CS100 Lecture 18

Contents

- Rvalue References
- Move Operations
  - Move Constructor
  - Move Assignment Operator
  - The Rule of Five
- `std::move`
- NRVO, Move and Copy Elision

Motivation: copy is slow

```cpp
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. Destroy it by calling 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. Destroy it by calling 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 needed 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 we can distinguish 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**:

```cpp
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 can only be bound to lvalues.
- Rvalue references can only be bound to rvalues.

Overload Resolution

```cpp
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 &)`.

Move Operations

Overview

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

```cpp
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 necessary!** (Will be covered in later lectures)

The 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;
  }
};
```

The 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.

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

```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?

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

Lvalues are Copied; Rvalues are Moved

Lvalues are copied; rvalues are moved …

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

The Rule of Five

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.

The Rule of Five

- The move constructor or the move assignment operator will not be generated if any of the rest four members have a user-provided 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-proided 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:

```
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?

```
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**:

```
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.**

```
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`

```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;
  }
};
```

Use `std::move`

```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;
  }
};
```

Use `std::move`

Recall the **slow** string concatenation:

```
std::string s;
for (int i = 0; i != n; ++i)
  s = s + 'a';
```

- To compute `s + 'a'`, a **copy** of `s` is made because `s` is an **lvalue**. (Slow)
- The result of `s + 'a'` is stored in a **temporary object**, say `tmp`.
- Then the assignment `s = tmp` is performed, which is a **move** since `tmp` is an rvalue.

Use `std::move`

Recall the **slow** string concatenation:

```
std::string s;
for (int i = 0; i != n; ++i)
  s = s + 'a';
```

- To compute `s + 'a'`, a copy of `s` is made because `s` is an **lvalue**.
- Since the original value of `s` is not used anymore (after computing `s + 'a'`), is it possible to avoid this copy?

Use `std::move`

Require a move instead of copy, explicitly, by calling `std::move`.

```
std::string s;
for (int i = 0; i != n; ++i)
  s = std::move(s) + 'a';
```

- If the left-hand side object is an rvalue, this concatenation is directly performed by appending the right-hand side object to it.
  - This is reasonable, since the value of a *moved-from* object do not need to be preserved.

NRVO, Move and Copy Elision

Returning a Temporary (pure rvalue)

```
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)

```
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

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

This is called **copy elision**.

Returning a Named Object

```cpp
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

```cpp
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

```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.

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.)

Summary

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

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