CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 25
May 3, 2016

Extended Objects, Custody, and the Double Delete Problem

New Features of C++ 11

Extended Objects, Custody, and the Double Delete Problem

Moving data

Much of computation involves moving data around.

Almost all non-functional languages support the assignment operator.

Assignment **copies** a value from the right hand side (RHS) to the variable on the left hand side (LHS).

After the assignment, there are two copies of the RHS value.

Temporaries

Conceptually, a **pure** value is a disembodied piece of information floating in space.

In reality, values always exist somewhere—in variables or in temporary registers.

Languages such as Java distinguish between **primitive values** like characters and numbers that can live on the stack, and **object values** that live in permanent storage and can only be accessed via pointers.

Custody

Primitive values are relatively small. Objects can be large. Objects with dynamic extensions can be huge, if the size of the extension is taken into account.

Copying large objects is expensive!

How to avoid copying large objects

Java approach: Assignment copies pointers, not the underlying object.

To make a copy of the object itself, one creates a new object and initializes it from the old.

Problem of **custody** arises: Who is responsible for managing and eventually freeing the storage occupied by large objects and their dynamic extensions?

Java approach

In Java, the custody problem is avoided by letting the system own all objects and relying on the garbage collector to decide when an object may be deleted.

This entails a huge performance penalty. Two options, both expensive:

- Traditional garbage collector runs when needed, causing large unpredictable latencies.
- Concurrent garbage collector runs continuously in another core, consuming a fraction of the entire CPU for this one purpose.

Original C++ approach

The C++ approach is to have the class manage its own dynamic storage.

The constructor allocates the storage; the destructor deletes it.

The programmer is responsible for ensuring that the storage management is done properly.

Double delete problem

How to do assignment—two approaches:

- **Shallow copy:** Copy the data members, not the dynamic extensions. This results in each copy **sharing** the same extension.

  When any copy is deleted, its destructor deletes the extension.

  If there are multiple copies, the extension is deleted multiple times, leading to memory management errors. This is called the **double delete problem**.

- **Deep copy:** Copy both the data members and the dynamic extensions.

  Avoids the double delete problem but at great cost.

Move semantics

C++ 11 introduces **move** semantics.

An object can be moved instead of copied. The data in the source object is removed from the source object and placed in the target object. The source object then becomes *empty*.

In the case of dynamic extensions, the pointer to the extension is copied from source to target, and the source pointer is set to `nullptr`.

Deleting `nullptr` is a no-op and causes no problems.

We say that **custody** has been transferred from source to target.

Motivation

A big motivation for move semantics comes from containers such as `vector`.

Containers need to be able to move objects around. Old-style containers can't work with dynamic extensions.

C++ containers support moving an object into or out of the container.

While in the container, the container has custody of the object.

Move is like a shallow copy, but it avoids the double-delete problem.

Implementation in C++

Here are the changes to C++ that enable move semantics.

1. The type system has been extended to include **rvalue references**. These are denoted by double ampersand, e.g., `int&&`.
2. Results in temporaries are marked as having rvalue reference type.
3. A class has now six required functions: constructor, destructor, copy constructor, copy assignment, move constructor, move assignment.

Move and copy constructors and assignment operators

Copy and move *constructors* are distinguished by their prototypes.

`class A`:

- *Copy constructor:* `A( const A& other ) { ...}`
- *Move constructor:* `A( A&& other ) { ...}`

Similarly, copy and move *assignment operators* have dfiferent prototypes.

`class A`:

- *Copy assignment:* `A& operator=( const A& other ) { ...}`
- *Move assignment:* `A& operator=( A&& other ) { ...}`

Default constructors and assignment operators

Under some conditions, the system will automatically create default move and copy constructors and assignment operators.

The default copy constructors and copy assignment operators do a shallow copy. Data members that are objects of type `T` are copied using the copy constructor/assignment operator defined for class `T`.

The default move constructors and move assignment operators do a shallow copy. Data members that are objects of type `T` are moved using the move constructor/assignment operator defined for class `T`.

Default definitions can be specified or inhibited by use of the keywords `=default` or `=delete`.

Moving from a temporary object

A mutable temporary object always has rvalue reference type.

Thus, the following code *moves* the temporary string created by the on-the-fly constructor `string("cat")` into the vector `v`:

```
#include <string>
#include <vector>
vector<string> v;
v.push_back( string("cat") );
```

Forcing a move from a non-temporary object

The function `std::move()` in the `utility` library can be used to force a move from a non-temporary object.

The following code *moves* the string in `s` into the vector `v`. After the move, `s` contains the null string.

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
vector<string> v;
string s;
cin >> s;
v.push_back( move(s) );
```

The full story

I've covered the most common uses for rvalue references, but there are many subtle points about how defaults work and what happens in unusual cases.

A good reference for further information is *Move semantics and rvalue references in C++11* by Alex Allain.

New Features of C++ 11

C++ 11 extensions

There are many other significant new features in C++ 11 besides move semantics. Here are some of the more important ones:

- Lambda expressions
- Automatic type deduction and decltype
- Initialization syntax
- Deleted and defaulted functions
- nullptr
- Delegating constructors
- C++11 standard library: threading, smart pointers, algorithms
- Range-based for loops

See *The Biggest Changes in C++11 (and Why You Should Care)* by Danny Kalev for a discussion and examples.