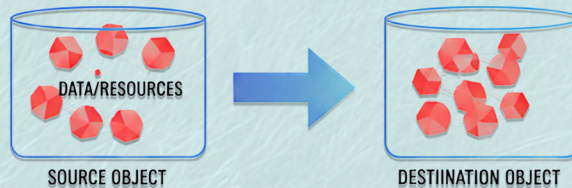
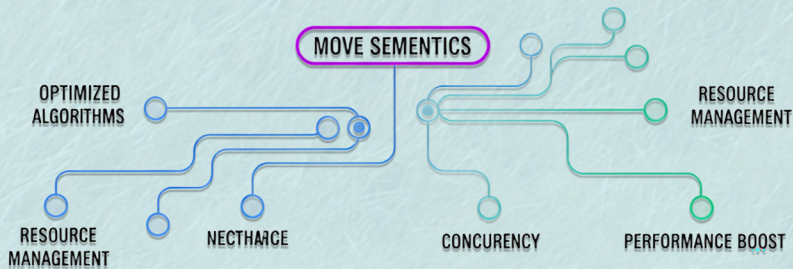


Move Semantics

why They Changed the Future of C++
and How They Really Work



MODERN C++ PROGRAMMING PARADIGMS



NEW PROGRAMMING PARADIGMS

Prepared By Ayman Alheraki

Move Semantics

Why They Changed the Future of C++ and How They Really Work

Prepared by Ayman Alheraki

simplifcpp.org

November 2025

Contents

Contents	2
1 Why Move Semantics Matter	5
1.1 The Core Problem Before C++11	5
1.2 What Move Semantics Solve	6
1.3 High-Level Intuition	7
2 Value Categories and Expressions	8
2.1 From lvalues vs rvalues to the New Taxonomy	8
2.2 Simple Mental Model	9
2.3 Examples: lvalues vs rvalues	9
3 Rvalue References and <code>std::move</code>	11
3.1 Rvalue References: T&&	11
3.2 <code>std::move</code> : A Cast to rvalue	12
3.3 When to Use <code>std::move</code>	13
4 Implementing Move Constructors and Move Assignment	14
4.1 Rule of Three, Five, and Zero	14
4.2 A Complete Example: Move-Aware RAII Class	15

4.2.1	Key Points	17
4.3	Move-Only Types	18
5	Move Semantics in the Standard Library	19
5.1	Containers and Move	19
5.2	<code>std::string</code> and Small String Optimization	20
5.3	Move and <code>noexcept</code>	20
6	Perfect Forwarding and <code>std::forward</code>	21
6.1	The Problem: Generic Functions and Value Categories	21
6.2	Universal (Forwarding) References	21
6.3	<code>std::forward</code> : Preserve Value Category	22
7	Design Patterns with Move Semantics	24
7.1	RAII and Ownership Transfer	24
7.1.1	Example: File Handle Wrapper	24
7.2	<code>pImpl</code> (Pointer-to-Implementation) and Move Semantics	26
8	Exception Safety, <code>noexcept</code>, and Performance	28
8.1	Why <code>noexcept</code> Matters	28
8.2	Strong Exception Safety with Copy-and-Swap	29
8.3	Performance Considerations	29
9	Common Pitfalls and Misconceptions	31
9.1	Misunderstanding <code>std::move</code>	31
9.2	Using Moved-From Objects	31
9.3	Accidental Copies Instead of Moves	32
9.4	Overusing Moves with Return Values	33

10 Practical Guidelines and Best Practices	34
10.1 For Beginners	34
10.2 For Intermediate Developers	34
10.3 For Advanced Developers	35
10.4 API Design Example: Take-By-Value and Move	35
11 Summary and Further Learning	37
11.1 Key Takeaways	37
11.2 Suggested Next Steps	38

Chapter 1

Why Move Semantics Matter

1.1 The Core Problem Before C++11

Before C++11, C++ code relied almost exclusively on:

- **Copying** objects.
- **Passing by reference** to avoid copies.
- **Manual resource management** with raw pointers, `new` and `delete`.

