Move Semantics and Perfect Forwarding

GKxx

August 6, 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`.

Compilers are allowed to perform **Return-value optimization** (RVO) or **Named Return-value Optimization** (NRVO) which eliminate copies in some cases.

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(some_char(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(some_char(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 Reference

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 Reference

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 &&)`.
  - But if `fun(std::string &&)` is not present, calls with rvalue arguments also match `fun(const std::string &)`.

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

```
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) {




  }
  return *this;
}
```

- 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;


  }
  return *this;
}
```

- 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;
  }
  return *this;
}
```

- 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 {
    swap(other);
    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(1, r); // move
```

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

```
struct Widget {
  // Widget(Widget &&) is not defined.
  Widget(const Widget &) = default;
};
Widget f();
Widget w = f(); // copy (before C++17)
```

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

Copy is Not Welcome!
Move Semantics
Rvalue Reference
Move Operations
std::move
The Rule of Five
Move Operations and Exceptions
Automatic Move and Copy Elision
Perfect Forwarding
Examples
Universal Reference
std::forward

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 `deleted` 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

Copy is Not Welcome!

Move Semantics
Rvalue Reference
Move Operations
std::move
The Rule of Five
Move Operations and Exceptions
Automatic Move and Copy Elision

Perfect Forwarding
Examples
Universal Reference
std::forward

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

Copy is Not Welcome!

Move Semantics
Rvalue Reference
Move Operations
std::move
The Rule of Five
Move Operations and Exceptions
Automatic Move and Copy Elision

Perfect Forwarding
Examples
Universal Reference
std::forward

Move from Local Variables and Parameters

When an object (non-`volatile`) being `return`ed is declared

- in the function body, or
- as a parameter of the function,

the overload resolution to select the constructor to use for initialization of the returned value is performed twice:

1. first as if the object were an **rvalue** (thus it may select the move constructor),
2. and if the first overload resolution failed, then overload resolution is performed as usual, with the object considered as an lvalue (so it may select the copy constructor).

Move from Local Variables and Parameters

In short, the returned value will be copy-initialized only when the move constructor is not available:

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

This causes only two moves.

Guaranteed Copy Elision

Since C++17, elision of copy/move operations are mandatory (instead of a compiler optimization) in some cases, e.g.

- returning a prvalue of the same class type (ignoring cv-qualification) as the function return-type:

```
std::string f() {
  return std::string(10, 'c');
}
f(); // only calls std::string(10, 'c')
     // No copy or move is made.
```

- Initializing an object with a prvalue initializer of the same class type (ignoring cv-qualification):

```
std::string s = f(); // No copy or move is made.
// equivalent to 'std::string s(10, 'c');'
```

Guaranteed Copy Elision

With C++17 copy elision, the following code causes only one move:

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

Guaranteed Copy Elision

With C++17 copy elision, the following code causes only one move:

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

This code compiles even with `Widget(Widget &&) = delete;`:

```
Widget fun() {
  return Widget{};
}
Widget w = fun(); // No copy or move is made.
```

Contents

Forward Arguments

Some functions need to forward the arguments to another function.

```
std::invoke(f, x, y, z);
std::vector<Type> v;
v.emplace_back(x, y, z);
```

- `std::invoke(f, args...)` calls `f(args...)`.
- `v.emplace_back(args...)` constructs the element by calling the constructor `Type(args...)`.
  - It is different from `v.push_back(Type(args...))` in that it does not copy the object.

Forward Arguments

Value categories must be preserved.

```
std::string some_value();
std::vector<std::string> vs;
std::string s = some_value();
vs.emplace_back(s);                 // copy
vs.emplace_back(some_value()); // move
```

Forward Arguments

Value categories must be preserved.

```
std::string some_value();
std::vector<std::string> vs;
std::string s = some_value();
vs.emplace_back(s);                 // copy
vs.emplace_back(some_value()); // move
```

cv-qualifications must be preserved.

```
struct Widget {
  Widget(const std::string &);
  Widget(std::string &);
};
std::vector<Widget> vw;
vw.emplace_back(s);      // Widget(std::string &)
vw.emplace_back("abc"); // Widget(const std::string &)
```

Forward Arguments

Value categories must be preserved.

```
std::string some_value();
std::vector<std::string> vs;
std::string s = some_value();
vs.emplace_back(s);                 // copy
vs.emplace_back(some_value()); // move
```

cv-qualifications must be preserved.

