

* Fundamentals of Classes.

The last session was designed to give you a rapid tour through classes, and expose you to some of the nice features of Object Oriented programming. Now, we're going to go over classes in a more thorough and disciplined way.

Keep your project directory. You're going to use it throughout the rest of the seminar.

* Definitions. Start by defining a few terms.

Class: a class is a new data type, defined by the programmer.

Object: an object is a variable which has a class type.

Example:

```
istack a;
```

istack is a **class**.

a is an **object** of class istack.

a is also said to be an **instance** of class istack.

The act of creating an object is also referred to as **instantiation**.

The functions defined with a class are called **member functions**.

The data within a class is called **member data**.

When one invokes a member function, e.g:

```
a.pop( );
```

He/she is said to be "**sending a message to the object a**".

Member functions and member data comes in two varieties:

- * public -- visible from clients.

- * private -- only visible from within member functions.

- * (a third variety, "protected" will be discussed later)

So, there are three kinds of scope: local scope, global scope, and class scope.

- * Inside the class declaration, one may alternate data declarations and member function declarations.

- * The keywords

```
public:  
private:
```

may be intermixed with the declarations. Each qualifier means that following data is "public" or "private".

- * It is generally a good idea to never allow client code to see any member data.

E.G. in a vector class:

```
vector v;
```

```
float x=v.x; // a bad idea!
```

```
float v=v.x() // a good idea!
```

Why? Because the first method ties client code to a particular representation. The second would allow the class to change without bothering the client.

E.G.

If the class was represented by cartesian coordinates:

```
double vector::x() const {  
    return _x;  
}
```

If it changes to spherical polar coordinates:

```
double vector::x() const {  
    return _r*sin(_theta)*cos(_phi);  
}
```

Nobody who uses the vector will ever know the difference.

You ask, "how important is this, really?" In practice it is **very** important. In a large project, it is extremely annoying when the interface changes, but causes no trouble whatsoever if the implementation changes.

*private member data is only visible from within member functions.

*private member functions can only be called from within member functions.

* In the implementation, the scope operator (`stack::`) is used to specify that the functions are members functions and to specify the class. (This is one of its many uses).

- * Most of what you learned about ordinary functions applies to member functions, except that these functions are bound to a particular instance of a class.
- * Member functions like ordinary functions can be overloaded.
- * One could imagine a method `init()` which would be called to initialize an object; i.e., to put it in a well defined state.
- * However, you might forget to call it.
- * C++ has a better way. This is called a constructor.
- * Constructors are called every time a variable is created.
- * If you don't provide any constructors, the compiler makes one for you!
- * It may not be the one you want.
- * For example, in the stack class, the count variable would come up with garbage if it is not properly initialized.
- * A constructor has the following syntax:

```
class XXX {  
    XXX(parameter-list);  
}
```

- * Because of function overloading, more than one constructor may appear.
- * Some of the parameters may have default values.
- * The compiler tries to match the parameter list.

* Overloaded constructors allow for different ways of constructing objects. Example:

```
class complex {  
public:  
    // construct with no arguments (default constructor)  
    complex();  
  
    // construct a complex number from a real number  
    complex(double x);  
  
    // construct with real and imaginary parts.  
    complex( double a, double b);  
};
```

* This gives three ways of constructing a vector

```
complex a;           // calls the first constructor.  
complex b(2.0);     // calls the second constructor  
complex c(1.0,-1.0); // calls the third constructor.
```

* The compiler tries to resolve the right constructor.

* The constructor is used in initialization, *not in assignment!*

* This makes a very big difference in user-defined classes.

* The following syntax is also allowed:

```
complex b=2.0;      // initialization. compiler  
                   // resolves second constructor
```

* This is also allowed

```
complex c = complex(1.0,-1.0); // initialization. third  
                               // constructor.
```

* This is also allowed

```
complex d(c)        // what constructor  
complex e=b;       // is this?? Answer: it's  
                   // a compiler-generated  
                   // "copy constructor"
```

Consider the body of a constructor. Lets take for example:

```
// construct with real and imaginary parts.  
complex( double a, double b);
```

Here is one possibility for the implementation of this constructor:

```
complex::complex(double a, double b) {  
    _real = a;  
    _imag = b;  
}
```

* This assigns value to the member data. This is not always possible.

* Some kinds of member data may be const. You cannot assign constants.

* You may however initialize them.

* Variable initialization can happen **only** in a constructor.

* How do you do this?

