Training Material

Back to Basics: Move Semantics

CppCon
Online
2022-09-12

Andreas Fertig

Write unique code!

Version: v1.0

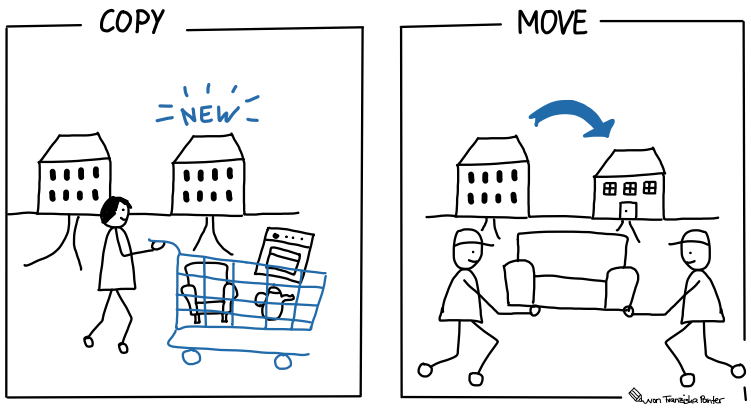
Planung, Satz und Einbandentwurf:
Andreas Fertig

Herstellung und Verlag:
Andreas Fertig

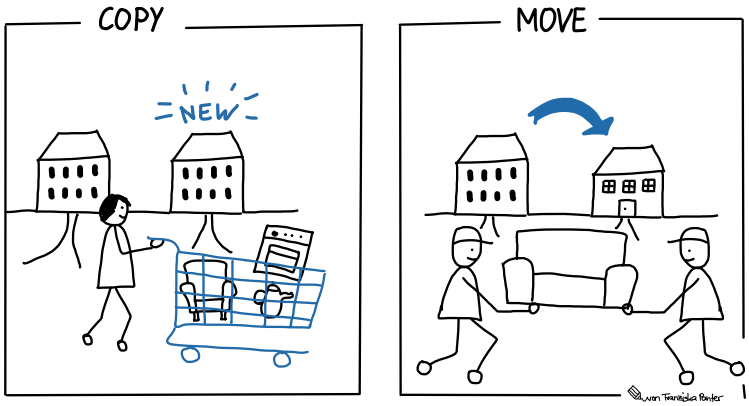
Back to Basics: Move Semantics

Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com
@Andreas__Fertig

Move semantics: move or duplicate



Move semantics: move or duplicate




```
void Copy(char** dst, char** src, size_t size)
{
  *dst = new char[size];
  memcpy( *dst, *src, size);
}
```

```
void Move(char** dst, char** src)
{
  *dst = *src;
  *src = nullptr;
}
```


Overloads

```
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}
```

Overloads

```
1 void Fun(std::vector<int>& byRef)
2 {
3   std::cout << "byRef\n";
4 }
5
6 void Fun(const std::vector<int>& byConstRef)
7 {
8   std::cout << "byConstRef\n";
9 }
```

Using Fun

```
1 void Use()
2 {
3   std::vector       v{2, 3, 4};
4   const std::vector cv{5, 6, 7};
5
6   Fun(v);           A We pass a lvalue
7   Fun(cv);          B We pass a const lvalue
8   Fun({3, 5, 6});   C We pass a temporary
9 }
```

Overloads

```
1 void Fun(std::vector<int>& byRef)
2 {
3   std::cout << "byRef\n";
4 }
5
6 void Fun(const std::vector<int>& byConstRef)
7 {
8   std::cout << "byConstRef\n";
9 }
```

Using Fun

```
1 void Use()
2 {
3   std::vector       v{2, 3, 4};
4   const std::vector cv{5, 6, 7};
5
6   Fun(v);           A We pass a lvalue
7   Fun(cv);          B We pass a const lvalue
8   Fun({3, 5, 6});   C We pass a temporary
9 }
```

```
$ ./a.out
byRef
byConstRef
byConstRef
```

Overloads

```
 1  void Fun(std::vector<int>& byRef)
 2 {
 3   std::cout << "byRef\n";
 4 }
 5
 6 void Fun(const std::vector<int>& byConstRef)
 7 {
 8   std::cout << "byConstRef\n";
 9 }
10
11 void Fun(std::vector<int>&& byRvalueRef)
12 {
13   std::cout << "byMoveRef\n";
14 }
```


Overloads

```
 1  void Fun(std::vector<int>& byRef)
 2 {
 3   std::cout << "byRef\n";
 4 }
 5
 6 void Fun(const std::vector<int>& byConstRef)
 7 {
 8   std::cout << "byConstRef\n";
 9 }
10
11 void Fun(std::vector<int>&& byRvalueRef)
12 {
13   std::cout << "byMoveRef\n";
14 }
```

```
$ ./a.out
byRef
byConstRef
byMoveRef
```

