

## FYI

**1. Read Chapter 3 - Templates**

This is a foundational chapter for the rest of this course. Study carefully.

**2. Homework due in class on Thursday, 24 June.**

Exercises 2.1-2.14 in the course text.

**3. New programming assignment announcement tomorrow! Due in class Thursday, June 27**

.....check the web page later today to get an early start

**4. There will be no office hours or classes the week of July 1- July 4.**

I will be working in my London office from June 28th until July 5th.

I have a programming assignment for you to work on while I am gone.

I will be reading email, but note that there is a 6 hour time difference.

I will schedule 3 makeup classes on 3 Fridays during the term, if everyone is agreeable.

**A Note on Inline Member Functions in .h files**

Note that when you define a member function, you can define an inline body, or you can define just the function declaration, and then provide inline implementations following the class declaration, using the **inline** keyword.

```
class String {
    char* _string;
    int _length;
public:
    String() { _string = NULL; _length = 0; }
    String(const String&);
    String(const char*);

    ~String() { delete _string; }
    int length() const { return _length; }
};

inline String::String(const String& s)
{
    ...
}

inline String::String(const char* s)
{
    ...
}
```

**Question: If you do not use the inline keyword, and you include String.h in multiple .C files, what happens when you attempt to link an executable program?**

**More on Static Member Data Initializers**

A static class variable initializer MUST be defined in a .C file, not a .h file.

```
/* Declared in String.h */
class String {
    ...
    static int _count;
public:
    ...
};

/* Defined in String.C */
int String::_count = 0;
```

The reason for this is that we can only ever have one static initialization. If the initialization we defined in a .h file, and we included that .h file in multiple .C files, what would happen?

**The Assignment Operator**

For every class, it is possible to overload the assignment operator. You should always implement operator=. In fact, you should ALWAYS define the following 4 methods for every class:

1. the default constructor
2. the "copy constructor"
3. the destructor
4. the assignment operator.

```
class String {
    char* _string;
    int _length;

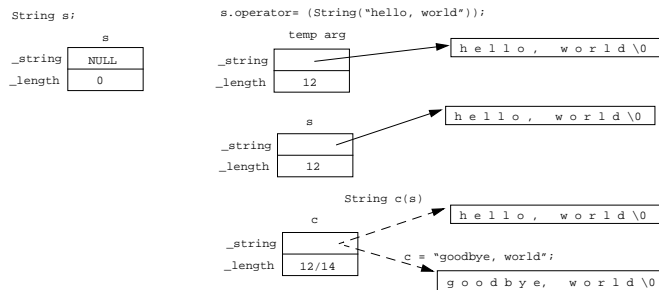
    void copy(const char* str) {
        if (str) {
            delete _string;
            _length = strlen(str);
            _string = new char[_length + 1];
            strcpy(_string, str);
        }
    }

public:
    String() : _string(NULL), _length(0) {}
    String(const String& s) : _string(NULL), _length(0) { copy(s._string); }
    String& operator=(const String& s) {
        if (this != &s) // VERY IMPORTANT -- check for self assignment
            copy(s._string);
        return *this;
    }
    ...
};
```

## The Assignment Operator

How does the assignment operator differ from a copy constructor?

```
String s;
s = String("hello, world");
String c(s);
c = "goodbye, world";
c = c; // what happens on "self assignment"?
```



8/5/00

5 of 12

## Type Name Collisions

The following are common definitions of a Boolean type, defined in the global name space:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

or maybe

```
enum Bool { FALSE = 0, TRUE = 1 };
```

or

```
enum bool_t { False = 0, True = 1 };
```

The problem is that you can well expect that either someone else already thought of these and you may eventually include one of a class header file from some class library, and get a compiler error because of a duplicate type definition for Bool. NOTE: Some new C++ compilers define 'bool' as a builtin type, and 'true' and 'false' are also defined. So, how do you avoid name collisions? Well, you can be obscure or trick:

```
enum Bool_t { flaze = 0, truee = 1 };
```

```
enum Boolean { notTrue = 0, notFalse = 1 };
```

The problem is that in a OO language where we are prone to reuse other class libraries, there is always an opportunity for name collision. How do we cope?