* Remember this unusual syntax:

```
complex::complex(double a, double b) : _real(a),_imag(b) {  
}
```

* You should prefer this kind of initialization to assignment.
(Meyers, Item 12)

- * The compiler can generate constructors.
- * If no constructors are given, it will generate a default constructor (a constructor with no arguments).
- * If no copy constructor is given, it will generate a default copy constructor.
- * The default copy constructor copies each piece of old member data into the newly created object
- * This is usually appropriate.
- * But under certain circumstances, it is not. We will return to these later.
- * To write your own copy constructor, declare it as follows:

```
class XXX {  
.  
.  
.  
public:  
    // copy constructor  
    XXX(const XXX & right);  
.  
.  
};
```

Exercises:

* Edit the program `example.cc` by adding the following line at the end:

```
istack c=b.  
c.print();
```

* Question: what is the most important difference between the statements lines:

`a=b;` `and istack c=b;` `?`

* Make your own copy constructor by following the syntax on the previous page. Make sure that the statement `c.print()` continues to work as before.

* Make your copy constructor print a message when it is called.

- * The compiler generates a copy constructor on its own initiative.
- * In certain circumstances, the copy constructor will be automatically called.
- * You should be aware of when this is happening.
- * It happens when a object is passed to a function by value.
- * It happens when a function returns an object.
- * Let's write a function that writes a "sawtooth" pattern into a stack:

```

istack sawtooth( istack s) {
    int nTeeth= 5; // number of teeth in the sawtooth
    int nSteps = 10; // number of steps per teeth
    for (int iTooth=0;iTooth<nTeeth;iTooth++) {
        for (int iStep=0;iStep<10;iStep++) {
            s.push(iStep);
        }
    }
    return s;
}

```

Excercise.

Make a test program that:

- * creates an istack.
- * fills it with a sawtooth.
- * how many times is the copy constructor called?
- * performing a memory-to-memory copy of an istack is expensive. If you haven't changed the dimension `_ISTACK_MAX`, each istack has over 10,000 integers or 40 kBytes.
- * Now, modify the sawtooth program so that it passes its parameter by reference, modifies the the contents of the stack parameter, and has return type `void`.

- * In some cases (like the last exercise), objects should never be copied.
- * If an object is too large, it may be inefficient to copy it.
- * Some objects (like cout) should simply not have multiple copies.
- * You can make it **illegal** to copy objects.
- * Simply make the copy constructor **private**.

Destructors.

- * Another member function that is generated automatically by the compiler, for each class, is the **destructor**.
- * The reason for the destructor is to clean up any dynamically allocated memory that the class may have created.
- * We will discuss this in detail at a later time.
- * Until we discuss dynamically allocated memory, the destructor does nothing.
- * The destructor has a special syntax.

```
class XXX {
.
.
public:
    // destructor
    ~XXX();
.
};
```

- * It takes no arguments and cannot be overloaded.
- * It is called automatically when a variable goes out of scope.
- * **Exercise:** write a destructor for istack that prints farewell messages from doomed istack objects.

Operator overloading

* Operator overloading lets you define arithmetic operations on your own classes.

* A large number of operators can be overloaded.

* Here are a few of them:

Binary operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
^	Bitwise exclusive OR
&	Bitwise exclusive AND
	Bitwise OR
<<	Left shift
>>	Right shift

Shortcuts

+=	increase
-=	decrease
*=	multiply by
/=	divide by
%=	remainder
^=	exclusive OR into
&=	AND into
=	OR into
<<=	shift left
>>=	shift right

Relational operators

==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Increment and decrement

++i	preincrement
i++	postincrement
--i	predecrement
i--	postdecrement

Unary operators

+	positive
-	negative
*	deference
&	address-of
~	ones complement

Logical operators

	OR
&&	AND
!	NOT
[]	Index
()	function call

cast operators e.g. double (); turns an XXX into a double.

* Use function overloading judiciously .E.G.

does it make sense to add one istack to another? Yes.

does it make sense to divide one istack by another? ...No.

overloaded operators should be used only if the operations is intuitive and unambiguous.

- * There are several ways to do this.
- * It takes some analysis and judgement to find the right one.
- * See Meyers, item 19 for a fuller discussion.
- * See Oualline chapter 18 for a completely contradictory point of view. (I happen to side with Meyers).

Way #1

- * Make a global function called operator+.
- * Its signature is

```
istack operator+ (const istack & a, const istack & b) {  
}
```

*You can call it in either of the following ways:

```
istack c=operator+(a,b); // not preferred, but works!  
istack c = a+b;         // intended syntax.
```

- * This function is cumbersome to implement.
- * We dont want to add b onto a in reverse order!
- * But we cannot see the private member data of object a or b!
- * An (inelegant) way around the problem is to declare this function a friend of the class.

Rap on "friends"

- * Private member data is only visible from within member functions.
- * Each class is, however, allowed to make exceptions to this rule.
- * A class can make a list of other classes and other functions that are allowed access to its private member data.
- * These are the friends of the class.
- * Friends look at private parts.

