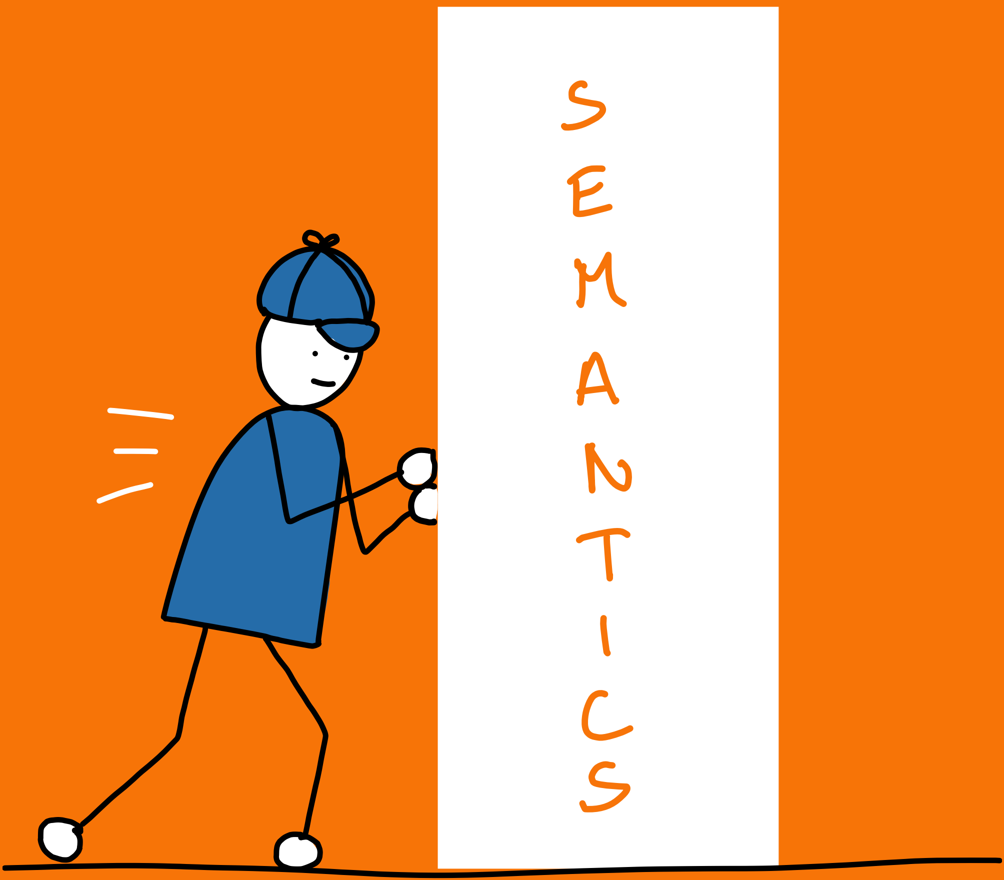
Notebook C++

About Move Semantics



Andreas Fertig

Andreas Fertig

Notebook C++

About Move Semantics

1. Edition

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

Planning and text:
Andreas Fertig

Cover art and illustrations:
Franziska Panter
https://franziskapanter.com

Published by:
Fertig Publications
https://andreasfertig.com

ISBN: 978-3-949323-03-4

This book is available as ebook at https://leanpub.com/notebookcpp-tips-and-tricks-with-templates

To Franziska, without her, I would not have accomplished this project. Never tired of reminding me of my talents, driving me when I'm tired, keeping my focus on. A lot of more could be written here, I like to close with: Thank You!

Foreword

This book is part of a series which is called *Notebook C++*. The idea is that most of us have some notes about do's and don'ts, how stuff works, or tips and tricks to keep in mind. It is probably one of the most frequent questions I get during training classes. I have such a list too. In this series, I will publish mine.

My idea is to create multiple short books (ok what is the number of pages required to call it a book or short?) about various topics. I currently plan to share tips about templates (this book), lambdas, and trap-like situations like dangling references. There will probably be more. They are available for early birds on Leanpub. Later they will also be available as a printed version.

Why several short books and not a single large one? Simply to give you a choice. Maybe, you are already fine with one topic but have an interest in tips for another topic. Why then by a large book where you need only a portion of it? Another thing is that I personally like printed books. There I find smaller ones more comfortable when carrying them around like on a train or airplane. Plus, they are not that heavy then, which is also an advantage.

