Course C^{++}, Explanation for Exercise 3

March 7, 2013

Topic of this exercise are what I call *the lifecycle operators* of a class. These are the *constructors*, *destructors*, and *copying assignment*.

- A *constructor* is a method with the same name as the class. Its task is to transform raw memory into a class object, which means that it has to make sure that the class invariants become true.

  In order to transform raw memory into a class object, the first thing that needs to be done is to make sure that for the fields, which at first still contain raw memory, the class invariants become true. This is done by the field initializers.

  If after field initialization something more needs to be done, this can be put in the body of the constructor. My experience is that this rarely happens.

```
class something
{
   aaaa a;
   bbbb b;

   something( par1, par2, par3, ... )
      : a( aaaa( pari, parj ))  // This is a field initializer.
                        // Since no initializer for b is present,
                        // it will be initialized by the default
                        // constructor of class bbbb.
   {
      // When this point is reached. All fields have already
      // been constructed. This is nearly always enough.

      b = bbbb( ... ); // I often see this in student code.
                // It is unacceptable C++ style.
                // Now b is first default initialized, and after
                // that overwritten by a second value.
   }
};
```

Since constructors are the guards of the class invariants, it is important that they are inserted everywhere, where a new class object is created. **The compiler does this for you**. If it cannot find a constructor definition, it will try to insert a default constructor. If no default constructor exists, the code will not compile.

- *Destructors* are another source of confusion. Their purpose is to transform a class object back into raw memory. If the class object holds any resources (heap memory, open files, or locks), the destructor must free these resources. This means that a destructor guards *global invariants*. It is essential that they are always called, when a class object goes out of scope. **The compiler does this for you**. I repeat, **you, as programmer do not call a destructor**. The compiler knows where they must be put.

```
class something
{
   aaaa a;
   bbbb b;

   ~something( )
   {
      // Write a destructor only when something
      // needs to be done before destroying the fields.
   }
      // Here destructors of a and b are called.
};
```

- A constructor with form `something( const something& s )` is called *copy constructor*. It does what its name suggests: copy `s` into raw memory to create another `something`.

  Since the compiler is obsessed with maintaining your class invariants, it will insert a copy constructor, whenever a parameter is copied in a function call, when a function returns a value, or when you write a statement of form `{ aaaa a = some_other_a; ... }`; If you don't write a copy constructor by yourself, the compiler will define one that copies the fields in the order of appearance in the class.

- Copying assignment has form `void operator = ( something a )` or `void operator = ( const something& a )`. The language allows other forms, but I am not in favour of using those.

  Copying assignment is used when an existing class object is overwritten by a new object. It should not be confused with the copy constructor. The copy constructor overwrites raw memory, while the assignment operator overwrites an existing class object.

This makes a difference when the existing class object holds resources. The assignment operator has to ensure that the class object that is being overwritten maintains its global invariants (returns all its resources). After that, it has to make sure that the copy maintains the class invariants.

One can view the assignment operator as a destructor, followed by a copy constructor, but don't ever write it like that.

I call these operators the life cycle operators, because they together control the life cycle of your objects. If you manage to get those right for every class that you write, then you are unlikely to have memory leaks, segmentation faults, running out of file handles, etc.