Object-Oriented Principles and Practice / C++

Alice E. Fischer

May 13, 2013

Construction and Destruction – Ch 10

Allocation and Deallocation: Review
Initializing Arrays
Six kinds of Constructors
But just one Destructor

Allocation and Deallocation – Ch 10.2

The new function in C++ builds upon C's malloc.

- Using new or malloc results in the storage you requested + enough to handle deallocation
- Minimally, the total allocated block length must be stored
- Thus, allocating a `new double` uses up 12 bytes of memory, 4 for the block length and 8 for the data.
- For a struct, it is the overhead + enough bytes to hold the data + padding.
- New and malloc differ for arrays because new also allocates memory for the array length. (A total of 8 bytes of overhead)

Initialization

C++11 gives us new ways to initialize arrays.

```
class Tester {
  private:
    static int a0[3];
    static constexpr int a1[3]{1,2,3};
    int a2[3] = {8};
    int a3[3];
    int* a4;
    int* a5 = new int[3]{5,6,7};
  public:
    Tester(): a3(), a4(new int[3]{7,8,9})  {}
}
const int Tester::a1[3];
      int Tester::a0[3]{4,5,6};
```

Six Kinds of Constructors

- Null constructor: initializes nothing but may print a trace comment. You get an empty null constructor for free if you do not define a real constructor.
- Default constructor: may do initializations but lacks parameters.
- Normal constructor: has parameters used to initialize the object.
- Default copy constructor: initializes an object by shallow-copying another.
- Copy constructor: replaces the default copy constructor. It can do anything, including making a full copy of the object.
- Move constructor: moves the contents of one object to another during initialization.

Several Constructors but just one Destructor

- A destructor must handle every class object the same way. Thus, every object must be the same general shape and have the same number of dynamic parts (or NULL pointers) to delete.
- According to the standard, it is harmless to delete a NULL pointer. Nothing is supposed to happen. You do not need to test a pointer before trying to delete it. However, apparently, some implementations of C++ do not comply to the standard.
- Normally, a destructor should free all objects that were created by calling `new` in the constructor or in any other class function, or by inserting an object (created elsewhere) into a class data structure.

Revisiting delete and delete[]

- For non-arrays, the block length is stored just before the data part (just like malloc)
- For arrays, the array length is stored before beginning of the data area and the block length is stored just before the array length.
- When you call `delete` it uses the block length to free memory.
- When you call `delete[]` it does two things:
  - First, the array length is used to control a loop that calls delete on each array element.
  - Then the block length is used to free the array allocation.

Construction and Destruction

RValue References
Default Class Operations: C++ 11
Control of Defaults: C++ 11

RValue References

An rvalue reference can bind to an rvalue (but not to an lvalue):

- ```
  X a;
  X& r1 = a;        // bind r1 to a (an lvalue)
  X&& rr1 = f();    // bind rr1 to the result of f().
  X&& rr2 = a;      // error: a is an lvalue
  ```
- This idea can be used to speed up execution of any operation that moves values around.
- If X is a type for which copying is expensive (string, vector) a simple swap becomes an expensive operation.
  ```

RValue Swapping

But when we swap, we don't really want new copies at all. We just want to rebind the existing copies.

- ```
  void swap(T& a, T& b) {
  T tmp = move(a);        // Save a's value.
  a = move(b);            // Move b's value to a.
  b = move(tmp);          // Move the saved value to b.
  ```
- Moving is faster than copying for many types because it does not construct a new object.
- This is very important when you design a template because the template parameter could be any primitive or class type.
  ```

Default Class Operations: C++ 11

By default, a class has these five related operations:

- destructor: By default, a null destructor.
- copy constructor: `X(const X&)`
  Initialize a new object from an old one of the same type. The contents of the source are unchanged.
- copy assignment: `Y& operator=(const Y&)`
  Copy one object into another of the same type. The contents of the source are unchanged.
- move constructor: `X(const X&&)`
  Initialize a new object by moving the contents of an old object into it. After the move, the contents of the source are set to the initial constructed state.
- move assignment: `Y& operator=(Y&&);`
  Move one object into another.

Default Class Operations: C++ 11

Copy, move, and delete are closely related operations.

- You can redefine all of them,
- but only a few combinations make sense.
- If you declare any of them you must explicitly define or default all the others.
- If you define any one, movers will not be generated automatically. Copiers will be generated automatically, but this is deprecated.
- Move constructor and move assignment takes non-const &&. They can, and usually do, write to their argument

Control of Defaults: C++ 11

The keywords delete and default and can be used to define methods.

- ```
  class X {
      // These definitions disallow copying.
      X& operator=(const X&) = delete;
      X(const X&) = delete;
  };
  ```
- ```
  class X {
      // These define the default copy behavior.
      X& operator=(const X&) = default;
      X(const X&) = default;
  };
  ```

Template Classes and Instantiation

Code with a Hole
Template Instantiation
Derive and Instantiate
Overriding functions

A Template is Code with Holes.