This led to several issues:

1. **Unnecessary copies** of large objects (such as `std::vector`, `std::string`, or user-defined types with heap allocations).
2. **Complex ownership conventions**: Who owns this pointer? Who should delete it?
3. **Suboptimal library interfaces**: Many APIs avoided returning large objects by value, and instead used output parameters or pointer-based APIs.

For example, returning a big container by value used to be considered expensive:

```
std::vector<int> make_big_vector() {  
    std::vector<int> v(1'000'000, 42);  
    return v; // Pre-C++11: usually a costly copy  
}
```

Compilers could sometimes apply Return Value Optimization (RVO), but:

- It was not guaranteed in all cases (before C++17).
- It did not help copies in other scenarios (like pushing into a container, reallocation, or assignment).

1.2 What Move Semantics Solve

Move semantics were introduced in C++11 to solve a simple but powerful problem:

If an object is about to be destroyed, why not “steal” its resources instead of copying them?

Key idea:

- Instead of copying data (expensive), we can **move** data by stealing internal pointers or handles (cheap).
- After moving, the source object is left in a **valid but unspecified** state.

This made:

- Returning large objects by value **fast and natural**.
- Temporary objects (rvalues) **very cheap to use**.
- Standard containers and strings significantly **more efficient**.

1.3 High-Level Intuition

Think of three levels of resource handling:

1. **Copy**: duplicate the resource (two owners).
2. **Move**: transfer ownership; old owner becomes empty or harmless.
3. **Destroy**: release the resource permanently.

Move semantics give the language a first-class way to express (2), which was missing before C++11.

Chapter 2

Value Categories and Expressions

To understand move semantics correctly, we must understand **value categories**.

2.1 From lvalues vs rvalues to the New Taxonomy

In C++03, the language mainly distinguished:

- **lvalues**: have a name and an address.
- **rvalues**: temporary objects, usually results of expressions.

C++11 refined and extended this into:

- **glvalues** (generalized lvalues).
- **prvalues** (pure rvalues).
- **xvalues** (expiring values).

In practice, for move semantics, two categories are most important:

- **lvalues**: objects with identity (you can take their address; they persist beyond the current expression).
- **rvalues** (including prvalues and xvalues): objects that are about to expire (temporary, disposable).

2.2 Simple Mental Model

You can adopt a simplified mental model:

- **lvalue** $\Rightarrow$ has a name, we should not steal from it unless explicitly asked.
- **rvalue** $\Rightarrow$ temporary or disposable; we are allowed to steal from it.

2.3 Examples: lvalues vs rvalues

```
int x = 10;           // x is an lvalue
int y = x;            // x is used as an lvalue

int foo();
int z = foo();        // foo() is an rvalue (a prvalue)
```

For user-defined types:

```
struct Vec {
    // ...
};

Vec v1;                // v1 is an lvalue
Vec v2 = v1;           // copy from lvalue

Vec make_vec();
Vec v3 = make_vec();   // make_vec() is an rvalue
```

With move semantics, constructors can distinguish between:

- **Copy from lvalue** (`const Vec&`).
- **Move from rvalue** (`Vec&&`).

Chapter 3

Rvalue References and `std::move`

3.1 Rvalue References: `T&&`

C++11 introduced **rvalue references**:

- A type like `T&&` binds (primarily) to rvalues.
- It allows you to implement move constructors and move assignment operators.

Example:

```
struct Buffer {
    std::size_t size{};
    int* data{};

    Buffer(std::size_t n) : size(n), data(new int[n]{} ) {}

    // Copy constructor
    Buffer(const Buffer& other)
        : size(other.size), data(new int[other.size])
```

```

{
    std::copy(other.data, other.data + size, data);
}

// Move constructor
Buffer(Buffer&& other) noexcept
    : size(other.size), data(other.data)
{
    other.size = 0;
    other.data = nullptr; // important: avoid double delete
}

~Buffer() {
    delete[] data;
}

};

```

3.2 std::move: A Cast to rvalue

Critical point: `std::move` does *not* move anything by itself.

