

ENCM 339 Fall 2005 L02

Lecture Notes for Friday, Oct. 21

Author: Dr. S. A. Norman

Electronic copies of handouts for L02 and T02 can be found at <http://www.enel.ucalgary.ca/People/Norman/encm339fall12005/>

Preface

The L02 lecture on Friday, October 21 is cancelled because I will be out of town, and I have not been able to find a colleague who is able to lecture in my place that day. (Note: The L01 lecture—Dr. Moussavi’s class—on October 21 is *not* cancelled.)

This handout is being provided as a substitute for a lecture October 21. Please study this material carefully—you will be expected to be familiar with it when normal lectures resume on Monday, October 24.

Related Reading

Related to this lecture:

- *Handout for L02 (Dr. Norman’s lecture) Wed., Oct. 19*—the files listed there will be discussed in detail in these lecture notes.
- Lippman, Lajoie, and Moo, *C++ Primer, 4th edition*, Sections 2.8, 12.1, and 7.7.

Preparation for Monday, October 24:

- Lippman, Lajoie, and Moo, *C++ Primer, 4th edition*, Sections 5.6, 12.2, 7.5 and 5.11.

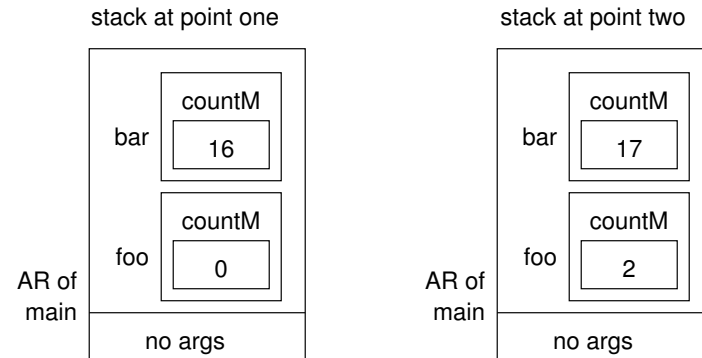
Lecture Outline

This lecture covers the most basic aspects of writing and using C++ class types:

- initialization of class objects using constructors;
- the kinds of operations that can be performed using `const` and `non-const` member functions;
- access control to members using the keywords `public` and `private`.

Review: Diagrams for points one and two

Diagrams for points one and two in `main` in `useCounter.cpp` were given in the lecture Wed., Oct. 19. Here they are again:



When the program is run the output is:

```
point 2 foo count: 2
point 2 bar count: 17
```

How does a class object do its job?

A class object performs its duties by executing code in *member function definitions*. The file `Counter.cpp` provides four examples of member function definitions.

How did `foo.countM` and `bar.countM` get initialized in `main`?

It’s easy to guess what the `increment` and `count` functions do. If you’ve never seen code for C++ class types before, it may be less easy to guess why the `countM` variable inside `foo` and the `countM` variable inside `bar` got their initial values of 0 and 16. The initializations happened because of special member functions called *constructors*.

What are constructors?

The role of a constructor function (often abbreviated as “ctor” in C++ comments and hand-written lecture notes) is to *initialize* class objects—to make them ready for use at the moments they are created. All classes should have one or more constructors.

The syntax for constructors is special: a constructor has the *same name as the class it belongs to*, and a constructor has *no return type* (not even `void`).

Looking at `Counter.h` and `Counter.cpp`, you should be able to see that the `Counter` class has two different constructors.

The default constructor

The *default constructor* is the constructor used to initialize a class object when no information is supplied for initialization of the object. The default constructor takes no arguments.

In `main` in `useCounter.cpp`, the default constructor will ensure that the initial value of `foo.countM` is 0.

Let's look in detail at the syntax of the default constructor function definition of the `Counter` class:

```

Counter::Counter( )
: countM(0)
{
}

```

says we are defining a member function of the Counter class

empty arg list indicates that this is a default constructor

says we are defining a constructor function

member initialization list

body is empty--in this example, all the work is done in the member initialization list

Member initialization lists

A *member initialization list* is a special feature of constructor function definitions; no other kind of C++ function definition is allowed to use them. The syntax is as follows:

```
: member-name(expression), member-name(expression), ...
```

You should be able to see that both constructor definitions in `Counter.cpp` follow this pattern.

Simple constructors such as those in `Counter.cpp` can get all of their work done with a member initialization list, leaving nothing to be done in the body of the function definition. Later in the course you'll see more complicated constructors, with non-empty bodies.

Other constructors

In addition to a default constructor, a class may provide one or more other constructors that can be used to initialize class objects using information passed as one or more arguments.

The `Counter` has a very simple example of such a constructor—the second constructor defined in `Counter.cpp` can be used to create a `Counter` class object with an initial count chosen by a programmer. In `main` in `useCounter.cpp`, this constructor will give `bar.countM` a value of 16.

Example constructor calls

In this code:

```
Counter foo;
```

the default constructor is called, because there is no information provided about how to initialize the class object `foo`.

And with this code:

```
Counter bar(16);
```

the compiler sets up a call to a constructor by matching argument types—it looks for a constructor that takes one argument, of type `int`.