Chapter 13

- A template is a class definition with parameters.
- A template is NOT a class and cannot be used directly.
- Most templates have one type-name parameter; some have two. Integer parameters are also possible.
- `template class NewTemp < class T > {...};`
- Within the class: `bool compareTo(T data) {...};`
- Outside the class braces:

```
template <class T> ostream&
NewTemp<T>::print(ostream& out) {...};
```

A Note about Templates

Given a template definition, there are three things you can do with it:

- Instantiate it to get a normal class.
- Derive from it to get another template.
- Derive and instantiate in the same step to get a class to which you can add functionality and/or data members.

From Template to Class.

- A class is constructed from a template by *instantiation*.
- `Stack<int>` instantiates the Stack template with type int to produce a class named `Stack<int>`. The instantiation is done at compile time, before the class can be compiled.
- The compiler will insert the type `int` everywhere the template code says `T`, then compile the resulting class.
- If we later instantiate with `double`, the resulting class is different and must also be compiled.
- We can instantiate a template and derive from it in the same line: `class myStack : public Stack<int>`
- Derivation + instantiation is a common technique because we often want to modify the basic type provided by the library templates.

Template problems

A template is an abstract data structure that will be instantiated with a specific base type at compile time.

- To be widely useful, the template should work with many or all possible base types. Functions called by the template must be defined for all instantiation-base types.
- This is not a problem for primitive numeric, character, and pointer types, since all of the common operations are defined for them: comparisons, input, output.
- If a class type is used to instantiate a template, then the class must define all the operators and functions used by the template functions.
- If the template makes copies of the base-type data, and if that base type has dynamic extensions great care must be taken to avoid shallow-copy followed by deep-delete.

Move Semantics

What we really want to do is move all of the data from one array to another, not create a duplicate copy of the data. We certainly do not want to create deep copies!

The latest revision of C++ allows us to define a move constructor for each class that is different from the copy constructor.

- When we deep-copy an object, all of its core parts and all its extensions are duplicated. This consumes time and space.
- A shallow copy duplicates the core parts but not the extensions, and two copies of the core object end up pointing at the same extensions.
- When we move an object, we want all parts of the object in the new location, and we want any pointers in the old location nulled-out.

The Stack Template

Two Versions

To define any linked data structure, we need a master class and a helper class. (e.g. List and Cell)

- The helper class should be controlled, managed, and used exclusively by its master class.
- The master class should have full access to helper instances, but they should otherwise be hidden from the world.
- There are two ways to achieve this: use a friend declaration and use a private inner class. See stack template examples 1 and 2.

Deriving a Template from a Template

The Stack template is derived from a FlexArray template.

- The FlexArray provides the growable array that implements the stack.
- There are some difficulties when a template is derived from a template.
- When the derived template is first parsed by the compiler, the base template has not yet been expanded, so the members of the base class are not yet visible to the compiler.
- Fix this by using `this->` in front of references to base class members, or by using the full name of the base class with ::

Allocation and Deallocation Example – Ch 10.2

Box and Van

The Van Class

The Output

Box and Van

This program illustrates allocation and deallocation semantics.

- Box has two constructors: with parameters and without parameters.
- They could be combined by using default parameters:

```
Box(int ln=1, int wd=1, int ht=1){
    length=ln; width=wd; high=ht;
    cout<<"\n Real Box ";
}
```

- The destructor prints trace information but does not deallocate anything. This is appropriate because the constructor does not allocate any new objects.
- Note the dump function: it backs off from the head of an array and prints the "secret" data.

The Van Class

This class aggregates several Boxes, stored as two arrays.

- Lines 60–61: the van constructs two arrays of default boxes using the default constructor. These will be overlaid by real boxes later.
- Line 68 allocates a Box in temporary memory. This Box is deallocated on line 69, immediately after its contents is copied into the box array, Load1.
- Construction, assignment, and deallocation of the original copy happen every time around the loop. You can follow the progress of this process using the trace information printed by the Box destructor.
- Again, note the dump function: it prints the address and contents of each part of the Van.

The Output

Looking at the output and the trace we see:

- 7 default boxes were constructed by lines 60 and 61.
- Then the loop on lines 64...69 is executed four times. Three sets of real dimensions are entered, followed by 0 dimensions to end the loop.
- For each set of real dimensions, line 68 creates a temporary box and copies it into the Load.
- At the end of the loop (line 69) the temporary box is deleted.
- The van dump shows that copies of the temporary boxes live on in Load1.
- After termination, Load2 (which was created last) is deallocated first. Two default boxes are deleted.
- Then Load1 is deleted, starting with the highest subscript and ending with the first real box.