- `std::move` is a cast that converts its argument to an rvalue (typically `T&&`).
- The actual move happens when a move constructor or move assignment operator is invoked.

```

#include <utility>

Buffer b1(1024);
Buffer b2 = std::move(b1); // calls Buffer(Buffer&&)

```

After this:

- `b2` now owns the buffer.
- `b1` is in a valid but unspecified state (likely empty).

3.3 When to Use `std::move`

Use `std::move` when:

- You have an **lvalue** that you know is no longer needed.
- You want to treat it as an **rvalue** and allow its resources to be moved.

Typical patterns:

- Moving out of a function parameter.
- Moving from local variables when returning or assigning.
- Moving from unique resources such as `std::unique_ptr`.

Chapter 4

Implementing Move Constructors and Move Assignment

4.1 Rule of Three, Five, and Zero

Historically, C++ had the **Rule of Three**:

- If you define any of
 - destructor
 - copy constructor
 - copy assignment operator

then you probably need to define all three.

With move semantics, we have the **Rule of Five**:

- destructor

- copy constructor
- copy assignment
- move constructor
- move assignment

And even better, the **Rule of Zero**:

If you use RAII types (like `std::vector`, `std::string`, `std::unique_ptr`), you often do not need to define any of the special member functions. Let the compiler generate them.

4.2 A Complete Example: Move-Aware RAII Class

```
#include <algorithm>
#include <cstdint>
#include <utility>

class IntArray {
public:
    IntArray() = default;

    explicit IntArray(std::size_t n)
        : size_(n), data_(n ? new int[n]{} : nullptr) {}

    // Destructor
    ~IntArray() {
        delete[] data_;
    }

    // Copy constructor
```

```
IntArray(const IntArray& other)
    : size_(other.size_), data_(other.size_ ? new int[other.size_] :
      ↪ nullptr)
{
    std::copy(other.data_, other.data_ + size_, data_);
}

// Copy assignment
IntArray& operator=(const IntArray& other) {
    if (this == &other) return *this;
    IntArray tmp(other);           // copy first (strong exception
    ↪ safety)
    swap(tmp);                     // swap with temporary
    return *this;
}

// Move constructor
IntArray(IntArray&& other) noexcept
    : size_(other.size_), data_(other.data_)
{
    other.size_ = 0;
    other.data_ = nullptr;
}

// Move assignment
IntArray& operator=(IntArray&& other) noexcept {
    if (this == &other) return *this;

    delete[] data_;                // free current resource

    size_ = other.size_;
    data_ = other.data_;
}
```

```
        other.size_ = 0;
        other.data_ = nullptr;

        return *this;
    }

    std::size_t size() const noexcept { return size_; }

    int& operator[](std::size_t i) { return data_[i]; }
    const int& operator[](std::size_t i) const { return data_[i]; }

    void swap(IntArray& other) noexcept {
        std::swap(size_, other.size_);
        std::swap(data_, other.data_);
    }

private:
    std::size_t size_{0};
    int* data_{nullptr};
};
```

4.2.1 Key Points

- **Move constructor** steals the pointer and nulls out the source.
- **Move assignment** must release the current resource, then steal.
- Mark move functions `noexcept` whenever possible (important for the standard library).

4.3 Move-Only Types

Some resources cannot be copied safely, but can be moved.

