Object-Oriented Principles and Practice / C++

Alice E. Fischer

October 10, 2016
Construction and Destruction

Kinds of Constructors

Move Semantics
  RValue References
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.
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.
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.

```cpp
▶ 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;
};
```
RValue References

An rvalue reference can bind to an rvalue (but not to an lvalue):
Classname && rref = &myClassInstance;

Refer to the Brackets program for examples of all these things.
RValue Swapping

When we swap, we don’t really want new copies at all. We just want to rebind the existing copies.

```cpp
template<typename T>
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.
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.