CS100 Lecture 17

Classes IV

Contents

Motivation: Copy is slow.

```
std::string a = some_value(), b = some_other_value();
std::string s;
s = a;
s = a + b;
```

Consider the two assignments: `s = a` and `s = a + b`.

How is `s = a + b` evaluated?

Motivation: Copy is slow.

```
s = a + b;
```

1. Evaluate `a + b` and store the result in a temporary object, say `tmp`.
2. Perform the assignment `s = tmp`.
3. The temporary object `tmp` is no longer needed, hence destroyed by its destructor.

Can we make this faster?

Motivation: Copy is slow.

```
s = a + b;
```

1. Evaluate `a + b` and store the result in a temporary object, say `tmp`.
2. Perform the assignment `s = tmp`.
3. The temporary object `tmp` is no longer needed, hence destroyed by its destructor.

Can we make this faster?

- The assignment `s = tmp` is done by **copying** the contents of `tmp`?
- But `tmp` is about to "die"! Why can't we just *steal* the contents from it (or *move* the contents from `tmp` to `s`)?

Motivation: Copy is slow.

Let's look at the other assignment:

```
s = a;
```

- **Copy** is necessary here, because `a` lives long. It is not destroyed immediately after this statement is executed.
- You cannot just "steal" the contents from `a`. The contents of `a` must be preserved.

Distinguish between the different kinds of assignments

```
s = a;
```

```
s = a + b;
```

What is the key difference between them?

- `s = a` is an assignment from an **lvalue**,
- while `s = a + b` is an assignment from an **rvalue**.

If we only have the copy assignment operator, there is no way of distinguishing them.

We want to invoke a **copy** operation for `s = a`, and a **move** operation for `s = a + b`.

Define two different assignment operators, one accepting an lvalue and the other accepting an rvalue

Rvalue references

A kind of reference that is bound to **rvalues**:

```
int &r = 42;              // Error: Lvalue reference cannot be bound to rvalue.
int &&rr = 42;            // Correct: `rr` is an rvalue reference.
const int &cr = 42;       // Also correct:
                          // Lvalue reference-to-const can be bound to rvalue.
const int &&crr = 42;     // Correct

int i = 42;
int &&rr2 = i;            // Error: Rvalue reference cannot be bound to lvalue.
int &r2 = i * 42;         // Error: Lvalue reference cannot be bound to rvalue.
const int &cr2 = i * 42; // Correct
int &&rr3 = i * 42;       // Correct
```

- Lvalue references (to non-`const`) can only be bound to lvalues.
- Rvalue references can only be bound to rvalues.

Overload resolution

Such overloading is allowed:

```
void fun(const std::string &);
void fun(std::string &&);
```

- `fun(s1 + s2)` matches `fun(std::string &&)`, because `s1 + s2` is an rvalue.
- `fun(s)` matches `fun(const std::string &)`, because `s` is an lvalue.
- Note that if `fun(std::string &&)` does not exist, `fun(s1 + s2)` also matches `fun(const std::string &)`.

We will see how this kind of overloading benefit us soon.

Move Operations

Overview

The **move constructor** and the **move assignment operator**.

```
class Widget {
public:
  Widget(Widget &&) noexcept;
  Widget &operator=(Widget &&) noexcept;

  // Compared to the copy constructor and the copy assignment operator:
  Widget(const Widget &);
  Widget &operator=(const Widget &);
};
```

- Parameter type is **rvalue reference**, instead of lvalue reference-to-`const`.
- **`noexcept` is (almost always) necessary!** $\Rightarrow$ We will talk about it in later lectures.

Move constructor

Take the `Dynarray` as an example.

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;

public:
  Dynarray(const Dynarray &other) // copy constructor
    : m_storage(new int[other.m_length]), m_length(other.m_length) {
    for (std::size_t i = 0; i != m_length; ++i)
      m_storage[i] = other.m_storage[i];
  }
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

Move constructor

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;

public:
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {


  }
};
```

1. *Steal* the resources of `other`, instead of making a copy.

Move constructor

```cpp
class Dynarray {
  int *m_storage;
  std::size_t m_length;

public:
  Dynarray(Dynarray &&other) noexcept // move constructor
    : m_storage(other.m_storage), m_length(other.m_length) {
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

1. *Steal* the resources of `other`, instead of making a copy.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

*** Take ownership of `other`'s resources!**

Move assignment operator

Take ownership of `other`'s resources!