- Example: owning file handles, sockets, mutexes, or unique pointers.

```
#include <memory>

struct UniqueResource {
    std::unique_ptr<int> ptr;

    UniqueResource() = default;
    explicit UniqueResource(int value) : ptr(std::make_unique<int>(value))
        ↪ {}

    // Move-only type: default move ctor/assign are fine
    UniqueResource(UniqueResource&&) noexcept = default;
    UniqueResource& operator=(UniqueResource&&) noexcept = default;

    // No copy
    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;
};
```

Chapter 5

Move Semantics in the Standard Library

5.1 Containers and Move

The standard containers (`std::vector`, `std::string`, `std::map`, etc.) are heavily optimized for move semantics.

Examples where moves happen:

- Pushing into a container: `push_back`, `emplace_back`.
- Reallocation: when a `std::vector` grows.
- Returning containers by value.

```
#include <vector>
#include <string>

std::vector<std::string> make_names() {
    std::vector<std::string> names;
    names.emplace_back("Alice");
    names.emplace_back("Bob");
}
```

```
names.emplace_back("Charlie");  
return names; // Efficient move (or RVO)  
}  
  
int main() {  
    auto names = make_names(); // cheap  
}
```

5.2 `std::string` and Small String Optimization

Many implementations of `std::string` use **Small String Optimization** (SSO):

- For short strings, they avoid heap allocation.
- For longer strings, they allocate on the heap.

Move semantics make longer strings cheaper to transfer:

- Moving a large string just swaps internal pointers and sizes.
- No need to reallocate and copy the characters.

5.3 Move and `noexcept`

Standard containers prefer moving elements when reallocation occurs:

- If the move constructor is `noexcept`, containers will use it.
- If moving might throw, containers may fall back to copying for exception safety.

Guideline:

If your type supports a move that cannot throw, **mark it `noexcept`**.

Chapter 6

Perfect Forwarding and `std::forward`

6.1 The Problem: Generic Functions and Value Categories

Consider a generic wrapper that forwards arguments:

```
#include <utility>

template <typename T, typename... Args>
T make(Args&&... args) {
    return T(std::forward<Args>(args)...);
}
```

Why do we need `std::forward`? Why not just pass `args...`?

6.2 Universal (Forwarding) References

In templates, a parameter of the form `T&&` can sometimes bind to both:

- lvalues

- rvalues

Example:

```
template <typename T>
void wrapper(T&& x); // x is a forwarding (universal) reference
```

- If you call `wrapper(10)`, `T` is `int`, and `T&&` is `int&&`.
- If you call `int a = 10; wrapper(a);`, `T` is `int&`, and `T&&` collapses to `int&`.

This is the **reference collapsing rule**:

$$\&\& + \& \Rightarrow \&$$

$$\&\& + \&\& \Rightarrow \&\&$$

6.3 `std::forward`: Preserve Value Category

`std::forward<T>(x):`

- Returns `x` as an rvalue if `T` is an rvalue reference.
- Returns `x` as an lvalue otherwise.

The result:

- When you pass an rvalue to the wrapper, the wrapper forwards it as an rvalue (move).
- When you pass an lvalue, the wrapper forwards it as an lvalue (copy or reference).

```
#include <iostream>
#include <utility>
```

```
void f(const int& x) {
    std::cout << "lvalue overload: " << x << '\n';
}

void f(int&& x) {
    std::cout << "rvalue overload: " << x << '\n';
}

template <typename T>
void wrapper(T&& x) {
    f(std::forward<T>(x)); // preserves value category
}

int main() {
    int a = 42;
    wrapper(a);           // calls lvalue overload
    wrapper(99);          // calls rvalue overload
}
```

Chapter 7

Design Patterns with Move Semantics

7.1 RAII and Ownership Transfer

Move semantics fit perfectly with RAII (Resource Acquisition Is Initialization):

- Objects own resources during their lifetime.
- Move operations transfer ownership safely between objects.

