# Move Semantics and Perfect Forwarding

GKxx

July 30, 2022

# Contents

# How many copies?

```
std::string foo() {
  std::string s = some_value();
  return s;
}
std::string t = foo();
```

# How many copies?

```
std::string foo() {
  std::string s = some_value();
  return s;
}
std::string t = foo();
```

1. Before C++11, the local variable `s` is returned by a **copy-initialization** of a temporary object,
2. which is then used to **copy-initialize** `t`.

# How many copies?

```
std::string foo() {
  std::string s = some_value();
  return s;
}
std::string t = foo();
```

1. Before C++11, the local variable `s` is returned by a **copy-initialization** of a temporary object,
2. which is then used to **copy-initialize** `t`.

Modern compilers perform **Return-value optimization** (RVO) which eliminates the first copy.

- But this is not guaranteed by the standard.

# How copy affects efficiency

```
char some_char(int);

std::string fun1(int n) {
  std::string s = "";
  for (auto i = 0; i != n; ++i)
    s += some_char(i);
  return s;
}

std::string fun2(int n) {
  std::string s = "";
  for (auto i = 0; i != n; ++i)
    s = s + some_char(i);
  return s;
}
```

# How copy affects efficiency

```
for (auto i = 0; i != n; ++i)
  s += some_char(i);
for (auto i = 0; i != n; ++i)
  s = s + some_char(i);
```

- `s += some_char(i)` is virtually the same as `s.push_back(i)`, which consumes little time.
- `s = s + some_char(i)` causes **two copies**: a temporary object generated by `s + some_char(i)`, and a **copy-assignment** to `s`.

# How copy affects efficiency

```
for (auto i = 0; i != n; ++i)
  s += some_char(i);
for (auto i = 0; i != n; ++i)
  s = s + some_char(i);
```

- `s += some_char(i)` is virtually the same as `s.push_back(i)`, which consumes little time.
- `s = s + some_char(i)` causes **two copies**: a temporary object generated by `s + some_char(i)`, and a **copy-assignment** to `s`.

As a result, the first code takes $O(n)$ time, while the second one takes $O(n^2)$ time (assuming `some_char(i)` is $O(1)$).

# Why is copy needed?

```
a = b;
```

- We may want `a` and `b` to be different and independent objects.
- We may want to make changes to `a` without affecting `b`.

# Why is copy needed?

```
a = b;
```

- We may want `a` and `b` to be different and independent objects.
- We may want to make changes to `a` without affecting `b`.

However, sometimes the "**copied-from** object" is about to die.

```
a = c + d;
```

- Can we just let `a` take the ownership of `b`'s resources?

# A special constructor/`operator=`?

We need a special constructor/`operator=` that

- is different than copy operations, and
- has the semantics of "taking ownership of resources".

What would the parameter type be?

# Contents

# Rvalue References

A kind of reference that is bound to rvalues:

```
int &r = 42;              // Error.
int &&rr = 42;            // Correct.
const int &cr = 42;       // Also correct.
const int &&crr = 42;     // Correct but useless.
int i = 42;
int &&rr2 = i;            // Error.
int &r2 = i * 42;         // Error.
const int &cr2 = i * 2;   // Correct.
int &&r3 = i * 42;        // Correct.
```

# Rvalue References

A kind of reference that is bound to rvalues:

```
int &r = 42;                // Error.
int &&rr = 42;              // Correct.
const int &cr = 42;         // Also correct.
const int &&crr = 42;       // Correct but useless.
int i = 42;
int &&rr2 = i;              // Error.
int &r2 = i * 42;           // Error.
const int &cr2 = i * 2;     // Correct.
int &&r3 = i * 42;          // Correct.
```

- (Lvalue) references can only be bound to lvalues.
- Rvalue references can only be bound to rvalues.
- (Lvalue) reference-to-`const` can also be bound to rvalues.
- Rvalue reference-to-`const` is useless in most cases (we will see why).

# Overload Resolution for References

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

- `fun("hello")` matches `fun(std::string &&)`.
- `fun(s)` matches `fun(const std::string &)`.
- `fun(s1 + s2)` matches `fun(std::string &&)`.

# Overload Resolution for References

```
void fun(int);
void fun(int &&);
```

- `fun(i)` matches `fun(int)`.
- `fun(42)` is **ambiguous** (compile-error).