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {


      m_storage = other.m_storage; m_length = other.m_length;


    return *this;
  }
};
```

1. *Steal* the resources from `other`.

Move assignment operator

```cpp
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {


      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;

    return *this;
  }
};
```

1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

Are we done?

Move assignment operator

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {

      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;

    return *this;
  }
};
```

0. **Avoid memory leaks!**
1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

Are we done?

Move assignment operator

```
class Dynarray {
public:
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      other.m_storage = nullptr; other.m_length = 0;
    }
    return *this;
  }
};
```

0. **Avoid memory leaks!**
1. *Steal* the resources from `other`.
2. Make sure `other` is in a valid state, so that it can be safely destroyed.

* **Self-assignment safe!**

Lvalues are copied; Rvalues are moved

Suppose we have a function that concatenates two `Dynarray`s:

```
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());
  for (std::size_t i = 0; i != a.size(); ++i)
    result.at(i) = a.at(i);
  for (std::size_t i = 0; i != b.size(); ++i)
    result.at(a.size() + i) = b.at(i);
  return result;
}
```

Which assignment operator should be called?

```
a = concat(b, c);
```

Lvalues are copied; Rvalues are moved

Lvalues are copied; rvalues are moved ...

```
a = concat(b, c); // call move assignment operator,
                  // because `concat(b, c)` is a temporary (rvalue).
a = b; // call copy assignment operator
```

Lvalues are copied; Rvalues are moved

Lvalues are copied; rvalues are moved ...

```
a = concat(b, c); // calls move assignment operator,
                  // because `concat(b, c)` generates an rvalue.
a = b; // copy assignment operator
```

... but rvalues are copied if there is no move operation.

```
// If Dynarray has no move assignment operator, this is a copy assignment.
a = concat(b, c)
```

Synthesized move operations

Like copy operations, we can use `=default;` to ask the compiler to synthesize a move operation that has the default behaviors.

```
class X {
public:
  X(X &&) = default;
  X &operator=(X &&) = default;
};
```

- The synthesized move operations call the move operations of each data member in the order in which they are declared.
- The synthesized move operations are `noexcept`.

Move operations can also be deleted by `=delete;`. [1]

The rule of five

Five special member functions:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator
- destructor

If one of them has a user-provided version, the class has some resources to manage, requiring the special behaviors of all the functions and thus their user-provided versions (Recall "the rule of three").

The rule of five

Five special member functions:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator
- destructor

Define zero or five of them.

How to invoke a move operation?

Suppose we give our `Dynarray` a label:

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
};
```

The move assignment operator of `Dynarray` should invoke the **move assignment operator** of `std::string` on `m_label`. But how?

```
m_label = other.m_label; // call copy assignment operator,
                         // because `other.m_label` is an lvalue.
```

`std::move`

`std::move`

Defined in `<utility>`

`std::move(x)` performs an **lvalue to rvalue cast**:

```
int ival = 42;
int &&rref = ival; // Error
int &&rref2 = std::move(ival); // Correct
```

Calling `std::move(x)` tells the compiler that:

- `x` is an lvalue,
- but we want to treat `x` as an **rvalue**.

`std::move`

`std::move(x)` indicates that we want to treat `x` as an **rvalue**, which means that `x` will be *moved from*.

The call to `std::move` **promises** that we do not intend to use `x` again,

- except to assign to it or to destroy it.

A call to `std::move` is usually followed by a call to some function that moves the object, after which **we cannot make any assumption about the value of the moved-from object.**

```
void fun(X &&x);       // move `x`
void fun(const X &x); // copy `x`
fun(std::move(x)); // match `fun(X&&)`, so that `x` is moved.
```

"`std::move` does not *move* anything. It just makes a *promise*."

Use `std::move`

Suppose we give every `Dynarray` a special "label", which is a string.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(std::move(other.m_label)) { // !!
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

Use `std::move`

Suppose we give every `Dynarray` a special "label", which is a string.