7.1.1 Example: File Handle Wrapper

```
#include <cstdio>

class File {
public:
    File() = default;

    explicit File(const char* path, const char* mode)
        : handle_(std::fopen(path, mode)) {}
```

```

~File() {
    if (handle_) {
        std::fclose(handle_);
    }
}

// Non-copyable
File(const File&) = delete;
File& operator=(const File&) = delete;

// Movable
File(File&& other) noexcept : handle_(other.handle_) {
    other.handle_ = nullptr;
}

File& operator=(File&& other) noexcept {
    if (this != &other) {
        if (handle_) std::fclose(handle_);
        handle_ = other.handle_;
        other.handle_ = nullptr;
    }
    return *this;
}

std::FILE* get() const noexcept { return handle_; }

private:
    std::FILE* handle_{nullptr};
};

```

This type is:

- **Move-only** (like `std::unique_ptr`).

- Perfect for representing exclusive ownership of a resource.

7.2 pImpl (Pointer-to-Implementation) and Move Semantics

The pImpl idiom hides implementation details behind a pointer. Move semantics make pImpl cheaper to use:

```
#include <memory>

class Widget {
public:
    Widget();
    ~Widget();

    Widget(const Widget&);
    Widget& operator=(const Widget&);

    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

    void do_something();

private:
    struct Impl;
    std::unique_ptr<Impl> impl_;
};
```

The move operations of `Widget` are usually defaulted:

```
Widget::Widget(Widget&&) noexcept = default;
Widget& Widget::operator=(Widget&&) noexcept = default;
```

Because:

- `std::unique_ptr` is already move-aware and `noexcept`.
- You get efficient moves almost for free.

Chapter 8

Exception Safety, `noexcept`, and Performance

8.1 Why `noexcept` Matters

Consider a `std::vector<T>`. When it grows and reallocates:

- It must move or copy elements to new memory.
- If moving `T` can throw, the vector must preserve a strong exception guarantee:
 - either reallocation completes successfully,
 - or the vector remains unchanged.

Therefore:

- If `T`'s move constructor is marked `noexcept`, `std::vector` can safely move elements.

- If it is not `noexcept`, the vector may need to fall back to copying, which is more expensive.

Guideline:

If your move operations cannot throw, **mark them `noexcept`**.

8.2 Strong Exception Safety with Copy-and-Swap

Sometimes we implement copy assignment using **copy-and-swap**, and move assignment using a similar approach:

```
IntArray& operator=(IntArray other) {  
    swap(other); // other is a copy or a moved-from temporary  
    return *this;  
}
```

Here:

- The parameter `other` is taken by value.
- Construction of `other` uses either copy or move constructor.
- `swap` is `noexcept`, so assignment cannot throw after `other` is constructed.

8.3 Performance Considerations

Move semantics give big performance improvements, especially when:

- Objects manage dynamically allocated memory or other heavy resources.
- Containers move large numbers of elements during reallocation or sorting.

- APIs frequently return objects by value.

However:

- Do not **overuse** `std::move` on things you still need.
- Avoid moving from objects that might be used again, unless you reassign them.

Chapter 9

Common Pitfalls and Misconceptions

9.1 Misunderstanding `std::move`

Misconception: `std::move` actually moves the object.

Reality: `std::move` only casts to an rvalue; the move operation is performed by the constructor or assignment operator.

```
std::string s = "Hello";  
std::string t = std::move(s); // move ctor  
// s is now valid but unspecified (often empty or minimal)
```

9.2 Using Moved-From Objects

The standard guarantees:

Moved-from objects are **valid but unspecified**.

This means:

- You can only assign to them, destroy them, or call functions that explicitly document support for moved-from objects.
- You **must not** assume any particular state (like empty string, size zero, etc.), unless documented.

Bad pattern:

```
std::string s = "Hello";  
std::string t = std::move(s);  
std::cout << s; // Undefined behavior? No, but unspecified state.
```

Better pattern:

```
std::string s = "Hello";  
std::string t = std::move(s);  
  
// Reinitialize s if you want to use it  
s = "New value";
```

9.3 Accidental Copies Instead of Moves

Remember:

- Named variables are always lvalues, even if their type is an rvalue reference.