# Overload Resolution for References

```
void fun(int);
void fun(int &&);
```

- `fun(i)` matches `fun(int)`.
- `fun(42)` is **ambiguous** (compile-error).

```
void test(int);
void test(int &);
```

- `test(42)` matches `test(int)`.
- `test(i)` is **ambiguous**.

# Contents

# Overview

The move constructor and the move assignment operator.

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

- Move operations should be noexcept in most cases (we will see this later).

# The Move Constructor

The resources owned by the "moved-from" object are *stolen* in move operations.

```
template <typename T>
class Array {
  std::size_t m_size;
  T *m_data;
 public:
  Array(Array &&other) noexcept
      : m_size(other.m_size), m_data(other.m_data) {
    other.m_size = 0;
    other.m_data = nullptr;
  }
};
```

# The Move Assignment Operator

```
template <typename T>
class Array {
  std::size_t m_size;
  T *m_data;
 public:
  Array &operator=(Array &&other) noexcept {
    if (this != &other) {
      delete[] m_data;
      m_size = other.m_size;
      m_data = other.m_data;
      other.m_size = 0;
      other.m_data = nullptr;
    }
    return *this;
  }
};
```

# The Move Constructor

```
template <typename T>
Array<T>::Array(Array &&other) noexcept
    : m_size(other.m_size), m_data(other.m_data) {


}
```

- Obtain the resources directly instead of making a copy.

# The Move Constructor

```cpp
template <typename T>
Array<T>::Array(Array &&other) noexcept
    : m_size(other.m_size), m_data(other.m_data) {
  other.m_size = 0;
  other.m_data = nullptr;
}
```

- Obtain the resources directly instead of making a copy.
- Make sure the “moved-from” object is in a valid state and can be safely destroyed.

# The Move-Assignment Operator

```
template <typename T>
Array<T> &Array<T>::operator=(Array &&other) noexcept {
  if (this != &other) {







  }
}
```

- Test self-assignment directly.

# The Move-Assignment Operator

```
template <typename T>
Array<T> &Array<T>::operator=(Array &&other) noexcept {
  if (this != &other) {
    delete[] m_data;
    m_size = other.m_size;
    m_data = other.m_data;



  }
}
```

- Test self-assignment directly.
- Obtain the resources.

# The Move-Assignment Operator

```
template <typename T>
Array<T> &Array<T>::operator=(Array &&other) noexcept {
  if (this != &other) {
    delete[] m_data;
    m_size = other.m_size;
    m_data = other.m_data;
    other.m_size = 0;
    other.m_data = nullptr;
  }
}
```

- Test self-assignment directly.
- Obtain the resources.
- Make sure the “moved-from” object is in a valid state and can be safely destroyed.

# Copy-and-Swap Still Works!

```
template <typename T>
class Array {
 public:
  void swap(Array &other) noexcept {
    using std::swap;
    swap(m_size, other.m_size);
    swap(m_data, other.m_data);
  }
  Array &operator=(Array other) noexcept {
    Array(other).swap(*this);
    return *this;
  }
};
```

- Surprisingly, we obtain both a copy-assignment operator and a move-assignment operator!

# Lvalues are Copied; Rvalues are Moved

Lvalues are copied; rvalues are moved...

```
Array<int> arr = some_value();
Array<int> arr2 = arr; // copy
Array<int> arr3 = arr.slice(l, r); // move
```

# Lvalues are Copied; Rvalues are Moved

Lvalues are copied; rvalues are moved...

```
Array<int> arr = some_value();
Array<int> arr2 = arr; // copy
Array<int> arr3 = arr.slice(l, r); // move
```

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

```
struct Widget {
  Widget(Widget &&) = delete;
  Widget(const Widget &) = default;
};
Widget f();
Widget w = f(); // copy (before C++17 and without RVO)
```

# Call Move Operations

```
class Widget {
  Array<int> m_array;
  std::string m_str;
 public:
  Widget(Widget &&other) noexcept
      : m_array(other.m_array), m_str(other.m_str) {}
};
```

# Call Move Operations

```
class Widget {
  Array<int> m_array;
  std::string m_str;
 public:
  Widget(Widget &&other) noexcept
      : m_array(other.m_array), m_str(other.m_str) {}
};
```

Unfortunately, this will call the **copy constructors** instead of move constructors.