8/5/00

6 of 12

## Namespaces

A recent addition to the C++ language, to deal with the problem of name collisions in classes developed as part of a class library. Namespaces do not define a class, they define a named (or unnamed) "scope" in which types, data, and functions may be declared. The name associated with the namespace is used to qualify access.

For example, you might implement a bool.h header file for that defines a boolean enumerated type as follows:

```
/* bool.h */
#ifndef _MFC_bool_h
#define _MFC_bool_h 1

namespace MicroserfFoundationClasses {

enum bool { false = 0, true = 1 };

}

#endif // _MFC_bool_h
```

If you then want to use this bool type, you use the namespace name as a scope qualifier.

```
#include <bool.h>
MicroserfFoundationClasses::bool b = false;
```

But this seems rather tedious, so we might define a shorter name using typedef to define a type alias:

```
#include <bool.h>
typedef MicroserfFoundationClasses::bool MFCBool;
...
MFCBool b = false;
```

8/5/00

7 of 12

## Namespaces

Alternatively, we can use define a namespace as an alias namespace:

```
namespace MFC = MicroserfFoundationClasses;
MFC::bool b = false;
```

Namespaces may also be "opened" with a using statement, when you are sure there will be no chance of a name collision:

```
using MicroserfFoundationClasses::bool // open namespace so that only bool type is visible
// bool is really MicroserfFoundationClasses::bool
bool b = false;
```

If you open the entire name space with a using directive, then everything declared in the namespace is visible without explicit qualification:

```
using namespace MicroserfFoundationClasses; // everything is visible
...
bool b = false;
```

8/5/00

8 of 12

## Namespaces

It is very likely the case that a namespace will have lots and lots of enumerated types, classes, and functions defined as part of the namespace, and they will not all be defined in the same .h file. So, it must be possible to define the same namespace in different translation units (.h files).

```
/* MFC String.h */
#ifndef _MFC_String_h
#define _MFC_String_h 1

namespace MicroserfFoundationClasses {

class String {
private:
    ...
public:
    ...
};

} // MicroserfFoundationClasses

#endif // _MFC_String_h
```

The namespace defined in bool.h is the same as the one defined on String.h, that is, the namespace can appear in multiple translation units (.h files) and it is the same namespace as far as the compiler is concerned. So, a single using directive “opens” multiple namespace declarations and as though it was one big namespace:

```
#include <bool.h>
#include <String.h>
using namespace MicroserfFoundationClasses; // bool and String now visible
```

## Simulating Namespaces

Since some compilers don’t implement the namespace syntactic construct, you can *almost* simulate a namespace using a struct or class:

```
#ifndef _bool_h
#define _bool_h 1

struct MyLibrary { // a struct is just a public class
enum bool { false = 0, true = 1 };
};

In some program:

#include "bool.h"

MyLibrary::bool flag = true;

Or

typedef MyLibrary::bool bool;
...
bool flag = true;
```

**Question:** Why is using a struct/class with a nested type definition NOT equivalent to a namespace? Can the same struct name be defined in multiple translation units?

## Namespaces

If you are using some set of classes that are not defined in a namespace, say because the class library predates the namespace feature of C++, you can introduce a namespace for those classes by “wrapping” all the declarations in an appropriate include file:

```
/* stl.h */

#ifndef _stl_h
#define _stl_h 1

namespace StandardTemplateLibrary {
#include <vector.h>
#include <stack.h>
#include <queue.h>
... // etc
}

namespace STL = StandardTemplateLibrary;

#endif /* _stl_h */

In some program:

#include "stl.h"

STL::Stack s;

Or just the following:

using STL;

Stack s;
```

## Classes with all static member functions

You can define a class with all static member functions and no data:

```
class Stdlib {
    /* no data elements */
public:
    static int strlen(const char* s) { return ::strlen(s); }
    static int strcmp(const char* a, const char* b) { return ::strcmp(a, b); }
    ...
};

// to use one of these functions, you use the class name as the scope qualifier
int x = Stdlib::strlen("hello, world");
```

**Question 1:** What is sizeof(Stdlib)?

**Question 2:** Would you ever need to instantiate an object of this class?

**Questions 3 & 4:** Can you accomplish the same thing using a namespace? If so, what is the advantage of using a namespace over a class?