

Static Member Functions and Variables

Static member functions are operations defined within the scope of a class, but there is no implicit 'this' pointer passed as the first argument to the function. This means that a static function is not invoked on an object, but is instead invoked on a class.

A static member function has access to the private data of a class. It can access static member variables, or given a pointer or reference to an object of the class as an argument, it can access the private instance variables of any object passed as an argument.

```
class String {
private:
    ... // as before
    static int _count; // a count of the total number of String objects allocated
public:

    String() { ++_count; ... }
    String(const char*) { ++_count; ... }
    String(const String&) { ++_count; ... }
    ~String() { --_count; ... }
    ...
    static int count() { return _count; }
};
```

Each constructor is required to increment the count, and the destructor decrements the count. The count() method is a static method, which is called on the String class. It can also be called on a string object, since each object of type String share the same _count object.

```
int c = String::count();
```

Static Member Variable Initialization

When you declare a static member variable, you must remember to initialize it. Since static data is allocated and initialized before main starts to execute, how does one accomplish static class variable initialization? The technique for initializing a static member variable is dependent on the compiler. All pre-ANSI compilers require you to initialize a static member variable in some implementation file (e.g., a .cpp file). So, to initialize the count of String objects, you would write the following in

```
int String::_count = 0;
```

Compilers that claim to be ANSI C++ compiler should allow initialization of a static member variable to occur at the point of declaration with a class. The reason this has to be allowed is because of a problem that occurs with template classes. For example:

```
template<class T> class Foo {  
    static int x; // pre-ANSI C++ requires initialization in Foo.C  
public:  
    ...  
};
```

If static variables must be initialized in an implementation file, how does the designer of class Foo arrange to initialize x for all possible types T with which Foo can be parameterized?

```
// in Foo.C  
int Foo<char>::x = -1;  
int Foo<int>::x = -1;  
... // and so on for all builtin types and all possible user-defined types!!!
```

Alternatively, the undesirable burden is placed on the user of template Foo to remember to initialize Foo<T>::x for whatever types are given as parameters to template Foo. Fortunately, ANSI C++ allows us to initialize x at the point of declaration inside of a class or template.

```
static int x = -1; // init at point of declaration in scope of Foo
```

Combining Static Methods and Static Variables

In C++, `operator new` and `operator delete` are static methods that can be overloaded on a class-by-class basis. Default versions of both operators are also defined in global scope. You might overload `new/delete` to provide alternative memory management of special kinds of objects for efficiency. For example:

```
class Window {
private:
    static Window* _root = 0;    // ANSI C++ allows inline initialization
    static list<Window*> _wlist;
    // hide new/delete and constructor/destructor to force heap allocation
    static void* operator new (size_t size);
    static void operator delete (void* p);
    Window();
    Window(Window&);
    ~Window();
public:

    static Window* create() {
        Window* w = new Window();
        if (_root == 0) _root = w;
        _wlist.push_back(w);
        return w;
    }
    // destroy finds s in _wlist, removes it, and then deletes it
    static void destroy(Window* s) { ...; delete(s); }
    ...
};

// Window objects can only be created by calling static create method
Window* win = Window::create();
...
Window::destroy(win);
```

Class Variables vs Constants vs Enums

Suppose we want a Stack of integers of a fixed size. How do each of the following differ?

```
class Stack {
public:
    static const int SIZE = 100;
private:
    int _stk[SIZE];
    ...
};
```

```
class Stack {
public:
    const int SIZE = 100; // how is this different from a static const int
private:
    int _stk[SIZE];
    ...
};
```

```
class Stack {
public:
    enum { SIZE = 100 }; // what about an 'anonymous' enum type??
private:
    int _stk[SIZE];
    ...
};
```

Q: Is there any advantage to using an enumeration over a static const int or a non-static const int?

Note: the public enum element SIZE is referred to as Stack::SIZE outside of the class.

Namespaces

ANSI C++ introduces the concept of a namespace. Namespaces are used to group related declarations into a common scope, so you must use the scope qualification operator '::' to refer to elements of the namespace. Alternatively, you can “open” the namespace by “using” it.

The purpose of a namespace is to group common abstractions together to avoid a name conflict with other types or operations that might be used together in a program. Name conflicts arise when classes you have defined conflict with classes of the same name in a class library you are using. So, namespaces give the programmer a way to define **packages** of related types, objects, and operations.