The syntax for this is:

```
friend istack operator+ (const istack & a, const istack & b);
```

To declare a friend class, the syntax is:

```
friend class className;
```

These declarations must occur **within the class declaration**.

Friend functions and friend classes should be used sparingly. If they proliferate, they break encapsulation.

Operators overloading is generally a sufficient reason.

Excercise. Implement and test an addition operator for the istack class.

There is another way of making an addition operator. You can define it within the class.

```
class istack {  
.  
.  
    istack operator+ (const istack & right) const;  
}
```

*note that this operator has only one argument.

*the member function is bound to the first operand.

*in other words, these two are completely equivalent

```
stack c=a.operator+(b);  
stack c=a+b;
```

Exercise: Now, try turning your global function (or friend function) into a member function.

Assignment.

* There is one operator that will be generator for you. In fact, you've already used the compiler-generated assignment operator.

* This compiler-generated operator performs a memberwise assignment from the right-hand side to the left-hand side.

* There are reasons (dynamically allocated memory) to write your own assignment operator rather than taking the compiler-generated assignment operator.

* We'll discuss these later; but now, we'll discuss how to write your own assignment operator.

```
class istack {  
.  
.  
    istack & operator= (const istack & right);  
}
```

* Remind yourself of what assignment should do:
-assign.
-return the thing that was assigned

(remember $a=b=c=(2+3)?$)

* performing the assignment is trivial.

* but what object does `operator=` assign?

* there is a special denomination used within a member function to denote "the object that I'm stuck onto right now".

* that denomination is `*this`;

* `operator=` should return a reference to that object.

* `operator =` should `return *this`;

* One more thing that operator = should do is to check for assignment to self, as in `b=b;`

* How do you do that?

```
istack & istack::operator= (const istack & right) {  
    if (this != &right) {  
        // carry out the assignment  
        .  
        .  
        .  
    }  
    // in any case, return reference to self:  
    return *this;  
}
```

* Exercise: write your own version of the assignment operator for the `istack` class. Arrange for it to print out an informational message when it is invoked.

* Exercise. The `+=` operator is an example of a unary operator. Like the assignment operator, it too should return a reference to itself. Implement a `+=` operator on the `istack` class.

Discussion: how to choose between member functions, global functions, and friend functions.

Conversions.

A constructor that takes a single argument can be used to perform a conversion.

Consider a class of complex numbers. Suppose that there is a constructor:

```
class complex {  
.  
// construct a complex number from a real number  
  complex(double x);  
}
```

Now what if there is a function that takes a complex number as a parameter?

```
double phase(complex u);
```

I call this function with a double precision constant

```
cout << phase(100.0) << endl;
```

The compiler finds a conversion from double precision → complex and is able to resolve the function call. One would expect 0.0 to be printed if phase was properly written.

* It is sometimes nice to have automatic conversions.

* It is sometimes very annoying, because it can happen at surprising times .. and do away with a lot of type-safety.

* You can turn it off by declaring the constructor explicit:

```
explicit complex (double x);
```

* Cast operators (which I'll skip) provide another method for automatic conversions. Same pros and cons...

`const` (as you have seen) is used to declare constant data.

```
const int MAX=100;
```

it is used to declare that parameters are constants:

```
double phase (const complex a);
```

it is used to promise that a function or member function won't modify the value of a parameter that it is passed by reference:

```
void print (const page & myPage);
```

or by pointer

```
void print (const page * page);
```

you can declare pointers to constant data:

```
const int *myIntegerThatImNeverEverGoingToModify.  
const int &anotherNameForSomethingIPromiseNotToTouch;
```

and, finally, you can declare that a member function won't change the value of the object:

In case of a stack:

```
a.print();
```

Should print a, not modify it.

```
a.pop() on the other hand modifies the stack.
```

A method is declared `const` by the following syntax:

```
void istack::print() const;
```

A non-constant method cannot be called from a constant method.

Use `const` wherever you can.

Inline member functions.

A member function call takes some overhead to perform. One can cut down on the overhead by making the function calls inline.

The meaning of this is, to expand the body of the function in the calling routine rather than actually creating a function, setting up the function call, and performing the function call.

The syntax for this is:

```
inline return-type function-name(parameters)
```

e.g.

```
inline void push(int i);
```

Both global functions and member functions may be inlined.

The implementation of the function needs to be available to all compilation units that include the header file. i.e. it needs to appear in the header file. It is customary to put these functions in a file called .icc and to include them at the end of the .hh file.

My advice is: inline a function **only** if you have proven that it is the cause of slow execution time.

Bad effects of inlining:

- * Longer compilation time.
- * Bigger executables (code bloat)
- * **Much** harder debugging