CS100 Lecture 17

Rvalue References and Move

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?

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.

*** 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, but useless:
                         // Rvalue reference-to-const is seldom used.

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**.

```
struct Widget {
  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.

The Move Constructor

Take the `Dynarray` as an example.

```
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;
  }
};
```

The Move Constructor

```
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.

The Move Constructor

```
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!**

The 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`.

The Move Assignment Operator

```
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?

The 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?

The 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

Before we move on, let's define a function for demonstration.

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); // calls move assignment operator,
                  // because `concat(b, c)` is an rvalue.
a = b; // calls 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 require a synthesized move operation that has the default behaviors.

```
struct X {
  X(X &&) = default;
  X &operator=(X &&) = default;
};
```

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

Move operations can also be deleted by `=delete`, but be careful ... ${}^{\textcolor{red}{1}}$

The Rule of Five: Idea

The updated *copy control members*:

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

If one of them has a user-provided version, the copy control of the class is thought of to have special behaviors. (Recall "the rule of three".)

The Rule of Five: Rules

- The move constructor or the move assignment operator will not be generated ${}^{\textcolor{red}{2}}$ if any of the rest four members have a user-declared version.

- The copy constructor or copy assignment operator, if not provided by the user, will be implicitly `delete`d if the class has a user-provided move operation.
- The generation of the copy constructor or copy assignment operator is **deprecated** (since C++11) when the class has a user-declared copy operation or a destructor.
  - This is why some of you see this error:

```
Implicitly-declared copy assignment operator is deprecated, because the
class has a user-provided copy constructor.
```

The Rule of Five

The *copy control members* in modern C++:

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

The Rule of Five: Define zero or five of them.

How to Invoke a Move Operation?

Suppose we give our `Dynarray` a label:

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

The move assignment operator should invoke the **move assignment operator** on `m_label`. But how?

```cpp
m_label = other.m_label; // calls 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**:

```cpp
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 assumptions about the value of the moved-from object.**

```cpp
void foo(X &&x);      // moves `x`
void foo(const X &x); // copies `x`
foo(std::move(x)); // matches `foo(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.

```cpp
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;
  }
};
```

The standard library facilities ought to define efficient and correct move operations.

Use `std::move`

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

```cpp
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;
  }
};
```

The standard library facilities ought to define efficient and correct move operations.

Use `std::move`

Why do we need `std::move`?

```cpp
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**.

- To move the object that the rvalue reference is bound to, we must call `std::move`.

```cpp
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** created in the course of evaluating expressions.

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

```cpp
std::string s1 = something(), s2 = some_other_thing();
std::string &&rr = s1 + s2; // The state of the temporary object is "captured"
                            // by the rvalue reference, without which the
                            // temporary object will be destroyed.
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.

NRVO, Move and Copy Elision

Returning a Temporary (pure rvalue)

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

- First, a temporary is generated to store the result of `a + b`.
- How is this temporary returned?

Returning a Temporary (pure rvalue)

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

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

```cpp
std::string s(a + b);
```

This is called **copy elision**.

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

- `result` is a local object of `concat`.
- Since C++11, `return result` performs a **move initialization** of a temporary object, say `tmp`.
- Then a **move assignment** to `a` is performed.

Named Return Value Optimization, NRVO

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

NRVO transforms this code to

```
// 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.

Named Return Value Optimization, NRVO

Note:

- NRVO was invented decades ago (even before C++98).
- NRVO is an **optimization**, but not mandatory.
- Even if NRVO is performed, the move constructor should still be available.
    - Because the compiler can choose not to perform NRVO.
    - The program should be syntactically correct ("well-formed"), no matter how the compiler treats it.

Summary

Rvalue references

- are bound to rvalues, and extends 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.
- `=default`
- The rule of five: Define zero or five of the special member functions.

Summary

`std::move`

- does not move anything. It only performs an lvalue-to-rvalue cast.
- `std::move(x)` makes a promise that `x` can be safely moved from.

In modern C++, unnecessary copies are greatly avoided by:

- copy-elision, which avoids the move or copy of temporary objects, and
- move, with the `return`ed lvalue treated as an rvalue, and
- NRVO, which constructs in-place the object to be initialized.