Triggering the rvalue-overload

```cpp
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}

void Fun(std::vector<int>&& byRvalueRef)
{
  std::cout << "byMoveRef\n";
}
```

```cpp
void Use()
{
  std::vector v{2, 3, 4};

  Fun(static_cast<std::vector<int>&&>(v));
}
```

```
$ ./a.out
byMoveRef
```

Triggering the rvalue-overload

```cpp
void Fun(std::vector<int>& byRef)
{
  std::cout << "byRef\n";
}

void Fun(const std::vector<int>& byConstRef)
{
  std::cout << "byConstRef\n";
}

void Fun(std::vector<int>&& byRvalueRef)
{
  std::cout << "byMoveRef\n";
}
```

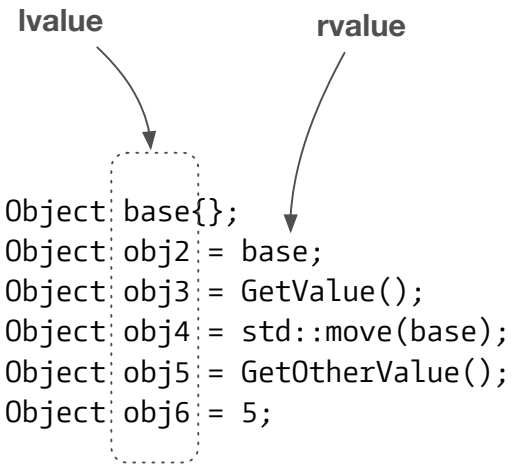
```cpp
#include <utility>

void UseMove()
{
  std::vector v{2, 3, 4};

  Fun( std::move(v) );
}
```

```
$ ./a.out
byMoveRef
```

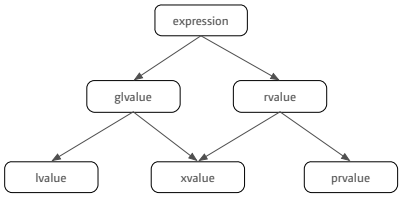
Some notes

- `std::move` is only a cast. The overloads can move if they exist.
- Temporary objects are picked up by default using move operations.
- **Only** if we have an object *we* no longer want to use we can say `std::move`.
- Move semantics is nothing else than an additional overload that is allowed and expected to steal data from a source object.

The value categories



The value categories



lvalue

xvalue

```
Object base{};
Object obj2 = base;
Object obj3 = GetValue();      // Object&& GetValue();
Object obj4 = std::move(base);
Object obj5 = GetOtherValue();  // Object GetOtherValue();
Object obj6 = 5;
```

glvalue := lvalue or xvalue

prvalue

The place of a potential performance win

```
1  class string {
2    size_t                  mLen{};
3    std::unique_ptr<char[]> mData{};
4
5  public:
6    string(const char* data);
7
8    string(const string& rhs);              // A Copy constructor
9    string& operator=(const string& rhs);   // B Copy assignment operator
10
11   string(string&& rhs);                   // C Move constructor
12   string& operator=(string&& rhs);        // D Move assignment operator
13
14   char* c_str() const { return mData.get(); }
15 };
```

The place of a potential performance win

```
1  string::string(const string& rhs)
2  : mLen{rhs.mLen}, mData{std::make_unique<char[]>(rhs.mLen)}
3  {
4    std::copy_n(rhs.mData.get(), mLen, mData.get());
5  }
6
7  string& string::operator=(const string& rhs)
8  {
9    if (&rhs != this) {
10     mLen  = rhs.mLen;
11     mData = std::make_unique<char[]>(mLen);
12     std::copy_n(rhs.mData.get(), mLen, mData.get());
13   }
14
15   return *this;
16 }
```


The place of a potential performance win

```
1  string::string(string&& rhs)
2  : mLen{std::exchange(rhs.mLen, 0)}, mData{std::exchange(rhs.mData, nullptr)}
3  {}
4
5  string& string::operator=(string&& rhs)
6  {
7    if (&rhs != this) {
8      mLen  = std::exchange(rhs.mLen, mLen);
9      mData = std::exchange(rhs.mData, std::move(mData));
10   }
11
12   return *this;
13 }
```


https://AndreasFertig.com
post@AndreasFertig.com

A moved from object is nothing special

```
1 string src{"Hello"};
2
3 string other{std::move(src)};
4
5 std::cout << src.c_str();
```

- After applying `std::move`, `other` becomes a *moved-from* object.
- Such an object is in a valid, yet unknown state.
- In general there are two types of move across programming languages:
  - destructive move
  - non-destructive move
- C++ implements the non-destructive move.
- The issue: Such an object is in a *valid, yet unknown state*.
  - Before we can use such an object we must bring the object in a *valid and known state*.
  - A moved-from object must be at least destroyable and assignable.
  - Every additional operation is up to the type author.