Stuttgart, September 2022 *Andreas Fertig*

Using Code Examples

This book exists to assist you during your daily job life or hobbies. All examples in this book are released under the MIT license.

The main reason for choosing the MIT license was to avoid uncertainty. It is a well-established open-source license and comes with few restrictions. That should make it easy to use it even in closed-source projects. If you need a dedicated license or have questions about the existing licensing, feel free to contact me.

Code download

The source code for this book's examples is available at
`https://github.com/andreasfertig/notebookcpp-about-move-semantics`.

Used Compilers

For those of you who would like to try out the code with the same compilers and revisions I used, here you go:

- GCC 13.2.0
- Clang 17.0.0

About the Author

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 23.

Andreas is involved in the C++ standardization committee, developing the new standards. 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 (cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus to understand constructs even better.

Before training and consulting, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.

You can find him online at andreasfertig.com and his blog at andreasfertig.blog.

About the Book

The idea of the *Notebook C++* series is to share some tips and tricks about various C++ elements. All books in this series are somewhat short and small books, one for each major topic. Such that the paperback version can be carried around easily.

About Move Semantics

This part of the series is all about move semantics. I will introduce the feature to you in an maybe unconventional but easy-to-understand way. Move semantics are nothing special, afterall.

We'll start looking at what move semantics is, how it works, and why we should most times stay away from `std::move`. We establish some rules about when to use `std::move`, when `std::forward`. You learn about why not to move return values or temporary objects. You want to get the best speed from your custom data type and the Standard Template Library (STL)? No problem, you will learn what your class must look like to achieve this.

In the end, you also learn about some feature that is not seen that often, ref-qualifiers, how they work, why they are there, and when to use them.

All in all, after having read this book, you have a solid understanding of move semantics.

The books style

The book corresponds to my general style, which is a mix of explaining and sharing code examples for better illustration and understanding. As I'm the creator of C++ Insights , which I created to be able to show more things rather than just telling, there will be examples that peek behind the scenes.

The headings are inspired by Scott Meyers *Effective C++* series [1].

As we have the joy to have multiple standards, we also need a way to address which standard is used. This is something I often experience when customizing my classes together with customers. They have only C++14 available. Or the plan to upgrade to C++17 but currently are on C++11. In this book, I use my experience and my system from my classes and talks. Therefore, my slides have a small marker on the upper right corner, stating which standard the functionality belongs too. The default assumption there is that it is C++11. However, a couple of Notes refer to C++98 and are still valid in newer versions of C++. All Notes are labeled with the standard it can first be used, for example, **C++11**. There are also overview pages for fast navigation and to skip promising Notes which are impossible because the standard is not available.

Style and conventions

As we are blessed with multiple versions of the ISO C++ standard, we need a way to specify the exact version of ISO C++ to which we are referring at any given time. I often run into this when customizing my classes with customers. Some may only have C++14, while others may be planning to upgrade to C++17 while temporarily sticking to C++11. In this book, I draw from my experiences and use a system I've developed through providing classes and talks. In presentations, my slides each have a small marker in the upper right corner indicating the standard to which the demonstrated functionality belongs. The default assumption there is that the code is C++11. However, a couple of Notes in this book refer to C++98 and are still valid in newer versions of C++. All Notes are labeled with the standard it can first be used, for example, C++11. There are also overview pages that allow you to navigate more quickly or skip otherwise promising Notes that may be inapplicable due to the standard to which you're bound.

The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in `a.out` as the program name.

```
$ ./a.out
Hello, Notebook C++!
```
Output

- `<string>` stands for a header file named `string`.
- `[[xyz]]` marks a C++ attribute with the name `xyz`.

Feedback

This book is published on Leanpub as a digital version. The printed version will follow. If you can specify the book type you're referencing when sending feedback, that would be a huge help.

In any case, I appreciate your feedback. Please report it to me, whether it be a typo, a grammatical error, an issue with naming variables or functions, or another logical concern. You can send your feedback to books@andreasfertig.com.

PDF/Paperback vs. epub

As with most of my material, the book is written in LaTeX. The epub version is generated by a custom script which first translates LaTeX into Markdown and then translates Markdown into epub with the help of pandoc. This comes with some limitations. Currently, the bibliography does not use the same style as the PDF, and the index is missing in the epub.

Another issue I have with the epub is that I do not own a reader device myself. I tested it with Apple’s Books.However, please tell me if you have better knowledge of optimizing the output.

Revision History

2022-09-12: First release (Leanpub)
2024-01-18: More notes and updates (Leanpub)

About the Tools

In this book, I will use two tools that you can use to verify the results of playing with different versions and combinations of the examples you find in this book.

Compiler Explorer

A website created by Matt Godbolt: `https://compiler-explorer.com`.

Initially, Compiler Explorer showed the assembly output resulting from a user's C+ code snippet. After some time, Compiler Explorer became an online Integrated Development Environment (IDE) with a wide variety of features. The website has a whole lot of compilers that you can use to compile your source code online. One of the newer features is the ability to execute your code online, optionally, by passing command line options.

C++ Insights

The following is a tool with a website I created: `https://cppinsights.io`.

C++ Insights is a clang Abstract Syntax Tree (AST) based source-to-source transformation tool. It shows C++ after the font-end stage of the compiler. With that, C++ Insights shows code from the point of view of a compiler. Making implicit conversions or template instantiation visible are just two among a lot of things it does. It is available as a command-line utility and as a website.

Table of Contents

Notes by Standard at a Glance **19**

Notes belonging to C++11 19

Notes belonging to C++17 20

Notes belonging to C++20 21

2 Move Semantics **23**

Note 1: Understand the type of move used in C++ 25

Note 2: Move is nothing special 27

Note 3: Move vs. copy 31

Note 4: Move is a partial swap 35

Note 5: `std::move` doesn't move 37

Note 6: Understand the value categories 39

Note 7: Make your rvalue parameters modifiable 43

Note 8: Only classes with dynamic memory profit from move semantics 45

Note 9: A moved-from object isn't special 47

Note 10: Never use `std::move` on a return value 49

Note 11: Remember to forward the move to all the base classes 51

Note 12: Never use `std::move` on a temporary 55

Note 13: When is it a forwarding reference 59

Note 14: When to use `std::forward` 61

Note 15: Your custom class and the STL 65

Note 16: Known when you lose a special member 69

Note 17: Even with a defaulted destructor, you lose the move operations 71
Note 18: Be aware of `std::initializer_list` 73
Note 19: Test for noexcept move 79
Note 20: Use ref-qualifiers for more efficiency 81
Note 21: Use reference qualifiers on assignment operators 85
Note 22: Understand user-declared and user-defined 89

Acronyms 93

Bibliography 95

Index 97

Notes belonging to C++11

Note 1: Understand the type of move used in C++ 25
Note 2: Move is nothing special 27
Note 3: Move vs. copy 31
Note 4: Move is a partial swap 35
Note 5: `std::move` doesn't move 37
Note 6: Understand the value categories 39
Note 7: Make your rvalue parameters modifiable 43
Note 8: Only classes with dynamic memory profit from move semantics 45
Note 9: A moved-from object isn't special 47
Note 10: Never use `std::move` on a return value 49
Note 11: Remember to forward the move to all the base classes 51
Note 12: Never use `std::move` on a temporary 55
Note 13: When is it a forwarding reference 59
Note 14: When to use `std::forward` 61
Note 15: Your custom class and the STL 65
Note 16: Known when you lose a special member 69
Note 17: Even with a defaulted destructor, you lose the move operations 71
Note 18: Be aware of `std::initializer_list` 73
Note 19: Test for noexcept move 79
Note 20: Use ref-qualifiers for more efficiency 81
Note 21: Use reference qualifiers on assignment operators 85
Note 22: Understand user-declared and user-defined 89

Notes belonging to C++17

Notes belonging to C++20

Part 2

Move Semantics

In this part, we discuss

- how move semantics works
- what you have to do to enable move semantics for your data-types
- how to use move semantics efficiently
- ... and more

“ … The only difference is that copying from `a` won’t change `a`, but moving from `a` might. … ”

— Sutter [2]

Note 1: Understand the type of move used in C++ C++11

Before diving in with various other tips, let's get a fundamental understanding of move semantics in C++.

In programming languages, in general, we have two different types of move semantics:

- destructive move
- non-destructive move

In C++, we have the latter, but let's look into the first form first.

A destructive move means that once we move an object, we will see later what that means and how to do it in C++; the source object gets destroyed afterward.

The destructive move approach has a couple of advantages. As the source object gets destroyed immediately after the move, regardless of whether the end of the scope is reached, we cannot touch it any longer. One of the biggest questions about move semantics in C++ is the so-called *moved-from state*, which you learn about in **Note 9**.

Aside from the inability to touch the source object, we do not have to worry about the state of the moved-from object. Another great advantage.

We don't have these advantages in C++ since we have a non-destructive move in C++. That means that the source object follows the standard lifetime rules of C++, which say that an object is destroyed either at the end of a full expression or at the end of the scope in which the object was declared. There are scenarios, like in a return statement, where we cannot touch the source object after the move since we go out of scope. However, plenty of other scenarios allow us in C++ to still touch the source object.

What does it mean to still have access to the source object? Well, since you can touch the source object after the move, the question about what the object's state is pops up. The one follows the other. So, is the source object still a valid object? Are its invariants still intact? These are the two biggest questions when it comes to a move-from object.

While a destructive move does not have to care about that, in C++, we must ensure that the move-from object is at least destroyable since that will happen at the end of the object's scope or lifetime in general.

Whether we do anything more than that is an implementation detail, making reasoning about a move-from object hard.

All that said, there is an advantage to a non-destructive move. We can implement a move as a constant operation. While a move operation more or less swaps pointers, the source object gets destroyed at some point. In C++, we can delay this operation, making the move itself a constant operation. For example, `delete` isn't constant as the time to really get the okay from your STL depends on the algorithm used to implement the dynamic memory management, potentially the memory currently consumed, and things like getting access to the global memory. All that can be avoided in a move constructor or move assignment operator, but this is a decision we can make.

If you reach difficulties with the moved-from state in C++, start treating all moved-from objects as if C++ had a destructive move and don't use them anymore. Only if you think there is performance wasted measure and see if reusing the moved-from objects gets you better performance.

Note 2: Move is nothing special C++11

While move semantics can positively impact your application’s performance, move semantics isn’t anything special.

“ The entire point of move semantics is to boost performance. ”

— Hinnant [3]

Keep that in mind because, in the end, move semantics is an optimization that you can apply but don’t have to. Yet, who wants slow code?

Before starting with move semantics, have a look at the following code example:

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

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

Listing Note 2.1

What you see here is function overloading, right? The first implementation of `Fun` takes its parameter `byRef` by reference. As you know, the reference allows the function to modify the contents of the parameter.

The second function `Fun`, the overload, takes the parameter as a `const` reference. We use this pattern to avoid copies while still preventing the source from modification.

I’m sure that’s all stuff you already know. The result is no surprise if we execute the following calls to `Fun`.

```cpp
void Use()
{
  std::vector       v{2, 3, 4};
  const std::vector cv{5, 6, 7};
```

Listing Note 2.2

```

  Fun(v);           A We pass a lvalue
  Fun(cv);          B We pass a const lvalue
  Fun({3, 5, 6});   C We pass a temporary
}
```

Listing Note 2.2

What's interesting in this example is C. Here, a temporary is passed as an argument to `Fun`. This call ends up selecting the `const &` overload of `Fun`, which is one additional reason to often use a `const` reference because this signature also binds to temporary objects. Hence, the following output is no surprise and makes us happy:

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

Output

Assume that `Fun` does take an object that acts like a container holding a pointer to dynamically allocated memory, like a `std::vector` or `std::string` as a parameter. Of course, there is nothing wrong with that change. `Fun` still works fine. You don't have to change a thing and can still be happy.

Remember, from the beginning, move semantics is a performance optimization. While you can be happy as your code works correctly, there is a potential for optimizing this code. We are now at a point where move semantics makes or can make a difference. As with all optimizations, the impact depends on your data.

The optimization we can apply is doing something useful with the temporary object. After all, there is a difference between an object we still use outside of `Fun`, which we want to protect from changes while passed around, and an object that we create in place only to pass this temporary object to `Fun`.

Well, we have overloads for a modifiable version of the parameter of `Fun`, and we have another overload for the `const` but still used outside case. What about adding another overload that takes care of the temporary object case? This is where move semantics comes to play. For the move semantics overload to get selected, we have to use the double ampersands `&&`:

Listing Note 2.3

```
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";
}
```

Notice that in the example, there is now a third version of `Fun` taking the `std::vector` by `&&`. Notice also that the parameter here is modifiable. It's not `const`. You can read more about why in **Note** 7.

Executing this code leads to the following output:

Output

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

As you can see, all overloads are now used. The temporary object case C picks the `byMoveRef` variant.

So, a straightforward way to see all these new tokens and their behavior is simply to have another overload that allows selective selection of temporary objects.

Inside that overload function, you can now use the contents of the temporary object in a more meaningful way than by taking the object as a `const` reference.

The great thing so far is that simply providing the function overload that can take advantage of the temporary object already gives you performance. We haven't done anything else special. Utilizing temporary objects is usually the most significant factor in terms of performance optimization.

Note 3: Move vs. copy C++11

So far, we have established that move semantics is nothing special. A move operation is a performance operation that we can address by writing the according function overload.

Let's revisit the performance gain. Say you're about to start a job at a new company in a different city, maybe also a continent, than you're currently living. Further, let's assume the company pays you a fortune of money. More than enough that you can consider affording two apartments. Keeping the existing apartment but renting a new one in the new company's city. Two options to carefully weigh. And congratulations on the new job!

Figure Note 3.1 illustrates your two options. On the left side, there is the copy option. You go for two apartments. Should you also enjoy spending time in furniture stores, that's probably a good choice. However, in terms of costs, you must spend hours there, purchasing all the stuff you already have. Maybe the money you spend there isn't the biggest factor, remember the new company pays you a fortune, but you also have to spend hours on all the purchases. Two resources are involved here.

Well, do you prefer spending the fortune on something better? Maybe environmental safety? Good. That would be the right side of Figure Note 3.1, the move. That's a common term: when we relocate from A to B, we move. Compared to the apartment duplication option above, a move is cheap in terms of money and time. You can hire a moving company. They tell you when they are available and how long the shipment from A to B will take. If you hire a decent moving company, the costs and the time will always be less than buying all the things again.

We can further say that the move option is constant in time and cost. Yes, the time may vary with the relocation distance, but it is constant compared to the hours you spend in furniture stores.

You can transfer this analogy to C++ when it comes to copy vs. move. Move semantics in C++ gives us a new option; previously, we had to copy, but now we can decide to move.

Here is the copy vs. move comparison in the code. A copy is illustrated in Listing Note 3.2.

Acronyms

API	Application Programming Interface.
AST	Abstract Syntax Tree.
CPL	Combined Programming Language.
IDE	Integrated Development Environment.
RVO	return value optimization.
SSO	Small String Optimization.
STL	Standard Template Library.
UB	Undefined Behavior.

Bibliography

[1] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.* O'Reilly Media, 2014.

[2] H. Sutter, "Move, simply." [Online]. Available: https://herbsutter.com/2020/02/17/move-simply/

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

[4] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey, "The Main Features of CPL," *The Computer Journal*, vol. 6, no. 2, pp. 134–143, 08 1963. [Online]. Available: https://doi.org/10.1093/comjnl/6.2.134

[5] B. Stroustrup, ""new" value terminology." [Online]. Available: https://www.stroustrup.com/terminology.pdf

[6] D. Abrahams, *Exception-Safety in Generic Components*. Generic Programming: International Seminar on Generic Programming Dagstuhl Castle, April 2000, pp. 69–79.

[7] "Compiler explorer - user-provided destructor." [Online]. Available: https://compiler-explorer.com/z/qa9o8ssKz

Index

A

API 85
AST 15

C

C++
  Insights 12
copy elision 49, 57
CPL 39

I

IDE 15

L

lvalue 37–40, 59, 81
  object 83

M

move semantics 25, 28, 39, 40, 45, 47, 49, 55, 59, 81

O

object
  temporary . . 29, 36, 43, 55, 57, 71, 82, 84, 86

P

perfect
  forwarding 73, 77

R

reference
  forwarding 59, 61, 62
  lvalue 37, 43
  rvalue 37, 41, 59, 83
rvalue 37–40, 43, 59, 81
  object 83
RVO 49

S

SSO 75
std::forward 63
std::move 37, 40, 47–49, 52, 55–57, 61, 63
std::move_if_noexcept 79, 80
STL 11, 26, 48, 66, 67, 79

U

UB 61