## Question

Is rvalue reference an lvalue or an rvalue?

# Lvalues Persist; Rvalues are Ephemeral

Roughly speaking,

- lvalues have persistent state, whereas
- rvalues are often **literals** or **temporary objects** that only live within an expression.
  - Rvalues are about to be destroyed and won't be used by anyone else.

# Lvalues Persist; Rvalues are Ephemeral

Roughly speaking,

- lvalues have persistent state, whereas
- rvalues are often **literals** or **temporary objects** that only live within an expression.
  - Rvalues are about to be destroyed and won't be used by anyone else.

By referring to an rvalue, an rvalue reference is **extending** the lifetime of it.

- Lvalue reference-to-`const` also does this.
- An rvalue reference is an **lvalue** because it has persistent state.

# Contents

# Generate an Rvalue

By casting to an rvalue reference using `static_cast`, we can produce an rvalue manually:

```
std::string s(t); // copy
std::string s2(static_cast<std::string &&>(t)); // move
```

# Generate an Rvalue

By casting to an rvalue reference using `static_cast`, we can produce an rvalue manually:

```
std::string s(t); // copy
std::string s2(static_cast<std::string &&>(t)); // move
```

The standard library function `std::move` does this.

```
std::string s3(std::move(s)); // move
```

Note: a function call whose return type is rvalue reference to object is treated as an rvalue.

## `std::move`

Defined in header `<utility>`.

- `std::move` performs a `static_cast` to rvalue reference, which produces an rvalue.
- `std::move` is used to *indicate* that an object may be "moved from".
  - **It does not move anything in fact!**

# std::move

Defined in header <utility>.

- `std::move` performs a `static_cast` to rvalue reference, which produces an rvalue.
- `std::move` is used to *indicate* that an object may be "moved from".
  - **It does not move anything in fact!**

Possible implementation:

```
template <typename T>
[[nodiscard]] constexpr auto move(T &&t) noexcept
    -> std::remove_reference_t<T> && {
  return static_cast<std::remove_reference_t<T> &&>(t);
}
```

* The parameter is a **universal reference**, which we will talk about later.

# Call Move Operations

```
class Widget {
  Array<int> m_array;
  std::string m_str;
 public:
  Widget(Widget &&other) noexcept
      : m_array(std::move(other.m_array)),
        m_str(std::move(other.m_str)) {}
  Widget &operator=(Widget &&other) noexcept {
    m_array = std::move(other.m_array);
    m_str = std::move(other.m_str);
    return *this;
  }
};
```

# The Moved-from Object

What might be the output?

```
int i = 42;
int j = std::move(i);
std::cout << i << '\n';
std::string s = "hello";
std::string t = std::move(s);
std::cout << s << '\n';
```

# The Moved-from Object

What might be the output?

```
int i = 42;
int j = std::move(i);
std::cout << i << '\n';
std::string s = "hello";
std::string t = std::move(s);
std::cout << s << '\n';
```

- After a move operation, the moved-from object remains a valid, destructible object,
- but users may make no assumptions about its value.

# The Moved-from Object

What might be the output?

```
int i = 42;
int j = std::move(i);
std::cout << i << '\n';
std::string s = "hello";
std::string t = std::move(s);
std::cout << s << '\n';
```

- After a move operation, the moved-from object remains a valid, destructible object,
- but users may make no assumptions about its value.
- The moved-from object is possibly modified in a move operation.
  - That's why rvalue reference-to-`const` is rarely used.

# Contents

# Synthesized Move Operations

```
class Widget {
  Array<int> m_array;
  std::string m_str;
 public:
  Widget(Widget &&) = default;
  Widget &operator=(Widget &&) = 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

# The Rule of Five

The updated copy control members:

- Copy constructor
- Copy-assignment operator
- Move constructor
- Move-assignment operator
- Destructor

If one of them is user-declared, the copy control of the class is thought of to have special behaviors.

- Therefore, the move ctor or move-assignment operator will not be generated if any of the rest four members has been declared by the user.

# The Rule of Five

- The move ctor or move-assignment operator will not be generated if any of the rest four members has been declared by the user.
- The copy ctor or copy-assignment operator, if not provided by the user, will be implicitly `delete`d if the class has a user-declared move operation.
- The generation of the copy ctor or copy-assignment operator is **deprecated** (since C++11) when the class has a user-declared copy operation or destructor.

To sum up, the five copy control members are thought of as a unit in modern C++: **If you think it necessary to define one of them, consider defining them all.**

# Contents

# Move Operations and Exceptions

Consider how `std::vector` grows:

```
template <typename T, typename Alloc>
void vector<T, Alloc>::reallocate(size_type cap) {
  using all_tr = std::allocator_traits<Alloc>;
  auto new_data = all_tr::allocate(s_alloc, cap), p = new_data;
  for (size_type i = 0; i != m_size; ++i, ++p)
    all_tr::construct(s_alloc, p, m_data[i]);
  m_free(); // destroys all elements and deallocates memory
  m_data = new_data;
  m_capacity = cap;
}
```

# Move Operations and Exceptions

To enable strong exception safety guarantee:

```
template <typename T, typename Alloc>
void vector<T, Alloc>::reallocate(size_type cap) {
  using all_tr = std::allocator_traits<Alloc>;
  auto new_data = all_tr::allocate(s_alloc, cap), p = new_data;
  try {
    for (size_type i = 0; i != m_size; ++i, ++p)
      all_tr::construct(s_alloc, p, m_data[i]);
  } catch (...) {
    while (p != new_data)
      all_tr::destroy(s_alloc, --p);
    all_tr::deallocate(s_alloc, new_data, cap);
    throw;
  }
  m_free();
  m_data = new_data;
  m_capacity = cap;
}
```

# Move Operations and Exceptions

With C++11, a natural optimization is to move-construct each element when `value_type` is move-constructible:

```
template <typename T, typename Alloc>
void vector<T, Alloc>::reallocate(size_type cap) {
  using all_tr = std::allocator_traits<Alloc>;
  auto new_data = all_tr::allocate(s_alloc, cap), p = new_data;
  try {
    for (size_type i = 0; i != m_size; ++i, ++p)
      all_tr::construct(s_alloc, p, std::move(m_data[i]));
  } catch (...) {
    while (p != new_data)
      all_tr::destroy(s_alloc, --p);
    all_tr::deallocate(s_alloc, new_data, cap);
    throw;
  }
  m_free();
  m_data = new_data;
  m_capacity = cap;
}
```

# Move Operations and Exceptions

What if the move constructor throws an exception?

```
try {
  for (size_type i = 0; i != m_size; ++i, ++p)
⚠ all_tr::construct(s_alloc, p, std::move(m_data[i]));
} catch (...) {
  while (p != new_data)
    all_tr::destroy(s_alloc, --p);
  all_tr::deallocate(s_alloc, new_data, cap);
  throw;
}
```

# Move Operations and Exceptions

What if the move constructor throws an exception?

```
try {
  for (size_type i = 0; i != m_size; ++i, ++p)
⚠   all_tr::construct(s_alloc, p, std::move(m_data[i]));
} catch (...) {
  while (p != new_data)
    all_tr::destroy(s_alloc, --p);
  all_tr::deallocate(s_alloc, new_data, cap);
  throw;
}
```

The preceding elements have been moved! How can we restore them?

# Move Operations and Exceptions

Exception is not welcome in move operations.

- Copy is to *create something else* in terms of existing things,
- whereas move is to *change* the existing things.

# Move Operations and Exceptions

Exception is not welcome in move operations.

- Copy is to *create something else* in terms of existing things,
- whereas move is to *change* the existing things.

Use `std::move_if_noexcept` to move the elements only when the move constructor does not throw.

```
for (size_type i = 0; i != m_size; ++i, ++p)
  all_tr::construct(s_alloc, p,
                    std::move_if_noexcept(m_data[i]));
```

# std::move_if_noexcept

Possible implementation:

```
template <typename T>
[[nodiscard]] constexpr std::conditional_t<
  !std::is_nothrow_move_constructible_v<T>
    && std::is_copy_constructible_v<T>,
  const T &,
  T &&
> move_if_noexcept(T &&x) noexcept {
  return std::move(x);
}
```

Note: for move-only types (for which copy constructor is not available), move constructor is used either way and the strong exception-safety guarantee may be waived.

# Contents

# Contents