A moved from object is nothing special

Decisions, decisions, decisions...

```
1  string::string(string&& rhs)
2  : mLen{std::exchange(rhs.mLen, 0)}, mData{std::exchange(rhs.mData, nullptr)}
3  {}
4
5  string& string::operator=(string&& rhs)
6  {
7    if (&rhs != this) {
8      mLen  = std::exchange(rhs.mLen, 0);
9      mData = std::move(rhs.mData);
10   }
11
12   return *this;
13 }
```

A moved from object is nothing special

- Simple Rule: Never touch a moved-from object.
- You know what you're doing rule: You can reuse a moved-from object once you brought the object back in a valid and known state. For all data types, assigning a new value to the moved-from object is a safe operation.

The Standard Template Library (STL), move, and custom object

```
1  struct Object {
2    Object()                           { printf("ctor\n"); }
3    Object(const Object&)              { printf("copy ctor\n"); }
4    Object& operator=(const Object&) { printf("copy assign\n"); return *this; }
5    Object(Object&&)                   { printf("move ctor\n"); }
6    Object& operator=(Object&&)        { printf("move assign\n"); return *this; }
7  };
8
9  int main()
10 {
11   std::vector<Object> v{};
12
13   v.push_back(Object{});
14
15   printf("second element\n");
16   v.push_back(Object{});
17 }
```

The STL, move, and custom object

```
1  struct Object {
2    Object()                                  { printf("ctor\n"); }
3    Object(const Object&)                     { printf("copy ctor\n"); }
4    Object& operator=(const Object&)          { printf("copy assign\n"); return *this; }
5    Object(Object&&)                noexcept  { printf("move ctor\n"); }
6    Object& operator=(Object&&)     noexcept  { printf("move assign\n"); return *this; }
7  };
8
9  int main()
10 {
11   std::vector<Object> v{};
12
13   v.push_back(Object{});
14
15   printf("second element\n");
16   v.push_back(Object{});
17 }
```


`std::move` is not always the right thing

```
1  void Innocent(std::string& value)
2  {
3    std::string local = std::move(value);
4  }
5
6  void Use()
7  {
8    std::string s{"A very very very very very very looooooong "
9                  "string to defeat the small string "
10                 "optimization SSO; hopefully."s};
11
12   Innocent(s);
13
14   std::cout << "'" << s << "'";
15 }
```

std::move is not always the right thing

```cpp
void Innocent(std::string& value)
{
  std::string local = std::move(value);
}

void Use()
{
  std::string s{"A very very very very very very looooooong "
                "string to defeat the small string "
                "optimization SSO; hopefully."s};

  Innocent(s);

  std::cout << "'" << s << "'";
}
```

```
$ ./a.out
''
```


std::move is not always the right thing

```cpp
template<typename T>
void Innocent(T&& value)
{
  std::string local = std::move(value);
}

void Use()
{
  std::string s{"A very very very very very very looooooong "
                "string to defeat the small string "
                "optimization SSO; hopefully."s};

  Innocent(s);

  std::cout << "'" << s << "'";
}
```

std::move is not always the right thing

```cpp
template<typename T>
void Innocent(T&& value)
{
  std::string local = std::forward<T>(value);
}

void Use()
{
  std::string s{"A very very very very very very looooooong "
                "string to defeat the small string "
                "optimization SSO; hopefully."s};

  Innocent(s);

  std::cout << "'" << s << "'";
}
```

```
$ ./a.out
'A very very very very very very looooooong string to defeat the small string /
    optimization SSO; hopefully.'
```

move or forward

Signature	Action
`void Fun(Object p);`	`x = std::move(p);`
`template<class T> void Fun(T&& p);`	`x = std::forward<T>(p);`
`Object o{/* */};`	`x = std::move(o);`
`auto&& o{/* */};`	`x = std::forward<decltype(o)>(o);`

- Use `std::forward` when you have a template parameter for the thing you want to pass around efficiently.
- Use `std::move` only if the above does not apply, you have a fixed type, and the type is not a reference.