```
class Dynarray {
  int *m_storage;
  std::size_t m_length;
  std::string m_label;
public:
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_storage = other.m_storage; m_length = other.m_length;
      m_label = std::move(other.m_label);
      other.m_storage = nullptr; other.m_length = 0;
    }
    return *this;
  }
};
```

Use `std::move`

Why do we need `std::move`?

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(other.m_label) { // Isn't this correct?
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

`other` is an rvalue reference, so ... ?

An rvalue reference is an lvalue.

`other` is an rvalue reference, **which is an lvalue**.

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_storage(other.m_storage), m_length(other.m_length),
        m_label(other.m_label) { // `other.m_label` is copied, not moved.
    other.m_storage = nullptr;
    other.m_length = 0;
  }
};
```

An rvalue reference is an lvalue! Does that make sense?

Lvalues persist; Rvalues are ephemeral.

The lifetime of rvalues is often very short, compared to that of lvalues.

- Lvalues have persistent state, whereas rvalues are either **literals** or **temporary objects** that only exist in their expressions or statements.

An rvalue reference **extends** the lifetime of the rvalue that it is bound to.

```
std::string s1 = something(), s2 = some_other_thing();
std::string &&rr = s1 + s2; // The rvalue reference gives a name `rr`
                            // to the temporary object, which will not
                            // be destroyed after this statement.
std::cout << rr << '\n'; // Now we can use `rr` just like a normal string.
```

Golden rule: **Anything that has a name is an lvalue.**

- The rvalue reference has a name, so it is an lvalue.

Return Value Optimization (RVO)

Returning an unnamed temporary

```
std::string fun(const std::string &a, const std::string &b) {
  return a + b; // a temporary
}
std::string a = "hello", b = "world";
std::string s = fun(a, b);
```

- First, a temporary `tmp1` is created to store the result of `a + b`.
- Then, a **move initialization** of another temporary `tmp2` with `tmp1` is performed.
- Finally, a **move initialization** of `s` with `tmp2` is performed.

(Unnamed) Return Value Optimization (RVO)

```cpp
std::string fun(const std::string &a, const std::string &b) {
  return a + b; // a temporary
}
std::string a = "hello", b = "world";
std::string s = fun(a, b);
```

Since C++17, **no copy or move** is made here. The initialization of `s` is the same as

```cpp
std::string s("helloworld"); // call the constructor directly
```

This is called **copy elision** that omits unnecessary copy or move of objects.

Returning a named object

```
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());
  for (std::size_t i = 0; i != a.size(); ++i)
    result.at(i) = a.at(i);
  for (std::size_t i = 0; i != b.size(); ++i)
    result.at(a.size() + i) = b.at(i);
  return result;
}
Dynarray a = concat(b, c); // Initialization
```

- `result` is a local object of `concat`.
- Since C++11, `return result;` performs a **move initialization** of a temporary `tmp` with `result` (`result` is about to "die" and thus can be moved from).
- Then, a **move initialization** of `a` with `tmp` is performed.

Named Return Value Optimization (NRVO)

```cpp
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());
  // ...
  return result;
}
Dynarray a = concat(b, c); // Initialization
```

NRVO (not mandatory) transforms this code to

```cpp
// Pseudo C++ code.
void concat(Dynarray &result, const Dynarray &a, const Dynarray &b) {
  // Pseudo C++ code. For demonstration only.
  result.Dynarray::Dynarray(a.size() + b.size()); // construct in-place
  // ...
}
Dynarray a@; // Uninitialized.
concat(a@, b, c);
```

so that no copy or move is needed.

Summary

Rvalue references

- Are bound to rvalues, and extend the lifetime of the rvalue.
- Functions accepting `X &&` and `const X &` can be overloaded.
- An rvalue reference is an lvalue.

Move operations

- Take ownership of resources from the other object.
- After a move operation, the moved-from object should be in a valid state that can be safely assigned to or destroyed.
- The rule of five: Define zero or five of the special member functions.

Summary

`std::move`

- It only performs an lvalue-to-rvalue cast, without moving anything.
- `std::move(x)` makes a promise that `x` can be safely moved from.

In C++, unnecessary copy or move of objects can be greatly avoided by copy elision.

- (Unnamed) RVO: for temporary objects returned by functions.
- NRVO: for named objects returned by functions.

Notes

[1] We seldom delete move operations. In most cases, we want rvalues to be copied if move is not possible. An explicitly deleted move operation will make rvalues not copyable, because deleted functions still participate in overload resolution.