ///:~**

***Listing 10-36. Illustrating the first Implementation File***

**//: C10:Dependency1StatFun.cpp {O}**

**#include "Dependency1StatFun.h" // To be INCLUDED from Header FILE**

**// above**

**Dependency1&d1() {**

**static Dependency1 dep1;**

**return dep1;**

**} ///:~**

***Listing 10-37. Illustrating the Second Implementation File***

**//: C10:Dependency2StatFun.cpp {O}**

**#include "Dependency1StatFun.h"**

**#include "Dependency2StatFun.h" // To be INCLUDED from Header FILE**

**// above**

**Dependency2&d2() {**

**static Dependency2 dep2(d1());**

**return dep2;**

**} ///:~**

***Listing 10-38.*** ***Illustrating that Initialization is Not affected by the Order of Linking***

**//: C10:Technique2b.cpp**

**//{L} Dependency1StatFun Dependency2StatFun**

**#include "Dependency2StatFun.h"**

**int main() { d2(); } ///:~**