Some C++ compilers lack the implementation of namespaces. For example, g++ 2.7 reports the following warning! g++ 2.8 and later support namespaces. The SunPro 5.0 C++ compiler does, but the SunPro 4.2 compiler does not.

```
% g++ -g namespace.C
namespace.C:1: warning: namespaces are mostly broken in this version of g++
namespace.C: In function `int main()':
namespace.C:14: parse error before `;'
```

ANSI C++ compilers do support namespaces. For example, the ANSI C++ classes like string, list, and iostream are all part of the 'std' package:

```
// in <list>
namespace std {
    template<class T> class list { ... }
};
```

Note that the same namespace name can be used in multiple include files. So you can split up the components of your class library package among multiple header files, but logically group them all within the same namespace using multiple namespace definitions spread across a set of files.

Namespaces

An element of a namespace is used by using the name of the namespace as a qualifier:

```
std::list<int> int_list;
```

Alternatively, you can define a **using directive** to “open” a namespace within some scope defined in your own code, which is usually a file scope:

```
#include <list>
using namespace std;    // a using directive to open the 'std' namespace package

list<int> int_list;
```

You can also define a **using declaration** to selectively use an element of a namespace without having to open up the entire namespace and possibly encounter a name conflict.

```
#include <string>
#include <list>
int main()
{
    using std::string;
    string x;    // short name for std::string
    std::list<string> slist; // still have to use name qualification for list
    ...
}
```

Resolving naming conflicts

Name spaces were invented in C++ as a way to avoid naming conflicts. A really simple but very annoying problem that occurred in pre-ANSI C++ was the definition of `bool`, which was not a builtin type in C++ originally. There are several ways to define `bool`. Two of the most common are:

```
typedef int bool
#define TRUE -1    // or const int True = -1
#define FALSE 0   // or const int False = 0;

enum bool { false, true };
```

When writing C++ code, it was often the case that a programmer would use class Library A which included a header file that defined `bool` using a typedef and another class library B that defined `bool` using an enumerated type. If you ended up including header files from both class libraries (a very common thing to do), there would be a type conflict.

The problem is that a common type name, 'bool', was not defined by the language, and so every programmer that needed a `bool` defined their own type in the global namespace. Namespaces allow a programmer to partition the global name space to avoid these types of naming collisions. Of course, there is now the chance that namespaces will have conflicting names. One suggestion is that organizations define name spaces based on Internet domain names. This avoids conflicts among different class library vendors, or classes from individuals. For example:

```
namespace ABC_Class_Lib { class String {...}; ... }
namespace XYZ_Class_Lib { class String {...}; ... }

namespace Freds_Cool_Internet_Stuff_Release_1_2 {
    using namespace ABC_Class_Lib;
    using namespace XYZ_Class_Lib;
    using ABC_Class_Lib::String; // prefer ABC String class
    typedef XYZ_Class_Lib::String XYZ_String; // rename
    ...
}
namespace Freds = Freds_Cool_Internet_Stuff_Release_1_2; // define alias namespace
```

Simulating Namespaces using a Class

Since all compilers are not yet ANSI C++ compliant, namespaces may not be available to use. However, it is possible to do a limited form of namespace by combining nested classes, a public class and static member functions. For example:

```
#include <stdio.h> // C standard I/O library

class Stdio {
public:
    typedef FILE File;

    static File* fopen(const char* file, const char* type) { return ::fopen(file, type); }
    static int fclose(File* fp) { return ::fclose(fp); }

    static void fprintf(FILE*, const char*, ...);
    static void printf(const char*, ...);
    ...
};

Stdio::File* foo = Stdio::fopen("/etc/passwd", "r");
...
```

The Stdio class introduces a new named scope in which public static functions are defined that just call the corresponding C functions. The effect is that the class introduces a scope that requires name qualification. This is helpful in some cases; however, this technique is not the same as a namespace, and is more limited since you cannot “spit” a class across multiple include files. When you define a class, its entire definition must occur within a single contiguous scope since the class keyword defines a type. A namespace defines a scope that is “open” in the sense that multiple namespace definitions with the same name define a logical grouping.

By the way, note that a class with all public members is the same as a ‘struct’, or alternatively, a struct is a public class.