Example:

```
Buffer make_buffer();  
  
Buffer b = make_buffer();           // OK: use move ctor  
Buffer b2 = b;                      // copy, not move
```

```
Buffer&& br = make_buffer();    // br is an lvalue of type Buffer&&
Buffer b3 = br;                // copy, not move!

Buffer b4 = std::move(br);     // now move
```

9.4 Overusing Moves with Return Values

You do **not** need `std::move` when returning a local variable:

```
Buffer make() {
    Buffer buf;
    // ...
    return buf;    // correct: NRVO or move
}

// Anti-pattern:
Buffer make2() {
    Buffer buf;
    // ...
    return std::move(buf); // might inhibit NRVO in older compilers
}
```

In modern C++, simply `return buf;` is best.

Chapter 10

Practical Guidelines and Best Practices

10.1 For Beginners

- Understand the basic idea: moving transfers ownership; copying duplicates.
- Use standard library types (`std::vector`, `std::string`, `std::unique_ptr`) and trust them.
- Return objects by value without fear; move semantics make it cheap.
- Avoid raw `new/delete`. Use RAII and move-aware types.

10.2 For Intermediate Developers

- Learn value categories: lvalue vs rvalue.
- Use `std::move` when you know a variable will no longer be used.
- Implement move constructors and move assignment in resource-owning types.

- Mark move operations `noexcept` whenever they cannot throw.
- Prefer `emplace_back` and other emplacement functions.

10.3 For Advanced Developers

- Master perfect forwarding and `std::forward`.
- Design APIs that exploit move semantics:
 - Take by value and move inside, when appropriate.
 - Provide overloads for lvalues and rvalues if needed.
- Analyze performance: profile and measure real gains from moves.
- Consider move-only types to model unique ownership.
- Understand the interaction between move semantics, concurrency, and atomics (e.g., moving futures, promises, thread wrappers).

10.4 API Design Example: Take-By-Value and Move

```
#include <string>
#include <utility>

class Person {
public:
    Person(std::string name, int age)
        : name_(std::move(name)), age_(age) {}

private:
```

```
std::string name_;  
int age_;  
};
```

Call sites:

- If the caller passes an rvalue: `Person p("Alice", 30);`
 - The constructor parameter `name` is constructed by moving.
 - `name_` is constructed by moving again (cheap).
- If the caller passes an lvalue: `std::string s = "Bob"; Person p(s, 25);`
 - The constructor copies into `name`.
 - Then `name_` is move-constructed from `name`.

This pattern balances simplicity and performance.

Chapter 11

Summary and Further Learning

11.1 Key Takeaways

- Move semantics solve the fundamental problem of efficiently transferring resources from temporary or disposable objects.
- Rvalue references (T&&) and `std::move` are the core language features behind move semantics.
- The standard library (containers, strings, smart pointers) is heavily optimized around move semantics.
- Perfect forwarding and `std::forward` allow generic code to preserve value categories and achieve maximum efficiency.
- Properly implemented move operations, especially when marked `noexcept`, enable containers and algorithms to be both fast and exception-safe.

- Moved-from objects are valid but unspecified: you are responsible for reinitializing or destroying them, not for using their state.

11.2 Suggested Next Steps

To fully master move semantics:

1. **Rewrite existing classes** to use RAII and move semantics instead of raw pointers.
2. **Experiment with performance:** benchmark code with and without moves.
3. **Study standard library implementations** (where available) to see how containers and strings use move semantics internally.
4. **Practice perfect forwarding:** implement generic factory functions, wrapper functions, and adaptors.
5. **Combine move semantics with modern C++ features:** smart pointers, ranges, coroutines, and concurrency abstractions.

With consistent practice, move semantics becomes a natural part of your C++ design vocabulary. Once you fully internalize it, you will write APIs and implementations that are both elegant and extremely efficient—and you will clearly understand why move semantics truly changed the future of C++.