Syntax warning

This might look like an attempt to create a class object and initialize it with a default constructor:

```
Counter quux(); // Doesn't do what you might think!
```

But in fact it's a prototype for a function called `quux` that takes no arguments and returns a `Counter` class object. To create `quux` and initialize it using the `Counter` default constructor, you should *not have any argument list, not even an empty one*:

```
Counter quux;
```

Other member function definitions for the Counter class

Here are the two definitions for the “normal” member functions of `Counter`—the functions that are not constructors—along with some annotations ...

```

    ↙ says that this is a member of the Counter class
void Counter::increment() ← const does NOT appear,
{                               matching the function prototype
    countM++; ← statement accesses and modifies a member
               variable of a class object
}

```

```

    ↙ says that this is a member of the Counter class
int Counter::count() const ← const appears here,
{                               matching the function prototype
    return countM; ← statement accesses but does NOT modify
                   a member variable of a class object
}

```

const and non-const member functions

A member function is declared to be **const** by putting the keyword **const** *immediately after the right parenthesis of the argument list* in both the function prototype and the function definition.

If you look at `Counter.h` and `Counter.cpp` you should be able to see that `count` has been properly declared as a **const** member function of `Counter`. [CORRECTION THU OCT 27, 2005: The original version of this handout said that `increment` was a **const** member function. That's not true; I am sorry for the confusion.]

const member functions are *not* allowed to modify values of member variables, but **non-const** member functions *are* allowed to do that.

When designing a class type in C++, it is important to decide which member functions should be **const** and which should be **non-const**, according to the following principle:

- A call to a **const** member function is a *question*, a request for information about the state of a class object.
- A call to a **non-const** member function is a *command*, a message telling a class object to modify its state.

Compilers enforce “const correctness”

Consider this example:

```
#include "Counter.h"
```

```

void func(const Counter& arg)
{
    int i;
    i = arg.count();           // statement 1
    arg.increment();          // statement 2
}

```

Statement 1 is fine, but statement 2 is not allowed. `func` must treat the referent of `arg` as a **const** object, which means that `func` can't call `increment` on that object. The function definition will be rejected by a C++ compiler.

By the way, the compiler error message might not be super-clear about what the problem is. For example, using `g++ 3.4.3`, which is what is installed in the labs in Fall 2005, the error message for the above code is

```

break-const.cpp: In function 'void func(const Counter&)':
break-const.cpp:7: error: passing 'const Counter' as 'this'
argument of 'void Counter::increment()' discards qualifiers

```

with the plain `g++` command; with `g++339` the error message changes to

```

break-const.cpp: In function 'void func(const Counter&)':
break-const.cpp:7: error: no matching function for call to
'Counter::increment() const'
Counter.h:8: note: candidates are: void Counter::increment()
<near match>

```

Neither of these messages is very helpful to a C++ beginner.

Access control: public and private

- **public** members are available for use anywhere in a program.
- **private** members can only be accessed by member functions of the same class.

Consider this example:

```

#include <iostream>
using namespace std;
#include "Counter.h"

int main()
{
    Counter foo;
}

```

```

    cout << "foo's count is " << foo.countM << endl;
    return 0;
}

```

`main` is not a member of `Counter`, and `countM` is a private member of `Counter`, so the compiler will not allow `main` to use the expression `foo.countM`. In this situation, the `g++` error message is pretty good:

```

access-control.cpp: In function 'int main()':
Counter.h:11: error: 'int Counter::countM' is private
access-control.cpp:8: error: within this context

```

(Line 8 in `access-control.cpp` is the `cout` statement in `main`.)

If the expression `foo.countM` were changed to `foo.count()`, the program would be acceptable to the compiler, because `count` is a public member of `Counter`.

Access control and object-based design

In object-based design, a program interacts with a class object by *sending messages* to the object, and the object responds by *performing operations*. This general concept applies to programming in all languages that have class types or their equivalents. In C++ “sending a message” is done by means of a calling a member function, and “performing an operation” is done by running the body of a member function definition.

When you design a class, first think, “What operations should objects of this class be able to perform?” Do *not* start by asking, “What should the member variables be?” After you have decided on the set of operations, decide what member variables are needed to support all the operations.

Operations should be *public* member functions, so that a program using a class can request class objects to perform operations. Member variables should be *private*, because code outside of a class should not need to directly access the pieces of data a class object uses to support the operations the class object offers.

Access control works on a class-by-class basis

It does not work on an object-by-object basis. To understand what I mean by this, consider the following code fragments. Here is a sketch of a class definition ...

```

class Alpha {
public:
    Alpha();
    // ... more ctors ...
    void beta(const Alpha& gamma);

```

```

    // ... more member functions ...
private:
    int deltaM;
};

```

...and here is a member function definition ...

```

void Alpha::beta(const Alpha& gamma)
{
    deltaM = gamma.deltaM + 2;
}

```

The member function is called in this code fragment:

```

Alpha x, y;
// ... code that works on x and y

x.beta(y);

```

When `beta` is running, it will access the `deltaM` members of *two different objects*: `x` and `y`. This is perfectly acceptable according to the rules of access control—a member function can access all private members of all class objects of that function’s class, so `beta` is allowed to access the `deltaM` member of any class object of type `Alpha`.