**C++ Activities and Lecture Notes – Chapters 14 and 15**

**Chapter 14 – Operator Overloading**

First we need a simple class to use for operator overloading. We’ll choose complex numbers, which have a real and an imaginary part.

*c = a + b i*

where *i* is the square root of negative one. Complex numbers are used throughout mathematics, science, and engineering; for example, a complete description of the AC that powers your home requires complex numbers.

For simplicity, we’ll break from convention and make the state variables public.

|  |
| --- |
| ComplexNumber |
| + real: double  + imaginary: double |
| + ComplexNumber() // default constructor, set both parts to zero  + ComplexNumber(double real, double imaginary)  …more stuff as we think of it |

[ Write code; ComplexNumber.h and ComplexNumberDemo.cpp ]

To add two complex numbers, we add the real parts and add the complex parts. Let’s write a function to accomplish this…

[ Add code… ]

OK, writing

c = ComplexNumbers.add(a,b);

or something like that is pretty awful. What we *really* want to write is

c = a + b;

To do so we need to overload operators. We’ll start with assignment.

**ASSIGNMENT OPERATOR**

The safe and correct way to overload the assignment operator is as follows:

ComplexNumber & operator= (const ComplexNumber & rhs){

// check for self-assignment

if (this = & rhs){ // self-assignment!

return \*this;

}

else{

// deallocate memory used by this

// allocate memory for rhs

// copy the values from rhs into this

real = rhs.real;

imaginary = rhs.imaginary;

}

// return \*this

return \*this;

}

In a class with constructors that allocate memory dynamically, you *must* check for self-assignment. If you don’t, you de-allocate the memory for the calling object, then copy from the right-hand side but rhs doesn’t exist anymore because you just de-allocated the memory.

Since our class doesn’t allocate memory dynamically ours is much simpler. Also, we get it for free anyway (in other words, a = b works just fine without overloading this operator).

Note that since we are returning a reference, which allows operator chaining:

a = b = c = d = e = f;

Also note that what we return is *not* a const reference, so we can write things like

(a = b) = c;

which is a little deranged but it is well-defined behavior for primitives so we ought to support it.

**COMPOUND ASSIGNMENT**

Compound assignment follows the same rules as assignment, with the addition of whatever you have to do to conduct the arithmetic operation.

[ write code ]

**BINARY ARITHMETIC OPERATORS**

Now we can implement (for example) binary addition. A great time-saver is to write compound assignment first, then code the binary operators using them. That is, if you can get it to work.

[ write code ]

**BINARY RELATIONAL OPERATORS**

Overloading == is relatively straightforward:

[ write code ]

And overloading != will be done in terms of the previously defined ==

[ write code ]

**STREAM OPERATION**

By this time we are rather tired of writing

cout << a.real << “ “ << a.imaginary << endl;

so let’s overload the stream operators << and >>. Unlike the arithmetic operations, these aren’t invoked by ComplexNumber objects on the left side of the operator, so we will add them as friend” functions, which isn’t discussed until a later chapter but we’ll do it anyway.

[ write code ]

With a bit of time we could implement pretty much every operator that works with (for example) ints and make them work with ComplexNumbers as well.

**Chapter 15 – Memory Management**

15.1 Categories of Memory

Code and Static Data pages in virtual memory

Run-time Stack default is 1 MB, can be changed at compile time

The sizes of data elements must be known at compile time

so that stack-pointer-relative addresses can be generated at that time.

int i;

double k;

might be stored in the symbol table as

i int SP - 01A0

k double SP - 01A4

In other words, the stack grows *down* in memory. This was a necessity back in the

day when memory was precious, and retained to maintain compatibility with older

programs.

Heap pages in virtual memory

It used to be that the heap grew up in memory (from the bottom) while the stack

grows down as described above. Now the “heap” is simply the virtual address

space of the system and the virtual memory manager runs it. When a program

requests new memory it’s address is whatever the vm manager had on hand at the

time.

Aside – static variables

[ write code ]

**C and C++ Memory Errors**

Using a value that has not been initialized.

The compiler doesn’t catch this, but preprocessors like LINT can. Visual Studio’s run-

time debugging will alert you if you attempt to use an uninitialized variable.

Using a pointer to reference a memory location that is no longer valid.

Array index out of bounds is a subset of this error. C and C++ don’t store array bounds,

and therefore you can access memory outside the array with an invalid index. This makes

C and C++ particularly vulnerable to “buffer overflow” attacks. Another is the “dangling

pointer” – a pointer to a value that has gone out of scope.

Using a pointer that was erroneously initialized.

Most C compilers won’t even allow pointers less than a certain value because a very

common programming error is to associate an integer with a pointer rather than the data

associated with the pointer.

Forgetting to delete dynamically-allocated memory, then re-using that pointer.

This is a memory leak. If it occurs in a loop, it will eventually consume all of the memory

resources of the computer.

Deleting allocated memory more than once.

Twice in a row isn’t bad. Once, then once again later, may deallocate memory that

actually belongs to something else.

Deleting memory that was never allocated.

Who knows what was actually there?

Note that *none* of these problems exist in Java or C#. Does this make Java and C# better languages than C and C++? The answer is (as always): It depends. The price for safety is performance. Which do you value more?

**Constructors**

The purpose of a constructor is to ensure that every allocated value is properly initialized.

One use of constructors we haven’t yet seen is type casting. Consider a constructor for ComplexNumber that accepts a single int and creates a ComplexNumber with that int as the real part and zero as the imaginary part…

[ write code ]

This constructor can also be used to type-cast an int to ComplexNumber, as in

… (ComplexNumber) k …

What if we want to cast a ComplexNumber to double?

[ write code ]

To cast anything to one of our user-written classes, the compiler looks for a single-argument constructor containing that data type as the parameter.

To cast one of our user-written classes to another type, we overload the operator (for the built-in types) or write the appropriate constructor in the other type.

**Destructors**

The purpose of a destructor is to ensure that every allocated value is properly deallocated.

This is only required if the constructor allocates memory dynamically.

SomeClass(){

double d = new double; // a dynamically allocated value

int \* a = new int[size]; // a dynamically allocated array

}

~SomeClass(){

delete d; // de-allocate the value

delete [] a; // de-allocate the array

}

To use the destructor, just “delete” the object.

SomeClass sc();

.

.

.

delete sc; // invokes our destructor