Perfect forwarding

```cpp
struct Apple {
  Apple(const std::string& s) { std::cout << "lvalue: " << s << '\n'; }

  Apple(std::string&& s) noexcept { std::cout << "rvalue: " << s << '\n'; }
};

template<typename T, typename U>
auto make(U&& value)
{
  return T{ std::forward<U>(value) };
}

void Use()
{
  std::string str{"Hello"};

  Apple x{make<Apple>(str)};
  Apple y{make<Apple>("World"s)};
}
```

```
$ ./a.out
lvalue: Hello
rvalue: World
```

Use move only rarely

- In general, the compiler is our friend.
- Temporary objects are automatically moved.
- For return values, the compiler might apply optimizations such as copy-elision. You don't beat copy-elision with move!

https://AndreasFertig.com
post@AndreasFertig.com

Compiler-generated special members

```
class Object {
  int* _data{};

public:
  Object() : _data{new int{6}} {}
  ~Object() { delete _data; }
};
```


Compiler-generated special members

```
class Object {
  int* _data{};

public:
  Object() : _data{new int{6}} {}
  ~Object() { delete _data; }

  Object(const Object& rhs) : _data{new int{*rhs._data}} {}

  Object& operator=(const Object& rhs)
  {
    if (&rhs != this) {
      delete _data;
      _data = new int{*rhs._data};
    }

    return *this;
  }
};
```

Compiler-generated special members

			compiler implicitly declares					
					copy		move	
			default ctor	dtor	ctor	assignment	ctor	assignment
user declares		Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
		Any ctor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
		default ctor	*user declared*	defaulted	defaulted	defaulted	defaulted	defaulted
		dtor	defaulted	*user declared*	defaulted	defaulted	not declared	not declared
	copy	ctor	not declared	defaulted	*user declared*	defaulted	not declared	not declared
		assignment	not declared	defaulted	defaulted	*user declared*	not declared	not declared
	move	ctor	not declared	defaulted	deleted	deleted	*user declared*	not declared
		assignment	not declared	defaulted	deleted	deleted	not declared	*user declared*

Source: [1]

Utilizing move semantics even more: ref-qualifiers

```
1 class string {
2   size_t                  mLen{};
3   std::unique_ptr<char[]> mData{};
4
5   void Concat(const char* s);
6 public:
7   string(const char* data);
8   string(const string& rhs);
9   string& operator=(const string& rhs);
10  string(string&& rhs);
11  string& operator=(string&& rhs);
12  char*   c_str() const { return mData.get(); }
13
14  string& append(const char* s)
15  {
16    Concat(s);
17    return *this;
18  }
19 };
```

```
1 string s{"Hello"};
2 s.append(", world!");
3
4 std::cout << s.c_str();
```

Utilizing move semantics even more: ref-qualifiers

```
class string {
  size_t                  mLen{};
  std::unique_ptr<char[]> mData{};

  void Concat(const char* s);
public:
  string(const char* data);
  string(const string& rhs);
  string& operator=(const string& rhs);
  string(string&& rhs);
  string& operator=(string&& rhs);
  char*   c_str() const { return mData.get(); }

  string& append(const char* s)
  {
    Concat(s);
    return *this;
  }
};
```

```
string s{"Hello"};
s.append(", world!");

std::cout << s.c_str();

string s2 = string{"Hello"}.append(", world!");

std::cout << s.c_str();
```

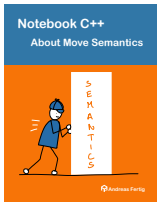

Utilizing move semantics even more: ref-qualifiers

```
class string {
  size_t            mLen{};
  std::unique_ptr<char[]> mData{};

  void Concat(const char* s);

public:
  string(const char* data);
  string(const string& rhs);
  string& operator=(const string& rhs);
  string(string&& rhs);
  string& operator=(string&& rhs);
  char*   c_str() const { return mData.get(); }

  string& append(const char* s) &
  {
    Concat(s);
    return *this;
  }

  string&& append(const char* s) &&
  {
    Concat(s);
    return  std::move(*this) ;
  }
};
```

```
string s{"Hello"};
s.append(", world!");

std::cout << s.c_str();

string s2 = string{"Hello"}.append(", world!");

std::cout << s.c_str();
```

}

I am Fertig.



fertig.to/babm

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 11.1.0
  - clang version 14.0.0 (https://github.com/tru/llvm-release-build fc075d7c96fe7c992dde351695a5d25fe084794a)

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

References

[1] Hinnant H., "Everything You Ever Wanted To Know About Move Semantics (and then some)", *ACCU*, Apr. 2014. https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

Images:

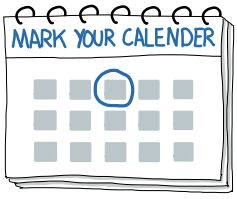
3: Franziska Panter
4: Franziska Panter
39: Franziska Panter

Upcoming Events

Training Classes

- *Programming with C++20*, CppCon, September 19 - 21
- *Programmieren mit C++20*, Andreas Fertig, October 05 - 07
- *C++1x for Embedded Systems*, QA Systems, October 18 - 21

For my upcoming talks you can check https://andreasfertig.com/talks/.
For my courses you can check https://andreasfertig.com/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.com/newsletter/.



https://AndreasFertig.com
post@AndreasFertig.com

About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 20.
Andreas is involved in the C++ standardization committee, in which the new standards are developed. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus understand constructs even better.
Before working as a trainer and consultant, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
You can find Andreas online at andreasfertig.com.