```
struct Widget {
  Widget(const std::string &);
  Widget(std::string &);
};
std::vector<Widget> vw;
vw.emplace_back(s);      // Widget(std::string &)
vw.emplace_back("abc"); // Widget(const std::string &)
```

Still, we need to avoid unnecessary copies.

Contents

Copy is Not Welcome!

Move Semantics
Rvalue Reference
Move Operations
std::move
The Rule of Five
Move Operations and Exceptions
Automatic Move and Copy Elision

Perfect Forwarding
Examples
Universal Reference
std::forward

Universal Reference

```
template <typename T>
void fun(T &&x) {
  std::cout << gkxx::get_type_name<T>() << '\n';
}
int main() {
  int i = 42;
  fun(i);
  fun(42);
  const int j = 42;
  fun(j);
  return 0;
}
```

Output:

```
int&
int
const int&
```

Universal Reference

```
template <typename T>
void fun(T &&x);
```

- If the argument is an rvalue of type `Tp`, this is a normal rvalue reference initialization and `T = Tp`.
- If the argument is an lvalue of type `Tp`, it follows the special rule:
  - `T` would be deduced to an lvalue reference, i.e. `T = Tp &`.
  - `x` is of type `Tp & &&`, which *collapses* to `Tp &` through the **reference collapsing rule**.

Universal Reference

```
template <typename T>
void fun(T &&x);
```

- If the argument is an rvalue of type `Tp`, this is a normal rvalue reference initialization and `T = Tp`.
- If the argument is an lvalue of type `Tp`, it follows the special rule:
  - `T` would be deduced to an lvalue reference, i.e. `T = Tp &`.
  - `x` is of type `Tp & &&`, which *collapses* to `Tp &` through the **reference collapsing rule**.
- cv-qualifications would be preserved since this is a reference initialization.
  - e.g. `x` will be an lvalue reference-to-`const` if the argument is a `const` lvalue.

Reference Collapsing

It is permitted to form "references to references" through type manipulations in **templates** or **type aliasing**, in which case the *reference collapsing* rules apply:

- `& &`, `& &&` and `&& &` collapse to `&`.
- `&& &&` collapses to `&&`.

```
using lref = int &;
using rref = int &&;
int n = 42;

lref&  r1 = n;      // int&
lref&& r2 = n;      // int&
rref&  r3 = n;      // int&
rref&& r4 = 42;     // int&&
```

Universal Reference vs Rvalue Reference

The form `T &&x` is a universal reference **iff** `T` is obtained through **type deduction** directly.

```
template <typename T>
void fun(T &&x);                    // universal reference

template <typename T>
class Widget {
  void fun(T &&x);                  // rvalue reference
};

auto &&x = y;                       // universal reference

template <typename T>
void f(std::vector<T> &&x); // rvalue reference
```

Contents

std::forward

Defined in header file <utility>.

```
template <typename Func, typename T>
auto invoke1(Func f, T &&arg) {
  return f(std::forward<T>(arg));
}
```

- `std::forward<T&>(x)` returns an lvalue reference, which produces an lvalue.
- `std::forward<T>(x)`, where `T` is not a reference, returns an rvalue reference, which produces an rvalue.
  - In this case, it is equivalent to `std::move(x)`.

std::forward

`std::forward` **does not actually "forward" anything!** It is used to preserve all the details about an argument's type (including value categories and cv-qualifiers).

std::forward

`std::forward` **does not actually "forward" anything!** It is used to preserve all the details about an argument's type (including value categories and cv-qualifiers).
Combining `std::forward` with universal reference and variadic template, we can perfectly forward any number of arguments with any types:

```
template <typename Func, typename... Args>
auto invoke(Func f, Args &&...args) {
  return f(std::forward<Args>(args)...);
}
```

Example: `std::invoke`

To avoid copying the callable object, we may forward `f` as well:

```
template <typename Func, typename... Args>
auto invoke(Func &&f, Args &&...args) {
  return (std::forward<Func>(f))(
    std::forward<Args>(args)...
  );
}
```

* The return-type might be problematic here...

Example: vector<T>::emplace_back

```
template <typename T, typename Alloc>
class vector {
 public:
  template <typename... Args>
  void emplace_back(Args &&...args) {
    check_and_realloc();
    using all_tr = std::allocator_traits<Alloc>;
    all_tr::construct(s_alloc, m_data + m_size,
                      std::forward<Args>(args)...);
    ++m_size;
  }
  void push_back(value_type &&x) {
    emplace_back(std::move(x)); // move
  }
  void push_back(const value_type &x) {
    emplace_back(x);            // copy
  }
};
```

Example: a Python-style print

```
template <typename First, typename... Rest>
inline void print(First &&first, Rest &&...rest) {
  std::cout << std::forward<First>(first);
  if constexpr (sizeof...(rest) == 0)
    std::cout << '\n';
  else {
    std::cout << ' ';
    print(std::forward<Rest>(rest)...);
  }
}
inline void print() {
  std::cout << '\n';
}
```

Reading Materials

Effective Modern C++:

- Item 26: Avoid overloading on universal references.
- Item 27: Familiarize yourself with alternatives to overloading on universal references.
- Item 29: Assume that move operations are not present, not cheap, and not used.
- Item 30: Familiarize yourself with perfect forwarding failure cases.
- Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied.