C++ Move Semantics

The Complete Guide

Nicolai M. Josuttis

C++ Move Semantics - The Complete Guide

First Edition

Nicolai M. Josuttis

This version was published on **2020-08-12**.

This book was typeset by Nicolai M. Josuttis using the LaTeX document processing system.

This book is for sale at **http://leanpub.com/cppmove**.

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book, and build traction once you do.

Contents

Preface

Move semantics, introduced with C++11, has become a hallmark of modern C++ programming. However, it also complicates the language in many ways. Even after several years of support, experienced programmers struggle with all the details of move semantics, style guides still recommend different consequences for programming even of trivial classes, and in the C++ standard committee we still discuss semantic details.

Whenever I taught what I learned about C++ move semantics so far, I said "*Somebody has to write a book about all this*" and the usual answer was: "*Yes, please do!*" So, I finally did.

As always when writing a book about C++, I was surprised about the amount of aspects to teach, the situations to clarify, and the consequences to describe. It was really time to write a book about all aspects of move semantics covering all C++ versions from C++11 until C++20. I learned a lot and am sure you will also do.

An Experiment

This book is an experiment in two ways:

- I am writing an in-depth book covering a complex core language feature without the direct help of a core language expert as a co-author. However, I can ask questions and I do.
- I am publishing the book myself on Leanpub and for printing on demand. That is, this book is written step by step and I will publish new versions as soon there is a significant improvement that makes the publication of a new version worthwhile.

The good thing is:

- You get the view of the language features from an experienced application programmer—somebody who feels the pain a feature might cause and asks the relevant questions to be able to motivate and explain the design and its consequences for programming in practice.
- You can benefit from my experience with move semantics while I am still writing.
- This book and all readers can benefit from your early feedback.

That means, you are also part of the experiment. So help me out: give feedback about flaws, errors, features that are not explained well, or gaps so that we all can benefit from these improvements.

Versions of This Book

Because this book is written incrementally, the following is the history of its major updates (newest first):

- **2020-08-10**: Describe move semantics for shared pointers.
- **2020-08-10**: Motivate the re-use of moved-from objects and clarify move assignments to itself.
- **2020-08-09**: Recommend not to return by-value with `const`.
- **2020-07-27**: Move semantics for strings and containers.
- **2020-07-24**: Add general remarks on using reference qualifiers.
- **2020-07-23**: Describe details of type deduction of universal references and reference collapsing for perfect forwarding.
- **2020-07-07**: Describe perfect returning for lambdas.
- **2020-06-21**: Provide the chapter about perfect returning (with small fixes for perfect forwarding).
- **2020-06-12**: Improving the chapter about "invalid" states and move it to a different location.
- **2020-06-08**: Adding the chapter about using `noexcept`.
- **2020-06-06**: Improving the chapter about invalid states.
- **2020-05-31**: Fixing several typos from feedback by readers.
- **2020-05-03**: Fixing *move semantics in class hierarchies* and moving it to usage chapter.
- **2020-05-03**: Fixing several typos from feedback by readers.
- **2020-04-29**: Fixing move semantics for classes and invalid states after review.
- **2020-04-25**: Fix code layout and missing figures in non-PDF versions.
- **2020-04-22**: Adding the chapter about invalid moved-from states.
- **2020-04-20**: Fixing several typos from feedback by readers.
- **2020-04-13**: When you can't avoid using `std::move()`
- **2020-02-01**: When automatically generated move operations are broken
- **2020-01-19**: Constructors with universal references
- **2020-01-17**: Several small fixes from reader feedback
- **2020-01-08**: Review Chapter 2 and small fixes
- **2020-01-07**: Review preface and Chapter 1
- **2020-01-06**: A few first small fixes
- **2020-01-04**: The initial published version of the book

Acknowledgments

First of all I would like to thank you, the C++ community, for making this book possible. The incredible design of all the features of move semantics, the helpful feedback, and the curiosity are the basis for the evolution of a successful language. In particular, thanks for all the issues you told me about and explained and for the feedback you gave.

I would especially like to thank everyone who reviewed drafts of this book or corresponding slides and provided valuable feedback and clarification. These reviews increased the quality of the book significantly again proving that good things need the input of many "wise guys." For this reason, so far (this list is

still growing) huge thanks to Javier Estrada, Howard Hinnant, Klaus Iglberger, Daniel Krügler, Marc Mutz, Aleksandr Solovev (alexolut), Peter Sommerlad, and Tony Van Eerd.

In addition, I would like to thank everyone in the C++ community and on the C++ standardization committee. In addition to all the work involved in adding new language and library features, these experts spent many, many hours explaining and discussing their work with me, and they did so with patience and enthusiasm.

Special thanks go to the LaTeX community for a great text system and to Frank Mittelbach for solving my LaTeX issues (it was almost always my fault).

This page is intentionally left blank

About This Book

This book teaches C++ move semantics. Starting from the basic principles it motivates and explains all features and corner cases of move semantics so that you as a programmer can understand and use move semantics right. The book is valuable for those that are beginning to learn about move semantics and is essential for those that are using it already.

As usual for my books, the focus lies on the application of the new features in practice and the book will demonstrate how features impact day-to-day programming and how you can benefit from them in projects. This applies to both application programmers and programmers who provide generic frameworks and foundation libraries.

What You Should Know Before Reading This Book

To get the most from this book, you should already be familiar with C++. You should be familiar with the concepts of classes and references in general, and you should be able to write C++ programs using components such as IOstreams and containers from the C++ standard library.

However, you do not have to be an expert. My goal is to make the content understandable for the average C++ programmer, not necessarily knowing all details of the latest features. You should also be familiar with the basic features of "Modern C++", such as `auto` or the range-based `for` loop.

Nevertheless, I will discuss basic features and review more subtle issues as the need arises. This ensures that the text is accessible to experts and intermediate programmers alike.

Overall Structure of the Book

This book covers *all* aspects of C++ move semantics up to C++20. This applies to both language and library features as well as both features that affect day-to-day application programming and features for the sophisticated implementation of (foundation) libraries. However, the more general cases and examples usually come first.

The different chapters are grouped so that didactically you should read the book from the front to the back. That is, later chapters usually rely on features introduced in earlier chapters. However, cross-references also

help to start with specific later topics to find places where they refer to features and aspects introduced earlier.

As a result, the book contains the following parts:

- **Part I** covers the basic feature of move semantics (especially for non-generic code).
- **Part II** covers the generic features of move semantics (especially used in templates and generic lambdas).
- **Part III** covers the use of move semantics in the C++ standard library (giving a good example how to use move semantics in practice).

How to Read This Book

Don't be afraid by the number of pages this book has. As always with C++, things can become pretty complicated when you look into details (such as implementing templates). For a basic understanding the first third of the book (Part I, especially chapters 1 to 5) is enough to read.

In my experience, the best way to learn something new is to look at examples. Therefore, you will find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files on the website for this book at `http://www.cppmove.com`.

The Way I Implement

Note the following hints about the way I write code and comments.

Initializations

I usually use the modern form of initialization (introduced in C++11 as *uniform initialization*) with curly braces:

```
int i{42};
std::string s{"hello"};
```

This form of initialization, which is called *brace initialization*, has the advantage that it

- can be used with fundamental types, class types, aggregates, enumeration types, and `auto`,
- can be used to initialize containers with multiple values,
- is able to detect narrowing errors (e.g., initialization of an `int` by a floating-point value), and
- can't be confused with function declarations or calls.

If the braces are empty, the default constructors of (sub)objects are called and fundamental data types are guaranteed to be initialized with `0`/`false`/`nullptr`.

Error Terminology

I often talk about programming errors. If there is no special hint, the term *error* or a comment such as

```
...     // ERROR
```

means a compile-time error. The corresponding code should not compile (with a conforming compiler).

If I use the term *runtime error*, the program might compile but not behave correctly or result in undefined behavior (thus, it might or might not do what is expected).

Code Simplifications

I try to explain all features with helpful examples. However, to concentrate on the key aspects to teach, I might often skip other details that should be part of code.

- Most of the time I use an ellipsis ("...") to signal additional code that is missing. Note that I don't use code font here. If you see an ellipsis with code font, it is necessary to have these three dots as language feature (such as for "`typename...`").
- In header files I usually skip the preprocessor guards, all header files should have:

  ```
  #ifndef MYFILE_HPP
  #define MYFILE_HPP
  ...
  #endif  // MYFILE_HPP
  ```

  So, please beware and fix the code when using these header files in your projects.

The C++ Standards

C++ has different versions defined by different C++ standards.

The original C++ standard was published in 1998 and was subsequently amended by a *technical corrigendum* in 2003, which provided minor corrections and clarifications to the original standard. This "old C++ standard" is known as C++98 or C++03.

The world of "Modern C++" began with C++11 and was extended with C++14 and C++17. The international C++ standard committee now aims to issue a new standard every three years. Clearly, that leaves less time for massive additions, but it brings the changes to the broader programming community more quickly. The development of larger features, therefore, takes time and might cover multiple standards.

And the next "Even more Modern C++" is at the horizon, introduced already with C++20. Again, several ways to program will probably change. However, as usual, compilers need some time to provide the latest language features. At the time of writing this book, C++17 is usually the latest supported version by major compilers.

Fortunately, the basic principles of move semantics were all introduced with C++11 and C++14. For that reason, the code examples of this book should usually compile on recent versions of all major compilers. If special features introduced with C++17 or C++20 are discussed, I will explicitly mention that.

Example Code and Additional Information

You can access all example programs and find more information about this book from its website, which has the following URL:

http://www.cppmove.com

Feedback

I welcome your constructive input—both negative and positive. I have worked very hard to bring you what I hope you will find to be an excellent book. However, at some point I had to stop writing, reviewing, and tweaking to "release the new revision." You may therefore find errors, inconsistencies, presentations that could be improved, or topics that are missing altogether. Your feedback gives me a chance to fix these issues, inform all readers about the changes through the book's website, and improve any subsequent revisions or editions.

The best way to reach me is by email. You will find the email address at the website for this book:

http://www.cppmove.com

Please be sure to have the latest version of this book available (remember it is written and published incrementally) and refer to the publishing date of this version when giving feedback. The current publishing date is **2020-08-12** (you can also find it on page ii, the page directly after the cover).

Many thanks.

Part I

Basic Features of Move Semantics

This part of the book introduces the basic features of move semantics that are not specific to generic programming (i.e., templates). They are particularly helpful for application programmers in their day-to-day programming and therefore every C++ programmer using Modern C++ should know them.

Move semantics features for generic programming are covered in Part II.

This page is intentionally left blank

Chapter 1

The Power of Move Semantics

This chapter demonstrates the basic principles and benefit of move semantics using a short code example.

1.1 Motivation for Move Semantics

To understand the basic principles of move semantics, let's look at the execution of a little piece of code first without move semantics (i.e., compiled with an old C++ compiler supporting C++03 only) and then with move semantics (compiled with a modern C++ compiler supporting C++11 or later).

1.1.1 Example with C++03 (Before Move Semantics)

Assume we have the following program:

basics/motiv03.cpp

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

std::vector<std::string> createAndInsert()
{
  std::vector<std::string> coll;   // create vector of strings
  coll.reserve(3);                 // reserve memory for 3 elements
  std::string s = "data";          // create string object

  coll.push_back(s);               // insert string object
  coll.push_back(s+s);             // insert temporary string
  coll.push_back(s);               // insert string

  return coll;                     // return vector of strings
}
```

```
int main()
{
  std::vector<std::string> v;       // create empty vector of strings
  ...
  v = createAndInsert();            // assign returned vector of strings
  ...
}
```

Let's look at the individual steps of the program (inspecting both the stack and the heap), when we compile this program with an old C++03 compiler ***not*** supporting move semantics.

- First, in `main()` we create the empty vector `v`:

  ```
  std::vector<std::string> v;
  ```

  which is placed on the stack as an object having 0 as the number of elements and no memory allocated for elements.
- Then, we call

  ```
  v = createAndInsert();
  ```

  where we create another empty vector `coll` on the stack and reserve memory for three elements on the heap:

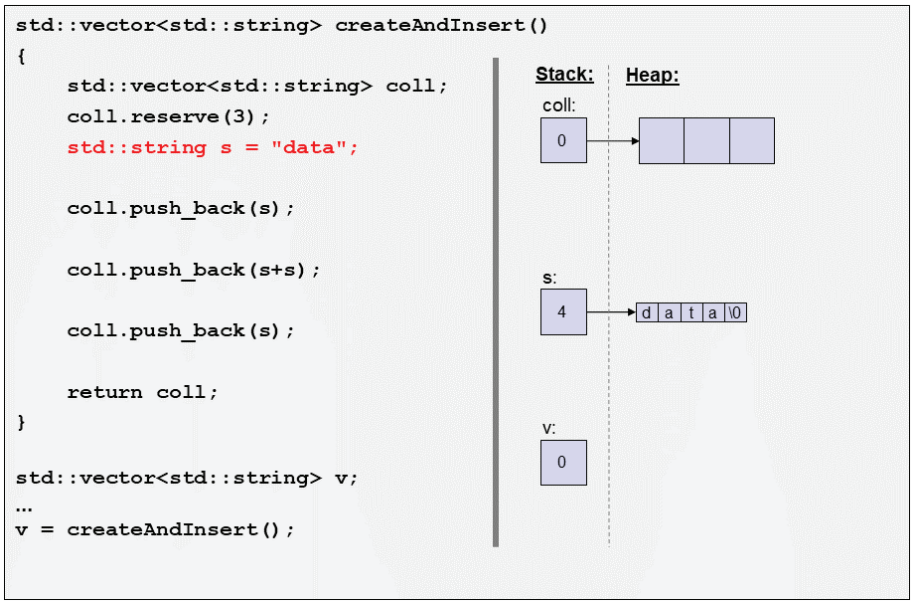
  ```
  std::vector<std::string> coll;
  coll.reserve(3);
  ```

  The allocated memory is not initialized, because the number of elements is still 0.
- Then, we create a string initialized with `"data"`. A string is something like a vector with `char` elements. So, essentially we create an object on the stack with a member for the size (having the value 4) and a pointer to the memory for the characters.[1]

  The whole program now has the following state: We have three objects on the stack: `v`, `coll`, and `s`. Two of them, `coll` and `s`, have allocated memory on the heap:[2]

[1] Strings internally usually also store a terminating null character to avoid allocating memory when they are asked for a C string representation of it with the member function `c_str()`.

[2] With the *small string optimization (SSO)*, the string `s` might store its whole value on the stack provided the value is not too long. However, for the general case, let's assume we don't have the small string optimization or the value of the string is long enough so that the small string optimization does not happen.

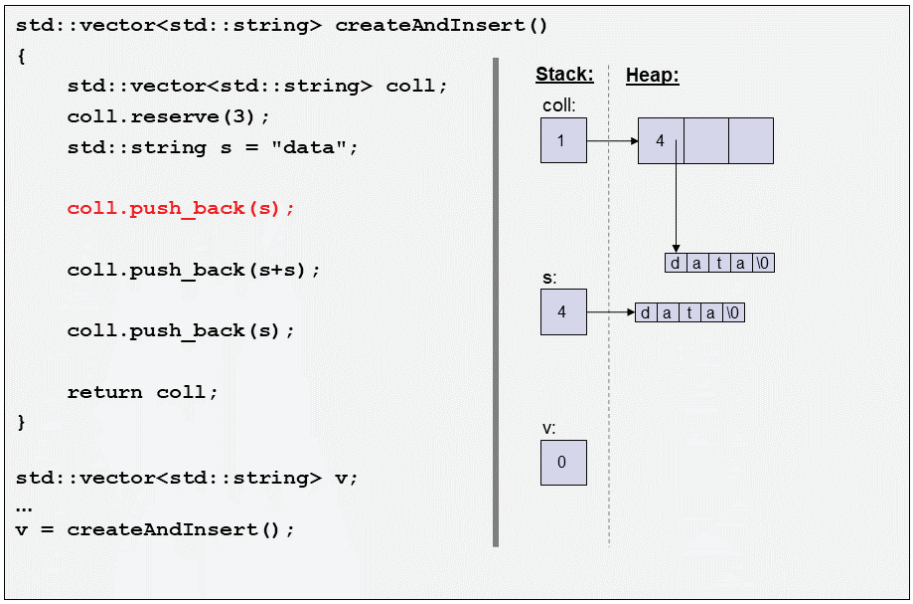


- The next step is the command to insert the string into the vector `coll`:

  ```
  coll.push_back(s);
  ```

  All containers in the C++ standard library have value semantics, which means that they create copies of the values passed to them.

  As a result, we get a first element in the vector which is a full (deep) copy of the passed value/object `s`:



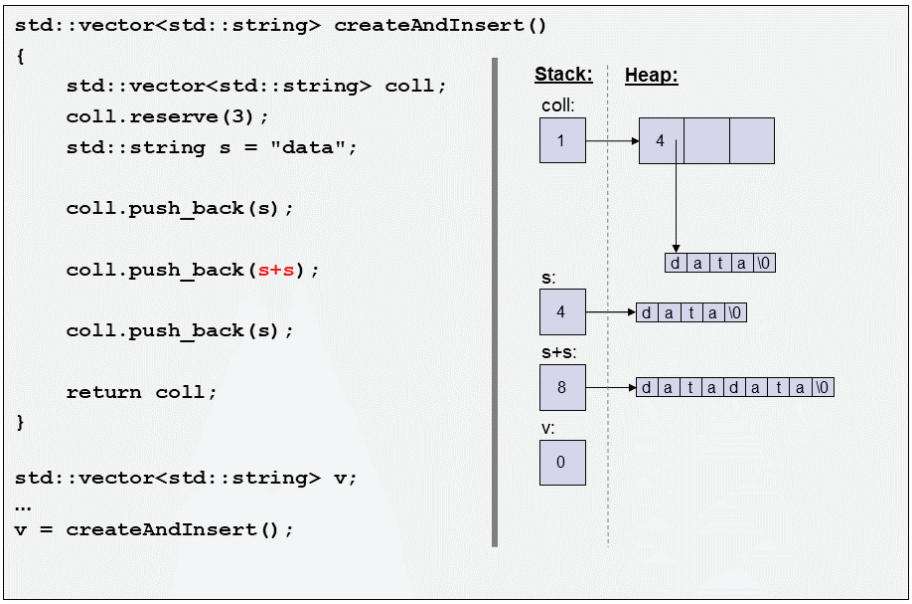
So far, in this program we have nothing to optimize. The current state is that we have two vectors, `v` and `coll`, and two strings, `s` and its copy, which is the first element in `coll`. They should all be separate objects with their own memory for the value, because modifying one of them should not impact any of the other objects.

- But now let's look at the next statement, which creates a new temporary string and again inserts it into the vector:

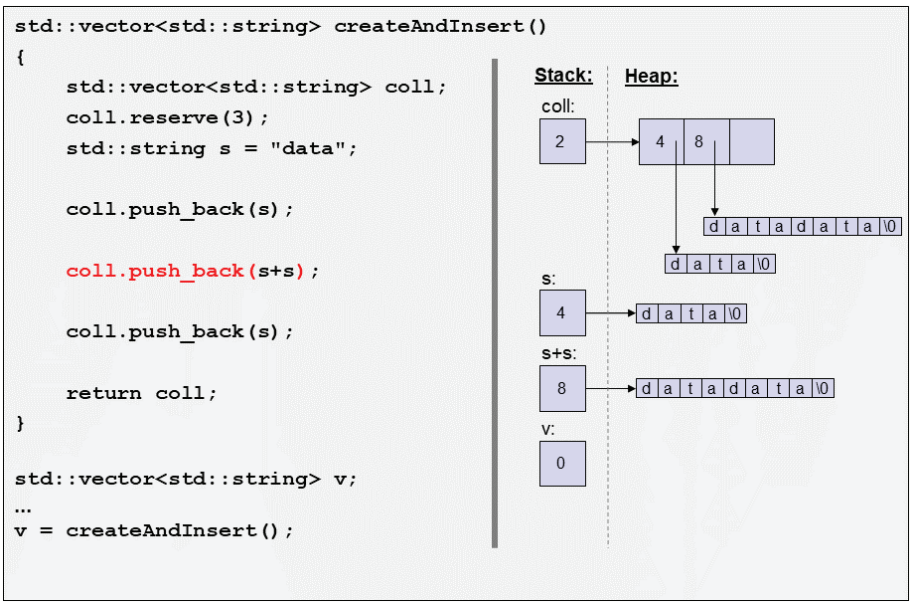
  ```
  coll.push_back(s+s);
  ```

  This statement is performed in three steps:

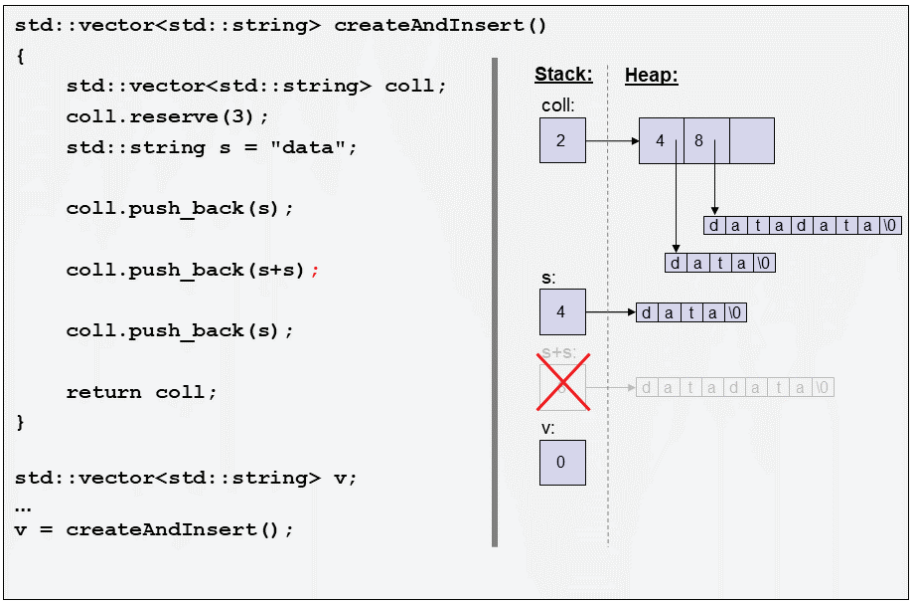
  – First, we create the temporary string `s+s`:



– Second, we insert this temporary string into the vector `coll`. As always, the container creates a copy of the passed value, which means that we create a deep copy of the temporary string, including allocating memory for the value:



– Last, at the end of the statement, the temporary string `s+s` is destroyed, because we no longer need it:

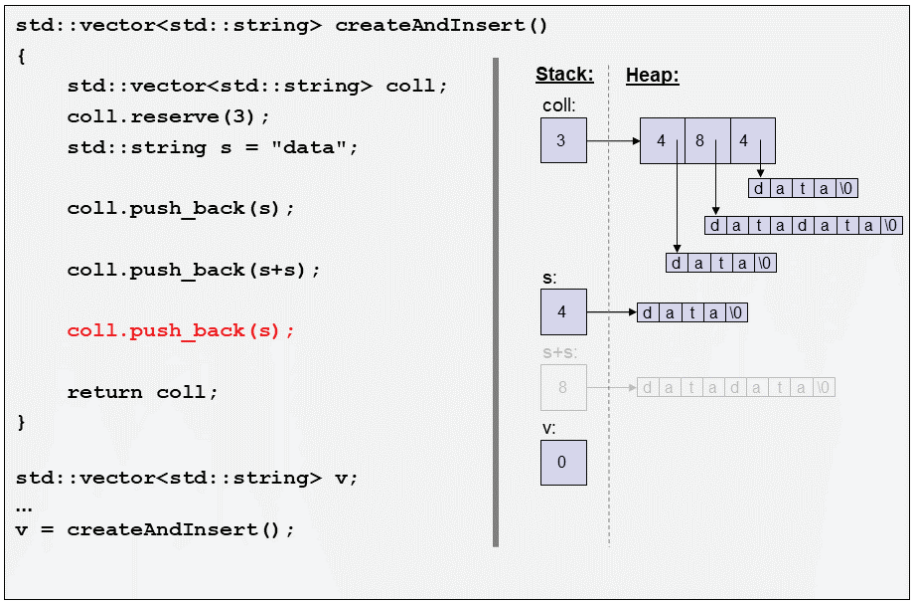


Here, we have the first moment where we generate code that is not performing well: we create a copy of a temporary string and destroy the source of the copy immediately afterwards, which means that we unnecessarily allocate and free memory, that we could have just moved from the source to the copy.

- With the next statement, again we insert `s` into `coll`:

  ```
  coll.push_back(s);
  ```

  Again, `coll` copies `s`:



This also is something to improve because the value of `s` is no longer needed. So we could at least see some optimization that handles this better.

- At the end of `createAndInsert()` we come to the return statement:

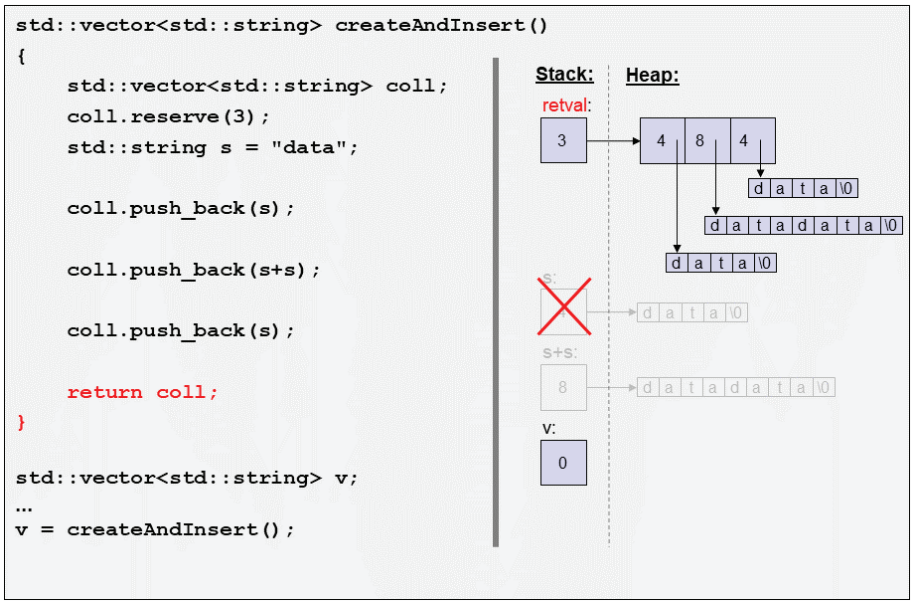
```
  return coll;
}
```

Here, it gets a bit more complicated. We return by value (the return type is not a reference), which in general should be a copy of the value in the `return` statement, `coll`. Creating a copy of `coll` means that we have to create a deep copy of the whole vector with all of its elements. Thus, we have to allocate heap memory for the array of elements in the vector and heap memory for the value each string allocates to hold its value. Here, we would have to allocate memory 4 times.

However, since at the same time `coll` gets destroyed because we leave the scope where it is declared, the compiler is allowed to perform the *named return value optimization* (*NRVO*). This means that the compiler can generate code so that `coll` is just used as the return value.

This optimization is allowed even if this would change the functional behavior of the program. If we had a print statement in the copy constructor of a vector or string, we would see that the program no longer has the output from the print statement. This means that this optimization changes the functional behavior of the program. But that's OK, because we explicitly allow this optimization in the C++ standard even if it has side effects. Nobody should expect that a copy is done here, in the same way that nobody should expect that it isn't, either. It is simply up to the compiler whether the named return value optimization is performed.

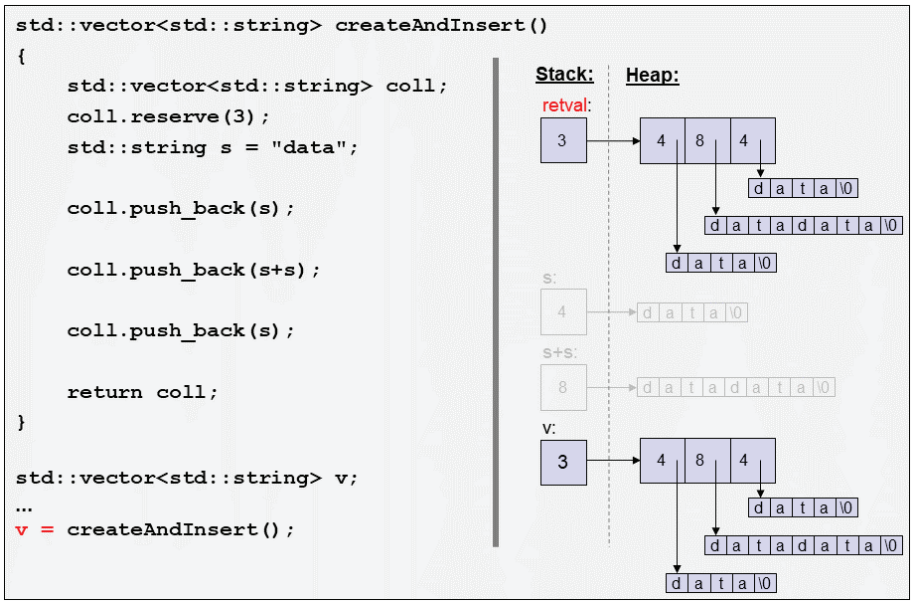
Let's assume that we have the named return value optimization. Then at the end of the return statement `coll` now became the return value and the destructor of `s` is called, which frees the memory allocated when it was declared:



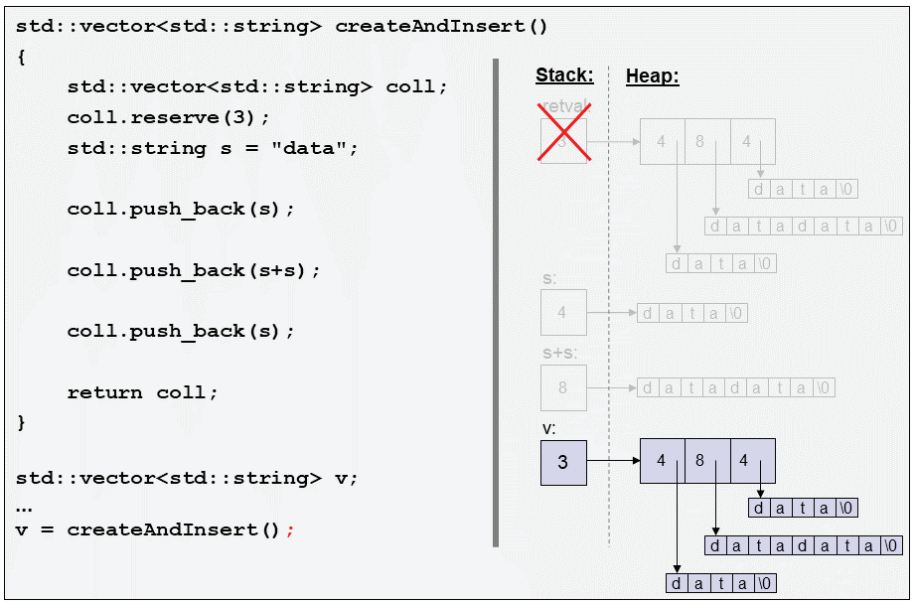
- Finally, we come to the assignment of the return value to `v`:

  ```
  v = createAndInsert();
  ```

  Here, we really get behavior that can be improved: The usual assignment operator has the goal to give `v` the same value as the source value that is assigned. And in general, any assigned value should not be modified and be independent from the object where the value was assigned to. So, the assignment operator will create a deep copy of the whole return value:



But right after that we no longer need the temporary return value and destroy it:



Again, we create a copy of a temporary and destroy the source of the copy immediately afterwards, which means that we again unnecessarily allocate and free memory. This time it applies to four allocations, one for the vector and one for each string element.

For the state of this program after the assignment in `main()`, we allocated memory 10 times and released it 6 times. The unnecessary memory allocations were caused by:

- inserting a temporary object into the collection
- inserting an object into the collection, where we no longer need the value
- assigning a temporary vector with all its elements

We can more or less avoid these performance penalties. Especially instead of the last assignment, we could

- pass the vector as an out parameter:

  ```
  createAndInsert(v);   // let the function fill vector v
  ```

- use `swap()`:

  ```
  createAndInsert().swap(v);
  ```

But the resulting code looks uglier (unless you see some beauty in complex code), and when inserting a temporary object there is not really a workaround.

But now we have another option: Compile and run the program with support for move semantics.

1.1.2 Example Since C++11 (Using Move Semantics)

Now let's recompile the program with a modern C++ compiler (C++11 or later) having support for move semantics:

basics/motiv11.cpp

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

std::vector<std::string> createAndInsert()
{
  std::vector<std::string> coll;   // create vector of strings
  coll.reserve(3);                 // reserve memory for 3 elements
  std::string s = "data";          // create string object

  coll.push_back(s);               // insert string object
  coll.push_back(s+s);             // insert temporary string
  coll.push_back(std::move(s));    // insert string (we no longer need the value of s)

  return coll;                     // return vector of strings
}

int main()
{
  std::vector<std::string> v;      // create empty vector of strings
  ...
  v = createAndInsert();           // assign returned vector of strings
  ...
}
```

There is a little modification, though: we add a `std::move()` call when we insert the last element into `coll`. We discuss this change when we come to this statement. Everything else is as before.

Again, let's look at the individual steps of the program by inspecting both the stack and the heap.

- First, in `main()` we create the empty vector `v`:

  ```
  std::vector<std::string> v;
  ```

  which is placed on the stack as an object having 0 elements.
- Then, we call

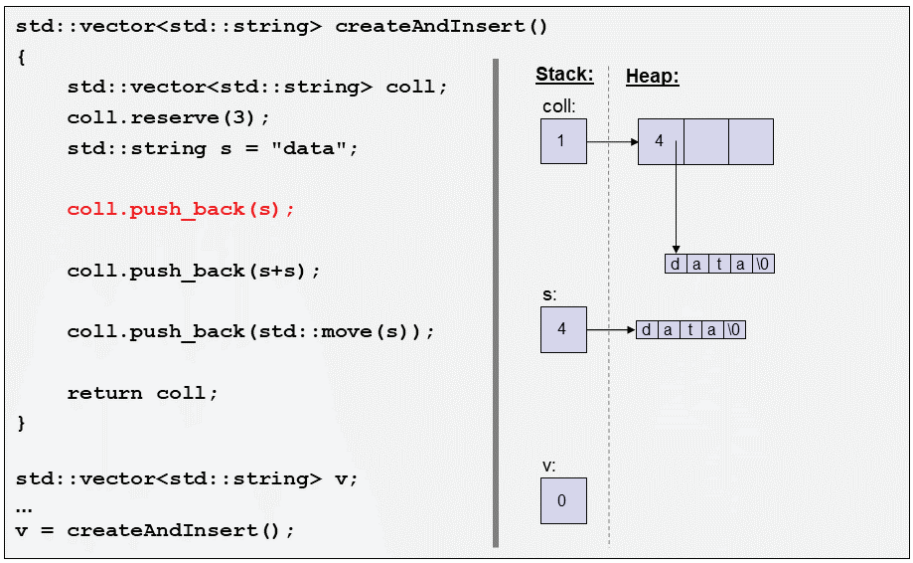
  ```
  v = createAndInsert();
  ```

  where we create another empty vector `coll` on the stack and reserve uninitialized memory for 3 elements on the heap:

  ```
  std::vector<std::string> coll;
  coll.reserve(3);
  ```
- Then, we create the string `s` initialized with `"data"` and insert it into `coll` again:

  ```
  std::string s = "data";

  coll.push_back(s);
  ```

  So far, there is nothing to optimize and we get the same state as with C++03:

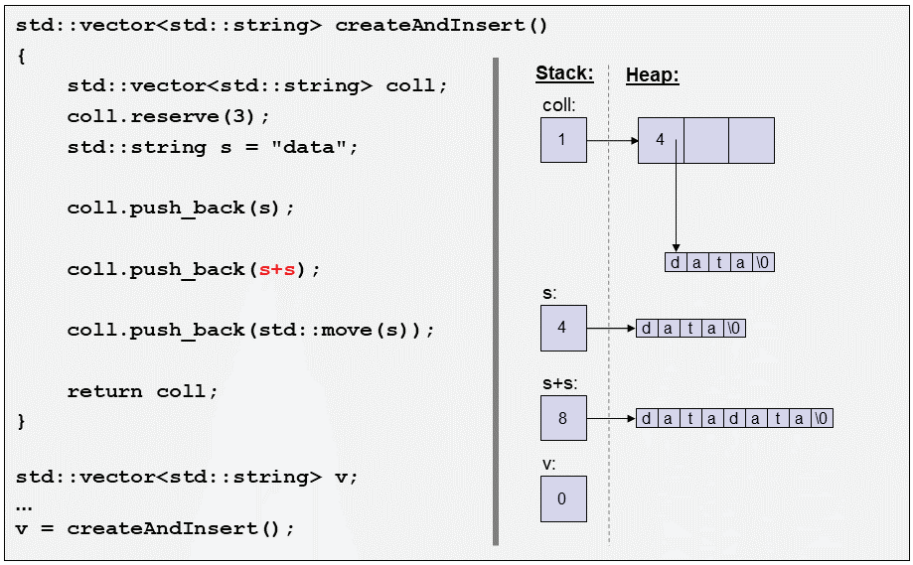


We have two vectors, `v` and `coll`, and two strings `s` and its copy, which is the first element in `coll`. They should all be separate objects with their own memory for the value, because modifying one of them should not impact any of the other objects.

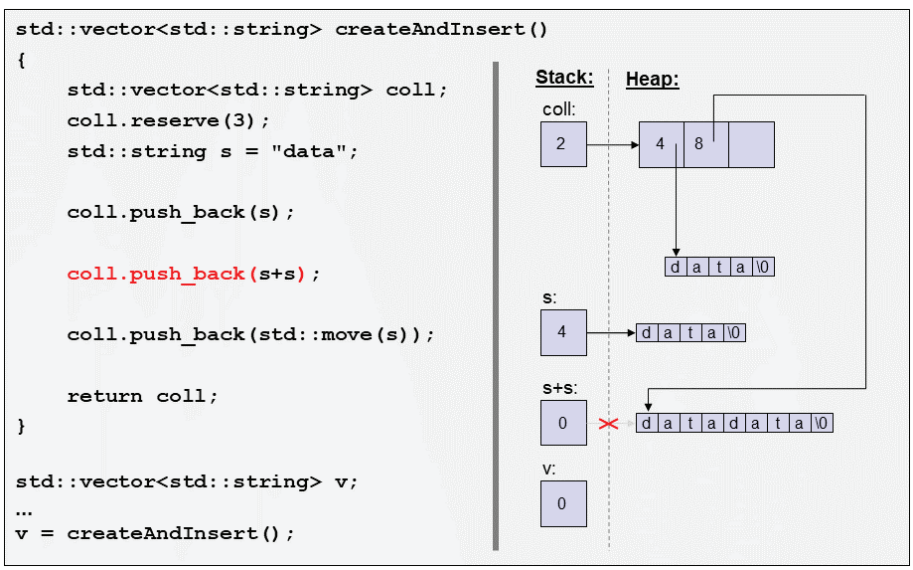
- But now things change. First, let's look at the statement that creates a new temporary string and inserts it into the vector:

  ```
  coll.push_back(s+s);
  ```

  Again, this statement is performed in three steps:
  - First, we create the temporary string `s+s`:

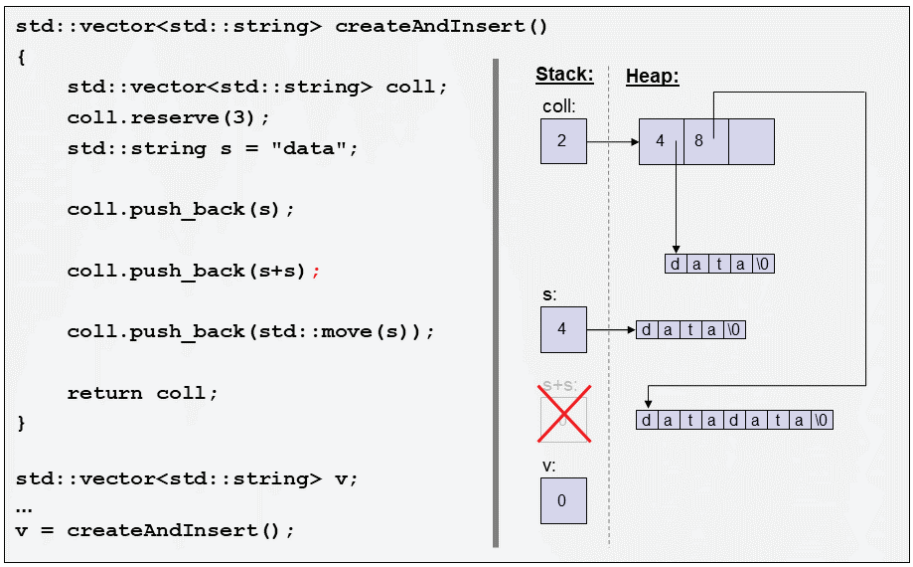


  - Second, we insert this temporary string into the vector `coll`. However, here something different happens now: we steal the memory for the value from `s+s` and move it to the new element of `coll`.



This is possible because since C++11 we can implement special behavior for getting a value which is no longer needed. The compiler can signal this fact, because it knows that right after performing the `push_back()` call the temporary object `s+s` will be destroyed. So, we call a different implementation of `push_back()` provided for the case when the caller no longer needs that value. And as we see, the effect is an optimized implementation of copying the string: Instead of creating an individual copy, we copy both the size and the pointer to the memory. But that shallow copy is not enough, we also modify the temporary object `s+s` by setting the size to `0` and assigning the `nullptr` as value. Essentially, `s+s` is modified getting the state of an empty string. It no longer owns its memory. And that is important because we still have a third step in this statement.

– Last, at the end of the statement, the temporary string `s+s` is destroyed because we no longer need it. But because the temporary string is no longer the owner of the initial memory, the destructor will not free this memory.



Essentially, we optimize the copying so that we move the ownership of the memory for the value of `s+s` to its copy in the vector.

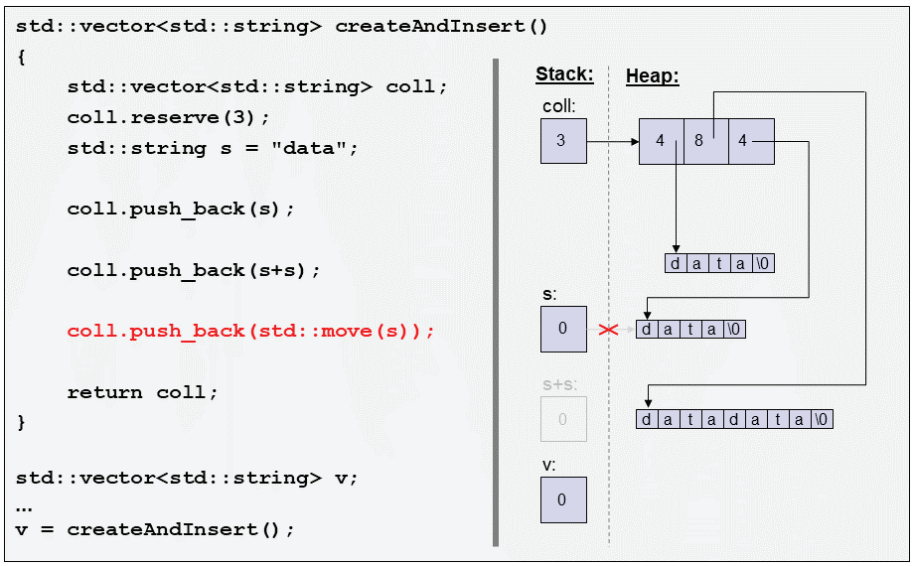
This is all done automatically by using a compiler that can signal that an object is about to die, so that we can use new implementations to copy a string value that steal the value from the source. It's not a technical move. It's a semantic move implemented by technically moving the memory for the value from the source string to its copy.

- The next statement we modified for the C++11 version. Again, we insert `s` into `coll`, but the statement has changed by calling `std::move()` for the string `s` that we insert:

```
coll.push_back(std::move(s));
```

Without `std::move()`, the same would happen as with the first call of `push_back()`: The vector would create a deep copy of the passed string `s`. But in this call we have marked `s` with `std::move()`, which semantically means "***I no longer need this value here.***" As a consequence, we have another call of the

other implementation of `push_back()`, which was used when we passed the temporary object `s+s`. The third element steals the value by moving the ownership of the memory for the value from `s` to its copy:



Note the following two very important things to understand about move semantics:

– `std::move(s)` only ***marks*** `s` to be movable in this context. It does not move. It only says "*I no longer need this value here*." It allows the implementation of the call to benefit from this mark by doing some optimization when copying the value, such as stealing the memory. Whether the value is moved is undefined.

– However, an optimization stealing the value has to ensure that the source object still is in a valid state. A moved-from object is neither partially nor fully destroyed. The C++ standard library formulates this for its types as follows: After an operation called for an object marked with `std::move()` the object is in a ***valid but unspecified state***.

That is, after calling

```
coll.push_back(std::move(s));
```

it is guaranteed that `s` is still a valid string. You can do whatever you want as long as it is valid for any string where you don't know the value. It is like using a string parameter, where you don't know which value was passed.

Note that it is also not guaranteed that the string either has its old value or is empty. It is up to the implementers of the (library) function, which value it has. The implementers can do with `s` whatever they like, provided they leave the string in a valid state.

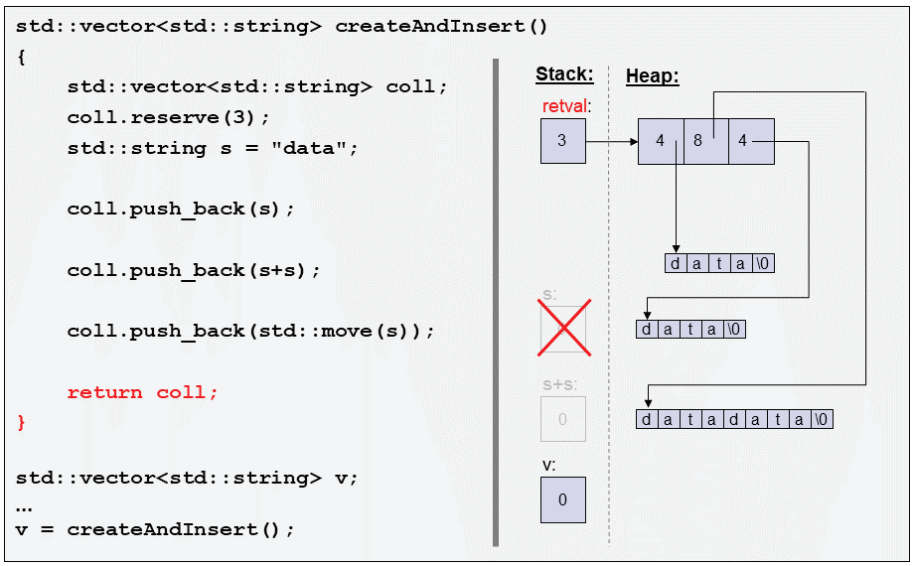
There are good reasons for this guarantee, which are discussed later.

- Again, at the end of `createAndInsert()` we come to the return statement:

```
return coll;
}
```

It is still up to the compiler whether it generates code with the *named return value optimization*, which would mean that `coll` just becomes the return value. However, if this optimization is not used, the return statement is still cheap, because again we have a situation where we create an object from a source that is about to die. That is, if the named return value optimization is not used, move semantics will be used, which means that the return value steals the value from `coll`. At worst, we have to copy the members for size, capacity, and the pointer to the memory (in total usually 12 or 24 bytes) from the source to the return value and assign new values to these members in the source.

Let's assume that we have the named return value optimization. Then at the end of the return statement `coll` now became the return value and the destructor of `s` is called, which no longer has to free any memory, because it was moved to the third element of `coll`:

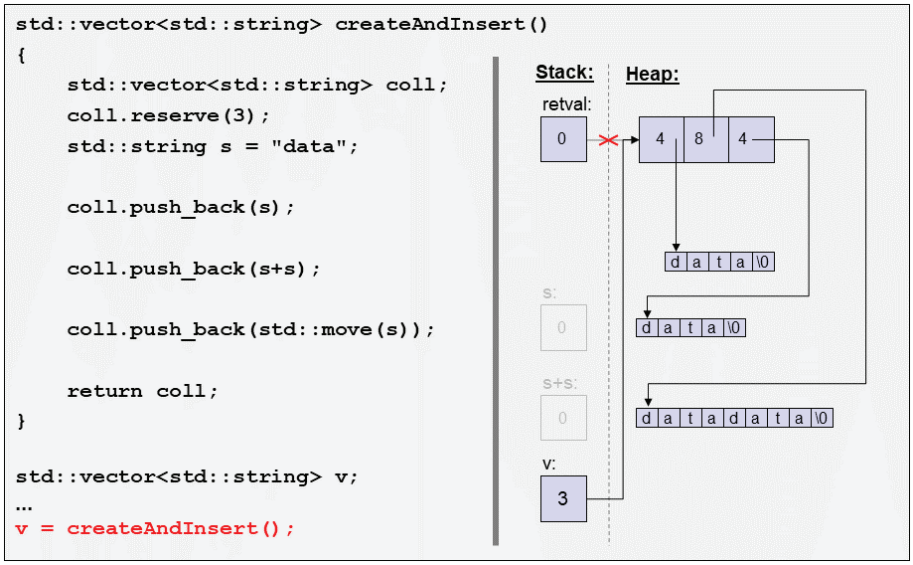


- So, finally, we come to the assignment of the return value to `v`:

  ```
  v = createAndInsert();
  ```

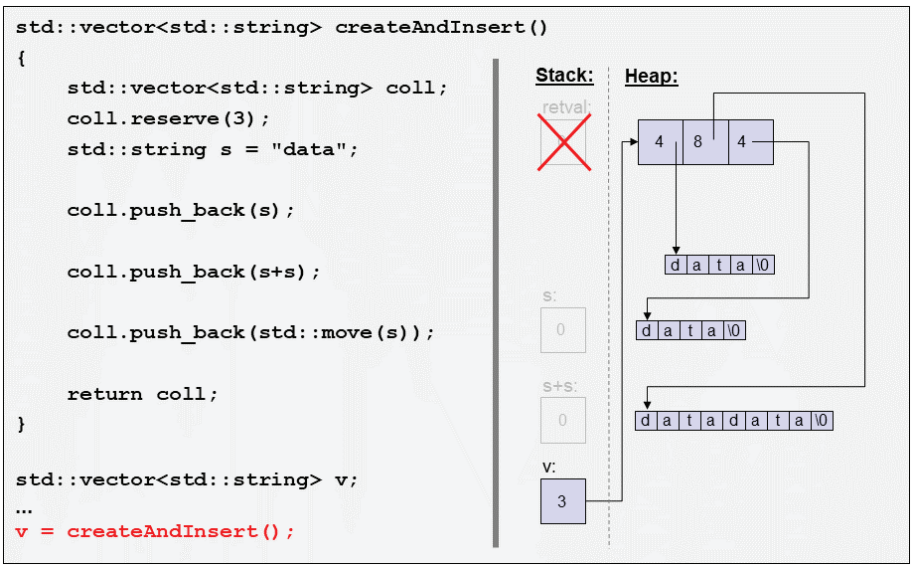
  Again, we can benefit from move semantics now, because we have a situation we already saw: We have to copy (here assign) a value from a temporary return value that is about to die.

  Now, move semantics allows us to provide a different implementation of the assignment operator for a vector that just steals the value from the source vector:



Again the temporary object is not (partially) destroyed. It gets in a valid state, but we don't know its values.

But right after the assignment the end of the statement destroys the (modified) temporary return value:



At the end we are in the same state as before using move semantics, but something significant has changed: We saved 6 allocations and releases of memory. All unnecessary memory allocations no longer happened:

- Allocations for inserting a temporary object into the collection

- Allocations for inserting an named object into the collection, when we use `std::move()` to signal that we no longer need the value
- Allocations for assigning a temporary vector with all its elements

In the second case the optimization was done with our help. By adding `std::move()` we had to say that we no longer needed the value of `s` there. All other optimizations happened because the compiler knows that an object is about to die so that it can call the optimized implementation, which uses move semantics.

This means that returning a vector of strings and assigning it to an existing vector is no longer a performance issue. We can use a vector of strings naively like an integral type and get way better performance. In practice, recompiling code with move semantics can improve speed by 10% to 40% (depending on how naive the existing code was).

1.2 Implementing Move Semantics

Let's use the previous example to see where and how move semantics is implemented.

Before move semantics was implemented, class `std::vector<>` only had one implementation of `push_back()` (the declaration of `vector` is simplified here):

```
template<typename T>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);
    ...
};
```

There was only one way to pass an argument to `push_back()` binding it to a `const` reference. `push_back()` is implemented in a way that the vector creates an internal copy of the passed argument without modifying it.

Since C++11, we have a second overload of `push_back()`:

```
template<typename T>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);

    // insert elem, when the value of elem is no longer needed:
    void push_back (T&& elem);
    ...
};
```

The second `push_back()` uses a new syntax introduced for move semantics. We declare the argument with two `&` and without `const`. Such an argument is called an ***rvalue reference***.[3] "Ordinary references" having only one `&` are now called ***lvalue references***. That is, in both calls we pass the value to be inserted by reference. However, the difference is

- With `push_back(const T&)` we promise not to modify the passed value.
  This function is called when the caller still needs the passed value.
- With `push_back(T&&)` the implementation can modify the passed argument (therefore it is not `const`) to "steal" the value. The semantic meaning is still that the new element gets the value of the passed argument, but we can use an optimized implementation, which moves the value into the vector.
  This function is called, when the caller no longer needs the passed value. The implementation has to ensure that the passed argument is still in a valid state. But the value may be changed. Therefore, after calling this, the caller can still use the passed argument as long as the caller does not make any assumption about its value.

However, a vector does not know *how* to copy an element (whether or not creating the copy is optimized). After making sure that the vector has enough memory for the new element, the vector delegates the work to the type of the elements.

In this case, the elements are strings. So, let's see what happens if we copy or move the passed string.

1.2.1 Using the Copy Constructor

`push_back(const T&)` for the traditional copy semantics calls the copy constructor of the string class, which initializes the new element in the vector. Let's look how this is implemented. The copy constructor of a very naive implementation of a string class would look like this:

```
class string {
  private:
    int len;       // current number of characters
    char* data;    // dynamic array of characters

  public:
    // copy constructor: create a full copy of s:
    string (const string& s)
     : len{s.len} {
        data = new char[len+1];  // new memory
        memcpy(data, s.data, len+1);
    }
    ...
};
```

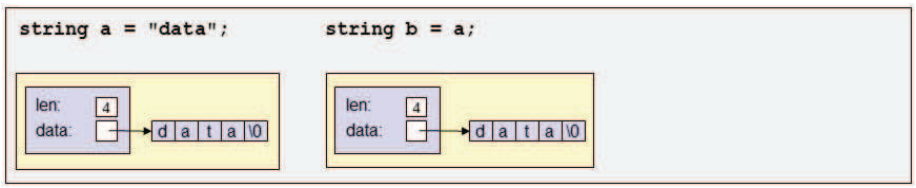
Given we call this copy constructor for a string having the value `"data"`:

```
std::string a = "data";
std::string b = a;        // create b as a copy of a
```

[3] The reason this is called an *rvalue reference* is discussed later in the chapter about value categories.

after initializing the string `a` as follows:

the copy constructor above would copy the member `len` for the number of characters, but assign to the `data` pointer new memory for the value and copy all characters from the source `a` (passed as `s`) to the new string:

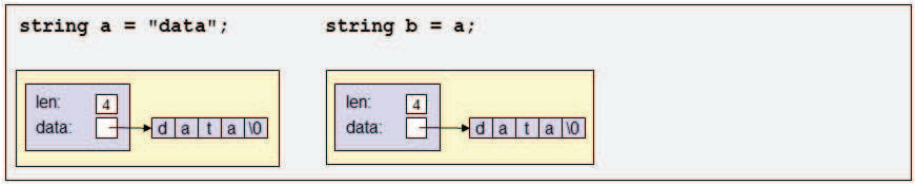


1.2.2 Using the Move Constructor

`push_back(T&&)` for the new move semantics calls a corresponding new constructor, the ***move constructor***. This is the constructor that creates a new string from an existing string, where we no longer need the value. As usual with move semantics, the constructor is declared with a non-`const` rvalue reference (`&&`) as its parameter:

```
class string {
  private:
    int len;       // current number of characters
    char* data;    // dynamic array of characters

  public:
    ...
    // move constructor: initialize the new string from s (stealing the value):
    string (string&& s)
     : len{s.len},
       data{s.data} {       // copy pointer to memory
        s.data = nullptr;   // release the memory for the value
        s.len = 0;          // and adjust number of characters accordingly
    }
    ...
};
```

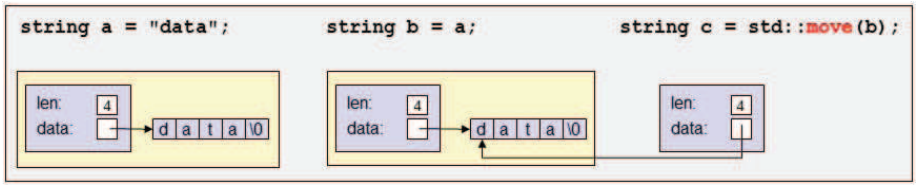
Giving the situation from the copy constructor above:



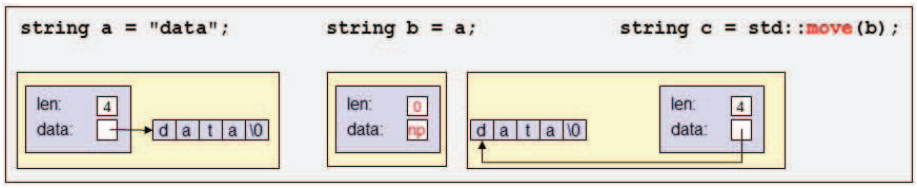
we can call this constructor for a string as follows:

```
std::string c = std::move(b);   // init c with the value of b (no longer needing its value here)
```

The move constructor first copies both the members `len` and `data`, so that the new string gets ownership of the value of `b` (passed as `s`).



But this is not enough, because the destructor of `b` would free the memory. So we also modify the source string to lose its ownership of the memory and bring it into a consistent state representing the empty string:



The effect is that `c` now has the former value of `b` and that `b` is the empty string. Again note that the only guarantee is that `b` is afterwards in a valid but unspecified state. Depending on the way the move constructor is implemented in a C++ library, it might not be empty (but it usually is because this is the easiest and best way here to improve performance).

1.3 Copying as a Fallback

We saw that by using temporary objects or marking objects with `std::move()` we can enable move semantics. Functions providing special implementations (by taking non-`const` rvalue references) can optimize the copying of a value by "stealing" the value from the source. However, ***if*** there is no optimized version of a function for move semantics, then the usual copying is used as a fallback.

For example, assume a container class like vector lacks the second overload of `push_back()`:

```
template<typename T>
class MyVector {
  public:
    ...
```

```
    void push_back (const T& elem);  // insert a copy of elem
    ...      // no other push_back() declared
};
```

We can still pass a temporary object or an object marked with `std::move()`:

```
MyVector<std::string> coll;
std::string s{"data"};
...
coll.push_back(std::move(s));  // OK, uses copy semantics
```

The rule is that for a temporary object or an object marked with `std::move()`, if available, a function declaring the parameter as rvalue reference is preferred. But if no such function exists, the usual copy semantics is used. That way, we ensure that the caller doesn't have to know whether an optimization exists. The optimization might not exist, because

- the function/class was implemented before move semantics was supported or without having move semantics support in mind
- there is nothing to optimize (a class having only numeric members would be an example of that)

Especially for generic code it is important that we can always mark an object with `std::move()` if we no longer need its value. The corresponding code compiles even if there is no move semantics support.

For the same reason, you can even mark objects of a fundamental data type such as `int` (or a pointer) with `std::move()`. The usual value semantics copying the value (the address) will still be used:

```
std::vector<int> coll;
int x{42};
...
coll.push_back(std::move(x));  // OK, but copies x (std::move() has no effect)
```

1.4 Move Semantics for `const` Objects

Finally, note that objects declared with `const` can't be moved, because any optimizing implementation requires that the passed argument can be modified. We can't steal a value if we are not allowed to modify it.

With the usual overloads of `push_back()`:

```
template<typename T>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);

    // insert elem, when the value of elem is no longer needed:
    void push_back (T&& elem);
    ...
};
```

the only valid function to call for `const` objects is the first overload of `push_back()` with the `const&` parameter:

```
std::vector<std::string> coll;
const std::string s{"data"};
...
coll.push_back(std::move(s));   // OK, calls push_back(const std::string&)
```

That means, a `std::move()` for `const` objects has essentially no effect.

In principle, we could provide a special overload for this case by declaring a function taking a `const` rvalue reference. But this makes no semantic sense. Again, the `const` lvalue reference overload serves as a fallback to handle this case.

1.4.1 `const` Return Values

When `const` disables move semantics, this also has consequences when returning values. A `const` return value can't be moved.

For that reason since C++11 it is no longer good style to return by-value with `const` (as some style guide have recommended in the past): For example:

```
const std::string getValue();

std::vector<std::string> coll;
...
coll.push_back(getValue());   // copies (because the return value is const)
```

When returning by-value, don't declare the return value as a whole to be `const`. Use `const` only to declare parts of your return type (such as the object a returned reference or pointer refers to):

```
const std::string getValue();   // BAD: disables move semantics for return values
const std::string& getRef();    // OK
const std::string* getPtr();    // OK
```

1.5 Summary

- Move semantics allows us to optimize copying of data. It can be used implicitly (for unnamed temporary objects or local return values) or explicitly (with `std::move()`).
- `std::move()` means *I no longer need this value here*. It marks the object to be movable. After using it, the object is valid, but you don't know its value anymore.
- By declaring a function with a non-`const` rvalue reference (such as `std::string&&`), you define an interface, where the caller semantically claims that it no longer needs the passed value. The implementer of the function can use this information to optimize its task by "stealing" the value or do any other modification with the passed argument. However that implementer has to ensure that afterwards the passed argument is in a valid state. The caller can count on that.
- Copy semantics is used as a fallback for move semantics. If there is no implementation taking an rvalue reference, any implementation taking an ordinary `const` lvalue reference (such as `const std::string&`) is used. This fallback is then even used if the object is explicitly marked with `std::move()`.

- Calling `std::move()` for a `const` object usually has no effect.
- If you return a value (and no reference), don't declare the return value as a whole to be `const`.

This page is intentionally left blank

Chapter 2

Core Features of Move Semantics

After the first motivating example, this chapter discusses the basic features of move semantics.

2.1 Rvalue References

To support move semantics we introduce a new type of references: Rvalue references. Let's discuss what they are and how to use them.

2.1.1 Rvalue References in Detail

They are declared with two ampersands. As ordinary references, which are declared with one ampersand and are called lvalue references now, rvalue references refer over their lifetime to an existing object, which has to be passed as an initial value. However, according to their semantic meaning, they can only refer to a temporary object not having a name or to an object marked with `std::move()`:

```
std::string returnStringByValue();          // forward declaration
...
std::string s{"hello"};
...
std::string&& r1{std::move(s)};             // OK
std::string&& r2{returnStringByValue()};    // OK, extends lifetime of return value
std::string&& r3{s};                        // ERROR
```

Temporary objects without a name and objects marked with move are so-called *rvalues*, a *value category* we formally discuss later in detail. The name *rvalue reference* comes from the fact that these objects can only refer to rvalues.

As usual for successful initializations of references from return values, references extend the lifetime of the return value until the end of the lifetime of the reference (ordinary `const` lvalue references already had this behavior).

It doesn't matter, which syntax is used to initialize the reference. We can use braces, the equal sign, or parentheses:

```
std::string s{"hello"};
...
std::string&& r1{std::move(s)};          // OK, reference to s
std::string&& r2 = std::move(s);         // OK, reference to s
std::string&& r3(std::move(s));          // OK, reference to s
```

All these references have the semantics of "we can steal/modify the object we refer to, provided the state of the object remains to have a valid state." Technically, these semantics are not checked by compilers, so we can modify an rvalue reference as we can do with any non-`const` object of the type. We might also decide not to modify the value. That is, if you have an rvalue reference to an object, the object might get a different value (which might or might not be the value of a default-constructed object) or it might keep its value.

As we have seen, move semantics allows us to optimize using a value of a source that no longer needs the value. If compilers automatically detect that a value is used from an object that is at the end of its lifetime, it will automatically switch to move semantics. This is the case when:

- We pass the value of a temporary object, which will automatically be destroyed after the statement.
- We pass an object marked with `std::move()`.

2.1.2 Rvalue References As Parameters

When we declare a parameter to be an rvalue reference, it has exactly the behavior and semantics as introduced above:

- It can only bind to a temporary object not having a name or to an object marked with `std::move()`.
- You can modify the object it refers to.
- According to its semantics:
  - The caller claims that it is no longer interested in the value. But it might still be interested to use the object.
  - Any modification should keep the referenced object in a valid state.

For example:

```
void foo(std::string&& rv);   // takes only objects where we no longer need the value
...
std::string s{"hello"};
...
foo(s);                       // ERROR
foo(std::move(s));            // OK, value of s might change
foo(returnStringByValue());   // OK
```

You can use a named object after passing it with `std::move()`, but usually you shouldn't. The recommended programming style is not to use an object anymore after a `std::move()`:

```
void foo(std::string&& rv);   // takes only objects where we no longer need the value
...
std::string s{"hello"};
...
foo(std::move(s));            // OK, value of s might change
std::cout << s << '\n';       // OOPS, you don't know which value is printed
```

```
foo(std::move(s));          // OOPS, you don't know which value is passed
s = "hello again";          // OK, but rarely done
foo(std::move(s));          // OK, value of s might change
```

For both lines marked with "*OOPS*" the call is technically OK as long as you make no assumption about the current value of `s`. Printing out the value is therefore fine, although usually not very useful.

2.2 `std::move()`

If you have an object for which the lifetime does not end when using it, you can mark it with `std::move()` to express "*I no longer need this value here*." `std::move()` does not move; it only sets a temporary marker in the context where the expression is used:

```
void foo(const std::string& lr);   // binds to the passed object without modifying it
void foo(std::string&& rv);        // binds to the passed object and might steal/modify the value
...
std::string s{"hello"};
...
foo(s);                            // calls the first foo(), s keeps its value
foo(std::move(s));                 // calls the second foo(), s might lose its value
```

Objects marked with `std::move()` can still be passed to a function taking an ordinary `const` lvalue reference:

```
void foo(const std::string& lr);   // binds to the passed object without modifying it
...                                // no other overload of foo()
std::string s{"hello"};
...
foo(s);                            // calls the first foo(), s keeps its value
foo(std::move(s));                 // also calls the first foo(), s keeps its value
```

Note that an object marked with `std::move()` cannot be passed to a non-`const` lvalue reference:

```
void foo(std::string&);            // modifies the passed argument
...
std::string s{"hello"};
...
foo(s);                            // OK
foo(std::move(s));                 // ERROR
```

Note that it doesn't make sense to mark a dying object with `std::move()`. In fact, this can even be counter productive for optimizations.

2.2.1 Header File for `std::move()`

`std::move()` is defined as a function in the C++ standard library. For this reason, to use it you have to include the header file `<utility>` where it is defined:

```
#include <utility>     // for std::move()
```

However, because in practice almost all header files include <utility> anyway, usually programs compile without including this header file. However, which header files include others is usually not defined. So, when using std::move() you better explicitly include <utility> to make your program portable.

2.2.2 Implementation of std::move()

std::move() is nothing but a static_cast to an rvalue reference. You can have the same effect by calling static_cast manually:

```
foo(static_cast<std::string&&>(s));  // same effect as foo(std::move(s))
```

However, note that the static_cast does a bit more than only changing the type of the object. It also enables passing the object to an rvalue reference (remember that usually it is not allowed to pass objects with names to rvalue references). We will discuss this in detail in the chapter about value categories.

2.3 Moved-from Objects

After a std::move(), moved-from objects are not (partially) destroyed. They are still valid objects, for which at least the destructor will be called. But they should also be valid in the sense that they have a consistent state and all operations work as expected. The only thing you don't know is their value. It is like using a parameter of the type where you have no clue which value was passed.

2.3.1 Valid but Unspecified State

The C++ standard library guarantees that moved-from objects are in a *valid but unspecified* state.

Consider the following code:

```
std::string s;
...
coll.push_back(std::move(s));
```

After passing s with std::move() you can ask for the number of characters, print out the value, or even assign a new value. But you cannot print the first character or any other character without checking the number of characters first:

```
foo(std::move(s));                // keeps s in a valid but unclear state

std::cout << s << '\n';           // OK (don't know which value is written)
std::cout << s.size() << '\n';    // OK (writes current number of characters)
std::cout << s[0] << '\n';        // ERROR (undefined behavior)
std::cout << s.front() << '\n';   // ERROR (undefined behavior)
s = "new value";                  // OK
```

Although you don't know the value, the string is in a consistent state. For example, s.size() will return the number of characters, so that you can iterate over all valid indexes:

```
foo(std::move(s));                // keeps s in a valid but unclear state

for (int i = 0; i < s.size(); ++i) {
```

```
  std::cout << s[i];              // OK
}
```

For user-defined types you should also ensure that moved-from objects are in a valid state, which sometimes requires to declare or implement move operations. Chapter *Moved-from States in Detail* will discuss details.

2.3.2 Re-using Moved-from Objects

You might wonder why moved-from objects are still valid objects and not (partially) destroyed. The reason is that there are useful applications of move semantics, where it makes sense to use moved-from objects again.

For example, consider code where we read line-by-line strings from a stream and move them into a vector:

```
std::vector<std::string> allRows;
std::string row;
while (std::getline(myStream, row)) {  // read next line into row
  allRows.push_back(std::move(row));   // and move it to somewhere
}
```

After each line read into `row`, we use `std::move()` to move its value into the vector of all rows. And then `std::getline()` uses the moved-from object `row` again to read the next line into it.

As a second example, consider a generic function swapping two values:

```
template<typename T>
void swap(T& a, T& b)
{
  T tmp{std::move(a)};
  a = std::move(b);      // assign new value to moved-from a
  b = std::move(tmp);    // assign new value to moved-from b
}
```

Here we move away the value of `a` into a temporary object to be able to move-assign the value of `b` afterwards. And the moved-from object `b` then gets the value of `tmp`, which is the former value of `a`.

Code like this is for example used in sorting algorithms, where we move the values of the different elements around to bring them into a sorted order. Assigning new values to moved-from objects happens there all the time. And it might even happen that we use the sorting criterion for such a moved-from object.

In general, moved-from objects should be valid objects that can be destroyed (the destructor should not fail), re-used to get other values, and provide all other operations objects of their type support without knowing the value. Chapter *Moved-from States in Detail* will discuss details.

2.3.3 Move Assignments to Itself

The rule that moved-from objects are in a *valid but unspecified state* usually also applies to objects after a direct or indirect self-move.

For example, after the following statement, object `x` is usually valid without knowing its value:

```
x = std::move(x);    // afterwards x is valid but has an unclear value
```

Again, the C++ standard library guarantees that for its objects.[1] User-defined types should usually also provide this guarantee, but sometimes it is necessary to implement something to fix the default-generated moved-from states.

2.4 Overloading by Different References

After introducing rvalue references, we now have three major ways of pass-by-reference:

- `void foo(const std::string& arg)`
  takes the argument as `const` lvalue reference.

  This means that you have only read access to the passed argument. Note that you can pass everything to a function declared that way if the type fits:
  - a modifiable named object
  - a `const` named object
  - a temporary object not having a name
  - an object marked with `std::move()`

  The semantic meaning is that we give `foo()` read access to the passed argument. The parameter is what we call an In-parameter.

- `void foo(std::string& arg)`
  takes the argument as non-`const` lvalue reference.

  This means that you have write access to the passed argument. Note that you can no longer pass everything to a function declared that way even if the type fits. You can only pass:
  - a modifiable named object

  For all other arguments the call does not compile.

  The semantic meaning is that we give `foo()` read/write access to the passed argument. The parameter is what we call an Out- or In/Out-parameter.

- `void foo(std::string&& arg)`
  takes the argument as non-`const` rvalue reference.

  This also means that you have write access to the passed argument. However, again you have restrictions on what you can pass. You can only pass:
  - a temporary object not having a name
  - an non-`const` object marked with `std::move()`

  The semantic meaning is that we give `foo()` write access to the passed argument to steal the value. It is an In-parameter with the additional constraint that the caller no longer needs the value.

Note that rvalue references take different arguments than non-`const` lvalue references. For this reason, we introduced a new syntax and didn't just implement move semantics as usage of functions modifying the passed argument.

[1] The guarantees for moved-from library objects were clarified with the *library working group issue* 2839 (see `http://wg21.link/lwg2839`).

2.4.1 `const` Rvalue References

Technically, there is a fourth way of pass-by-reference:

- `void foo(const std::string&& arg)`
  takes the argument as `const` rvalue reference.

  This also means that you have read access to the passed argument. Here, the restrictions would be that you can only pass:
  - a temporary object not having a name
  - a `const` or non-`const` object marked with `std::move()`

However, there is no useful semantic meaning of this case. As an rvalue reference it's allowed to steal the value, but being `const` we disable modifying the passed argument. This is a contradiction in itself.

Nevertheless, creating objects with this behavior happens pretty easy: Just mark a `const` object with `std::move()`:

```
const std::string s{"data"};
...
foo(std::move(s));      // would call a function declared as const rvalue reference
```

This might indirectly happen when declaring a function to return a value with `const`:

```
const std::string getValue();
...
foo(getValue());        // would call a function declared as const rvalue reference
```

Semantically, this case is usually covered by the `const` lvalue reference overload of a function for read access. A specific implementation of this case is possible, but usually makes no sense. However, the standard library class `std::optional<>` uses `const` rvalue references.

2.5 Passing by Value

If you declare a function taking an argument by value, move semantics might also (automatically) be used.

For example:

```
void foo(std::string str);   // takes the object by value
...
std::string s{"hello"};
...
foo(s);                      // calls foo(), str becomes a copy of s
foo(std::move(s));           // calls foo(), s is moved to str
foo(returnStringByValue()); // calls foo(), return value is moved to str
```

If the caller no longer needs the value (by using `std::move()` or passing a temporary object without a name), we create a new parameter `s` from an object by using move semantics if supported.

That means that with move semantics pass-by-value suddenly can become cheap if a temporary object is passed or the passed argument is marked with `std::move()`. Note that like returning a local object by value this move can be optimized away. However, if it is not optimized away, the call is guaranteed to be cheap now (if move semantics is cheap).

Note the following difference:

```
void fooByVal(std::string str);      // takes the object by value
void fooByRRef(std::string&& str);   // takes the object by rvalue reference
...
std::string s1{"hello"}, s2{"hello"};
...
fooByVal(std::move(s1));             // s1 is moved
fooByRRef(std::move(s2));            // s2 might be moved
```

Here, we compare two functions, one taking the string by value, one taking the string as an rvalue reference. In both cases we pass a string with `std::move()`.

- The function taking the string by value **will** use move semantics, because a new string is created with the value of the passed argument.
- The function taking the string by rvalue reference **might** use move semantics. We don't create a new string. Whether we steal/modify the value of the passed argument depends on the implementation of the function.

Thus

- A function declared to support move semantics, might not use move semantics.
- A function declared to take an argument by value, will use move semantics.

However, again note that the effect of move semantics does not guarantee that any optimization happens at all or what the effect is. All we know is that the passed object afterwards is in a valid but unspecified state.

2.6 Summary

- Rvalue references are declared with `&&` and no `const`.
- They can be initialized by temporary objects not having a name or non-`const` objects marked with `std::move()`.
- Rvalue references extend the lifetime of objects returned by value.
- `std::move()` is a `static_cast` to the corresponding rvalue reference type. This allows us to pass a named object to an rvalue reference.
- Objects marked with `std::move()` can also be passed to functions taking the argument by `const` lvalue reference, but not taking a non-`const` lvalue reference.
- Objects marked with `std::move()` can also be passed to functions taking the argument by value. in that case move semantics is used to initialize the parameter, which can make pass-by-value pretty cheap.
- `const` rvalue references are possible, but implementing against them usually makes no sense.
- Moved-from objects are in a *valid but unspecified* state. You can still (re-)use them providing you don't make any assumptions about their value.

Chapter 3

Move Semantics in Classes

This chapter demonstrates how classes can benefit from move semantics. It demonstrates how ordinary classes automatically benefit from move semantics and how to explicitly implement move operations in classes.

3.1 Move Semantics in Ordinary Classes

Assume you have a pretty simple class with members of types where move semantics can make a difference:

basics/customer.hpp

```
#include <string>
#include <vector>
#include <iostream>
#include <cassert>

class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  Customer(const std::string& n)
   : name{n} {
      assert(!name.empty());
  }

  std::string getName() const {
    return name;
  }
```

```
  void addValue(int val) {
    values.push_back(val);
  }

  friend std::ostream& operator<< (std::ostream& strm, const Customer& cust) {
    strm << '[' << cust.name << ": ";
    for (int val : cust.values) {
      strm << val << ' ';
    }
    strm << ']';
    return strm;
  }
};
```

This class has two (potentially) expensive members, a string for the name and a vector of integral values:

```
class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
  ...
};
```

Both members are expensive to copy

- To copy the name, we have to allocate memory for the characters of the string (unless we have a short name and strings are implemented using the *small string optimization* (SSO)).
- To copy the values, we have to allocate memory for the elements of the vector.

It would be even more expensive, if we had a vector of strings or another pretty expensive element type. For example, a deep copy of a vector of strings would have to allocate memory for both the dynamic array of elements and the memory each element needs.

The good news are that **move semantics is usually automatically supported** by such a class. Since C++11, the compiler usually generates a ***move constructor*** and a ***move assignment operator*** (similar to the automatic generation of a copy constructor and a copying assignment operator).

This has the following effect:

- Returning a local `Customer` by value will use move semantics (if it is not optimized away).
- Passing an unnamed `Customer` by value will use move semantics (if it is not optimized away).
- Passing a temporary `Customer` (e.g., returned by another function) by value will use move semantics (if it is not optimized away).
- Passing a `Customer` object marked with `std::move()` by value will use move semantics (if it is not optimized away).

For example:

basics/customer1.cpp

```
#include "customer.hpp"
#include <iostream>
```

```
#include <random>
#include <utility>  // for std::move()

int main()
{
  // create a customer with some initial values:
  Customer c{"Wolfgang Amadeus Mozart" };
  for (int val : {0, 8, 15}) {
    c.addValue(val);
  }
  std::cout << "c: " << c << '\n';      // print value of initialized c

  // insert the customer twice into a collection of customers:
  std::vector<Customer> customers;
  customers.push_back(c);               // copy into the vector
  customers.push_back(std::move(c));    // move into the vector
  std::cout << "c: " << c << '\n';      // print value of moved-from c

  // print all customers in the collection:
  std::cout << "customers:\n";
  for (const Customer& cust : customers) {
    std::cout << "  " << cust << '\n';
  }
}
```

Here we create and initialize a customer `c`. To avoid SSO, we use a pretty long name. So after the initialization of `c` the first output is as follows:

```
c: [Wolfgang Amadeus Mozart: 0 8 15 ]
```

Then we insert this customer twice in a vector, once copying it and once moving it.

```
customers.push_back(c);               // copy into the vector
customers.push_back(std::move(c));    // move into the vector
```

Afterwards, the next output of the value of `c` will typically be as follows:

```
c: [: ]
```

With the second call of `push_back()` both the name and the values were moved away into the second element of the vector. However, don't forget that a moved-from object is in a *valid but unspecified state*. Thus, the second output could have any value `name` and `values`:

- It might still have the same value:

  ```
  c: [Wolfgang Amadeus Mozart: 0 8 15 ]
  ```

- It might have a totally different value:

  ```
  c: [value was moved away: 0 ]
  ```

However, because move semantics is provided to optimize performance and restoring a different value is not necessarily a way to improve performance, it is pretty typical that implementations make both the string and the vector empty.

In any case, we can see that move semantics is automatically enabled for class `Customer`. For the same reason, it is now guaranteed that the following code is cheap:

```
Customer createCustomer()
{
  Customer c{...};
  ...
  return c;   // uses move semantics if not optimized away
}

std::vector<Customer> customers;
...
customers.push_back(createCustomer());   // uses move semantics
```

See *basics/customer2.cpp* for the complete example.

The important message is that since C++11 classes automatically benefit from move semantics if they use members that benefit from it. These classes have:

- A ***move constructor*** that moves the members if we create a new object from a source where we no longer need the value:

  ```
  Customer c1{...}
  ...
  Customer c2{std::move(c1)};   // move members of c1 to members of c2
  ```

- A ***move assignment operator*** that move assigns the members if we assign the value from a source where we no longer need the value.

  ```
  Customer c1{...}, c2{...};
  ...
  c2 = std::move(c1);           // move assign members of c1 to members of c2
  ```

Note that such a class can benefit even further from move semantics by explicitly implementing to

- use move semantics when initializing members
- use move semantics to make getters both safe and fast

3.1.1 When is Move Semantics Automatically Enabled in Classes?

As just introduced, compilers may automatically generate special move member functions (move constructor and move assignment operator). However, there are constraints. The constraint is that the compiler has to assume that the generated operations do the right thing. The right thing is that we optimize the normal copy behavior: Instead of copying members we move them because the source values are no longer needed.

If classes have changed the usual behavior of copying or assignment, they probably also have to make something different when optimizing these operations. For this reason, the automatic generation of move operations is disabled when at least one of the following special member functions are user-declared:

- copy constructor

- copy assignment operator
- another move operation
- destructor

Note that I wrote "user-declared." Any form of an explicit declaration of a copy constructor, copy assignment operator, or destructor disables move semantics. If we, for example implement a destructor that does nothing, we have disabled move semantics:

```
class Customer {
  ...
  ~Customer() {  // automatic move semantics is disabled
  }
};
```

Even the following declaration is enough to disable move semantics:

```
class Customer {
  ...
  ~Customer() = default;  // automatic move semantics is disabled
};
```

A destructor explicitly requested to have its default behavior is user-declared and therefore disables move semantics. As usual, copy semantics will be used as a fallback, then.

For this reason: **Don't implement or declare a destructor without a specific need** (a rule a surprising number of programmers don't follow).

This also means that a polymorphic base class by default has disabled move semantics:

```
class Base {
  ...
  virtual ~Base() {  // automatic move semantics is disabled
  }
};
```

Note that this does only means that for members of *this* base class move semantics is not automatically supported. For members of derived classes, move semantics is still automatically generated (if the derived class does not explicitly declare a special member function). *Move Semantics in Class Hierarchies* discusses details.

3.1.2 When Generated Move Operations are Broken

Note that it might happen that generated move operations introduce problems although the generated copy operations work fine. Especially, you have to be careful, when

- Members have restrictions on values
  - Values of members have restrictions
  - Values of members depend on each other
- Members with references semantics are used (pointers, smart pointers, ...)
- Objects have no default constructed state

The problem that can occur is that moved-from objects might no longer be valid: invariants might be broken or even the destructor of the object might fail. For example, objects of the `Customer` class in this chapter might suddenly have an empty name although we have assertions to avoid that. Chapter *Invalid Moved-From States* will discuss details. Section *Dealing with Broken Invariants* will especially discuss the problem of broken assertions.

3.2 Implementing Special Copy/Move Member Functions

You can implement the special move member functions yourself. This happens roughly the way a copy constructor and an assignment operator is implemented. The only difference is that the parameter is declared as a nonconst rvalue reference and that inside the implementation you have to specify where to optimize the usual copying by something better.

Let's look at a class having both special copy and special move member functions implemented to print out when objects are copied and when they are moved:

basics/customerimpl.hpp

```
#include <string>
#include <vector>
#include <iostream>
#include <cassert>

class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  Customer(const std::string& n)
   : name{n} {
      assert(!name.empty());
  }

  std::string getName() const {
    return name;
  }

  void addValue(int val) {
    values.push_back(val);
  }

  friend std::ostream& operator<< (std::ostream& strm, const Customer& cust) {
    strm << '[' << cust.name << ": ";
    for (int val : cust.values) {
      strm << val << ' ';
    }
```

```
    strm << ']';
    return strm;
  }

  // copy constructor (copy all members):
  Customer(const Customer& cust)
   : name{cust.name}, values{cust.values}  {
      std::cout << "COPY " << cust.name << '\n';
  }

  // move constructor (move all members):
  Customer(Customer&& cust)
   : name{std::move(cust.name)}, values{std::move(cust.values)} {
      std::cout << "MOVE " << name << '\n';
  }

  // copy assignment (assign all members):
  Customer& operator= (const Customer& cust) {
    std::cout << "COPYASSIGN " << cust.name << '\n';
    name = cust.name;
    values = cust.values;
    return *this;
  }

  // move assignment (move all members):
  Customer& operator= (Customer&& cust) {
    std::cout << "MOVEASSIGN " << cust.name << '\n';
    name = std::move(cust.name);
    values = std::move(cust.values);
    return *this;
  }
};
```

Let's look at the implementations of all special copy/move member functions in detail.

3.2.1 Copy Constructor

The copy constructor is implemented as follows:

```
class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  ...
```

```
    // copy constructor (copy all members):
    Customer(const Customer& cust)
     : name{cust.name}, values{cust.values}  {
        std::cout << "COPY " << cust.name << '\n';
    }
    ...
};
```

The automatically generated copy constructor does just copy all members. In our implementation we only add a print statement that a specific customer is copied.

3.2.2 Move Constructor

The move constructor is implemented as follows:

```
class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  ...
  // move constructor (move all members):
  Customer(Customer&& cust)
   : name{std::move(cust.name)}, values{std::move(cust.values)} {
      std::cout << "MOVE " << name << '\n';
  }
  ...
};
```

Again, this is what the default generated move constructor would do with the additional print statement.

The key difference to the copy constructor is to declare the parameter as non-`const` rvalue reference and then move the members.

Note something very important here: **move semantics is not passed through**. When we initialize the members with the members of `cust` we have to mark them with `std::move()`. Without this, we would just copy them (the move constructor would have the performance of the copy constructor).

You might wonder *why* move semantics is not passed through. Didn't we declare the parameter `cust` to only accept objects with move semantics? However note the semantics here: This function is called when the ***caller*** no longer needs the value. Inside the move constructor, ***we*** now have the value to deal with and ***we*** have to decide where and how long we need it. Especially, we might need the value multiple times not losing it with its first use.

So, the fact that move semantics is not passed through is a feature, not a bug. If we would pass move semantics through, we could not use an object that was passed with move semantics twice. For example:

```
void insertTwice(std::vector<std::string>& coll, std::string&& str)
{
  coll.push_back(str);              // copy str into coll
  coll.push_back(std::move(str));   // move str into coll
```

```
}
```

If all usages of `str` would implicitly have move semantics, the value of `str` would be moved away with the first `push_back()` call.

The important lesson to learn here is that a parameter being declared as rvalue reference restricts what we can pass to this function, but behaves as any other non-`const` object of this type. We again have to specify when and where we no longer need the value. See the formal discussion *When Rvalues become Lvalues* for more details.

One additional note: While the print statement of the copy constructor prints the name of the passed customer:

```
Customer(const Customer& cust)
 : name{cust.name}, values{cust.values}  {
    std::cout << "COPY " << cust.name << '\n';   // cust.name still there
}
```

the move constructor can't use `cust.name` because in the initialization of the constructor the value might have been moved away. We have to use the member of the new objects instead:

```
Customer(Customer&& cust)
 : name{std::move(cust.name)}, values{std::move(cust.values)} {
    std::cout << "MOVE " << name << '\n';   // have to use name (cust.name moved away)
}
```

3.2.3 Copy Assignment Operator

The copy assignment operator is implemented as follows:

```
class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  ...
  // copy assignment (assign all members):
  Customer& operator= (const Customer& cust) {
    std::cout << "COPYASSIGN " << cust.name << '\n';
    name = cust.name;
    values = cust.values;
    return *this;
  }
  ...
};
```

Like the automatically generated copy assignment operator we simply assign all members.[1] The only difference is the print statement at the beginning.

3.2.4 Move Assignment Operator

The move assignment operator is implemented as follows:

```
class Customer {
 private:
  std::string name;          // name of the customer
  std::vector<int> values;   // some values of the customer
 public:
  ...
  // move assignment (steal all members):
  Customer& operator= (Customer&& cust) {
    std::cout << "MOVEASSIGN " << cust.name << '\n';
    name = std::move(cust.name);
    values = std::move(cust.values);
    return *this;
  }
};
```

Again, we have to declare the move constructor as a function taking a non-`const` rvalue reference. Then in the body we have to implement how to improve the usual copying by benefiting that we can steal values from the source object.

In this case, we do what the automatically generated move assignment operator would do: we move assign the members in the body instead of copy assigning them. In addition, we add our print statement. Everything else is as you have learned from implementing assignment operators:

- Return the object for further use.
- You might want to check for assignments to itself before assigning the members:

```
...
Customer& operator= (Customer&& cust) {
  std::cout << "MOVEASSIGN " << cust.name << '\n';
  if (this != &cust) {  // move assignment to itself?
    name = std::move(cust.name);
    values = std::move(cust.values);
  }
  return *this;
}
```

Doing nothing on self-assignment ensures that objects keep their value if this happens, because otherwise the members moved to themselves get an unspecified state.

[1] In addition, you can (and may be should) check for assignments of an object to itself, but note that the default generated assignment operator doesn't do that.

Note that you should usually implement move assignment operators with a `noexcept` specification to improve the performance of reallocations of a `vector` of `Customers`.

3.2.5 Using the Special Copy/Move Member Functions

Let's test the class `Customer` with a small program:

basics/customerimpl.cpp

```
#include "customerimpl.hpp"
#include <vector>
#include <algorithm>

int main()
{
  std::vector<Customer> coll;
  for (int i=0; i<12; ++i) {
    coll.push_back(Customer{"TestCustomer " + std::to_string(i-5)});
  }

  std::cout << "---- sort():\n";
  std::sort(coll.begin(), coll.end(),
            [] (const Customer& c1, const Customer& c2) {
              return c1.getName() < c2.getName();
            });
}
```

Initialization

The first part is the initialization of the vector with 12 customers:

```
std::vector<Customer> coll;
for (int i=0; i<12; ++i) {
  coll.push_back(Customer{"TestCustomer " + std::to_string(i-5)});
}
```

This initialization might have the following output:

```
MOVE TestCustomer -5
MOVE TestCustomer -4
COPY TestCustomer -5
MOVE TestCustomer -3
COPY TestCustomer -5
COPY TestCustomer -4
MOVE TestCustomer -2
MOVE TestCustomer -1
COPY TestCustomer -5
COPY TestCustomer -4
```

```
COPY TestCustomer -3
COPY TestCustomer -2
MOVE TestCustomer 0
MOVE TestCustomer 1
MOVE TestCustomer 2
MOVE TestCustomer 3
COPY TestCustomer -5
COPY TestCustomer -4
COPY TestCustomer -3
COPY TestCustomer -2
COPY TestCustomer -1
COPY TestCustomer 0
COPY TestCustomer 1
COPY TestCustomer 2
MOVE TestCustomer 4
MOVE TestCustomer 5
MOVE TestCustomer 6
```

Each time we insert a new object, a temporary object is created and moved into the vector. Therefore, for each element we have a `MOVE`.

In addition, we have several copies marked with `COPY`. These copies are caused by the fact that a vector from time to time reallocates it's internal memory (capacity) to be big enough for all elements. In this case the vector grows form having memory for 1, 2, 4, and 8 to 16 elements. So we have to copy first 1, then 2, then 4, and then 8 elements already in the vector. Calling `coll.reserve(20)` before the loop would avoid these copies. However, you might wonder why no move is used here, which we will discuss in the chapter about exception handling.

Note that the exact policy of a vector to grow its capacity is implementation specific. Thus, the output might differ when implementations grow differently (such as by 50% only).

Sorting

Next we sort all elements by name:

```
std::sort(coll.begin(), coll.end(),
          [] (const Customer& c1, const Customer& c2) {
            return c1.getName() < c2.getName();
          });
```

This sorting might have the following output:

```
MOVE TestCustomer -4
MOVEASSIGN TestCustomer -5
MOVEASSIGN TestCustomer -4
MOVE TestCustomer -3
MOVEASSIGN TestCustomer -5
MOVEASSIGN TestCustomer -4
MOVEASSIGN TestCustomer -3
MOVE TestCustomer -2
MOVEASSIGN TestCustomer -5
MOVEASSIGN TestCustomer -4
MOVEASSIGN TestCustomer -3
MOVEASSIGN TestCustomer -2
```

```
MOVE TestCustomer -1
MOVEASSIGN TestCustomer -5
MOVEASSIGN TestCustomer -4
MOVEASSIGN TestCustomer -3
MOVEASSIGN TestCustomer -2
MOVEASSIGN TestCustomer -1
MOVE TestCustomer 0
MOVEASSIGN TestCustomer 0
MOVE TestCustomer 1
MOVEASSIGN TestCustomer 1
MOVE TestCustomer 2
MOVEASSIGN TestCustomer 2
MOVE TestCustomer 3
MOVEASSIGN TestCustomer 3
MOVE TestCustomer 4
MOVEASSIGN TestCustomer 4
MOVE TestCustomer 5
MOVEASSIGN TestCustomer 5
MOVE TestCustomer 6
MOVEASSIGN TestCustomer 6
```

As you can see, the whole sorting is only moving around elements. Sometimes to create a new temporary object (`MOVE`), sometimes to assign a value to a different location (`MOVEASSIGN`).

Again, the output depends on the exact implementation of `sort()`, which is implementation specific.

How Much Copies We Saved

The output of this program demonstrates again the benefit of move semantics. Before move semantics support, we would only have copies when inserting and sorting elements. But even in this small program having a container with 12 elements move semantics converted 44 expensive element copies into cheap moves. For 10,000 customers we would save more than 150,000 copies. And note that each copy of a `Customer` means up to 2 memory allocations with the corresponding deallocations later on.

Note again:

- The implementation of `std::sort()` is implementation-specific so that the number of saved copies might slightly differ.
- The only copies performed are cause by the reallocation of the vector for the memory of the elements. They should also become moves, which will be discussed in the chapter about exception handling.

3.3 Rules for Special Member Functions

Let's talk about special member functions and especially specify exactly, when and how the special copy/-move member functions are generated.

3.3.1 Special Member Functions

First we have to talk a bit about the term *special member function*, because it is used in different ways. The C++ standard defines the following 6 operations as *special member functions*:

- Default constructor
- Copy constructor
- Copy assignment operator
- Move constructor (since C++11)
- Move assignment operator (since C++11)
- Destructor

However, many times such as in *the rule of three* or *the rule of five* we only talk about all but one of them, because the default constructor is a bit different from the other five (or three before C++11) operations. The 5 other operations usually are not declared and have more complex dependencies. So make sure you always know what is meant by *the* special member functions (I tried to avoid this term so far).

Figure 3.1 gives an overview when special member function are automatically generated depending on which (other) constructors and special member functions are declared:[2]

user declaration of ↓ / forces →	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	undeclared	undeclared	user declared	undeclared (fallback disabled	defaulted
move assignment	defaulted	undeclared	undeclared	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

Figure 3.1. Rules for Automatic Generation of Special Member Functions

There are a few basic rules you can see in this table:

- A default constructor is only automatically declared if no other constructor is user-declared.

[2] This table is adopted from Howard Hinnant with friendly permission.

- The special copy member functions and the destructor disable move support. The automatic declaration of special move member functions is disabled so that only the fallback can be used (unless the moving operations are also declared).
- The special move member functions disable the normal copying and assignment. The automatic declaration of special copy member functions is disabled so that you can only move (assign) but not copy (assign) an object (unless the copying operations are also declared).

Let's look at some details.

3.3.2 By Default, We Have Copying and Moving

As written, by default both copying and moving special member functions are generated for a class.

With the following declaration of a class `Person`:

```
class Person {
    ...
  public:
    ...
    // NO copy constructor/assignment declared

    // NO move constructor/assignment declared

    // NO destructor declared
};
```

In this case a `Person` can be both copied and moved:

```
std::vector<Person> coll;

Person p{"Tina", "Fox"};
coll.push_back(p);                    // OK, copies p
coll.push_back(std::move(p));         // OK, moves p
```

3.3.3 Declared Copying Disables Moving (Fallback Enabled)

When declaring a copying special member function (or the destructor), we have the automatic generation of the moving special member functions disabled:

With the following declaration of a class `Person`:

```
class Person {
    ...
  public:
    ...
    // copy constructor/assignment declared:
    Person(const Person&) = default;
    Person& operator=(const Person&) = default;

    // NO move constructor/assignment declared
```

```
};
```

Because the fallback mechanism works, copying and moving a `Person` compiles but the move is performed as a copy:

```
std::vector<Person> coll;

Person p{"Tina", "Fox"};
coll.push_back(p);                      // OK, copies p
coll.push_back(std::move(p));           // OK, copies p
```

Thus, declaring a special member function with `=default` is not the same as not declaring it at all. Copy constructor and copy assignment are *user-declared*, which disables move construction and move assignment so that moves fall back to copies. A declared destructor has the same effect.

3.3.4 Declared Moving Disables Copying

If you have user-declared move semantics, you have disabled copy semantics. The copying special member function are no longer generated.

In other words, if the move constructor or the move assignment operator is explicitly declared (implemented, generated with `=default`, or disabled with `=delete`) you have disabled the implicit generation of the copy constructor and the copy assignment operator.

With the following declaration of a class `Person`:

```
class Person {
    ...
  public:
    ...
    // NO copy constructor declared

    // move constructor/assignment declared:
    Person(Person&&) = default;
    Person& operator=(Person&&) = default;
};
```

In this case we have a move-only type. A `Person` can be moved but not copied:

```
std::vector<Person> coll;

Person p{"Tina", "Fox"};
coll.push_back(p);                      // ERROR: copying disabled
coll.push_back(std::move(p));           // OK, moves p

coll.push_back(Person{"Ben", "Cook"});  // OK, moves temporary person into coll
```

Again, declaring a special member function with `=default` is not the same as not declaring it at all. But this time the consequences for the caller are more severe: the trial to copy an object will no longer compile.

A class supporting move operations but no copy operations can make sense. You can use such a ***move-only*** type to pass around ownership or handles of resources without sharing or copying them. For example,

this is used in I/O stream classes, thread classes, and unique pointers. See the chapter about move-only types for details.

3.3.5 Deleting Moving Makes No Sense

For the same reason, if you declare the move constructor as deleted, you can't move (you have disabled this operation; any fallback is not used) and can't copy (because a declared move constructor disables copy operations):

```
class Person {
  public:
    ...
    // NO copy constructor declared

    // move constructor/assignment declared as deleted:
    Person(Person&&) = delete;
    Person& operator=(Person&&) = delete;
    ...
};

Person p{"Tina", "Fox"};
coll.push_back(p);                      // ERROR: copying disabled
coll.push_back(std::move(p));           // ERROR: moving disabled
```

Even if you enable the copy operations with deleted move operation, deleting the move operations makes no sense:

```
class Person {
  public:
    ...
    // copy constructor explicitly declared:
    Person(const Person& p) = default;
    Person& operator=(const Person&) = default;

    // move constructor/assignment declared as deleted:
    Person(Person&&) = delete;
    Person& operator=(Person&&) = delete;
    ...
};

Person p{"Tina", "Fox"};
coll.push_back(p);                      // OK: copying enabled
coll.push_back(std::move(p));           // ERROR: moving disabled
```

That is, a `=delete` disables the fallback mechanism (and is therefore also not the same as not declaring it at all). The compiler finds the declaration and reports the call as an error.

A type supporting copying but failing when moving is called doesn't make any sense. For the user of such a class sometimes copying would work, sometimes not. As a guideline: Never `=delete` the special move member functions.[3] If you want to disable both copying and moving, deleting the copying special member functions is enough.

3.3.6 Disabling Move Semantics

Based on what we just discussed, we know now how to disable move semantics. Declaring the special move member functions as deleted does *not* work:

```
class Customer {
  ...
 public:
  ...
  Customer(Customer&&) = delete;              // OOPS: does disable move calls
  Customer& operator=(Customer&&) = delete;   // OOPS: does disable move calls
};

std::vector<Customer> customers;
...
customers.push_back(createCustomer());        // ERROR: move calls disabled
```

The right way to disable move semantics is to declare one of the other special member functions (copy constructor, assignment operator, or destructor). I recommend defaulting copy constructor and assignment operator (declaring one of them would be enough, but might cause unnecessary confusion):

```
class Customer {
  ...
 public:
  ...
  Customer(const Customer&) = default;              // disable move semantics
  Customer& operator=(const Customer&) = default;   // disable move semantics
};
```

Because no generated special move member function are found, a nameless temporary customer or even a customer marked with `std::move()` is now copied:

```
std::vector<Customer> customers;
...
customers.push_back(createCustomer());          // OK, falls back to copying
customers.push_back(std::move(customers[0]));   // OK, falls back to copying
```

However, usually it is better to implement the special move member functions to fix any problem the default generated move operation has. Chapter *Invalid Moved-From States* will discuss examples from practice.

Note that only declaring the special copy member functions breaks the common "rule of five", which we discuss later in detail. You have to declare the special copy member functions but can't in addition declare

[3] Thanks to Howard Hinnant for pointing that out.

the special move member functions (both deleting and defaulting would not work, implementing them makes the class unnecessary complicated). So, if you explicitly declare a copying special member function just to disable move semantics, place a big comment to ensure that this declaration is neither removed nor extended by a declaration of the special move member functions.

3.3.7 Moving for Members with Disabled Move Semantics

Note that unavailable or deleted move semantics of a type has no influence on the generation of move semantics for classes where the type is used. The default generated move constructor and assignment decide member-by-member whether to copy or to move it. If a move is not possible (even if it is deleted), a copy is generated.

For example, given the following class:

```
class Customer {
  ...
 public:
  ...
  Customer(const Customer&) = default;              // copying calls enabled
  Customer& operator=(const Customer&) = default;   // copying calls enabled
  Customer(Customer&&) = delete;                    // moving calls disabled
  Customer& operator=(Customer&&) = delete;         // moving calls disabled
};
```

If this class is used by a member in the other class:

```
class Invoice {
  std::string id;
  Customer cust;
 public:
  ...   // no special member functions
};
```

the generated move constructor will move the `id` string, but copy the customer:

```
Invoice i;
Invoice i1{std::move(i)};    // OK, moves id, copies cust
```

3.3.8 Exact Rules for Generated Special Member Functions

Now we can summarize the new rules for special member functions (when they are generated and how they behave).

As an example, assume we have the following derived class:

```
class MyClass : public Base
{
 private:
  MyType value;
  ...
};
```

The one thing missing here is the `noexcept` specification, which we introduce later in the chapter about `noexcept`. However, here we already mention the corresponding guarantees.

Copy Constructor

The copy constructor is automatically generated when

- no move constructor is user-declared and
- no move assignment operator is user-declared

If generated (implicitly or with `=default`), it has the following behavior:[4]

```
MyClass(const MyClass& obj)
 : Base{obj}, value{obj.value} {
}
```

The generated copy constructor ensures that the copy constructor of the base class is called with the given source object (remember that copy constructors are always called top-down), and copies all members of its class.

The generated copy constructor guarantees not to throw if all copy operations (the copy constructor of the base class and the copy constructors of all members) give this guarantee.

Move Constructor

The move constructor is automatically generated when

- no copy constructor is user-declared and
- no copy assignment operator is user-declared and
- no move assignment operator is user-declared and
- no destructor is user-declared

If generated (implicitly or with `=default`), it has the following behavior:

```
MyClass(MyClass&& obj)
 : Base{std::move(obj)}, value{std::move(obj.value)} {
}
```

The generated move constructor ensures that the move constructor of the base class is called (with the given source object marked with `std::move()` again to pass through its move semantics) and moves all members of its class. If any member can't be moved but can be copied, the generated move constructor uses its copy constructor instead.

The generated move constructor guarantees not to throw if all called move/copy operations (the copy or move constructor of the base class and the copy or move constructors of all members) give this guarantee.

[4] The generated copy constructor takes the argument as non-`const` reference if one of the copy constructors used are implemented without `const`.

Copy Assignment Operator

The copy assignment operator is automatically generated when

- no move constructor is user-declared and
- no move assignment operator is user-declared

If generated (implicitly or with `=default`), it roughly has the following behavior:[5]

```
MyClass& operator= (const MyClass& obj) {
  Base::operator=(obj);      // - perform assignments for base class members
  value = obj.value;         // - assign new members
  return *this;
}
```

The generated copy assignment operator first ensures that the members of the base class(es) are assigned (remember that assignment operators are not called top-down as it is the case for constructors) and assigns all members of its class.

Note that the generated assignment operator does not check for assignments to itself. If this is critical, you have to implement it yourself.

In addition, the generated copy assignment operator guarantees not to throw if all assignment operations (the assignment of the base class members and the assignment of the new members) give this guarantee.

Move Assignment Operator

The move assignment operator is automatically generated when

- no copy constructor is user-declared and
- no move constructor is user-declared and
- no copy assignment operator is user-declared and
- no destructor is user-declared

If generated (implicitly or with `=default`), it roughly has the following behavior:

```
MyClass& operator= (MyClass&& obj) {
  Base::operator=(std::move(obj));   // - perform move assignments for base class members
  value = std::move(obj.value);      // - move assign new members
  return *this;
}
```

The generated move assignment operator ensures that the move assignments of the base class members is called and moves assigns all new members (again with using copy assignments if move assignments are not possible). If any member can't be move but can be copy assigned, the generated move assignment operator uses its copy assignment operator instead.

Note that the generated assignment operator does not check for assignments to itself. Thus, in its default behavior it will move assign members to itself, which usually means that members get an unspecified value. If this is critical, you have to implement it yourself.

[5] The generated copy assignment operator takes the argument as non-`const` reference if one of the copy assignment operators used are implemented without `const`.

In addition, the generated move assignment operator guarantees not to throw if all called assignment operations (the assignments for base class members and the assignments for new members) give this guarantee.

Other Special Member Functions

Other special member functions play not such an important role for move semantics:

- **Destructors** are nothing special with move semantics except that their declaration disables the automatic generation of move operations.

  However, you have to ensure that the moved-from state of an object can be handled by the destructor. In some cases generated move operations create non-destructible states for the moved-from objects, which you have to fix then.
- The **default constructor** (a "not-so-special" special member function) is still automatically generated if no other constructor is declared. That is, the declaration of a move constructor disables the generation of a default constructor.

3.4 The Rule of Five or Three

When which special member function is automatically generated is a combination of several rules just described. Many programmers don't know all these rules. For this reason even before C++11 there was a guideline to always have all or none of *the* special member functions for copying, assignment, and destruction.

- Before C++11, it was called the ***Rule of Three***: You should either declare all three (copy constructor, assignment operator, and destructor) or none of them.
- Since C++11, the rule became the ***Rule of Five***, which usually is formulated as:[6] You should either declare all five (copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor) or none of them.

Declaring means here

- either to implement(`{...}`)
- or to declare as defaulted (`=default`)
- or to declare as deleted (`=delete`)

That is, when one of these special member functions is either implemented, defaulted, or deleted, you should do one of these things with all four other special member functions.

However, you should be careful with this rule. I recommend to take it more as a guideline to ***carefully think*** about all of these five special member function when one of them is user-declared.

As we saw, to disable move semantics you should `=default` the copying special member functions without declaring the special move member functions (deleting and defaulting them would not work, implementing them makes the class unnecessary complicated). This is especially one recommended option if the generated move semantics creates invalid states, as we discuss in the section about invalid moved-from states.

[6] E.g., see http://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-five.

When applying this rule of five it also turned out that sometimes programmers use it to add declarations for the new move operations without understanding what this means. Programmers were just declaring move operations with `=default` because copy operations were implemented and they wanted to follow the rule of five.

For this reason, I usually teach the ***Rule of Five or Three***:

- If you declare copy constructor, move constructor, copy assignment operator, move assignment operator, or destructor, ***think very carefully*** about how to deal with the other of these special member functions.
- If you don't understand move semantics, only think about copy constructor, copy assignment operator, and destructor if declaring one of them. In doubt, disable move semantics by declaring the copying special member functions (or the destructor) with `=default`.

3.5 Summary

- Move semantics is not passed through.
- For every class, move constructor and move assignment operator are automatically generated if this can't be a problem
- User-declaring a copy constructor, copy assignment operator, or destructor disables the automatic support of move semantics in a class. This does not impact the support in derived classes.
- User-declaring a move constructor or move assignment operator disables the automatic support of copy semantics in a class. By default, you get move-only types.
- Never `=delete` a special move member function.
- Don't declare a destructor without a specific need. There is no need in classes derived from a polymorphic base class.

This page is intentionally left blank

Chapter 4

How to Benefit from Move Semantics

Most of the time programmers benefit from move semantics automatically. However, even as an ordinary application programmer you can benefit from move semantics even more when programming slightly differently.

Therefore, this chapter discusses how to benefit from move semantics in ordinary application code, classes, and class hierarchies beyond the automatic generation of special move member functions, and introduces corresponding new guidelines.

Note that even several years after introducing move semantics these recommendations are not widely known in the community. In fact, one reason for me to write this book was to make them state of the art of modern C++ programming.

4.1 Avoid Objects with Names

As we have seen, move semantics allows us to optimize using a value of a source that no longer needs the value. If compilers automatically detect that a value is used from an object that is at the end of its lifetime, it will automatically switch to move semantics. This is the case when:

- We pass a temporary object, which will automatically be destroyed after the statement.
- We return a local object by value.

In addition, we can force move semantics by marking an object with with `std::move()`.

Because it is easier to just to let compilers do the work, a first consequence of move semantics is the following advice: **Avoid objects with names.**

Instead of

```
MyType x{42, "hello"};
foo(x);      // x not used afterwards
```

you better program:

```
foo(MyType{42,"hello"});
```

to automatically enable move semantics.

Of course, the advice to avoid objects with name might collide with other important style guide such as readability and maintainability of source code. Instead of having a complicated complex statement, it might be better to use multiple statements. In that case you should use `std::move()` if you no longer need an object (and know that copying the object might take significant time):

```
foo(std::move(x));
```

4.1.1 When You Can't Avoid Using `std::move()`

There are cases, where you can't avoid using `std::move()`, because you have to give objects names. The most obvious typical examples are:

- You have to use an object multiple times. For example, you might get a value to process it twice in a function or loop:

  ```
  std::string str{getData()};
  ...
  coll1.push_back(str);                  // copy (still need the value of str)
  coll2.push_back(std::move(str));       // move (no longer need the value of str)
  ```

  The same applies when inserting the value in two different collections or calling two different functions doing that.
- You have to deal with a parameter. The most common example of this is the following loop:

  ```
  // read and store line by line from myfile in coll
  std::string line;
  while (std::getline(myStream, line)) {
    coll.push_back(std::move(line));     // move (no longer need the value of line)
  }
  ```

4.2 Avoid Unnecessary `std::move()`

As we saw, returning a local object by value automatically uses move semantics if supported. However, just to be safe, programmers might try to force this by an explicit `std::move()`:

```
std::string foo()
{
  std::string s;
  ...
  return std::move(s);   // don't do this
}
```

Remember that `std::move()` is just a `static_cast`. So we have an expression that yields the type `std::string&&`. However, this does no longer match the return type and therefore disables the *return value optimization*, which usually let the returned object be used as return value.

Note that for types where move semantics is not implemented, this might even force to copy the return value instead of just using the returned object as return value.

So, if you return local objects by value, don't use `std::move()`:

```
std::string foo()
{
  std::string s;
  ...
  return s;   // best performance (return value optimization or move)
}
```

The same problem arises when initializing a new object. Instead of

```
std::string s{std::move(foo())};
```

just write

```
std::string s{foo()};
```

Otherwise, you prevent copy elision in the initialization.

Compilers might (have options to) warn about any counter-productive or unnecessary use of `std::move()`. For example, gcc has the options `-Wpessimizing-move` (enabled with `-Wall`) and `-Wredundant-move` (enabled with `-Wextra`).

There are applications, though, where a `std::move()` in a return statement might be appropriate. One example is moving out the value of a member. Another example is returning a parameter with move semantics.

4.3 Initialize Members with Move Semantics

A surprising consequence is that you can benefit from move semantics even in trivial classes with members of types that benefit from move semantics (such as string members or containers).

Let's understand this on a pretty simple example.

4.3.1 Initialize Members the Classical Way

Consider a class with two string members, which we can initialize in the constructor. Such a class will typically be implemented like this:

basics/initclassic.hpp

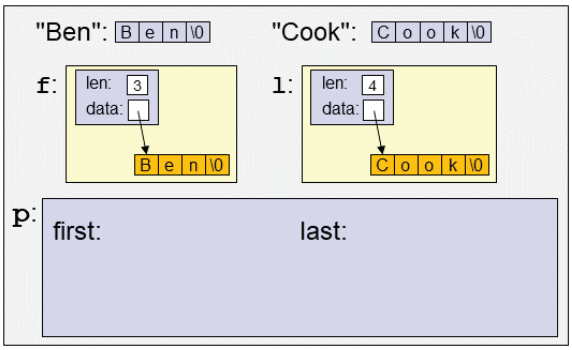
```
#include <string>

class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
  Person(const std::string& f, const std::string& l)
   : first{f}, last{l} {
  }
  ...
```

```
};
```

Now, let's look at what happens when we initialize an object of this class with two string literals:

```
Person p{"Ben", "Cook"};
```

The compilers finds out that this initialization is possible by the provided constructor. However, the types of the parameters does not fit. So, the compiler first creates code to create two `std::string`s from two string literals, to which the parameters `f` and `l` bind to:

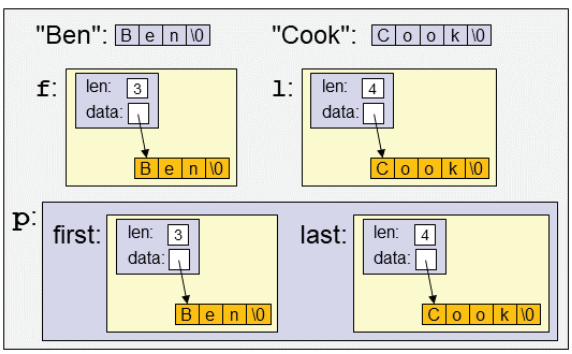


In general (if the *small string optimization (SSO)* is not used because it is not available or the strings are too long), this means that code is generated that allocates memory for the value of each `std::string`.

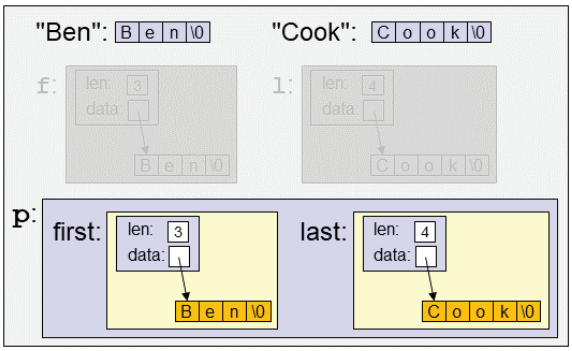
However, the temporary created strings are not directly used as members `first` or `last`. Instead, they are used to initialize these members. Unfortunately move semantics is not used here for two reasons:

- With the parameters `f` and `l` we have objects with name that exists longer than the initialization of the members (you can still use them in the body of the constructor).
- The parameters are declared to be `const`, which disables move semantics even if we use `std::move()`.

As a consequence, the copy constructor for strings is called on each member initialization, again allocating memory for the values:



And then, at the end of the constructor, the temporary strings are destroyed:



This means that we have 4 memory allocations although only 2 are necessary. Using move semantics we can do better.

Using non-`const` Lvalue References?

You may wonder why we can't simply use non-`const` lvalue references here:

```
class Person {
  ...
  Person(std::string& f, std::string& l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};
```

But passing `const std::string`s and temporary objects (e.g., created from a type conversion) would not compile:

```
Person p{"Ben", "Cook"};   // ERROR: cannot bind a non-const lvalue reference to a temporary
```

The reason is that in general a non-`const` lvalue reference does not bind to a temporary object. So this constructor could not bind `f` and `l` to temporary strings created from the passed string literals.

4.3.2 Initialize Members via Moved Parameters Passed by Value

With move semantics, there is now a simple alternative way for constructors to initializing members: the Constructor takes each argument by value and moves it into the member:

basics/initmove.hpp

```
#include <string>

class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
```

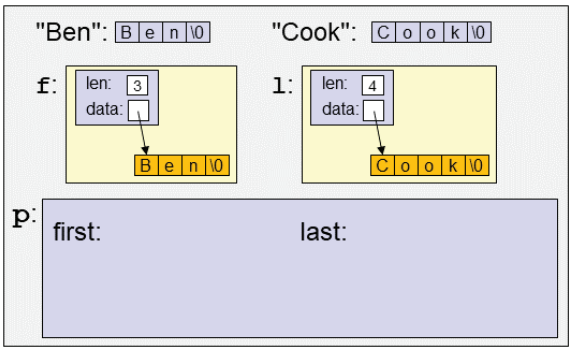
```
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};
```

This single constructor takes all possible arguments and ensures that we only have one allocation for each argument.

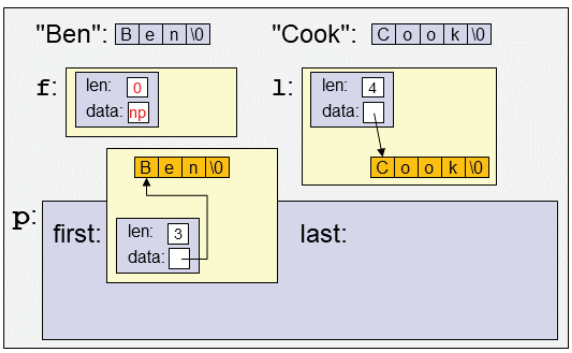
For example, if we pass two string literals:

```
Person p{"Ben", "Cook"};
```

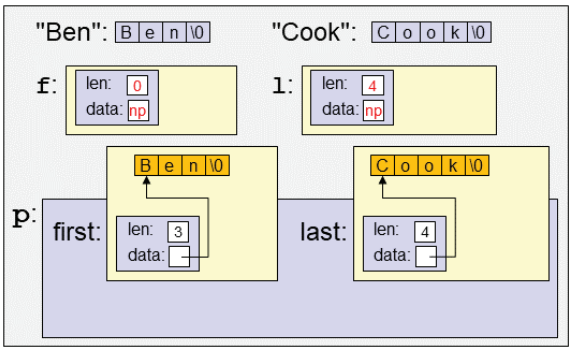
we first use them to initialize the parameters `f` and `l`:



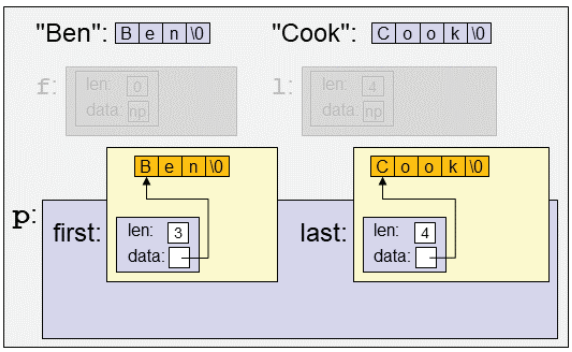
By using `std::move()` we move the values of the parameters to the members. First, the member `first` steals the value from `f`:



Then, the member `last` steals the value from `l`:



Again, at the end of the constructor, the temporary strings are destroyed, which however no longer have to free allocated memory:



If instead we pass two existing strings without marking them with `std::move()`,

```
std::string name1{"Jane"}, name2{"White"};
...
Person p{name1, name2};     // OK, copy names into parameters and move them into the members
```

If instead we pass two existing strings where we no longer need the value, we even need no allocation at all:

```
std::string name1{"Jane"}, name2{"White"};
...
Person p{std::move(name1), std::move(name2)};     // OK, move value to parameters and then to members
```

Passing the strings by value uses move semantics to initialize the parameters `f` and `l`. And then again we use move semantics to move the values to the members.

Provided a move is cheap, with this implementation any initialization is possible and cheap with only one constructor.

4.3.3 Initialize Members via Rvalue References?

There are still other alternatives to initialize members, which however need multiple constructors. To support move semantics we learned already that we can declare a parameter as non-`const` rvalue reference. That way we steal the value from a passed temporary object or an object marked with `std::move()`.

Using Rvalue References

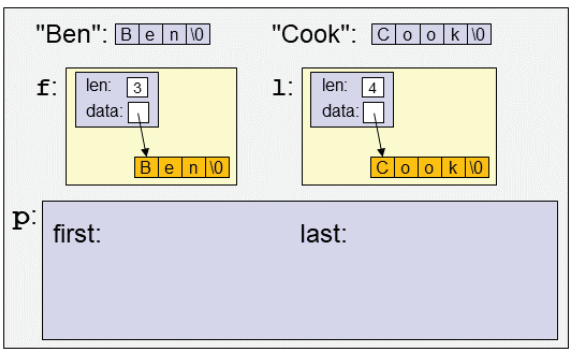
The following constructor would do the work:

```
class Person {
  ...
  Person(std::string&& f, std::string&& l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};
```

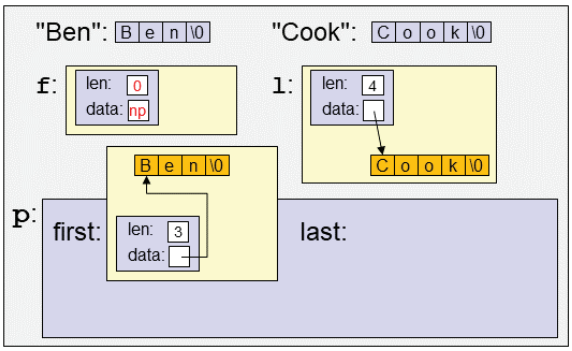
This initialization would work for our passed string literals:

```
Person p{"Ben", "Cook"};
```

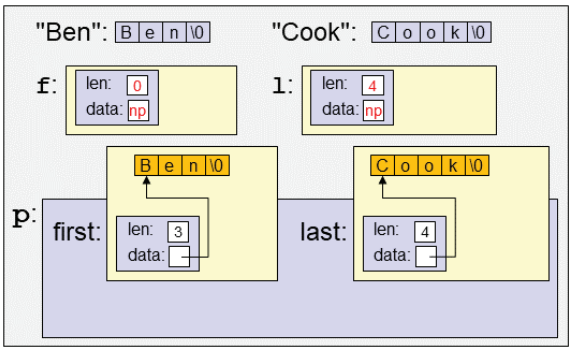
Again, because the constructor needs strings, we would create two temporary strings to which `f` and `l` bind:



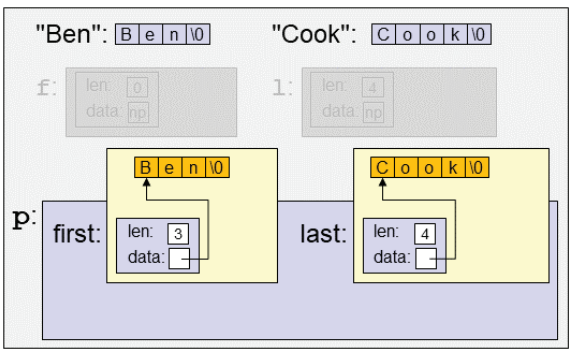
However, because we now have non-`const` objects, we can modify them. In this case, we mark them with `std::move()` so that the initialization of the members can steal the values. First, the member `first` steals the value from `f`:



Then, the member `last` steals the value from `l`:



Again, at the end of the constructor, the temporary strings are destroyed, which however no longer have to free allocated memory:



Overloading for Rvalue and Lvalue References

While the constructors taking rvalue references work fine when we pass string literals to them, they also have restrictions: We can't pass named objects that still need the value afterwards. So, with only a constructor taking rvalue references we can't pass an existing string:

```
class Person {
  ...
  Person(std::string&& f, std::string&& l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};

Person p1{"Ben", "Cook"};  // OK

std::string name1{"Jane"}, name2{"White"};
...
Person p2{name1, name2};    // ERROR: can't pass a named object to an rvalue reference
```

For the case of `p2` we would need the classical constructor, taking a `const` lvalue reference. However, it could also happen that we pass a string literal and an existing string. So, we need in total 4 constructors for all possible combinations:

```
class Person {
  ...
  Person(const std::string& f, const std::string& l)
   : first{f}, last{l} {
  }
  Person(const std::string& f, std::string&& l)
   : first{f}, last{std::move(l)} {
  }
  Person(std::string&& f, const std::string& l)
   : first{std::move(f)}, last{l} {
  }
  Person(std::string&& f, std::string&& l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};
```

That way we can pass both string literals and existing strings in any combination and always have only one allocation per member.

Overloading even for String Literals

To improve the performance even more, we might even have specific implementations taking string literals as ordinary pointers. That way, we could avoid even some moves. However, implementing all constructors becomes a bit tedious:

basics/initall.hpp

```
#include <string>
```

```
class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
  Person(const std::string& f, const std::string& l)
   : first{f}, last{l} {
  }
  Person(const std::string& f, std::string&& l)
   : first{f}, last{std::move(l)} {
  }
  Person(std::string&& f, const std::string& l)
   : first{std::move(f)}, last{l} {
  }
  Person(std::string&& f, std::string&& l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  Person(const char* f, const char* l)
   : first{f}, last{l} {
  }
  Person(const char* f, const std::string& l)
   : first{f}, last{l} {
  }
  Person(const char* f, std::string&& l)
   : first{f}, last{std::move(l)} {
  }
  Person(const std::string& f, const char* l)
   : first{f}, last{l} {
  }
  Person(std::string&& f, const char* l)
   : first{std::move(f)}, last{l} {
  }
  ...
};
```

4.3.4 Compare the Different Approaches

With the introduction of new ways of initializing member, the obvious question is, which technique should we use when. And also: which technique should we teach?

In general, there should be a good reason not to use the simple approach with `const &`. This reason usually is performance. So, let's measure. Let's measure how long it takes to initialize `Persons` by three kinds of arguments: passing string literals, passing existing strings, passing strings marked with `std::move()`:

```
std::string fname = "a first name";
```

```
std::string lname = "a last name";

// measure how long this takes:
Person p1{"a firstname", "a lastname"};
Person p2{fname, lname};
Person p3{std::move(fname), std::move(lname)};
```

However, to avoid the *small string optimization (SSO)*, which would mean that the strings do not allocate any memory at all, we should use strings of significant length. So, here is a full function to measure the different approaches:

basics/initmeasure.hpp

```
#include <chrono>

// measure num initializations of whatever is currently defined as Person:
std::chrono::nanoseconds measure(int num)
{
  std::chrono::nanoseconds totalDur{0};
  for (int i = 0; i < num; ++i) {
    std::string fname = "a firstname a bit too long for SSO";
    std::string lname = "a lastname a bit too long for SSO";

    // measure how long it takes to create 3 Persons in different ways:
    auto t0 = std::chrono::steady_clock::now();
      Person p1{"a firstname too long for SSO", "a lastname too long for SSO"};
      Person p2{fname, lname};
      Person p3{std::move(fname), std::move(lname)};
    auto t1 = std::chrono::steady_clock::now();
    totalDur += t1 - t0;
  }
  return totalDur;
}
```

The function `measure()` returns the duration of `num` iterations doing the three initializations above with string of a significant length.

Now we combine the different definitions of class `person` above with the measuring function in programs with `main()` functions that call the measurements and print the resulting durations. For example:

basics/initclassicperf.cpp

```
#include "initclassic.hpp"
#include "initmeasure.hpp"
#include <iostream>
#include <cstdlib>    // for std::atoi()

int main(int argc, const char** argv)
```

```
{
  int num = 1000;   // num iterations to measure
  if (argc > 1) {
    num = std::atoi(argv[1]);
  }

  // a few iterations to avoid measuring initial behavior:
  measure(5);

  // measure (in integral nano- and floating-point milliseconds):
  std::chrono::nanoseconds nsDur{measure(num)};
  std::chrono::duration<double, std::milli> msDur{nsDur};

  // print result:
  std::cout << num << " iterations take:  "
            << msDur.count() << "ms\n";
  std::cout << "3 inits take on average:  "
            << nsDur.count() / num << "ns\n";
}
```

The other two programs, *basics/initallperf.cpp* and *basics/initmoveperf.cpp*, just use different header files for the other declarations of class `Person`.

Running this code on three different platforms with three different compilers with significant optimization turned on,

- In general, the initializations using the classic lvalue references (`const &`) take significant more time than the other initializations. I have seen factors of up to 2.
- There is no big difference between implementing all 9 constructors and just the constructor taking the argument by value and move. Sometimes one approach was a bit faster, sometimes the other; often there was no significant difference.

If we benefit from the *small string optimization (SSO)*, by using pretty short strings, which means that we don't allocate any memory at all (and move semantics should not be of significant help), the numbers are pretty close. But you can still measure a slight drawback using the classic constructor taking a `const` lvalue reference. However, this definitely requires good optimization.

Beside trying the different test programs you can use *basics/initperf.cpp* as one file doing all these different measurements by one CPP file on your favorite platform.

Only if a `Person` has a very expensive member to initialize that does not benefit from move semantics (such as an array of 10000 double values):

```
class Person {
 private:
  std::string name;
  std::array<double, 10000> values;   // move can't optimize here
 public:
  ...
```

```
};
```

we see a problem with the approach taking the argument by value and move. We have to copy the argument twice, which takes almost twice as much time. See *basics/initbigperf.cpp* for a complete example program to measure this.

4.3.5 Summary for Member Initialization

As a summary, to initialize members for which move semantics makes a significant difference (strings, containers, or classes/arrays having such members) you should use move semantics as follows:

- Either switch from taking the parameter by lvalue reference to taking it by value and move it into the member.
- Or overload the constructors for move semantics.

The first option allows us to have only one constructor so that it is easier to maintain. However, it results in more move operations than necessary. So, if move operations might still take significant time, you might better use multiple overloads.

For example, if we have a class with a string and a vector of values, taking by value and move is usually the right approach:

```
class Person {
 private:
  std::string name;
  std::vector<std::string> values;
 public:
  Person(std::string n, std::vector<std::string> v)
   : name{std::move(n)}, values{std::move(v)} {
  }
  ...
};
```

However, if we have a `std::array` member, you better overload because moving a `std::array` still takes significant time even if the members are moved:

```
class Person {
 private:
  std::string name;
  std::array<std::string, 1000> values;
 public:
  Person(std::string n, const std::array<std::string, 1000>& v)
   : name{std::move(n)}, values{v} {
  }
  Person(std::string n, std::array<std::string, 1000>&& v)
   : name{std::move(n)}, values{std::move(v)} {
  }
  ...
};
```

4.3.6 Should we now Always Pass by Value and Move?

The discussion above leads to the question whether we should now always take arguments by value and move them to set an internal value or member. The answer is **no**.

The special case discussed here is that we *create and initialize* a new value. Then this strategy pays off. But if we already have a value, which we update or modify, using this approach would be counter-productive.

A simple example would be setters. Consider the following implementation of class `Person`:

```
class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  void setFirstname(std::string s) {
    first = std::move(s);
  }
  ...
};
```

If we now call

```
Person p{"Ben", "Cook"};
std::string name1{"Ann"};
std::string name2{"Constantin Alexander"};

p.setFirstname(name1);
p.setFirstname(name2);
p.setFirstname(name1);
p.setFirstname(name2);
```

Each time we set a new firstname we create a new temporary parameter `s` which allocates its own memory, which is then moved to the value of the member. Thus, we have 4 allocations (provided we don't have SSO).

Now consider we implement the setter as we learned it taking a `const` lvalue reference:

```
class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  void setFirstname(const std::string& s) {
```

```
    first = s;
  }
  ...
};
```

Binding `s` to the passed argument will not create a new string. And the assignment operator will only allocate new memory if the new length exceeds the current amount of memory allocated for the value. That means, because we already have a value, the approach taking the argument by value and move might be counter-productive.

However, you might wonder whether to overload the setter so that we can benefit from move semantics if the new length exceeds the existing length:

```
class Person {
 private:
  std::string first;  // first name
  std::string last;   // last name
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
  void setFirstname(const std::string& s) {
    first = s;
  }
  void setFirstname(std::string&& s) {
    first = std::move(s);
  }
  ...
};
```

But even then this might be counter-productive, because a move assignment might shrink the capacity of `first`:

```
Person p{"Ben", "Cook"};

p.setFirstname("Constantin Alexander");  // would allocate enough memory
p.setFirstname("Ann");                   // would reduce capacity
p.setFirstname("Constantin Alexander");  // would have to allocate again
```

Even with move semantics the best approach to set existing values is to take the new values by `const` lvalue reference and assign without using `std::move()`

Taking a parameter by value and moving it to where the new value is needed is only useful, when we store the passed value somewhere as a ***new*** value (so that we anyway need new memory for it). When modifying an existing value, this policy might be counter-productive.

However, initializing members is not the only application of "*take by value and move*." A (member) function ***adding*** a new value to a container would be another:

```
class Person {
 private:
```

```
  std::string name;
  std::vector<std::string> values;
 public:
  Person(std::string n, std::vector<std::string> v)
   : first{std::move(n)}, values{std::move(v)} {
  }
  ...
  // better pass by value and move to create a new element:
  void addValue(std::string s) {
    values.push_back(std::move(s));
  }
  ...
};
```

4.4 Move Semantics in Class Hierarchies

As we have seen, any declaration of a copy constructor, copy assignment, or destructor disables the automatic support for move semantics. This also applies to polymorphic base classes. For example:

```
class GeoObj {
 public:
  virtual void moveBy(Coord) = 0;    // pure virtual function (introducing the API)
  virtual void draw() const = 0;     // pure virtual function (introducing the API)
  ...

  virtual ~GeoObj() = default;       // let delete call the right destructor
  ...                                // other special member functions due to slicing problem
};
```

In this base class, move semantics is disabled, which means that if we move a geometric object the members declared *here* in the base class have no automatic support for move semantics. This also applies if we have a protected copy constructor and a deleted assignment operator, which you should usually have in a polymorphic base class to avoid the problem of slicing.

As long as the base class doesn't introduce members, not supporting move semantics has no effect. However, if we have an expensive member in this base class, you have disabled move support for it. For example:

```
class GeoObj {
 protected:
  std::string name;                  // name of the geometric object
 public:
  ...
  virtual void moveBy(Coord) = 0;    // pure virtual function (introducing the API)
  virtual void draw() const = 0;     // pure virtual function (introducing the API)

  virtual ~GeoObj() = default;       // disables move semantics for name
```

```
  ...                                   // other special member functions due to slicing problem
};
```

To re-enable move semantics again, you can explicitly declare the move operations as defaulted. However, as we just learned, this disables the copying special member functions. So if you want to have them, you explicitly have to provide them.

However, due to the problem of slicing, you should anyway disable assignment operation in polymorphic class hierarchies. And if the class is not abstract, you should also avoid having public copy constructors. For this reason, a polymorphic base class with move semantics (and members) should be declared as follows:

```
class GeoObj {
 protected:
  std::string name;                     // name of the geometric object

  GeoObj(std::string n)
   : name{std::move(n)} {
  }
 public:
  virtual void moveBy(Coord) = 0;       // pure virtual function (introducing the API)
  virtual void draw() const = 0;        // pure virtual function (introducing the API)

  virtual ~GeoObj() = default;                  // would disable move semantics for name
 protected:
  GeoObj(const GeoObj&) = default;              // enable protected copy and move semantics
  GeoObj(GeoObj&&) = default;
  GeoObj& operator= (GeoObj&&) = delete;        // disable assignment operators
  GeoObj& operator= (const GeoObj&) = delete;
  ...
};
```

See *poly/geoobj.hpp* for the complete header file.

Also ensure that the derived classes have no special member functions declared. That is, don't declare a virtual destructor again:[1]

```
class Polygon : public GeoObj {
 protected:
  std::vector<Coord> points;
 public:
  Polygon(std::string s, std::initializer_list<Coord> = {});   // constructor
  virtual void moveBy(Coord c) override;        // implementation of moveBy()
  virtual void draw() const override;           // implementation of draw()
};
```

See *poly/polygon.hpp* for the complete header file.

[1] `virtual` is not necessary if member functions are declared with `override`. However I prefer to have it again for better alignment and the rule that either all or no member functions should be `virtual`.

Declaring a destructor again (whether or not to be `virtual`) would disable automatic support of move semantics for the members of the derived class (here for the vector `points`):

```
class Polygon : public GeoObj {
 protected:
  std::vector<Coord> points;
 public:
  Polygon(std::string s, std::initializer_list<Coord> = {});  // constructor
  virtual void moveBy(Coord c) override;        // implementation of moveBy()
  virtual void draw() const override;           // implementation of draw()

  virtual ~Polygon() = default;  // OOPS: don't do that because it disables move semantics
};
```

Now, move semantics works for both `Polygon` members, `name` and `points`. For example, the following program:

poly/polygon.cpp

```
#include "geoobj.hpp"
#include "polygon.hpp"

int main()
{
  Polygon p0{"TestPolygon",
             {Coord{1,1}, Coord{1,9}, Coord{9,9}, Coord{9,1}}};

  Polygon p1{p0};              // copy
  Polygon p2{std::move(p0)};   // move

  p0.draw();
  p1.draw();
  p2.draw();

  p2.moveBy(Coord{10, 10});
  p2.draw();
}
```

has the following output:

```
polygon '' over
polygon 'TestPolygon' over (1,1) (1,9) (9,9) (9,1)
polygon 'TestPolygon' over (1,1) (1,9) (9,9) (9,1)
polygon 'TestPolygon' over (11,11) (11,19) (19,19) (19,11)
```

4.5 Summary

- Avoid objects with names.
- Avoid unnecessary `std::move()`. Especially don't use it when returning a local object.
- Constructors initializing members from parameters, for which move operations are cheap, should take the argument by value and move it to the member.
- Constructors initializing members from parameters, for which move operations take significant time, should be overloaded for move semantics for best performance.
- In general, creating and initializing new values from parameters, for which move operations are cheap, should take the arguments by value and move. But don't take by value and move to update/modify existing values.

Chapter 5

Overloading on Reference Qualifiers

This chapter discusses overloading member functions for different reference qualifiers. As we will see, this will end the common community question, whether getters should return by value or by constant reference.

5.1 Return Type of Getters

When implementing getters for members that are expensive to copy, before C++11 we have the following alternatives:

- Return by value
- Return by lvalue reference

Let's discuss these alternatives a bit.

5.1.1 Return by Value

A getter returning by value would look like this: Remember: don't return by value using `const`. Otherwise you disable move semantics.

```
class Person
{
 private:
  std::string name;
 public:
  ...
  std::string getName() const {
    return name;
  }
};
```

This code is safe, but each time we ask for the name, we might copy the name.

For example, just checking whether we have a person with an empty name would have significant overhead:

```
std::vector<Person> coll;
...
for (const auto& person : coll) {
  if (person.getName().empty()) {    // OOPS: copies the name
    std::cout << "found empty name\n";
  }
}
```

If you compare this approaches with an approach returning a reference, you can see that the version returning the string by value has a performance overhead by a factor between 2 and 100 (provided the names have a significant length so that SSO does not help). Giving access to an member that is an image or a collection of thousands of elements might even be worse. Such a factor is huge, so that getters in this case often return by (`const`) reference.

5.1.2 Return by Reference

A getter returning by reference would look like this:

```
class Person
{
 private:
  std::string name;
 public:
  ...
  const std::string& getName() const {
    return name;
  }
};
```

This is faster, but a bit unsafe, because the caller has to ensure that the object it calls lives as long as the return value.

However, there is a lifetime risk that you use the return value of the getter longer than the object you call the getter for.

One subtle way to fall into this trap is to use the range-based `for` loop as follows:

```
for (char c : returnPersonByValue().getName()) {  // OOPS: undefined behavior
  if (c == ' ') {
    ...
  }
}
```

Note that on the right side in the header of the loop there is a function returning a temporary object to which we refer with our getter. However, the range-based `for` loop is defined so that the code above is equivalent to the following:

```
auto&& range = returnPersonByValue().getName();  // OOPS: returned temporary destroyed here
for (auto pos = range.begin(), end = range.end(); pos != end; ++pos) {
  char c = *pos;
  if (c == ' ') {
```

```
    ...
  }
}
```

Before we start to iterate, we initialize a reference (of type `auto&&`), because we have to use the passed range twice (once to call `begin()` and once to call `end()` for it) and want to avoid creating a copy of the range (which might be expensive or even not possible). In general, references extend the lifetime to what they refer to. However, here we don't refer to the returned `Person`; here we refer to a reference to a returned `Person`. Thus, we extend the lifetime of the reference but not of the temporary that reference refers to. So with the semicolon of the first statement the returned temporary object gets destroyed and we use a reference to a destroyed object to iterate over its elements.

At best, we get a core dump here so that we see that something went significantly wrong. At worst, we get fatal undefined behavior once we ship the software.

Code like this would not be a problem, if the getter would return the name by value. Then we would extend the lifetime of the return value until the end of the lifetime of the reference `range` referring to it.

5.1.3 Using Move Semantics to Solve the Dilemma

With move semantics, we now have way to solve the dilemma. We can return by reference if it's safe and return by value if we might run into lifetime issues.

The way to program this is as follows:

```
class Person
{
 private:
  std::string name;
 public:
  ...
  std::string getName() && {    // when we no longer need the value
    return std::move(name);     // we steal and return by value
  }
  const std::string& getName() const& {    // in all other cases
    return name;                            // we give access to the member
  }
};
```

We overload the getter by different reference qualifiers as when passing argument to a function parameter overloaded for `&&` and `const&`:

- The version with the `&&` qualifier is used when we have an object where we no longer need the value (an object that is about to die or which we have marked with `std::move()`).
- The version with the `const&` qualifier is used in all other cases. It always fits but is only the fallback if we cannot take the `&&` version. Thus, this function is used if we have an object that is not about to die or marked with `std::move()`.

Now we have both good performance and safety:

```
Person p{"Ben"};
std::cout << p.getName();                    // 1) fast (returns reference)
```

```
std::cout << returnPersonByValue().getName();      // 2) fast (uses move())

std::vector<Person> coll;
...
for (const auto& person : coll) {
  if (person.getName().empty()) {                  // 3) fast (returns reference)
    std::cout << "found empty name\n";
  }
}

for (char c : returnPersonByValue().getName()) {   // 4) safe and fast (uses move())
  if (c == ' ') {
     ...
  }
}
```

Statements 1) and 3) use the version for `const&` because we have an object with a name not marked with `std::move()`. Statements 2) and 4) use the version for `&&` because we call `getName()` for an temporary object not having a name. Because the objects are about to die, the getter can move out the name member as the return value, which means that we don't have to allocate new memory for the return value. We steal the value from the member.

You might remember that that `return` statements should not have a `std::move()` to move out local objects that die anyway. However, in this case we have no local object; we have a member, for which the lifetime does not end with the end of the member function.

5.2 Overloading on Qualifiers

Since C++98, we can overload member functions for implementing a const and a non-const version. For example:

```
class C {
 public:
  ...
  void foo();        // foo() for non-const objects
  void foo() const;  // foo() for const objects
};
```

The qualifiers after the parenthesis allow us to qualify the one object that is not passed to a parameter: the object we call this member function for.

Now, with move semantics we have new ways to overload functions by qualifiers, because we have different reference qualifiers. Consider the following program:

basics/refqual.cpp

```
#include <iostream>

class C {
```

```
 public:
  void foo() const& {
    std::cout << "foo() const&\n";
  }
  void foo() && {
    std::cout << "foo() &&\n";
  }
  void foo() & {
    std::cout << "foo() &\n";
  }
  void foo() const&& {
    std::cout << "foo() const&&\n";
  }
};

int main()
{
  C x;
  x.foo();              // foo() &
  C{}.foo();            // foo() &&
  std::move(x).foo();   // foo() &&

  const C cx;
  cx.foo();             // foo() const&
  std::move(cx).foo();  // foo() const&&
}
```

Note that this program demonstrates which reference qualifiers are possible and called. Usually, we only have two of these overloads, such as using `&&` and `const&` for getters.

Also note that it is not allowed to overload for both reference and non-reference qualifiers:

```
class C {
 public:
  void foo() &&;
  void foo() const;   // ERROR: can't overload by reference and value qualifiers
};
```

5.3 When to Use Reference Qualifiers

Reference qualifiers allow us to implement functions differently when called for objects of a specific value category. The approach is to provide a different implementation, when we call a member function for an object that no longer needs its value.

While we have this feature, it is not used as much as we could. We could (and should) especially use it to ensure that operations modifying objects are not called for temporary objects, which are closed to die.[1]

5.3.1 Reference Qualifiers for Assignment Operators

One example of better using reference qualifiers would even mean to modify the implementation of assignment operators. As proposed in http://wg21.link/n2819 we should better declare the assignment operator with reference qualifiers wherever we can.[2]

For example, the assignment operator for strings are declared as follows:

```
namespace std {
 template<typename charT, ...>
 class basic_string {
  public:
    ...
    constexpr basic_string& operator=(const basic_string& str);
    constexpr basic_string& operator=(basic_string&& str) noexcept(...);
    constexpr basic_string& operator=(const charT* s);
    ...
 };
}
```

This enabled to accidentally assign a new value to a temporary string:

```
std::string getString();

getString() = "hello";    // OK
foo(getString() = "");    // passes string instead of bool
```

By declaring the assignment operator with reference qualifiers:

```
namespace std {
 template<typename charT, ...>
 class basic_string {
  public:
    ...
    constexpr basic_string& operator=(const basic_string& str) &;
    constexpr basic_string& operator=(basic_string&& str) & noexcept(...);
    constexpr basic_string& operator=(const charT* s) &;
    ...
 };
}
```

Code like that would no longer compile:

---

[1] Thanks to Peter Sommerlad for pointing this out.

[2] The paper was formulated after even discussing to change the behavior of the generated assignment operators to have reference qualifiers. See https://wg21.link/cwg733.

```
std::string getString();

getString() = "hello";    // ERROR
foo(getString() = "");    // ERROR
```

Note that especially for types that can be used as Boolean values this would help to find some typical bug:

```
std::optional<int> getValue();

if (getValue() = 0) {     // OOPS: compiles, but is probably wrong
  ...
}
```

Essentially, we give temporary objects back a property they have for fundamental types: they are rvalues, which means that they can't be on the left-hand side of an assignment.

Note that all these proposals to fix the C++ standard accordingly so far were rejected. The main reason were concerns about backward compatibility. However, when implementing your own class, you can make it better as follows:

```
class MyType {
 public:
  ...
  // disable assigning value to temporary objects:
  MyType& operator=(const MyType& str) & =default;
  MyType& operator=(MyType&& str) & =default;
  // because this disables the copy/move constructor, also:
  MyType(const MyType&) =default;
  MyType(MyType&&) =default;
  ...
};
```

In general, you should do it for every member function that might modify an object.

5.3.2 Reference Qualifiers for Other Member Functions

As demonstrated by the example of getters, reference qualifiers could and should also be use when references to objects are returned. That way, we can reduce the danger that we ask for access to a member of a temporary objects that is getting destroyed.

Again, the current declaration of standard strings might serve as an example:

```
namespace std {
 template<typename charT, ...>
 class basic_string {
  public:
   ...
   constexpr const charT& operator[](size_type pos) const;
   constexpr charT&       operator[](size_type pos);
   constexpr const charT& at(size_type n) const;
   constexpr charT&       at(size_type n);
```

```
    constexpr const charT& front() const;
    constexpr charT&       front();
    constexpr const charT& back() const;
    constexpr charT&       back();
    ...
  };
}
```

Instead, we might better have the following overloads:

```
namespace std {
 template<typename charT, ...>
 class basic_string {
  public:
    ...
    constexpr const charT& operator[](size_type pos) const&;
    constexpr charT&       operator[](size_type pos) &;
    constexpr charT        operator[](size_type pos) &&;
    constexpr const charT& at(size_type n) const&;
    constexpr charT&       at(size_type n) &;
    constexpr charT        at(size_type n) &&;

    constexpr const charT& front() const&;
    constexpr charT&       front() &;
    constexpr charT        front() &&;
    constexpr const charT& back() const&;
    constexpr charT&       back() &;
    constexpr charT        back() &&;
    ...
 };
}
```

Again, a corresponding change in the C++ standard might be a problem because of backward compatibility. But you could and should do it with your types. But don't forget that the implementations for rvalue references should move out expensive members, then.

5.4 Summary

- You can overload member functions on different reference qualifiers.
- Overload getters for expensive members with reference qualifiers to make them both safe and fast.
- Use reference qualifiers in assignment operators.

Chapter 6

Moved-from States in Detail

While move semantics as generated or naively implemented usually works fine, we at least should have a look at the possible case that a move operation can bring objects into an state that is not supported by the C++ standard library or breaks invariants of the type. We clarify what an "invalid" state is according to the guarantee of the C++ standard library that moved-from objects are in a *valid but unspecified* state.

6.1 Required and Guaranteed States of Moved-from Objects

After a move operation, the moved-from objects are neither partially nor fully destroyed. The destructor hasn't been called yet and still will be called when the lifetime of the moved-from object ends. So, the destructor at least has to run smoothly.

But the C++ standard library guarantees more for its moved-from types. Moved-from objects are in a "valid but unspecified state". That means you can use a moved-from object just like any object of its type where you don't know its value. It is as if you use a non-`const` reference parameter of the type having no idea about the value of the passed object. By knowing that we can do more than just destroying moved-from objects, we can, for example, use move semantics to implement sorting and mutating algorithms.

To understand in more detail how to deal with moved-from objects, it is better to distinguish between two aspects relevant for them:[1]

- What are the **requirements** for moved-from objects so that you can use them safely with the C++ standard library?
- What are the **guarantees** you should give moved-from objects of your types so that users of these types know how to use them with well-defined behavior?

Usually, the guarantees you give should at least fulfill the requirements of the C++ standard library, but they might guarantee more.

[1] Thanks to Sean Parent for pointing that out.

6.1.1 Required States of Moved-from Objects

The requirements for moved-from objects in the C++ standard library are nothing special. That is, for all functions, the requirements they formulate for the passed types and objects also apply to moved-from objects.

Basically, you always have to be able to destruct a moved-from object. In addition, in many functions you have to be able to assign a new value to a moved-from object.

Consider, for example, how we swap the values of two objects `a` and `b` (which might be part of a sorting). Swapping is usually implemented as follows (see Figure 6.1):

- Move `a` to a new temporary object `tmp` (so that `a` becomes a moved-from object).
- Move assign `b` to the moved-from object `a` (so that `b` becomes a moved-from object).
- Move assign `tmp` to the moved-from object `b` (so that `tmp` becomes a moved-from object).
- Destroy the moved-from object `tmp`.

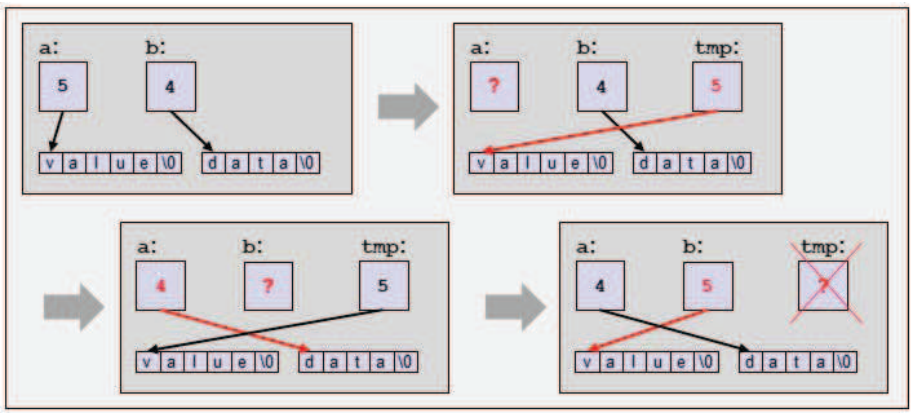


Figure 6.1. Moved-from objects in swap functions

Note that according to the fact that we might call self-swapping (passing the same object to both parameters), it might even happen that we use a moved-from value to assign it to another object.

So, for moved-from object we have the usual requirements. To call `swap()` they have to

- support the creation of a new copy (using the move or copy constructor)
- support the assignment of a value from another object (using the move or copy assignment operator)
- be destructible

To sort objects we in addition have to support to call `operator<` or the sorting criterion for all objects. This also applies to moved-from objects. You might argue that in your sorting algorithm you should know which object was moved so that you can avoid comparing it, but the C++ standard library doesn't require that. This, by the way, also means that you can pass moved-from objects to sort them. As long as they support all required operations, everything is fine.

Note that the definition of the C++ standard library requires more than what is called *partially formed* in "*Elements of Programming*" by *Alexander Stepanov* and *Paul McJones*[2]

So, for all objects and types you use in the C++ standard library, **ensure that moved-from objects also support all requirements** of the called functions.

6.1.2 Guaranteed States of Moved-from Objects

The guarantees for moved-from objects define which code is well-defined when using them. In general, it is up to the designer of a class which guarantees are given. However, because we always at the end of its lifetime destroy an objects the minimum guarantee you always give for moved-from states is that calling the destructor is well defined.

Usually more guarantees are given. For the requirements of the C++ standard library, supporting assignments from other objects of the same type is enough. However, users usually expect that your object also can deal with other ways of "assignments." Therefore, a useful additional guarantee is that you can use any form to "assign" a new value to a moved-from object.

Consider the different ways you can assign a new value to a standard string `s`:

```
s = "hello";               // assign "hello"
a.assign(s2);              // assign the state of s2
s.clear();                 // assign the empty state
std::cin >> s;             // assign the next word from standard input
std::getline(myfile, s);   // assign the next line from a file
```

For example, the following loop is a common way to read line-by-line from a stream into a vector:

```
std::string row;
while (std::getline(myStream, row)) {
  coll.push_back(std::move(row));    // move the line into the vector
}
```

As another example, you can implement the following in generic code dealing with containers:

```
foo(std::move(obj));       // pass obj to foo()
obj.clear();               // ensure the object is empty afterwards
```

Similar code can be useful to release memory for an object a unique pointer uses:

```
draw(std::move(up));       // the unique pointer might or might not give up ownership
up.reset();                // ensure we give up ownership and release any resource
```

By claiming that moved-from objects are in a *valid but unspecified state*, the C++ standard guarantees for its types in the library that *any* operation is well-defined provided it works in general for all possible states.

For example, the following is defined behavior according to the C++ standard:

```
std::stack<int> stk;
...
foo(std::move(stk));       // stk gets unspecified state
stk.push(42);
```

[2] See http://elementsofprogramming.com.

```
...                          // do something else without using stk
int i = stk.top();
assert(i == 42);             // should never fail
```

Although we don't know the value of `stk` after passing it with `std::move()` to `foo()`, we can use it as a valid stack as long as we only use it to hold our value 42 until we need it again.

Of course, it is up to you how far the guarantees you give go. But ensure the users of your type are aware of them. They might expect that your types gives the same guarantee as the C++ library, so that you can use a moved-from object just as any other object of the type not knowing its value.

6.1.3 Broken Invariants

The C++ standard library defines what it means that all moved-from objects are in a "valid but unspecified state" as follows:

> The *value of an object is not specified except that the object's* ***invariants are met*** *and* ***operations*** *on the object* ***behave as specified*** *for its type*

Invariants are the guarantees that apply to **all** of the objects that can be created. With this guarantee you can assume that a moved-from object are in a state so that their invariants are not broken. You can use it like a non-const reference parameter not know anything about the argument passed: You can call any operation that has no constraint or precondition and the effect/result of this call is as specified for any other object of this type.

For example, for a moved-from string we can perform all operations that have no precondition:

- ask for its size
- print it out
- iterate over the characters
- convert it to a C string
- assign a new value
- append a character

And all of these functions still have the usual specified semantics:

- The returned size can be used to safely call the index operator.
- The returned size matches the number of characters when iterating over them.
- The printed characters match the sequence of characters when iterating over them.
- Appending a character will have that character at the end of the value (still not knowing whether and which other characters are in front).

Programmers can make use of these guarantees (see the `std::stack<>` example above).

However, sometimes you have to explicitly ensure that these invariants are not broken. The default special move member functions might not work properly. We will discuss examples in the following subsections.

Restricted Invariants

It might also also be useful *not* to give the full guarantee of the C++ standard library to moved-from states. That is, you might intentionally restrict the possible operations for moved-from objects. For example, when a valid state of an object always need resources such as memory, having only a partially supported state might be better to make move operations cheaper.

We have such a case in the C++ standard library: Some implementations need always memory for all states of node-based containers (so that even the default constructor has to allocate memory). For these containers, it might have been better to restrict what we can do with moved-from objects instead of guaranteeing that moved-from objects are always in a "valid" state. But we decided to give the full guarantee.

Ideally, a moved-from state that doesn't support all operations should be supported by the type. The objects should know this state and rejects to perform the corresponding operation. That way, you can also argue that you just extended your invariants and added additional requirements to your operations. In any case, make sure users of your types know about what is well-defined and what not.

6.2 Destructible and Assignable

Let's discuss how to ensure that moved-from objects fulfill the basic requirements to support assignment and destruction.

6.2.1 Assignable and Destructible Moved-from Objects

In most of the classes, the generated special move member functions bring moved-from objects into a state where the assignment operator and destructor just work fine. However, provided each moved-from member is assignable and destructible both an assignment and the destruction of the moved-from object as a whole should work well:

- The assignment will overwrite the unspecified state of the member by assigning the state from the corresponding source member
- The destructor will destroy the member (having the unspecified state)

Because we can and should usually expect for the members that move-from objects are in a valid but unspecified state, generated move operations usually just work and create the right state.

For example, consider the following class:

```
class Customer {
 private:
  std::string name;
  std::vector<int> values;
  ...
};
```

When the value of a customer is moved away, both `name` and `values` are guaranteed to have a valid state so that the destructor for them (called by the destructor of `Customer`) works fine:

```
void foo()
{
  Customer c{"Michael Spencer"};
```

```
    ...
    process(std::move(c));
    // both name and values have valid but unspecified states
    ...
} // destructor of c will clean-up name and values (whatever their state is)
```

Also, assigning a new value to `c` will work, because we assign both the `name` and the `values`.

6.2.2 Non-Destructible Moved-from Objects

However, in rare cases problems might occur. In these cases we usually have implemented an assignment operator or a destructor to do more than just assigning or destroying the members.[3]

Consider the following class, where we use a fixed-size array of a variable number of thread objects, for which we explicitly call `join()` in the destructor to wait for their end:

basics/tasks.hpp

```
#include <array>
#include <thread>

class Tasks {
 private:
  std::array<std::thread,10> threads;   // array of threads for up to 10 tasks
  int numThreads{0};                    // current number of threads/tasks
 public:
  Tasks() = default;

  // pass a new thread:
  template <typename T>
  void start(T op) {
    threads[numThreads] = std::thread{std::move(op)};
    ++numThreads;
  }
  ...

  // at the end wait for all started threads:
  ~Tasks() {
    for (int i = 0; i < numThreads; ++i) {
      threads[i].join();
    }
  }
};
```

[3] We see here a reason why move semantics is by default disabled when a destructor is implemented.

So far, the class doesn't support move semantics because we have a user-declared destructor. You also can't copy `Tasks` objects because copying `std::threads` is disabled. But you can start multiple tasks and wait for their end:

basics/tasks.cpp

```
#include "tasks.hpp"
#include <iostream>
#include <chrono>

int main()
{
  Tasks ts;
  ts.start([]{
             std::this_thread::sleep_for(std::chrono::seconds{2});
             std::cout << "\nt1 done" << std::endl;
           });
  ts.start([]{
             std::cout << "\nt2 done" << std::endl;
           });
}
```

Now consider enabling move semantics by generating the default implementation of the move operations (see *basics/tasksbug.cpp*):

```
class Tasks {
 private:
  std::array<std::thread,10> threads;   // array of threads for up to 10 tasks
  int numThreads{0};                    // current number of threads/tasks
 public:
  ...
  // OOPS: enable default move semantics:
  Tasks(Tasks&&) = default;
  Tasks& operator=(Tasks&&) = default;

  // at the end wait for all started threads:
  ~Tasks() {
    for (int i = 0; i < numThreads; ++i) {
      threads[i].join();
    }
  }
};
```

In this case you are in trouble because the default generated move operation may create invalid `Tasks` states. Consider:

basics/tasksbug.cpp

```
#include "tasksbug.hpp"
#include <iostream>
#include <chrono>
#include <exception>

int main()
{
  try {
    Tasks ts;
    ts.start([]{
               std::this_thread::sleep_for(std::chrono::seconds{2});
               std::cout << "\nt1 done" << std::endl;
             });
    ts.start([]{
               std::cout << "\nt2 done" << std::endl;
             });

    // OOPS: move tasks:
    Tasks other{std::move(ts)};
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
  }
}
```

At the beginning we start two tasks by passing them to the `Tasks` object `ts`. So, in `ts` the `threads` array has two entries and `numThreads` is 2. Unfortunately, the move operations of containers move the elements, so that after the `std::move() ts` no longer contains any thread objects representing a running thread. But `numThreads` is just copied, so that we create an inconsistent/invalid state. The destructor will finally loop over the first two elements calling `join()`, which throws an exception (which is a fatal error in a destructor).

The general problem is that two members together define a valid state and the default generated move semantics creates an inconsistency so that even the destructor fails. You can't always avoid this problem, because it can always happen that you explicitly have to do something with a subset of the elements an object contains when the object gets destroyed. In all these cases, the default generated move operations might not work and you should either disable or fix them.

A fix might be:

- Fixing the destructor to deal with the moved-from state (in this case we could, for example, only call `join()` if the thread object is `joinable()`).
- Implement the move operations yourself.
- Disable move semantics (which would be the behavior here without declaring special move member functions).

The whole problem is also a side effect of a design mistake of class `std::thread`. The type doesn't follow the RAII principle. You have to call `join()` for all running threads in the destructor (otherwise the

destructor of a thread throws). Since C++20, you could and should use class `std::jthread` instead, which will automatically call `join()` only there is a thread that is still joinable.

6.3 Dealing with Broken Invariants

Unfortunately, moved-from objects can a lot easier break the "valid but unspecified state" guarantee than breaking the requirement to be destructible. We can accidentally bring objects in a state where we break its invariants.

Fortunately, this is only a problem if `std::move()` is used, because temporary objects are destroyed immediately anyway.

In principle, if the invariants of a class are broken by a (generated) move operation, you have the following options:

- Fix the move operations to bring the moved-from objects into a state not breaking the invariants.
- Disable move semantics.
- Relax the invariants defining all possible moved-from states also as valid. This especially might mean that member functions and functions using the objects have to be implemented differently to deal with the new possible states.
- Document the broken invariants so that users of the type don't use an object of this type after it was marked with `std::move()` (or only use a limited set of operations).

Let's look at some example of broken invariants and discuss how to fix them.

6.3.1 Breaking Invariants Due to a Moved Value Member

The first reason for move operations breaking invariants are objects where the moved-from state of a member is a problem in itself, because the state is valid but should not happen.

Consider we have a class for the cards of a card game.[4] Assume each object is a valid card, such as eight-of-hearts or king-of-diamonds. Assume also that for whatever reason the value is a string and that the invariant of the class is that each and every object has a state representing a valid card. This would mean that we probably don't have a default constructor and that an initializing constructor asserts that the value is valid. For example:

```
class Card {
 private:
  std::string value;            // rank + "-of-" + suit
 public:
  Card(const std::string& v)
   : value{v} {
      assertValidCard(value);   // ensure the value is always valid
  }

  std::string getValue() const {
```

[4] Thanks to Tony Van Eerd for this example.

```
    return value;
  }
};
```

In this class the generated special move member functions create an invalid state breaking the invariant of the class that the value is always the *rank* followed by `"-of-"` followed by the suit (such as `"queen-of-hearts"`).

As long as we don't use `std::move()` this is not a problem (the destructor of the type works fine for a moved-from string. But when we call `std::move()` we can get into trouble. Assigning a new value works fine:

```
std::vector<Card> deck;
...
Card c{std::move(deck[0])};        // deck[0] has invalid state
...
deck[0] = Card{"ace-of-hearts"};   // deck[0] is valid again
```

But printing the value of the card might fail:

```
std::vector<Card> deck;
...
Card c{std::move(deck[0])};        // deck[0] has invalid state
...
print(deck[0]);                    // passing an object with broken invariant
```

If the print function assumes that the invariant is not broken, we might get a core dump:

```
void print(const Card& c) {
  std::string val{c.getValue()};
  auto pos = val.find("-of-");     // requires that we find such a position
  std::cout << val.substr(0, pos) << ' '
            << val.substr(pos+4) << '\n';
}
```

This code might fail at runtime for a moved-from card it is no longer guaranteed that the value contains `"-of-"`. In that case, `find()` initializes `pos` with `std::string::npos`, which causes a `std::out_of_range` exception. when `pos+4` is used as the first argument of `substr()`.

See *basics/card.hpp* and *basics/card.cpp* for the full example.

The options to fix this class are as follows:

- Disable move semantics:

```
class Card {
  ...
  Card(const Card&) = default;              // disable move semantics
  Card& operator=(const Card&) = default;   // disable move semantics
};
```

  However that makes move operations (which for example are called by `std::sort()`) more expensive.

- Disable copying and moving at all:

```
class Card {
  ...
```

```
  Card(const Card&) = delete;              // disable copy/move semantics
  Card& operator=(const Card&) = delete;   // disable copy/move semantics
};
```

However, then you can no longer shuffle or sort cards.

- Fix the broken special move member functions.

  However, what would be a valid fix (is always assigning a "default value" such as `"ace-of-clubs"` fine)? And how to ensure that objects with the default value perform well not allocating memory?
- Internally allow the new state but disallow calling `getValue()` or other member functions.

  You can only document this ("For moved-from objects you are only allowed to assign a new value. All other member functions have the precondition that the object is not in a moved-from state.") or even check this inside the member functions.
- Extend the invariant by introducing a new state that a `Card` might have no value (`value` is empty).

  This means you have to implement the moving special member functions, because you have to ensure that for a moved-from object the member `value` is empty. This also is an opportunity to provide a default constructor, which initialized cards having the same state as a moved-from object.

  But with this change the users of this class have to take into account that the value of the string might be empty and update their code accordingly. For example:

```
void print(const Card& c) {
  std::string val{c.getValue()};
  auto pos = val.find("-of-");
  If (pos != std::string::npos) {
    std::cout << val.substr(0, pos) << ' '
              << val.substr(pos+4) << '\n';
  }
  else {
    std::cout << "no value\n";
  }
}
```

It seems that there is no obvious perfect solution. You have to think about what each of these fixes means to the larger invariants of your program (i.e., that there is only one ace-of-clubs, or that all cards are valid cards, etc.) and decide which one to take.

Note that this class worked fine before C++11, where move semantics wasn't supported (which might mean the first option is the best one). Thus, C++11 might create new states classes that were not possible when the class was implemented. It is a rare case but that way move semantics can break existing code.

6.3.2 Breaking Invariants due to Moved Consistent Value Members

The second reason for move operations to break invariants are objects where two members have to be consistent, which might be broken by a moving special member function. As seen in the example of an array of threads this can even create an inconsistency that breaks the destructor. However, more often the destructor works fine, but the moved-from state breaks an invariant.

Consider a class where we have two different representations of a value, an integral and a string value:

basics/intstring.hpp

```
#include <iostream>
#include <string>

class IntString
{
  private:
    int val;            // value
    std::string sval;   // cached string representation of the value
  public:
    IntString(int i = 0)
     : val{i}, sval{std::to_string(i)} {
    }
    void setValue(int i) {
      val = i;
      sval = std::to_string(i);
    }
    ...
    void dump() const {
      std::cout << " [" << val << "/'" << sval << "']\n";
    }
};
```

In this class, we usually make sure that the member `val` and the member `sval` are just two different representations of the same value. That means that in the implementation and use of this class we would usually expect that both the `int` and the `string` representation of its state are consistent. However, if we call a move operation here, we will keep the value `val`, but `sval` will no longer be guaranteed to have the string representation of `val`.

For example, the following program

basics/intstring.cpp

```
#include "intstring.hpp"
#include <iostream>

int main()
{
  IntString is1{42};
  IntString is2;
  std::cout << "is1 and is2 before move:\n";
  is1.dump();
  is2.dump();

  is2 = std::move(is1);
```

```
  std::cout << "is1 and is2 after move:\n";
  is1.dump();
  is2.dump();
}
```

typically has the following output (but don't forget that a moved-from string becoming empty is typical but not guaranteed):

```
is1 and is2 before move:
 [42/'42']
 [0/'0']
is1 and is2 after move:
 [42/'']
 [42/'42']
```

That is, the automatically generated move operation breaks our invariant that both members always match each other.

How big is this problem? Well, it is at least a possible trap. Again, you might argue that after a move we should no longer use the value (until we set the value again). However, programmers might expect a policy as for objects of the C++ standard library that objects are in a valid but unspecified state.

The fact is that with corresponding getters, the class no longer guarantees that the value as `int` and `string` match, which is probably a class invariant here (implicitly or explicitly stated). You might think that the worst consequence is that the value (which is unspecified now) looks different depending how you use it, but there is no problem using it because it is still a valid `int` or `string`.

However, code counting on this invariant might be broken. That code might assume that the string representation has at least one digit. So if it for example searches for the first or last digit you will definitely find one. For a moved-from string, which typically is empty, this no longer is the case. So, code, not double-checking that there is any character in the string value, might run into unexpected undefined behavior.

Again, it is up to the designer of the class how to deal with this problem. But if you follow the rule of the C++ standard library, you should better bring your moved-from object in a valid state, which might mean that you introduce a possible state not having any value.

In general, when the state of an object has members that depend on each other in some way, you have to explicitly ensure that the moved-from state is a valid state. Examples where this might be broken are:

- We have different representations of the same value, but some of them were moved away
- A member such as a counter corresponds with the number of elements in a member.
- A Boolean value claims that a string value was validated but the validated value was moved away
- A cached value for the average values of all elements is still there but the values (being in a container member) were moved away

Note again that this class worked fine before C++11, where move semantics wasn't supported. The invariant was broken when switching to C++11 or later.

6.3.3 Breaking Invariants due to Moved Members with Reference Semantics

The third reason for move operations breaking invariants are objects with members that have reference semantics such as a (smart) pointer.

Consider the following example of a class where objects use a `std::shared_ptr<>` to share an integral value:[5]

```
class SharedInt {
  private:
    std::shared_ptr<int> sp;
  public:
    explicit SharedInt(int val)
     : sp{std::make_shared<int>(val)} {
    }

    std::string asString() const {
      return std::to_string(*sp);      // OOPS: assume there is always a string value
    }
};
```

The objects of this class get an initial integral value that can be shared with copies of them. As long as new objects are only copied everything is fine:

```
SharedInt si1{42};
SharedInt si2{si1};                    // si1 and si2 share the value 42

std::cout << si1.asString() << '\n';   // OK
```

Being only copied, the `SharedInt` member `sp` always has allocated memory for its value (either from `std::make_shared<>()` or from copying an existing shared pointer with allocated memory).

However, the moment we use move semantics, we run into undefined behavior if we still use the moved-from object:

```
SharedInt si1{42};
SharedInt si3{std::move(si1)};         // OOPS: moves away the allocated memory in si1

std::cout << si1.asString() << '\n';   // undefined behavior (probably core dump)
```

The problem is that inside the class we don't deal correctly with the fact that the value might have been moved away, which can happen because the default generated move operations call the move operations of the shared pointer, which moves away the ownership from the original object. So the moved-from state of a `SharedInt` brings the member `sp` in the situation that it no longer owns an object, which is not handled properly in its member function `asString()`.

You might argue that calling `asString()` for an object with a moved-from state makes no sense, because you are using an unspecified value. but at least the standard library guarantees for its types that moved-from

[5] Thanks to Geoffrey Romer and Herb Sutter for providing the idea of this example in an email discussion of the C++ standard library working group.

types are in a valid state so that you can call all operations that have no constraints. Not giving the same guarantee in a user-defined type can be surprising for users of the type.

From a perspective of robust programming (avoiding surprises, traps, and undefined behavior), I would usually recommend to follow the rule of the C++ standard library. That is: move operations should not bring objects into a state that breaks invariants.

In this case we have to do one of the following:

- Fix all broken operations of the class by also dealing correctly with all possible moved-from states
- Disable move semantics so that there is no optimization when copying objects
- Implement move operations explicitly
- Adjust and document the invariant (constraints/preconditions) for the class or specific operations (such as "It is undefined behavior to call `asString()` for a moved-from-object")

Because allocating memory is expensive, probably the best fix in this case is to deal correctly with the fact the ownership of the integral value might be moved away. That would create a state a default constructor would have, which we could introduce with this change.

The following subsections demonstrate this and the other code fixes.

Fixing Broken Member Functions

The first option, fixing all broken operations, essentially means that we extend the invariant of the class (possible states of all objects) so that all operations can even deal with a moved-from state. We still have to make design decisions. For example, when calling `asString()` for a moved-from object (or more general an object where the shared pointer does not own an integral value), we can:

- still return a fallback value:

```
class SharedInt {
    ...
    std::string asString() const {
      return sp ? std::to_string(*sp) : "";
    }
    ...
};
```

- throw an exception:

```
class SharedInt {
    ...
    std::string asString() const {
      if (!sp) throw ...
      return std::to_string(*sp);
    }
    ...
};
```

- force a runtime error:

```
class SharedInt {
    ...
```

```
    std::string asString() const {
      assert(sp);
      return std::to_string(*sp);
    }
    ...
};
```

Disabling Move Semantics

The second option is to disable move semantics so that only copy semantics can be used. We described earlier how to disable move semantics. You have to user-declare another special member function. Usually you `=default` the copying special member functions:

```
class SharedInt {
  ...
  SharedInt(const SharedInt&) = default;              // disable move semantics
  SharedInt& operator=(const SharedInt&) = default;   // disable move semantics
  ...
};
```

Implementing Move Semantics

The third option is to implement the move operations so that they don't break the invariants of the class.

For this, we have to decide what the state of a moved-from object should be. To support that `asString()` can call `operator*` without checking whether a value exists, we always would have to provide a value. We could, for example, have a static moved-from value which we assign to objects where the value is moved away:[6]

basics/sharedint.hpp

```
#include <memory>
#include <string>

class SharedInt {
  private:
    std::shared_ptr<int> sp;
  public:
    explicit SharedInt(int val)
     : sp{std::make_shared<int>(val)} {
    }

    std::string asString() const {
      return std::to_string(*sp);    // OOPS: unconditional deref
```

[6] Note that `inline static` members are only supported since C++17. Before C++17 you would have to define the member in a separate CPP file to follow the *one definition rule* (ODR).

```
    }

    // fix moving special member functions
    inline static std::shared_ptr<int> moveFromValue{std::make_shared<int>(0)};

    SharedInt (SharedInt&& si)
     : sp{std::move(si.sp)} {
        si.sp = moveFromValue;
    }
    SharedInt& operator= (SharedInt&& si) {
      if (this != &si) {
        sp = std::move(si.sp);
        si.sp = moveFromValue;
      }
      return *this;
    }

    SharedInt (const SharedInt&) = default;
    SharedInt& operator= (const SharedInt&) = default;
};
```

Note that we have to `=default` the copying special member functions, because these are disabled when having user-declared special move member functions.

6.4 Summary

- Clarify the possible state of moved-from objects. You have to ensure that they are at least destructible (which without implementing special member functions is usually the case). But users of your type might expect/require more.
- The requirements of functions of the C++ standard library also apply to moved-from objects.
- Generated special move member function might bring moved-from objects into an state so that a class invariant is broken. This especially might happen if:
  - Classes have no default constructor (and therefore no natural moved-from state)
  - Values of members have restrictions (such as assertions)
  - Values of members depend on each other
  - Members with reference semantics are used (pointers, smart pointers, etc.)
- If the moved-from state breaks invariants or invalidates operations, you should fix this by one of the following options:
  - Disable move semantics
  - Fix the implementation of move semantics
  - Deal with broken invariants inside the class and hide them to the outside

- Relax the invariants of the class by documenting the constraints and preconditions for moved-from objects

Chapter 7

Move Semantics and `noexcept`

When move semantics was almost done for C++11, we detected a problem: Vector reallocations couldn't use move semantics. As a consequence the new keyword `noexcept` was introduced.

This chapter explains the problem and what this means for the use of `noexcept` in C++ code.

7.1 Move Constructors with and without `noexcept`

Let's introduce the topic by a small example motivating the `noexcept` keyword.

7.1.1 Move Constructors without `noexcept`

Consider the following class, which introduces a type with a string member and implements copy and move constructor to make it visible when these are called:

basics/person.hpp

```
#include <string>
#include <iostream>

class Person {
  private:
    std::string name;
  public:
    Person(const char* n)
     : name{n} {
    }

    std::string getName() const {
      return name;
    }
```

```
    // print out when we copy or move:
    Person(const Person& p)
     : name{p.name} {
        std::cout << "COPY " << name << '\n';
    }
    Person(Person&& p)
     : name{std::move(p.name)} {
        std::cout << "MOVE " << name << '\n';
    }
    ...
};
```

Now let's create and initialize a vector of Persons and insert a Person later when it exists:

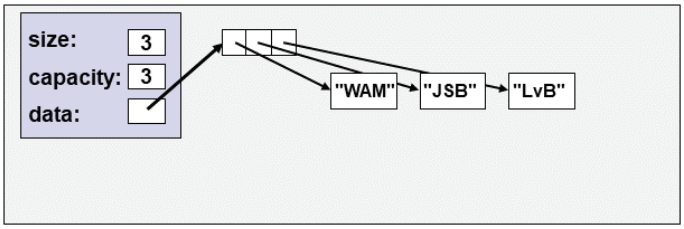
basics/person.cpp

```
#include "person.hpp"
#include <iostream>
#include <vector>

int main()
{
  std::vector<Person> coll{"Wolfgang Amadeus Mozart",
                           "Johann Sebastian Bach",
                           "Ludwig van Beethoven"};
  std::cout << "capacity: " << coll.capacity() << '\n';
  coll.push_back("Pjotr Iljitsch Tschaikowski");
}
```

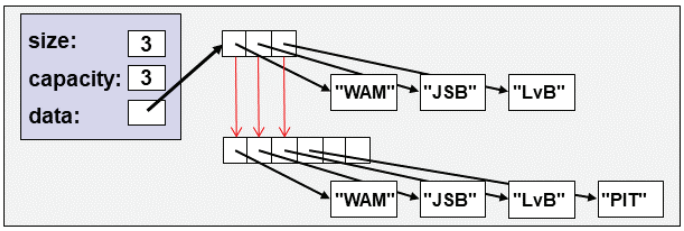
The output of the program is as follows:

```
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
MOVE Pjotr Iljitsch Tschaikowski
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
```

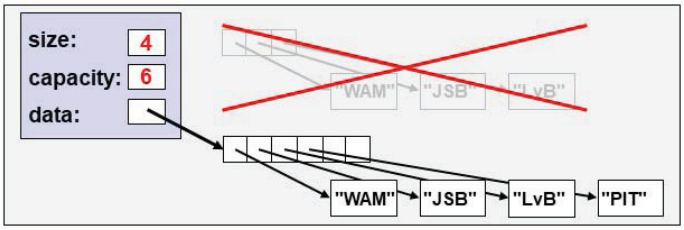
We first copy the initial values into the vector (the reason is that the std::initializer_list<> constructor of a value takes the passed arguments by value) As a result, the vector typically allocates memory for 3 elements:



The important thing here is what happens next: Because the vector internally needs more memory, it allocates the new memory (for say 6 elements), moves in the fourth string (we create a temporary `Person` and move it with `push_back()` into the vector), but also **copies** the existing elements to the new memory:



And at the end it destroys the old elements, releases the old memory for the elements and updates its members:



The question is, why doesn't the vector use the move constructor to move the elements from the old to the new memory.

Strong Exception Safety Guarantee

The reason that vector reallocation doesn't use move semantics is the *strong exception handling guarantee* we give for `push_back()`: When in the middle of the reallocation of the vector and exception gets thrown, we roll back the vector to its previous state. That is, `push_back()` give kind of a transactional guarantee: Either it succeeds or it has no effect.

We could give this guarantee for C++98, because we could only copy the elements, and if something went wrong, we simply destroy the copies created so far and had the vector in its previous state (we assume and require that destructors don't throw; otherwise we could not roll back).

Reallocation is a perfect place for move semantics, because we move elements from one location to the other. So, since C++11 we want to use move semantics here. However, then we are in trouble: If during the reallocation an exceptions gets thrown, we might not be able to roll back, because the elements in the new

memory have already stolen the values of the elements in the old memory. So, destroying the new elements would not work, we have to move them back. But how do we know that moving them back does not fail?

Now you might argue that a move constructor should never throw. This might be correct for strings (because we just move integral values and pointers around), but because we require that moved-from objects are in a valid state, it can happen that this state needs memory, what means that the move might throw, if we are out of memory (node-based containers of Visual C++ are, for example, implemented that way).

We could also not give up the guarantee, because programs might have used this feature to avoid creating a backup of a vector and losing that data could be (safety) critical. And no longer supporting `push_back()` would be a nightmare for the acceptance of C++11.

So, the final decision was to use move semantics on reallocation only when the move constructor of the element types guarantees not to throw.

7.1.2 Move Constructors with `noexcept`

So, our small example program changes its behavior when we guarantee that the move constructor of class `Person` never throws:

basics/personmove.hpp

```
#include <string>
#include <iostream>

class Person {
  private:
    std::string name;
  public:
    Person(const char* n)
     : name{n} {
    }

    std::string getName() const {
      return name;
    }

    // print out when we copy or move:
    Person(const Person& p)
     : name{p.name} {
        std::cout << "COPY " << name << '\n';
    }
    Person(Person&& p) noexcept    // guarantee not to throw
     : name{std::move(p.name)} {
        std::cout << "MOVE " << name << '\n';
    }
    ...
};
```

If we now use Person the same way:

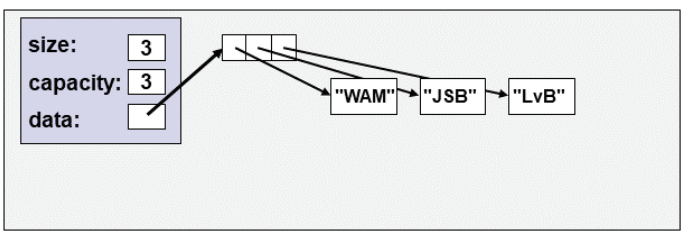
basics/personmove.cpp

```
#include "personmove.hpp"
#include <iostream>
#include <vector>

int main()
{
  std::vector<Person> coll{"Wolfgang Amadeus Mozart",
                           "Johann Sebastian Bach",
                           "Ludwig van Beethoven"};
  std::cout << "capacity: " << coll.capacity() << '\n';
  coll.push_back("Pjotr Iljitsch Tschaikowski");
}
```

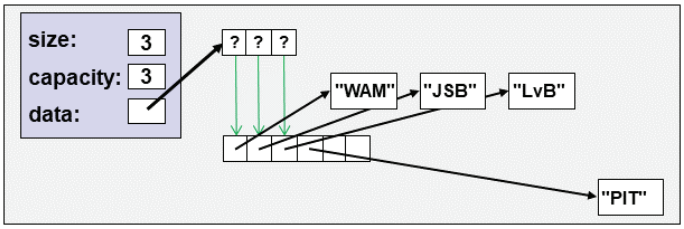
we get the following output:

```
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
MOVE Pjotr Iljitsch Tschaikowski
MOVE Wolfgang Amadeus Mozart
MOVE Johann Sebastian Bach
MOVE Ludwig van Beethoven
```

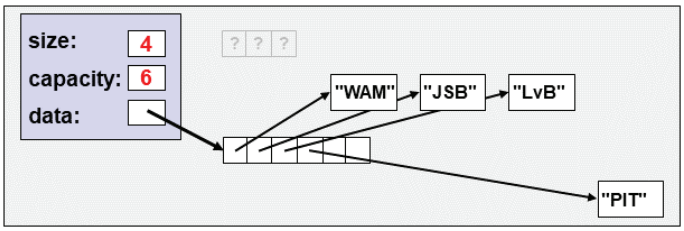
Again, we first copy the initial values into the vector



But as you can see, the vector now uses the move constructor to move its elements to the new reallocated memory:



So that at the end it only has to release the old memory and update its members:



Conditional `noexcept` Declarations

However, is it OK to mark the move constructor with `noexcept`? Well, we move the name (a `std::string`) and write something to the standard output stream. If we throw there, we violate the guarantee not to throw. In that case the program will at runtime call `std::terminate()`, which usually calls `std::abort()` to signal an abnormal end of the program (and often creates a core dump).

So we better give the guarantee not to throw if the string member and the output does not throw. The keyword `noexcept` was introduced because you can use it to specify a conditional guarantee not to throw. It would look as follows (see *`basics/personcond.hpp`* for the full example):

```
class Person {
  private:
    std::string name;
  public:
    ...
    Person(Person&& p)
         noexcept(std::is_nothrow_move_constructible<std::string>::value
                  && noexcept(std::cout << name))
     : name{std::move(p.name)} {
        std::cout << "MOVE " << name << '\n';
    }
    ...
```

With `noexcept(...)` we guarantee not to throw if the compile-time expression within the parentheses is `true`. Here we require two things to give the guarantee:

- With `std::is_nothrow_move_constructible<std::string>::value` we use a standard type trait (type function) to tell us, whether the move constructor of `std::string` guarantees not to throw.

- With `noexcept(std::cout << name)` we ask whether the call of the output expression for the name guarantees not to throw. Here we use `noexcept` as an operator telling us whether the corresponding operations all guarantee not to throw.

As you might imagine, with this declaration reallocation will use the copy constructor again. The move constructor for strings does guarantee not to throw, but the output operator does not. However, the move constructor usually does not output anything; so in general when members do not throw, we can give the guarantee not to throw for the move constructor as a whole.

The good news is that the compiler will detect noexcept guarantees for you if you don't implement the move constructor yourself. For classes where all members guarantee not to throw in the move constructor a generated or defaulted move constructor will give the guarantee as a whole.

Consider with the following declaration:

basics/persondefault.hpp

```
#include <string>
#include <iostream>

class Person {
  private:
    std::string name;
  public:
    Person(const char* n)
     : name{n} {
    }

    std::string getName() const {
      return name;
    }

    // print out when we copy:
    Person(const Person& p)
     : name{p.name} {
        std::cout << "COPY " << name << '\n';
    }
    // force default generated move constructor:
    Person(Person&& p) = default;
    ...
};
```

Now, we request to provide a default generated move constructor:

```
class Person {
    ...
    // force default generated move constructor:
    Person(Person&& p) = default;
    ...
```

```
};
```

This means, we print only when we copy. When the defaulted move constructor is used, we only see that we don't perform a copy.

Now let's use out usual program with printing out some Persons at the end:

basics/persondefault.cpp

```
#include "persondefault.hpp"
#include <iostream>
#include <vector>

int main()
{
  std::vector<Person> coll{"Wolfgang Amadeus Mozart",
                           "Johann Sebastian Bach",
                           "Ludwig van Beethoven"};
  std::cout << "capacity: " << coll.capacity() << '\n';
  coll.push_back("Pjotr Iljitsch Tschaikowski");

  std::cout << "name of coll[0]: " << coll[0].getName() << '\n';
}
```

We get the following output:

```
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
name of coll[0]: Wolfgang Amadeus Mozart
```

We only see the copies for the elements in the initializer list. For everything else, including the reallocation, the default move constructor is used. As we can see, the name of the first person is correct in the reallocated memory.

If we skip to specify any member functions at all, we have the same behavior. That means:

- If you ***implement*** a move constructor, you should declare, whether and when it guarantees not to throw.
- If you don't have to implement the move constructor, you don't have to specify anything at all.

If the performance of a class or the reallocation objects of this class is important for you, you might also want to double check at compile-time that the move constructor of your class guarantees not to throw:

```
static_assert(std::is_nothrow_move_constructible<Person>::value, "");
```

or since C++17 just:

```
static_assert(std::is_nothrow_move_constructible_v<Person>);
```

7.1.3 Is noexcept Worth It?

You might wonder whether declaring move constructors with more or less complicated noexcept expressions is worth it. Howard Hinnant demonstrated the effect with a simple program (slightly adopted for this book):

basics/movenoexcept.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <chrono>

// string wrapper with move constructor:
struct Str
{
  std::string val;

  // ensure each string has 100 characters:
  Str()
   : val(100, 'a') {    // don't use braces here
  }

  // enable copying:
  Str(const Str&) = default;

  // enable moving (with and without noexcept):
  Str (Str&& s) NOEXCEPT
   : val{std::move(s.val)} {
  }
};

int main()
{
  // create vector of 1 Million wrapped strings:
  std::vector<Str> coll;
  coll.resize(1000000);

  // measure time to reallocate memory for all elements:
  auto t0 = std::chrono::steady_clock::now();
  coll.reserve(coll.capacity() + 1);
  auto t1 = std::chrono::steady_clock::now();

  std::chrono::duration<double, std::milli> d{t1 - t0};
  std::cout << d.count() << "ms\n";
}
```

We provide a class wrapping a string of significant length (to avoid the *small string optimization*). Note that we have to initialize the value with parentheses because braces would interpret the 100 as the initial character with the value 100.

In the class we mark the move constructor with `NOEXCEPT`, which can be replaced by the preprocessor by nothing or `noexcept` (e.g., compile with `-DNOEXCEPT=noexcept`). And then we measure how long it takes to reallocate 1 Million of these objects in a vector.

On almost all platforms declaring the move constructor with `noexcept` makes the reallocation faster by a factor of up to 10 (make sure some optimization level is turned on). That is, a reallocation (typically forced by inserting a new element) might take about 20 instead of 200 Milliseconds. These are 180 Milliseconds less, in which we can't use the vector for anything. That can be huge.[1]

7.2 Details of `noexcept`

******* This chapter/section is at work *******

7.3 When and Where to Use `noexcept`

As seen, `noexcept` was introduced in C++11 as a response of the problem that we can't use move semantics, when reallocating elements in a vector.

But in principle you can now mark every function with a (conditional) `noexcept`. This might not only help in situations like vector reallocations (note that the move constructor might call other functions, where we also need the `noexcept` guarantee then), but we might also help the compiler to optimize code because no code to handle exceptions has to be generated. So the question is, where to place `noexcept` in your code.

For the C++11 standard library, we were in a hurry to decide where to place `noexcept` (the reallocation problem was detected very late and the final semantic meaning of `noexcept` was clarified right in the week where we shipped C++11). So, we followed a pretty conservative approach that was proposed in `http://wg21.link/n3279`, which could also be a useful guideline for your code. Roughly speaking the guideline is:

- Each library function that the library working group agree cannot throw, and having a "wide contract" (i.e., does not specify undefined behavior due to a precondition), should be marked as unconditionally noexcept.
- If a library swap function, move constructor, or move assignment operator is "conditionally wide" (i.e., can be proven not to throw by applying the `noexcept` operator) then it should be marked as conditionally `noexcept`. No other function should use a conditional `noexcept` specification.
- No library destructor should throw. It shall use the implicitly supplied (non-throwing) exception specification.
- Library functions designed for compatibility with C code may be marked as unconditionally `noexcept`.

[1] Note that there is a bug in Visual C++ 2015 so that move is always used on reallocation and we don't see a difference. This bug is fixed with Visual C++ 2017 (which might slow down the reallocation, if a move constructor does not guarantee not to throw).

What the first item means demonstrates the following example:

- For containers and strings the member functions `empty()` and `clear()` are marked with `noexcept`, because there is no useful way to implement them in a way that they might throw.
- The index operator of vectors and strings is *not* marked with `noexcept` although when correctly used it guarantees not to throw. However, when passing an invalid index, we have undefined behavior and implementations are allowed to do whatever they want. By not declaring them with `noexcept` these implementations could throw in this case.

For move semantics the second item is the guideline. We propose that you only use a conditional `noexcept` declaration when implementing a move constructor, a move assignment operator, or a `swap()` function. For all other functions it is usually not worth the effort to think about the detailed conditions and you might reveal too much implementation details.

With http://wg21.link/p0884 we added that wrapping types also should have conditional `noexcept` declarations:

- If a library type has wrapping semantics to transparently provide the same behavior as the underlying type, then the default constructor, copy constructor, and copy-assignment operator should be marked as conditionally `noexcept` so that the underlying exception specification still holds.

Finally note that by rule even an implemented destructor is declared as `noexcept` unless you explicitly specify your own `noexcept` condition (such as `noexcept(false)`).

As a consequence for your code:

- You should declare the move constructor, the move assignment operator and a `swap()` function with a (conditional) `noexcept`.
- For all other functions mark them just with an unconditional `noexcept` if you know that they can't throw.
- Destructors don't need a `noexcept` specification.

7.4 Summary

- With `noexcept` you can declare a (conditional) guarantee not to throw.
- If you implement a move constructor, move assignment operator, or `swap()`, declare it with a (conditional) `noexcept` expression.
- For other functions, you just might want to mark them with an unconditional `noexcept` if they don't throw.

This page is intentionally left blank

Chapter 8

Value Categories and Reference Binding

This chapter introduces the formal terminology and rules for move semantics. We formally introduce terms like lvalue, rvalue, prvalue, and xvalue and discuss their behavior when binding objects to references. That way we also introduce the important rule that move semantics is not automatically passed through and a very subtle behavior of `decltype` when called for expressions.

This chapter is the most complicated chapter of the book. You probably will see facts and features that are tricky and for some people hard to believe. Come back to it later, whenever you read again about value categories, binding references to objects, and `decltype`.

8.1 Value Categories

Each expression in a C++ program has a value category. Beside the type, the category describes what can be done with an expression.

However, value categories changed over time in C++.

8.1.1 History of Value Categories

Historically (taken from Kernighan&Ritchie C, *K&R* C), we had only *lvalue* and *rvalue*. The terms came from what was allowed in an assignment:

- An ***lvalue*** could occur on the ***l****eft* side of an assignment
- An ***rvalue*** could only occur on the ***r****ight* side of an assignment

However, these categories did matter not only for assignments. They were in general used to specify whether and where and expression can be used. For example:

```
int x;                  // x is an lvalue when used in an expression

x = 42;                 // OK, because x is an lvalue and the type matches
42 = x;                 // ERROR: 42 is an rvalue and can be only on the right side of an assignment
```

```
int* p1 = &x;          // OK: & is fine for lvalues (object has a specified location)
int* p2 = &42;         // ERROR: & is not allowed for rvalues (object has no specified location)
```

However, things already became more complicated with ANSI-C because an `x` declared as `const int` could not stand on the left side of an assignment, but could still be used at several other places an lvalue could be used:

```
const int c = 42;      // Is c an lvalue or rvalue?

c = 42;                // now an ERROR
const int* p1 = &c;    // still OK
```

The decision in C was that such a `c` is still an *lvalue*, because most of the operations for lvalues can still be called for `const` objects of a specific type.

The only thing you couldn't do anymore was to have a `const` object on the left side of an assignment. As a consequence, the meaning of *l* changed in ANSI-C to *locator value*. An ***lvalue*** is now an object that has a specified location in the program (so that you can take the address, for example). The same way an ***rvalue*** can now better be considered as just a *readable value*.

C++98 adopted these definitions of value categories. And everything was fine until we introduced move semantics. The question was which value category an object marked with `std::move()` has.

The decision was that, in general, objects marked with `std::move()` follow the following rules for class types:

```
std::string s;
...
std::move(s) = "hello";    // OK (behaves like an lvalue)
auto ps = &std::move(s);   // ERROR (behaves like an rvalue)
```

However, note that fundamental data types (FDTs) behave as follows:

```
int i;
...
std::move(i) = 42;         // ERROR
auto pi = &std::move(i);   // ERROR
```

So, except for fundamental data types, an object marked with `std::move()` should still behave like an lvalue by allowing that you can modify its value. On the other hand, there are restrictions such as you should not be able to take the address.

For this reason, a new category *xvalue* was introduced, which is only used for objects marked with `std::move()`. However, most of the rules for former rvalues also apply to xvalues. For this reason, the former value category *rvalue* was renamed to *prvalue* and *rvalue* became a common term for either and *xvalue* or an *prvalue*. See `http://wg21.link/n3055` for the paper proposing these changes.

8.1.2 Value Categories Since C++11

Since C++11, the value categories are as described in Figure 8.1.

We have the following primary categories:

- ***lvalue*** ("locator value")

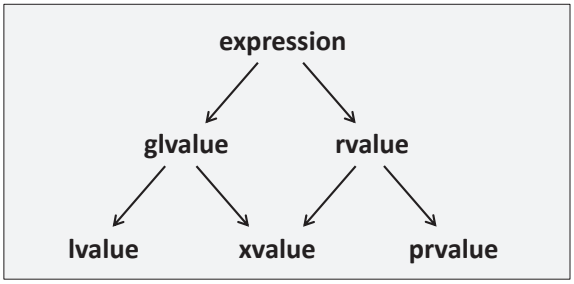


Figure 8.1. Value categories since C++11

- ***prvalue*** ("pure readable value")
- ***xvalue*** ("eXpiring value")

The composite categories are:

- ***glvalue*** ("generalized lvalue") as a common term for "*lvalue* or *xvalue*"
- ***rvalue*** as a common term for "*xvalue* or *prvalue*"

Value Categories of Basic Expressions

Examples of ***lvalues*** are:

- An expression that is just the name of a variable, function, or data member (except member of an xvalue)
- An expression that is just a string literal (e.g., `"hello"`)
- The result of a function returned by lvalue reference (return type *Type*`&`)
- The result of the built-in unary `*` operator (i.e., what dereferencing a raw pointer yields)

Examples of ***prvalues*** are:

- Expressions that consist of a literal that is not a string literal (e.g., `42`, `true`, or `nullptr`)
- The result of a function returned by value (return type *Type*)
- The result of the built-in unary `&` operator (i.e., what taking the address of an expression yields)
- A lambda expression

Examples of ***xvalues*** are:

- The result of marking an objects with `std::move()`
- The result of a function returned by rvalue reference (return type *Type*`&&`)
- A cast to an rvalue reference of an object type
- A data member of an xvalue

For example:

```
class X {
};

X v;
const X c;
```

```
f(v);                // passes a modifiable lvalue
f(c);                // passes a non-modifiable lvalue
f(X());              // passes a prvalue (old syntax of creating a temporary)
f(X{});              // passes a prvalue (new syntax of creating a temporary)
f(std::move(v));     // passes an xvalue
```

Roughly speaking as a rule of thumb:

- All names used as expressions are *lvalues*.
- All string literals used as expression are *lvalues*.
- All non-string literals (`4.2`, `true`, or `nullptr`) are *prvalues*.
- All temporaries without a name (especially objects returned by value) are *prvalues*.
- An object marked with `std::move()` and its members are *xvalues*.

It is worth emphasizing that strictly speaking, glvalues, prvalues, and xvalues are terms for expressions and *not* for values (which means that these terms are misnomers). For example, a variable itself is not an lvalue; only an expression denoting the variable is an lvalue:

```
int x = 3;           // here, x is a variable, not an lvalue
int y = x;           // here, x is an lvalue
```

In the first statement, `3` is a prvalue which initializes the variable (not the lvalue) `x`. In the second statement, `x` is an lvalue (its evaluation designates an object containing the value `3`). The lvalue `x` is used as a rvalue, which is what initializes the variable `y`.

8.1.3 Value Categories Since C++17

C++17 has the same value categories, but clarified the semantic meaning of value categories as described in Figure 8.2.

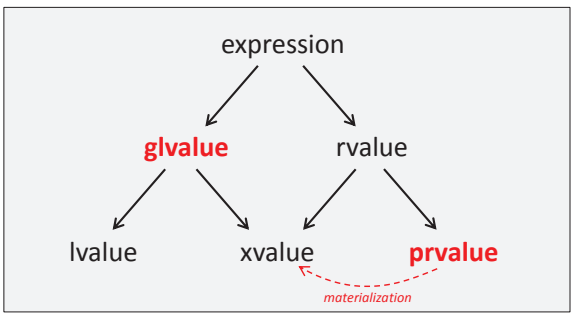


Figure 8.2. Value categories since C++17

The key approach for explaining value categories now is that in general, we have two major kinds of expressions:

- **glvalues**: expressions for *locations* of long-living objects or functions
- **prvalues:** expressions for short-living values for *initializations*

An **xvalue** is then considered a special location, representing an (long-living) object whose resources/values are no longer needed.

Passing Prvalues by Value

With this change, we can now pass around prvalues by value as unnamed initial values even if no valid copy and no valid move constructor is defined:

```
class C {
 public:
  C(...);
  C(const C&) = delete;   // this class is neither copyable ...
  C(C&&) = delete;        // ... nor movable
};

C createC() {
  return C{...};          // Always creates a conceptual temporary prior to C++17.
}                         // In C++17, no temporary object is created at this point.

void takeC(C val) {
  ...
}

auto n = createC();       // OK since C++17 (error prior to C++17)

takeC(createC());         // OK since C++17 (error prior to C++17)
```

Before C++17, passing a prvalue such as the created and initialized return value of `createC()` around is not possible without either copy or move support. However, since C++17 we can pass prvalues around by value as long as we don't need an object with a location.

Materialization

C++17 then introduces a new term, called *materialization* (of an unnamed temporary), for the moment a prvalue becomes a temporary object. Thus, a *temporary materialization conversion* is a (usually implicit) prvalue-to-xvalue conversion.

Any time a prvalue validly appears where a glvalue (lvalue or xvalue) is expected, a temporary object is created and initialized with the prvalue (remember that prvalues are primarily "initializing values"), and the prvalue is replaced by an *xvalue* designating the temporary. So in the example above, strictly speaking, we have:

```
void f(const X& p);   // accepts an expression of any value category, but expects a glvalue

f(X{});               // creates a temporary prvalue and passes it materialized as xvalue
```

Because `f()` in this example has a reference parameter, it expects a glvalue argument. However, the expression `X{}` is a prvalue. The "temporary materialization" rule therefore kicks in, and the expression `X{}` is "converted" into an xvalue designating a temporary object initialized with the default constructor.

Note that materialization does not mean that we create a new/different object. The lvalue reference p still *binds* to both an xvalue and a prvalue, although the latter now always involves a conversion to an xvalue.

8.2 Impact of Values Categories When Binding to References

When we bind objects to references, value categories play an important role. They, for example, define already in C++98/C++03 that you can assign or pass an rvalue (a temporary object without name or an object marked with `std::move()`) to a `const` lvalue reference, but not to a non-`const` lvalue reference:

```
std::string createString();                 // forward declaration

const std::string& r1{createString()};     // OK

std::string& r2{createString()};           // ERROR
```

The typical error message printed by the compiler here is "cannot bind a non-`const` lvalue reference to an rvalue."

You also get this error message with the call of `foo2()` here:

```
void foo1(const std::string&);   // forward declaration
void foo2(std::string&);         // forward declaration

foo1(std::string{"hello"});      // OK
foo2(std::string{"hello"});      // ERROR
```

8.2.1 Overload Resolution with Rvalue References

So, let's see the exact rules when passing an object to a reference.

Assume we have a non-`const` variable `v` and a `const` variable `c` of a class `X`:

```
class X {
  ...
};

X v;
const X c{...};
```

Table *Rules for Binding References* lists the formal rules for binding references to passed arguments if we provide all reference overloads of a function `f()`:

```
void f(const X&);    // read-only access
void f(X&);          // OUT parameter (usually long-living object)
void f(X&&);         // can steal value (object might be about to die)
void f(const X&&);   // no clear semantic meaning
```

The numbers list the priority in overload resolution so that you can see which function is called when multiple overloads are provided. Note that you can only pass rvalues (prvalues (such as temporary objects

Call	f(X&)	f(const X&)	f(X&&)	f(const X&&)
f(v)	1	2	error	error
f(c)	error	1	error	error
f(X{})	error	3	1	2
f(move(v))	error	3	1	2
f(move(c))	error	2	error	1

Table 8.1. Rules for Binding References

without a name) or xvalues (objects marked with `std::move()`) to rvalue references. That's where they have their name from.

Usually you can ignore the last column of the table, because semantically it makes not much sense, so that we get the following rules:

- A non-`const` lvalue reference only takes non-`const` lvalues.
- An rvalue reference only takes non-`const` rvalues.
- A `const` lvalue reference can take everything and serves as the fallback mechanism in case other overloads are not provided.

The following extract from the middle of the table is the rule for the fallback mechanism of move semantics:

Call	f(const X&)	f(X&&)
f(X)	3	1
f(move(v))	3	1

If we pass an rvalue (temporary object or object marked with `std::move()`) to a function and there is no specific implementation for move semantics (declared by taking an rvalue reference), the usual copy semantics is used taking the argument by `const&`.

Please note that we will extend this table later, when we have introduced universal references.

There we will also learn, that sometimes you can pass an lvalue to an rvalue reference (when a template parameter is used). So, beware, not every declaration with `&&` follows the same rules. The rules here apply if we have a **concrete type** declared with `&&`.

8.2.2 Overloading by Reference and Value

We can declare functions by both references and value parameters: For example:

```
void f(X);          // pass-by-value
void f(const X&);   // pass-by-reference
void f(X&);
void f(X&&);
void f(const X&&);
```

In principle, declaring all these overloads is allowed. However, there is no specific priority between pass-by-value and pass-by-reference. If you have a function declared to take an argument by value (which can take any argument of any value category), any matching declaration taking the argument by reference creates an ambiguity.

For this reason, you should usually only take an argument either by value or by reference (with as many reference overload you think are useful) but never both.

8.3 When Lvalues become Rvalues

As we learned, when a function is declared with an rvalue reference parameter of a concrete type, you can only bind these parameters to rvalues. For example:

```
void rvFunc(std::string&&);      // forward declaration

std::string s{...};
rvFunc(s);                       // ERROR: passing an lvalue to an rvalue reference
rvFunc(std::move(s));            // OK, passing an xvalue
```

However, note that sometimes passing an lvalue works. For example:

```
void rvFunc(std::string&&);      // forward declaration

rvFunc("hello");                 // OK, although "hello" is an lvalue
```

Remember that string literals are lvalues when used as an expression. So passing it to an rvalue reference should not compile. However, there is a hidden operation involved, because the type of the argument (array of 6 constant characters) does not match the type of the parameter. We have an implicit type conversion, performed by the string constructor, which creates a temporary object not having a name.

So what we really call is the following:

```
void rvFunc(std::string&&);      // forward declaration

rvFunc(std::string{"hello"});    // OK, "hello" converted to a string is a prvalue
```

8.4 When Rvalues become Lvalues

Now, let's look at the implementation of a function declaring the parameter as rvalue reference:

```
void rvFunc(std::string&& str) {
  ...
}
```

As we learned we can only pass rvalues:

```
std::string s{...};
rvFunc(s);                       // ERROR: passing an lvalue to an rvalue reference
rvFunc(std::move(s));            // OK, passing an xvalue
rvFunc(std::string{"hello"});    // OK, passing a prvalue
```

However, when we use the parameter `str` inside the function, we deal with an object that has a name. That means that we use `str` as an lvalue. We can do (only) everything we are allowed to do with an lvalue.

That means, we can't recursively call our own function:

```
void rvFunc(std::string&& str) {
```

```
  rvFunc(str);                    // ERROR: passing an lvalue to an rvalue reference
}
```

We have to make it with `std::move()` again:

```
void rvFunc(std::string&& str) {
  rvFunc(std::move(str));         // OK, passing an xvalue
}
```

This is the formal specification of the rule that **move semantics is not passed through** we already discussed. Again note that this is a feature, not a bug. If we would pass move semantics through, we could not use an object that was passed with move semantics twice, because the first time we use it it would lose its value. Or we would need a feature temporarily disabling move semantics here.

If we bind an rvalue reference parameter to an rvalue (prvalue or xvalue), the object is used as an lvalue, which we have to convert to an rvalue again to pass it to an rvalue reference.

Now remember that `std::move()` is nothing but a `static_cast` to the an rvalue reference. That is, what we program in a recursive call is just the following:

```
void rvFunc(std::string&& str) {
  rvFunc(static_cast<std::string&&>(str));   // OK, passing an xvalue
}
```

We cast the object `str` to its own type. So far that would be a no-op. However, with the cast we do something else: we change the value category. By rule, the lvalue becomes an xvalue and therefore allows us to pass the object to an rvalue reference.

This is nothing new, even before C++11 a parameter declared as lvalue reference followed the rules of lvalues when being used. The key point is that a reference in a declaration specifies what can be passed to a function. For the behavior inside a function references are irrelevant.

Confusing? Well that's just how we define the rules of move semantics and value categories in the C++ standard. Take it as it is. Compilers fortunately know these rules.

If there is one thing for you to learn here it is that move semantics is not passed through. If you pass an object with move semantics you have to mark it with `std::move()` again to forward its semantics to another function.

8.5 Checking Value Categories with `decltype`

Together with move semantics, C++11 introduced a new keyword `decltype`. The primary goal of this keyword is to get the exact type of a declared object. But it also can be used to check the value category of an expression.

8.5.1 Using `decltype` to Check the Type of Names

In a function taking an rvalue reference parameter we can use `decltype` to query and use the exact type of the parameter. Just pass the name of the parameter to `decltype`. For example:

```
void rvFunc(std::string&& str)
{
  std::cout << std::is_same<decltype(str), std::string>::value;    //false
```

```
  std::cout << std::is_same<decltype(str), std::string&>::value;   //false
  std::cout << std::is_same<decltype(str), std::string&&>::value;  //true

  std::cout << std::is_reference<decltype(str)>::value;            //true
  std::cout << std::is_lvalue_reference<decltype(str)>::value;     //false
  std::cout << std::is_rvalue_reference<decltype(str)>::value;     //true
}
```

The expression `decltype(str)` always yields the type of `str`, which is `std::string&&`. We can use this type wherever we need this type in an expression. Type traits (type functions such as `std::is_same<>`) help us to deal with these types.

For example, to declare a new object of the passed parameter type that is not a reference we can declare:

```
void rvFunc(std::string&& str)
{
  std::remove_reference<decltype(str)>::type tmp;
  ...
}
```

`tmp` has type `std::string` in this function (which we could also explicitly declare but if we make this a generic function for objects of type T, the code would still work).

8.5.2 Using `decltype` to Check the Value Category

So far we have only passed names to `decltype` to ask for its type. However, you can also pass *expressions* (that are not just names) to `decltype`. In that case, `decltype` yields also the value category according to the following convention:

- For a **prvalue** it just yields its value type: *type*
- For an **lvalue** it yields its type as lvalue reference: *type*`&`
- For an **xvalue** it yields its type as rvalue reference: *type*`&&`

For example:

```
void rvFunc(std::string&& str)
{
  decltype(str + str)    // yields std::string because s+s is a prvalue
  decltype(str[0])       // yields char& because the index operator yields an lvalue
  ...
}
```

This means that if you just pass a name inside parentheses, which is an expression, `decltype` also yields the type and value category. The behavior is as follows:

```
void rvFunc(std::string&& str)
{
  std::cout << std::is_same<decltype((str)), std::string>::value;    //false
  std::cout << std::is_same<decltype((str)), std::string&>::value;   //true
  std::cout << std::is_same<decltype((str)), std::string&&>::value;  //false
```

```
  std::cout << std::is_reference<decltype((str))>::value;           //true
  std::cout << std::is_lvalue_reference<decltype((str))>::value;    //true
  std::cout << std::is_rvalue_reference<decltype((str))>::value;    //false
}
```

Compare with the implementation of this function not using additional parentheses.

Now because `str` is an lvalue `decltype` of `(str)` yields `std::string&`.

Check for a Value Category Inside Code

In general, you can now check for a specific value category inside code:

- `!std::is_reference<decltype((`*expr*`))>::value`
  checks whether *expr* is a prvalue.
- `std::is_lvalue_reference<decltype((`*expr*`))>::value`
  checks whether *expr* is an lvalue.
- `std::is_rvalue_reference<decltype((`*expr*`))>::value`
  checks whether *expr* is an xvalue.

Note again the additional parentheses used here to ensure that we use the value-category checking form of `decltype` even if we only pass a name as *expr*.

Note also that for `decltype` it makes a difference when we put additional parentheses around a passed name. That will have significant consequences later when we discuss `decltype(auto)`.

8.6 Summary

- Any expression in a C++ program is of exactly one of these primary value categories:
  - **lvalue** (roughly, for a named object or a string literal)
  - **prvalue** (roughly, for an unnamed temporary object)
  - **xvalue** (roughly, for an object marked with `std::move()`
- Rvalue references of concrete types can only bind to rvalues (prvalues or xvalues).
- Whether a call or operation in C++ is valid depends on both the type and the value category.
- Implicit operations might change the value category of a passed argument.
- Passing an rvalue to an rvalue references binds it to an lvalue.
- Move semantics is not passed through.
- `decltype` checks for the type of a name of for type and value category of an expression.

This page is intentionally left blank

Part II

Move Semantics in Generic Code

This part of the book introduces the move semantics features that C++ provides for generic programming (i.e., templates).

This page is intentionally left blank

Chapter 9

Perfect Forwarding

This chapter introduces move semantics in generic code. This means how to (perfectly) forward parameters with and without move semantics. For that purpose, universal references are introduced.

The following chapters then discuss tricky details of perfect forwarding and universal references and how to deal with return values that may or may not have move semantics in generic code.

9.1 Motivation for Perfect Forwarding

We learned already that move semantics is not automatically passed through. This has consequences for generic code.

9.1.1 What we Need to Perfectly Forward Arguments

To forward an object that is passed with move semantics to a function, it not only has to be bound to an rvalue reference. You have to use `std::move()` again to pass its move semantics to somewhere else.

For example, remember the overload resolution rules for the main cases of functions overloaded by reference:

```
class X {
  ...
};

// forward declarations:
void foo(const X&);     // for constant values (read-only access)
void foo(X&);           // for variable values (out parameters)
void foo(X&&);          // for values that are no longer used (move semantics)
```

We have the following rules when calling them:

```
X v;
const X c;
foo(v);                 // calls foo(X&)
```

```
foo(c);                 // calls foo(const X&)
foo(X{});               // calls foo(X&&)
foo(std::move(v));      // calls foo(X&&)
foo(std::move(c));      // calls foo(const X&)
```

Now assume, we want to call `foo()` for the same arguments indirectly via a helper function `callFoo()`.

That helper function would also needs the three overloads:

```
void callFoo(const X& arg) { // arg binds to all const objects
  foo(arg);                  // calls foo(const X&)
}
void callFoo(X& arg) {       // arg binds to lvalues
  foo(arg);                  // calls foo(X&)
}
void callFoo(X&& arg) {      // arg binds to rvalues
  foo(std::move(arg));       // needs std::move() to call foo(X&&)
}
```

In all cases `arg` is used as an lvalue (being on object with a name). The first version forwards it as a `const` object, but the other two cases implement the two different ways to forward the non-`const` argument:

- Arguments declared as lvalue references (having no move semantics) are just passed as they are.
- Arguments declared as rvalue references (having move semantics) are passed with `std::move()`.

That way we forward move semantics perfectly: we keep move semantics for an argument that was passed having move semantics, and we don't add move semantics when we got an argument that doesn't have it.

Only with this implementation the use of `callFoo()` to call `foo()` is transparent:

```
X v;
const X c;
callFoo(v);                 // calls foo(X&)
callFoo(c);                 // calls foo(const X&)
callFoo(X{});               // calls foo(X&&)
callFoo(std::move(v));      // calls foo(X&&)
callFoo(std::move(c));      // calls foo(const X&)
```

Remember that an rvalue passed to an rvalue reference becomes an lvalue when used, so that we need `std::move()` to pass it as rvalue again. But we can't use `std::move()` everywhere. For the other overloads, using `std::move()` would call the overload of `foo()` for rvalue references when an lvalue is passed.

In general, if we need *perfect forwarding* in generic code, we would always need all these overloads for each parameter. This would mean that for 2 generic arguments we need 9 overloads, and for 3 generic arguments we need 27 overloads.

The goal was therefore to have way to write code that always *perfectly forwards* a given argument so that it keeps its type and value category.

Perfectly Forwarding `const` Rvalue References

Although we have no semantic meaning for `const` rvalue references, if we wanted to keep the exact type and value category of a constant object marked with `std::move()`, we would even need a fourth overload:

```
void callFoo(const X&& arg) { // arg binds to const rvalues
  foo(std::move(arg));        // needs std::move() to call foo(const X&&)
}
```

Otherwise, we would call `foo(const X&)`. This usually is fine, but there might be cases that we want to keep the knowledge that we passed a `const` rvalue reference (e.g., when for whatever reason a `foo(const X&&)` overload is provided).

For two or three parameters we would need then even 16 and 64 overloads in generic code.

9.2 Implementing Perfect Forwarding

To avoid overloading functions for parameters with different value categories, C++ introduced a special mechanism for *perfect forwarding*: You have to take the argument as an rvalue reference to a function template parameter and use a helper function called `std::forward<>()`:

```
template<typename T>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));  // becomes foo(std::move(arg)) for passed rvalues
}
```

That way, we get the same behavior as the three (or four) overloads of `callFoo()` above, because `std::forward<>()` is a conditional `std::move()`:

- If we pass an rvalue to `arg`, we call `foo(std::move(arg))`
- If we pass an lvalue to `arg`, we call `foo(arg)`

The same way, we can perfectly forward two arguments:

```
template<typename T1, typename T2>
void callFoo(T1&& arg1, T2&& arg2) {
  foo(std::forward<T1>(arg1), std::forward<T2>(arg2));
}
```

And we can apply the forwarding to each argument of a variadic number of parameters to perfectly forward them all:

```
template<typename... Ts>
void callFoo(Ts&&... args) {
  foo(std::forward<Ts>(args)...);
}
```

Note that we don't call `forward<>()` once for all arguments, we call it for each argument. Therefore, we have to place the ellipsis (`...`) behind the end of the `forward()` expression instead of behind `args`.

However, what exactly is happening here is pretty tricky and needs careful explanation.

9.2.1 Universal (or Forwarding) References

First, note that we declare `arg` as an rvalue reference parameter:

```
template<typename T>
void callFoo(T&& arg);   // arg is universal/forwarding reference
```

This might give the impression that the usual rules of rvalue references apply. But that is not the case. An **rvalue reference of a function template parameter** doesn't follow the rules of ordinary rvalue references. It's a different thing.

Two Terms: Universal and Forwarding Reference

Such a reference is called an ***universal reference***. Unfortunately, there is also another term for it, which is mainly used in the C++ standard ***forwarding reference***. There is no difference between these two terms, it's just that we have an intentional mess here for no good reason.

Both terms describe basic aspects of universal/forwarding references:

- They can universally bind to objects of all value categories.
- They are usually used to forward arguments (but there is another usage).

Universal References Bind to and Keep All Value Categories

The important feature of universal references is that they can bind to all objects (of any value category):

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

X v;
const X c;
callFoo(v);                 // OK
callFoo(c);                 // OK
callFoo(X{});               // OK
callFoo(std::move(v));      // OK
callFoo(std::move(c));      // OK
```

In addition, they preserve the constness and value category of the object they are bound to:

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

X v;
const X c;
callFoo(v);                 // OK, arg is X&
callFoo(c);                 // OK, arg is const X&
callFoo(X{});               // OK, arg is X&&
callFoo(std::move(v));      // OK, arg is X&&
callFoo(std::move(c));      // OK, arg is const X&&
```

By rule the type `T&&`, which is the type of `arg`, is

- an lvalue reference if we refer to an lvalue
- an rvalue reference if we refer to an rvalue

Note that I didn't talk about the type of `T`. What type `T` is here in all these cases I will explain later.

9.2.2 `std::forward<>()`

Inside `callFoo()`, we then use the universal reference as follows:

```
template<typename T>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));   // becomes foo(std::move(arg)) for passed rvalues
}
```

The expression `std::forward<T>(arg)` is essentially implemented as follows:

- If an rvalue of type `T` is passed to the function, the expression expands to `std::move(arg)`.
- If an lvalue of type `T` is passed to the function, the expression expands to `arg`.

That is, `std::forward<>()` is a `std::move()` only for passed rvalues.

Like `std::move()`, `std::forward<>()` is also defined as a function in the header file `<utility>`.

Again, the details of what `std::forward<>()` is and how it works is explained later.

9.2.3 The Effect of Perfect Forwarding

Combining the behavior of declaring a universal reference and using `std::forward<>()` we get the following behavior:

```
void foo(const X&);    // for constant values (read-only access)
void foo(X&);          // for variable values (out parameters)
void foo(X&&);         // for values that are no longer used (move semantics)

template<typename T>
void callFoo(T&& arg) {        // arg is a universal/forwarding reference
  foo(std::forward<T>(arg));   // becomes foo(std::move(arg)) for passed rvalues
}

X v;
const X c;

callFoo(v);                 // OK, expands to func(arg), so it calls foo(X&)
callFoo(c);                 // OK, expands to func(arg), so it calls foo(const X&)
callFoo(X{});               // OK, expands to func(std::move(arg)), so it calls foo(X&&)
callFoo(std::move(v));      // OK, expands to func(std::move(arg)), so it calls foo(X&&)
callFoo(std::move(c));      // OK, expands to func(std::move(arg)), so it calls foo(const X&)
```

Whatever we pass to `callFoo()` becomes an lvalue (as we have an objects with a name "`arg`"). But the type of `arg` depends on what we pass:

- If we pass an lvalue, `arg` is an lvalue reference (`X&`, when we passed a non-`const` `X`, or `const X&`, when we pass a `const X`).
- If we pass an rvalue (an unnamed temporary an object marked with `std::move()`), `arg` is an rvalue reference (`X&&` or `const X&&`).

By using `std::forward<>()` we forward the parameter with `std::move()` if and only if we have an rvalue reference (i.e., `arg` binds to an rvalue).

9.3 Rvalue References versus Universal References

Unfortunately, universal/forwarding references use the same syntax as ordinary rvalue references (formally, they are special rvalue references). This is a serious source of trouble and confusion.[1] If you see a declaration with two ampersands, you have to double check, whether a concrete type or a function template parameter is used.

In other words, there is a significant difference between

```
void foo(Coll&& arg)        // arg is an ordinary rvalue reference of type Coll
```

and

```
template<typename Coll>
void foo(Coll&& arg)        // arg is a universal/forwarding reference of any type
```

Let's discuss the differences in detail.

9.3.1 Rvalue References of Actual Types

When having an ordinary rvalue reference to a name that is not template parameter of the function called, we can only bind these references to rvalues. In addition, we know that the passed argument is not `const`:

```
using Coll = std::vector<std::string>;

void foo(Coll&& arg)        // arg is an ordinary rvalue reference
{
  Coll coll;                // coll can't be const
  ...
  bar(std::move(arg));      // perfectly forward to bar()  (don't use std::forward<>() here)
}

Coll v;
const Coll c;

foo(v);                     // ERROR: can't bind rvalue reference to lvalue
foo(c);                     // ERROR: can't bind rvalue reference to lvalue
foo(Coll{});                // OK, arg binds to a non-const prvalue
foo(std::move(v));          // OK, arg binds to a non-const xvalue
```

[1] We discuss later the drawbacks of having the same syntax and why no different syntax was introduced.

```
foo(std::move(c));          // ERROR: can't bind non-const rvalue reference to const xvalue
```

Inside `foo()`:

- We know that the type of `arg`, `Coll` is never `const`.
- It doesn't make sense to use `std::forward<>()`. It only makes sense to use `std::move()` when we no longer need the value and want to forward it to another function (using `std::forward<>()` is possible and would always be equivalent to `std::move()`).

9.3.2 Rvalue References of Function Template Parameters

When having an rvalue reference to a function template parameter, we can pass objects of *all* value categories. The passed argument might or might not be `const`:

```
using Coll = std::vector<std::string>;

template<typename Coll>
void foo(Coll&& arg)              // arg is a universal/forwarding reference
{
  Coll coll;                      // coll may be const
  ...
  bar(std::forward<Coll>(arg));   // perfectly forward to bar() (don't use std::move() here)
}

Coll v;
const Coll c;

foo(v);                           // OK, arg binds to a non-const lvalue
foo(c);                           // OK, arg binds to a const lvalue
foo(Coll{});                      // OK, arg binds to a non-const prvalue
foo(std::move(v));                // OK, arg binds to a non-const xvalue
foo(std::move(c));                // OK, arg binds to a const xvalue
```

Inside `foo()`:

- Now the type of `arg` may or may not be `const`
- It doesn't make sense to use `std::move()`. It only makes sense to use `std::forward<>()` when we no longer need the value and want to forward it to another function (using `std::move()` is possible, but would forward all arguments with move semantics).

9.4 Overload Resolution with Universal Reverences

We already have discussed the formal rules when binding objects to ordinary references. Now, after introducing universal references, we have to extend these rules.

Again, assume we have a class `X`, and non-`const` variable of it and a `const` variable of it:

```
class X {
  ...
```

```
};

X v;
const X c{...};
```

Table *Rules for Binding All References* lists the formal rules for binding all kinds of references to passed arguments if we provide all possible overloads of a function `f()`:

```
void f(const X&);     // read-only access
void f(X&);           // OUT parameter (object lives long)
void f(X&&);          // can steal value (object usually close to die)
void f(const X&&);    // no clear semantic meaning
template<typename T>
void f(T&&);          // to use perfect forwarding
```

Call	`f(X&)`	`f(const X&)`	`f(X&&)`	`f(const X&&)`	`template<typename T>` `f(T&&)`
`f(v)`	1	3	error	error	2
`f(c)`	error	1	error	error	2
`f(X{})`	error	4	1	3	2
`f(move(v))`	error	4	1	3	2
`f(move(c))`	error	3	error	1	2

Table 9.1. Rules for Binding All References

Again, the numbers list the priority in overload resolution so that you can see which function is called when multiple overloads are provided.

Note that the universal reference is always the second-best option. A perfect match is always better, but the need to convert the type (such as making it `const` or converting an rvalue to an lvalue) is a worse match that just instantiating the function template for the exact type.

9.5 Overload Resolution with Universal Reverences

The fact that a universal reference binds better than a type conversion in overload resolution, has a very nasty side effect: If you have a constructor taking a single universal reference, this is a better match than

- the copy constructor if passing a non-`const` object
- the move constructor if passing a `const` object

For this reason, you should be very careful when implementing a constructor having one single universal reference parameter.

Consider the following program:

generic/universalconstructor.cpp

```
#include <iostream>
```

```
class X {
 public:
  X() = default;

  X(const X&) {
    std::cout << "copy constructor\n";
  }
  X(X&&) {
    std::cout << "move constructor\n";
  }

  template<typename T>
  X(T&&) {
    std::cout << "universal constructor\n";
  }
};

int main()
{
  X xv;
  const X xc;

  X xcc{xc};              // OK: calls copy constructor
  X xvc{xv};              // OOPS: calls universal constructor
  X xvm{std::move(xv)};   // OK: calls move constructor
  X xcm{std::move(xc)};   // OOPS: calls universal constructor
}
```

As indicated in the comments, the program has the following output:

```
copy constructor
universal constructor
move constructor
universal constructor
```

For this reason, you should better avoid implementing generic constructors for one universal reference.

The other option is to disable the constructors if the passed type is the same so that better the copy or move constructor should be used. Since C++20 this looks as follows:

```
class X {
 public:
  ...
  template<typename T>
  requires (!std::is_same_v<std::remove_cvref_t<T>, X>)
  X(T&&) {
    std::cout << "universal constructor\n";
```

```
  }
};
```

Before C++20 you need something like this:

```
class X {
 public:
  ...
  template<typename T,
           typename
            = typename std::enable_if<!std::is_same<typename std::decay<T>::type,
                                                    X>::value
                                      >::type>
  X(T&&) {
    std::cout << "universal constructor\n";
  }
};
```

9.6 Summary

- Declarations with two ampersands (*name*`&&`) can be two different things:
  - If *MyType* is not a function template parameter it is an ordinary rvalue reference, binding only to rvalues.
  - If *MyType* is a function template parameter it is a *universal reference*, binding to all objects.
- A *universal reference* (sometimes called *forwarding reference*) is a reference that can refer to all objects of any type and value category. Its type is
  - an lvalue reference (*type*`&`), if it binds to an lvalue
  - an rvalue reference (*type*`&&`), if it binds to an rvalue
- To perfectly forward a passed argument, declare the parameter as a universal reference of a template parameter of the function and use `std::forward<>()`.
- `std::forward<>()` is a conditional `std::move()`. It expands to `std::move()` if and only if its template parameter is an rvalue.
- Universal references are the second best options of all overload resolutions.
- Avoid implementing generic constructors for one universal reference (or constrain them for specific types).

Chapter 10

Tricky Details of Perfect Forwarding

After introducing (perfect) forwarding and universal references in general, this chapter discusses tricky details of perfect forwarding and universal references, such as

- Non-forwarding universal references
- When exactly do we have a universal reference

The following chapters then discuss and how to deal with return values that may or may not have move semantics in generic code.

10.1 Universal References as Non-Forwarding References

There are applications of universal references which have nothing to do with forwarding, because there can be other benefits from being able to bind to all objects while still knowing the value category and whether the object is `const`.

Here we discuss some first examples. After introducing a second kind of universal reference, we discuss other practical examples of using universal references as non-forwarding references.

10.1.1 Universal References and `const`

According to the formal rules for binding references a universal reference is the only way we can bind a reference to any object preserving whether or not it is `const`. The only other reference that binds to all objects, `const&`, loses the information, whether the object passed was `const` or not.

Forwarding Constness

That is, if we want to avoid overloading, but want to have different behavior for `const` and non-`const` arguments, we have to use universal references.

Consider the following example:

generic/universalconst.cpp

```
#include <iostream>
```

```
#include <string>

void usePos(std::string::iterator)
{
  std::cout << "do some non-const stuff with the iterator\n";
}
void usePos(std::string::const_iterator pos)
{
  std::cout << "do some const stuff with the iterator\n";
}

template<typename T>
void takeAll(T&& arg)
{
  usePos(arg.begin());
}

int main()
{
  std::string v{"v"};
  const std::string c{"c"};

  takeAll(v);                  // OK, arg binds to a non-const lvalue
  takeAll(c);                  // OK, arg binds to a const lvalue
  takeAll(std::string{"t"});   // OK, arg binds to a non-const prvalue
  takeAll(std::move(v));       // OK, arg binds to a non-const xvalue
  takeAll(std::move(c));       // OK, arg binds to a const xvalue
}
```

Here, we declare `takeAll()` with a parameter being a universal reference. For that reason, we can pass strings of all possible value categories. Inside `takeAll()` we can still behave differently depending on whether or not the argument is `const`. In this case we call different implementations of `usePos()`, one provided for `std::string::const_iterators` and one provided just for `std::string::iterators`.

Thus, the program has the following output:

```
do some non-const stuff with the iterator
do some const stuff with the iterator
do some non-const stuff with the iterator
do some non-const stuff with the iterator
do some const stuff with the iterator
```

Note that we don't use (perfect) forwarding here; we just want to refer universally to both `const` and non-`const` objects. This example demonstrates that using the term *forwarding references* for universal references is confusing and should be avoided.[1]

Constness Depending Code

If we have a universal reference, we can still check the constness of the passed argument and behave differently:

```
template<typename T>
void takeAll(T&& arg)
{
  if constexpr(std::is_const_v<std::remove_reference_t<T>>) {
    std::cout << "do some const stuff with " << arg << "\n";
  }
  else {
    std::cout << "do some non-const stuff with " << arg << "\n";
  }
}
```

Here, we use `if constexpr` (a compile-time `if`, introduced with C++17) to do different things, depending whether the passed argument is `const`.

Note that it is important to use `std::remove_reference<>` to remove the referenceness `T` before we check for its constness. A reference to a `const` type is not considered to be `const` as a whole:

```
std::is_const_v<int>                                  //false
std::is_const_v<const int>                            //true
std::is_const_v<const int&>                           //false
std::is_const_v<std::remove_reference_t<const int&>>  //true
```

In case you don't know yet:

- `std::remove_reference_t<T>` is a shortcut for `typename std::remove_reference<T>::type` (available since C++14).
- `std::is_const_v<T>` is a shortcut for `std::is_const<T>::value` (available since C++17).

10.1.2 Universal References of Specific Types

The fact that ordinary rvalue references and universal references share the same syntax creates multiple problems. It's it not only that it is not easy to find out what we have, you also have the problem that you can't declare a universal reference of a specific type.

For example, consider a function declared to take universal references:

```
template<typename T>
void takeAll(T&& arg);
```

[1] In my opinion, the term *forwarding reference* was an intentional mistake.

According to this declaration `arg` could have any type. The function can be called for all arguments supporting all operations in the template.

This is usually a good thing, but if for whatever reason we want to constraint this function to only take strings (both `const` and non-`const` without losing this information), we can't easily do that.

Before C++20, we need the `enable_if<>` type trait, to disable the visibility of the template unless it is called for strings. The trait is usually placed in an additional dummy template argument.

For example, to allow types convertible to a string:

```
template<typename T,
         typename = typename std::enable_if<
                                  std::is_convertible<T, std::string>::value
                                            >::type>
void foo(T&& args);
```

To restrict to type `std::string` we even need:

```
template<typename T,
         typename = typename std::enable_if<
                                  std::is_same<typename std::decay<T>::type,
                                               std::string
                                              >::value
                                             >::type>
void foo(T&& arg);
```

The type trait `std::decay<>` is used here to remove both referenceness and constness from type `T` (since C++20 you can use also use `remove_cvref<>` for this purpose).

Since C++20, restricting a universal reference to specific types become easier:

```
template<typename T>
requires std::is_convertible_v<T, std::string>
void takeAll(T&& arg);
```

But all this would not be necessary if we had a specific syntax for universal references. For example:

```
void takeAll(std::string&&& arg);   // assume &&& declares a universal reference
```

We have not, and now we have existing code with both applications of `&&`.

10.2 Universal or Ordinary Rvalue Reference?

The fact that we use the same syntax for ordinary rvalue references and universal/forwarding references introduces also some interesting corner cases for the question whether an object is an ordinary rvalue reference or a universal reference.

10.2.1 Rvalue References of Members of Generic Types

What is an rvalue reference to a member type of a template parameter? Does it follow the rules of rvalue or universal references?

The answer is: An rvalue reference to a member type of a template parameter is ***not*** a universal reference.

For example:

```
template<typename T>
void foo(typename T::value_type&& arg);   // no universal reference
```

Here is a complete example:

generic/universalmem.cpp

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

template<typename T>
void insert(T& coll, typename T::value_type&& arg)
{
  coll.push_back(arg);
}

int main()
{
  std::vector<std::string> coll;
  ...
  insert(coll, std::string{"prvalue"});  // OK
  std::string str{"lvalue"};
  insert(coll, str);              // ERROR: T::value_type&& is no universal reference
  insert(coll, std::move(str));   // OK
  ...
}
```

10.2.2 Rvalue References of Parameters in Class Templates

What is an rvalue reference to a template parameter of a class template? Does it follow the rules of rvalue or universal references?

The answer is: An rvalue reference to a template parameter of a class template is ***not*** a universal reference.

For example:

```
template<typename T>
class C {
  T&& member;          // member is no universal reference
  ...
  void foo(T&& arg);   // arg is no universal reference
};
```

Here is a complete example:

generic/universalclass.cpp

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

template<typename T>
class Coll {
 private:
  std::vector<T> values;
 public:
  Coll() = default;

  // function in class template:
  void insert(T&& val) {
    values.push_back(val);
  }
};

int main()
{
  Coll<std::string> coll;
  ...
  coll.insert(std::string{"prvalue"});  // OK
  std::string str{"lvalue"};
  coll.insert(str);                     // ERROR: T&& of Coll<T> is no universal reference
  coll.insert(std::move(str));          // OK
  ...
}
```

In general, a function in a class template does not follow the rule of function templates. It is what we call a *temploid*, generic code that follows the rules of ordinary functions when we instantiate the class.

10.2.3 Rvalue References of Parameters in Full Specializations of Function Templates

What is an rvalue reference to a template parameter of a full specialization of a function template? Does it follow the rules of rvalue or universal references?

The answer is: An rvalue reference to a template parameter of a function template is ***also*** a universal reference (just as it is in the primary template).

For example:

```
template<typename T>
void foo(T&& arg);              // arg is a universal reference
...
template<>
void foo(std::string&& arg);    // arg is a universal reference
```

Here is a complete example:

generic/universalspec.cpp

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

template<typename Coll, typename T>
void insert(Coll& coll, T&& arg)
{
  coll.push_back(arg);
}

template<>
void insert(std::vector<std::string>& coll, std::string&& arg)
{
  coll.push_back(arg);
}

int main()
{
  std::vector<std::string> coll;
  ...
  insert(coll, std::string{"prvalue"});  // OK
  std::string str{"lvalue"};
  insert(coll, str);                     // OK: std::string&& arg is a universal reference
  insert(coll, std::move(str));          // OK
  ...
}
```

10.3 Perfect Forwarding in Lambdas

If you want to perfectly forward parameters of lambdas you also have to use universal references and `std::forward<>()`. However, in that case you have to declare the universal references with `auto&&` as will be explained in the chapter about `auto&&`.

10.4 How the Standard Specifies Perfect Forwarding

To finally understand all rules of perfect forwarding, let's see how the rules are formally specified in the C++ standard.

Again, consider you have the following declaration:

```
template<typename T>
```

```
void f(T&& t)           // t is universal/forwarding reference
{
  g(std::forward<T>(t));   // perfectly forward (move() only for passed rvalues)
}
```

Normally, `T` would just have the type of the passed argument:

```
MyType v;

f(MyType{});            // T is deduced as MyType, so t is declared as MyType&&
f(std::move(v));        // T is deduced as MyType, so t is declared as MyType&&
```

However, for universal references there is a special rule, if lvalues are passed (see section [temp.deduct.call]):[2]

> If the parameter type is an rvalue reference to a cv-unqualified template parameter and the argument is an lvalue, the type "lvalue reference to T" is used in place of T for type deduction.

That means in this case:

- If the type of the parameter `t` is declared with `&&`
- and the type is a template parameter, which is neither qualified with `const` nor with `volatile` (as it is the case with `T` here)
- then `T` is deduced as `T&` instead.

In our example, that means:

```
template<typename T>
void f(T&& t);          // t is universal/forwarding reference

MyType v;
const MyType c;

f(v);                   // T is deduced as MyType&
f(c);                   // T is deduced as const MyType&
```

However, we have `t` declared as `T&&`. So, if `T` is `T&` instead, what does that mean? Here the *reference collapsing* rule of C++ (see section [dcl.ref]) gives an answer:

- *Type*`&` `&` becomes *Type*`&`
- *Type*`&` `&&` becomes *Type*`&`
- *Type*`&&` `&` becomes *Type*`&`
- *Type*`&&` `&&` becomes *Type*`&&`

That means:

```
MyType v;
const MyType c;

f(v);   // T is deduced as MyType& and t has this type
f(c);   // T is deduced as const MyType& and t has this type
```

[2] In new versions of the C++ standard the term *forwarding reference* is used in this definition.

Now consider how `std::forward<>()` is defined in contrast to `std::move()`:

- `std::move()` always converts any type to an rvalue reference:

  ```
  static_cast<remove_reference_t<T>&&>(t)
  ```

  It removes any referenceness and converts to the corresponding rvalue reference type (removes any `&` and adds `&&`).

- `std::forward<>()` only adds rvalue referenceness to the given type as it is:

  ```
  static_cast<T&&>(t)
  ```

  That means, that here the reference collapsing rules apply again. If type `T` is an lvalue reference, the `&&` has no effect and we cast to an lvalue reference. That keeps the value category.

  But if `T` is an rvalue reference (or no reference at all), we cast to an rvalue reference and that way we change the value category to an xvalue, which is the effect of `std::move()`.

So, in total we get:

```
template<typename T>
void f(T&& t)          // t is universal/forwarding reference
{
  g(std::forward<T>(t));   // perfectly forward (move() only for passed rvalues)
}

MyType v;
const MyType c;

f(v);                  // T and t are MyType&, forward() has no effect
f(c);                  // T and t are const MyType&, forward() has no effect
f(MyType{});           // T is MyType, t is MyType&&, forward() is a move()
f(std::move(v));       // T is MyType, t is MyType&&, forward() is a move()
```

10.4.1 Template Parameter Deduction with Universal References

The special rule for deducing template parameters of universal references (deducing the type as lvalue reference when lvalues are passed) can lead to unexpected errors for code that looks correct.

Programmers are usually surprised when code like the following does not compile:

```
template<typename T>
void insert(std::vector<T>& vec, T&& elem)
{
  vec.push_back(std::forward<T>(elem));
}

std::vector<std::string> coll;
std::string s;
...
insert(coll, s);    // ERROR: no matching function call
```

The problem is that both parameters can be used to deduce parameter `T`, but the deduced types are not the same:

- Using parameter `coll` type `T` it is deduced as `std::string`.
- But according to the special rule for universal references, parameter `elem` let's deduce parameter `T` as `std::string&`.

Therefore, the compiler raises an error.

The solution is simply to use two template parameters, instead:

```
template<typename T1, typename T2>
void insert(std::vector<T1>& vec, T2&& elem)
{
  vec.push_back(std::forward<T2>(elem));
}
```

or just:

```
template<typename Coll, typename T>
void insert(Coll& coll, T&& elem)
{
  coll.push_back(std::forward<T>(elem));
}
```

10.5 Nasty Details of Perfect Forwarding

Whenever something new and complex like move semantics is invented, people make mistakes. That's human. In the C++ standard we also have made a few mistakes regarding perfect forwarding and universal references.

10.5.1 "Universal" versus "Forwarding" Reference

As written, unfortunately we have two different terms for references that can refer to all objects of any value category: *universal* and *forwarding* reference. The history of this mess is as follows.

With C++11, the C++ standard introduced no special term for universal references. They just introduced special rules for rvalue references of function templates parameters.

When describing the behavior of these references in his books, Scott Meyers, one of the key authors of the C++ community, introduced the term *universal reference* for them. In consistency with the names *rvalue reference* and *lvalue reference* describing what the references bind to, he decided to introduce *universal reference* as description that this kind of references can universally bind to all value categories.

Unfortunately, the C++ standard committee decided later to introduce a different term with C++17: *forwarding reference*. The official reason is that these references are not that "universal" that they can bind to everything and that their main purpose is to perfectly forward parameters.

Well, while forwarding is a major use case of these references, it's not their only use case. As we have seen another very important use case is just to bind a reference, well universally, to any objects just to keep the information about its value category and/or whether it is `const`. And as we will see later:

- One reason might be to refer to the same object (whatever it is) twice, (which is why it is used in the implementation of the range-based `for` loop).
- Another reason might be the need to refer to all kinds of objects without losing their value category because that would make some code invalid (such as using it in a range-based `for` loop).

So, the question is why the C++ standard committee just didn't adopt the established term. Well, in the C++ community there is rumor for another reason for coming up with a different term: The key people in the standard committee didn't like the term *universal reference*, because it came from Scott Meyers, a guy who wrote a lot about the outcome of the standardization of C++ without joining the committee to make things better. Yes, even in the C++ standardization committee we sometime have politics. We are human...

In any case, it was the intentional decision of the C++ standard committee to create this mess. Definitely *universal reference* was good enough (we have far worse terms in the C++ standard). But not adopting this term and using another or better term was for a critical mass of people in the standard committee even more important.

I still prefer using *universal reference* (at least the books should be consistent; the standard anyway often uses its own terminology only valid for experts). But for clarification I also might just write *universal/forwarding reference*.

For you this means that whenever you hear the term *forwarding reference* you have to translate it to *universal reference* (or the other way round).

10.5.2 Why `&&` for Both Ordinary Rvalues and Universal References?

Universal/forwarding references use the same syntax as ordinary rvalue references, which is a serious source of trouble and confusion. So why didn't we introduce a specific syntax for universal references?

An alternative proposal might, for example, have been:

- Use ***two*** ampersands for ordinary rvalue references:

```
void foo(Coll&& arg)                // arg is an ordinary rvalue reference
```

- Use ***three*** ampersands for universal references:

```
template<typename Coll>
void foo(Coll&&& arg)               // arg is universal/forwarding reference
```

However, these three ampersands might just look too ridiculous (whenever I show this option people laugh). Unfortunately, it would have been better to use the three ampersands, because it would make code more intuitive.

As discussed when restricting universal references to concrete types we would then just have to declare

```
void takeAll(std::string&&& arg);   // assume &&& declares a universal reference
```

instead of

```
template<typename T,
         typename =
           typename std::enable_if<std::is_convertible<T, std::string>::value
                                   >::type>
void foo(T&& args);
```

or:

```
template<typename T>
requires std::is_convertible_v<T, std::string>
void takeAll(T&& arg);
```

There is an important lesson to learn here: It's better to have a ridiculous but clear syntax than having a cool but confusing syntax.

10.6 Summary

- You can use universal references to bind to all `const` and non-`const` objects without losing the information about the constness of the object.
- To have universal reference of a specific type you need some template tricks (till C++17) or concepts/requirements (since C++20).
- Only rvalue references of function template parameters and their (full) specializations are universal references. Rvalue references of class template parameters or members of template parameters are ordinary rvalue references, you can only bind to rvalues.
- The C++ standard committee introduced forwarding references as a "better" term for universal references. Unfortunately, this term restricts the purpose of universal references to a common specific use case and creates unnecessary confusion of having two terms for the same thing.

Chapter 11

Perfect Passing with `auto&&`

After discussing the usual case of move semantics in generic code, perfect forwarding a parameter, we now have to talk about perfectly dealing with return values. In this chapter we discuss how to forward return values perfectly to somewhere else. For this purpose we introduce the other universal reference, `auto&&`. But we also discuss applications of `auto&&` that have nothing to do with forwarding values.

The next chapter then discusses how to perfectly return values.

11.1 Default Perfect Passing

It happens pretty often that we have to pass a return value to another function:

```
// pass return value of compute() to process():
process(compute(t));   // OK, uses perfect forwarding of returned value
```

In non-generic code you know the types involved. However, in generic code you want to make sure that the return value of `compute()` is perfectly passed to `process()`.

The good news is: if you directly pass a return value to another function, the value is passed perfectly keeping its type and value category. You don't have to worry about move semantics. It will automatically be used if supported.

11.1.1 Default Perfect Passing in Detail

Here is a complete example:

generic/perfectpassing.cpp

```
#include <iostream>
#include <string>

void process(const std::string&) {
  std::cout << "process(const std::string&)\n";
}
```

```
void process(std::string&) {
  std::cout << "process(std::string&)\n";
}
void process(std::string&&) {
  std::cout << "process(std::string&&)\n";
}

const std::string& computeConstLRef(const std::string& str) {
  return str;
}
std::string& computeLRef(std::string& str) {
  return str;
}
std::string&& computeRRef(std::string&& str) {
  return std::move(str);
}
std::string computeValue(const std::string& str) {
  return str;
}

int main()
{
  process(computeConstLRef("tmp"));       // calls process(const std::string&)

  std::string str{"lvalue"};
  process(computeLRef(str));              // calls process(std::string&)

  process(computeRRef("tmp"));            // calls process(std::string&&)
  process(computeRRef(std::move(str)));   // calls process(std::string&&)

  process(computeValue("tmp"));           // calls process(std::string&&)
}
```

- If `compute()` returns a `const` lvalue reference:

  ```
  const std::string& computeConstLRef(const std::string& str) {
    return str;
  }
  ```

  the value category of the return value is an lvalue so that the return value is perfectly forwarded with the best match for a `const` lvalue reference:

  ```
  process(computeConstLRef("tmp"));          // calls process(const std::string&)
  ```

- If `compute()` returns a non-`const` lvalue reference:

  ```
  std::string& computeLRef(std::string& str) {
    return str;
  ```

```
}
```

the value category of the return value is an lvalue so that the return value is perfectly forwarded with the best match for a non-`const` lvalue reference:

```
std::string str{"lvalue"};
process(computeLRef(str));                    // calls process(const std::string&)
```

Note that we have to pass a non-`const` lvalue to the function returning is a non-`const` lvalue.

- If `compute()` returns an rvalue reference:

```
std::string&& computeRRef(std::string&& str) {
  return std::move(str);
}
```

the value category of the return value is an xvalue so that the return value is perfectly forwarded with the best match for an rvalue reference, so that `process()` could steal its value:

```
process(computeRRef("tmp"));                  // calls process(std::string&&)
process(computeRRef(std::move(str)));         // calls process(std::string&&)
```

- If `compute()` returns a new temporary object by value:

```
std::string computeValue(const std::string& str) {
  return str;
}
```

the value category of the return value is an prvalue so that the return value is perfectly forwarded with the best match for an rvalue reference, so that `process()` could also steal its value:

```
process(computeValue("tmp"));                 // calls process(std::string&&)
```

Note that by returning a `const` value:

```
const std::string computeConstValue(const std::string& str) {
  return str;
}
```

or a `const` rvalue reference:

```
const std::string&& computeConstRRef(std::string&& str) {
  return std::move(str);
}
```

you have again disabled move semantics:

```
process(computeConstValue("tmp"));            // calls process(const std::string&)
process(computeConstRRef("tmp"));             // calls process(const std::string&)
```

If we had declaration for `const&&` that overload would be taken.

For this reason: Don't mark values returned by value with `const` and don't mark returned non-`const` rvalue references with `const`.

11.2 Universal References with auto&&

However, how can you program in generic code to pass a return value *later* by keeping its type and value category?

The answer is that you again need a universal/forwarding reference, which, however, is not declared as a parameter. For this reason, auto&& was introduced.

That is, instead of:

```
// pass return value of compute() to process():
process(compute(t));   // OK, uses perfect forwarding of returned value
```

you can also implement the following:

```
// pass return value of compute() to process() with some delay:
auto&& ret = compute(t);                // initialize a universal reference with the return value
...
process(std::forward<decltype(ret)>(ret));   // OK, uses perfect forwarding of returned value
```

Or when using brace initialization:

```
// pass return value of compute() to process() with some delay:
auto&& ret{compute(t)};                 // initialize a universal reference with the return value
...
process(std::forward<decltype(ret)>(ret));   // OK, uses perfect forwarding of returned value
```

See *generic/perfectautorefref.cpp* for a complete example with all cases.

11.2.1 auto&&

If you declare something with auto&& you also declare a *universal reference*. The type of this reference preserves the type and the value category of its initial value.

If you declare

```
auto&& ref
```

By rule the type auto&&, which is the type of ref, is

- an lvalue reference if we refer to an lvalue
- an rvalue reference if we refer to an rvalue

This is exactly the same rule as for universal references being function template parameters:

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

That is, by rule the type T&&, which is the type of arg, is

- an lvalue reference if we refer to an lvalue
- an rvalue reference if we refer to an rvalue

11.2.2 Perfectly Forwarding an auto&& Reference

The same way we can perfectly forward the value passed to a universal reference being a function template parameter:

```
template<typename T>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));   // becomes foo(std::move(arg)) for passed rvalues
}
```

we can perfectly forward the universal reference declared with auto&&:

```
auto&& ref{...};

foo(std::forward<decltype(ref)>(ref));   // becomes foo(std::move(ref)) for initial rvalues
```

Remember that the expression std::forward<decltype(ref)>(ref) is essentially implemented as follows:

- If an rvalue of type decltype(ref) is passed to the function, the expression expands to std::move(arg).
- If an lvalue of type decltype(ref) is passed to the function, the expression expands to arg.

That is, because ref has a name and therefore is an lvalue, std::forward<>() is used to expand to std::move() if arg is declared as an rvalue reference, which means that it was initialized by an rvalue. That is, forward<>() gives the initial value back its move semantics if it had some.

11.3 auto&& as Non-Forwarding Reference

Note again that a universal reference is the only way we can bind a reference to any object of any type and value category preserving its value category and the information whether it is const. This also applies to universal references declared with auto&&.

11.3.1 Universal References and the Range-Based for Loop

Non-forwarding universal references declared with auto&& play an important role when using the range-based for loop.

Specification of the Range-based for Loop

In the C++ standard, the range-based for loop is specified as a shortcut for iterating over the elements of a range with an ordinary for loop.

A call such as:

```
std::vector<std::string> coll;
...
for (const auto& s : coll) {
  ...
}
```

is equivalent to the following:

```
std::vector<std::string> coll;
...
auto&& range = coll;          // initialize a universal reference
auto pos = range.begin();     // to use the given range coll here
auto end = range.end();       // and here
for ( ; pos != end; ++pos ) {
  const auto& s = *pos;
  ...
}
```

The reason we declare `coll` as a universal reference is that we want to be able to bind it to *every* range, to use it twice (once to ask for its beginning, and once to ask to its end) without creating a copy or losing the information whether or not the range is `const`.

The loop should work for:

- a non-`const` lvalue:

```
std::vector<int> coll;
...
for (int& i : coll) {
  i *= 2;
}
```

- a `const` lvalue:

```
const std::vector<int> coll{0, 8, 15};
...
for (int i : coll) {
  ...
}
```

- a prvalue:

```
for (int i : std::vector<int>{0, 8, 15}) {
  ...
}
```

Note that there is no other way to declare `range` in the equivalent code:

- With `auto` we would create a copy of the range (which takes time and disables modifying the elements).
- With `auto&` we would disable initializing the range with a temporary prvalue.
- With `const auto&` we would lose any non-constness of the range we iterate over.

However, note that there is a significant problem with the range-based `for` loop the way it is specified now. Code like this:

```
std::vector<std::string> createStrings();
...
for (char c : createStrings().at(0)) {     // fatal runtime error
  ...
}
```

becomes:

```
std::vector<std::string> createStrings();
...
auto&& range = createStrings().at(0);      // OOPS: universal reference to reference
auto pos = range.begin();                  // return value of createStrings() destroyed here
auto end = range.end();
for ( ; pos != end; ++pos ) {
  char c = *pos;
  ...
}
```

All valid references extend the lifetime of the value they bind to. However, we don't bind to the return value of `createString()` (that would work fine), we bind to a reference to the return type of `createStrings()` returned by `at()`. That means that we extend the lifetime of the reference but not of the return value of `createString()`. And the loop iterates over strings that were already destroyed.

Using the Range-based `for` Loop

But even when using the range-based `for` loop, a universal reference can make sense.

We can iterate over the elements by value and by reference:

- When copying elements is cheap, we can iterate by value:

```
std::vector<int> coll;
...
for (int i : coll) {
  std::cout << i << '\n';
}
```

- When copying elements is expected to be expensive, we better iterate by `const` reference:

```
std::vector<std::string> coll;
...
for (const std::string& s : coll) {
  std::cout << s << '\n';
}
```

- When modifying the elements, we have to iterate by non-`const` reference:

```
std::vector<int> coll;
...
for (int& i : coll) {
  i *= 2;
}
```

So, when we use a range-based `for` loop, it isn't enough to use lvalue references:

- Either we don't modify the elements:

```
template<typename T>
void print(const T& coll) {
  for (const auto& elem : coll) {
```

```
    std::cout << elem << '\n';
  }
}
```

- Or we modify the elements:

```
template<typename T>
void twice(T& coll) {
  for (auto& elem : coll) {
    elem = elem + elem;
  }
}
```

And this usually works:

```
std::vector<int> coll1{0, 8, 15};
...
twice(coll1);  // OK
print(coll1);  // OK

std::vector<std::string> coll2{"hi", "ho"};
...
twice(coll2);  // OK
print(coll2);  // OK
```

However, here it suddenly doesn't work:

```
std::vector<bool> coll{true, false, true};
...
twice(coll);   // ERROR: cannot bind non-const lvalue reference to an rvalue
print(coll);   // OK
```

What happened? Well, let's look at the code the range-based `for` loop expands to here:

```
std::vector<bool> coll{true, false, true};
...
{
  auto&& range = coll;              // OK: universal reference to reference
  auto pos = range.begin();         // OK
  auto end = range.end();           // OK
  for ( ; pos != end; ++pos ) {     // OK
    auto& elem = *pos;              // ERROR: cannot bind non-const lvalue reference to an rvalue
    elem = elem + elem;
  }
}
```

The problem is that the elements in a `std::vector<bool>` internally are not objects of type `bool`, they are single bits. The way this is implemented is that the type for a reference to an element is *not* a reference. It is an ordinary type you can use like a reference:

```
namespace std {
  template<typename Allocator>
```

```
  class vector<bool, Allocator> {
   public:
    ...
    class reference {
      ...
    };
    ...
  };
}
```

A value of this type is returned when dereferencing an iterator. For that reason

```
auto& elem = *pos;
```

tries to bind a non-`const` lvalue reference to a temporary prvalue, which is not allowed.

But there is a solution to the problem, use a universal reference:

```
template<typename T>
void twice(T& coll) {
  for (auto&& elem : coll) {   // NOTE: use a universal reference here
    elem = elem + elem;
  }
}
```

Because a universal reference can bind to any object (even to a temporary prvalue), now using this generic code with `vector<bool>` compiles:

```
std::vector<bool> coll{true, false, true};
...
twice(coll);    // OK (universal reference used to bind to an element reference)
print(coll);    // OK
```

So, we found another reason to use non-forwarding universal references: They bind to reference types that are not implemented as references. Or more generally: they allow us to bind to non-`const` objects provided as proxy types to manipulate objects.

11.4 Perfect Forwarding in Lambdas

If you want to perfectly forward parameters of generic lambdas you also have to use universal references and `std::forward<>()`. However, you simply use `auto&&` to declare the universal reference.

For example, the following generic lambda perfectly forwards all passed arguments:

```
[] (auto&&... args) {
  ...
  foo(std::forward<decltype(args)>(args)...);
};
```

Remember that lambdas are just an easy way to define function objects (objects with `operator()` defined to be able to use them as functions). The definition above expands to a compiler-defined class with universal references defined as template parameters:

```
class NameDefinedByCompiler {
    ...
  public:
    template<typename... Args>
    auto operator() (Args&&... args) {
      ...
      foo(std::forward<decltype(args)>(args)...);
    }
};
```

Since C++20, you can also declare ordinary functions with `auto&&`, which is then handled in the same way: it declares a function template with a universal reference.

11.5 Summary

- Don't return by-value with `const` (otherwise you disable move semantics for return values).
- Don't mark returned non-`const` rvalue references with `const`.
- `auto&&` can be used to declare a universal reference that is not a parameter.
  As any *universal reference* it can refer to all objects of any type and value category and its type is
  - an lvalue reference (*type*`&`), if it binds to an lvalue
  - an rvalue reference (*type*`&&`), if it binds to an rvalue
- To perfectly forward a universal reference `r` declared with `auto&&`, use `std::forward<decltype(r)>(r)`.
- You can use universal references to refer to both `const` and non-`const` objects and use them multiple times without losing the information about their constness.
- You can use universal references to bind to references that are proxy types.
- Consider using `auto&&` in generic code that iterates over elements of a collection to modify them. That way the code works for references that are proxy types.
- Using `auto&&` when declaring parameters of a lambda (or function since C++20), is a shortcut for declaring a function template with a universal reference as a parameter.

Chapter 12

Perfect Returning with `decltype(auto)`

After discussing perfect forwarding a parameter and perfectly passing through a returned value, we now have to talk about perfectly returning values. This introduces the new placeholder type `decltype(auto)`. We will also see surprising consequences such as the strong recommendation not to use unnecessary parentheses in return statements.

12.1 Perfect Returning

In generic code we often compute a value we return then to the caller. The question is how to perfectly return it keeping whatever type and value category it has. In other words: how should we declare the return type of the following function:

```
template<typename T>
??? callFoo(T&& arg)
{
  return foo(std::forward<T>(arg));
}
```

In this function we call a function called `foo()` with the perfectly forwarded parameter `arg`. We don't know what `foo()` returns for this type. It might be a temporary value (prvalue), an lvalue reference, or an rvalue reference. The return type might be `const` or not `const`.

So how do we perfectly return the return value of `foo()` to the caller of `callFoo()`. A couple of options do not work:

- Return type `auto` will remove the referenceness of the return type of `foo()`. If we, for example, give access to the element of a container (consider `foo()` is the `at()` member function or index operator of a vector), `callFoo()` would no longer give access to the element. In addition, we might create an unnecessary copy (if not optimized away).

- Any return type that is a reference (auto&, const auto&, and auto&&) will return a reference to a local object if foo() returns a temporary object by value. Fortunately, compilers can warn when they detect such a bug.

That is, we need a way to say:

- return by-value if we got/have a value
- return by-reference if we got/have a reference

keeping both the type and the value category of what we return.

C++14 introduced a new placeholder type for that purpose: decltype(auto).

```
template<typename T>
decltype(auto) callFoo(T&& arg)          // since C++14
{
  return foo(std::forward<T>(arg));
}
```

With this declaration, callFoo() returns by-value, if foo() returns by-value, and callFoo() returns by-reference, if foo() returns by-reference. In all cases it keeps both the type and the value category.

12.2 decltype(auto)

Just like the other placeholder type auto, decltype(auto) is a placeholder type that lets the compiler deduce the type at initialization time. However, in this case the type is deduced according to the rules of decltype:

- If you initialize it with or return a plain name, the return type is the type of the object with that name.
- If you initialize it with or return an expression, the return type is the type and value category of the evaluated expression as follows with the following encoding:
  - For a **prvalue** it just yields its value type: *type*
  - For an **lvalue** it yields its type as lvalue reference: *type*&
  - For an **xvalue** it yields its type as rvalue reference: *type*&&

For example:

```
std::string s = "hello";
std::string& r = s;

// initialized with name:
decltype(auto) da1 = s;              //std::string
decltype(auto) da2(s);               //  same
decltype(auto) da3{s};               //  same
decltype(auto) da4 = r;              //std::string&

// initialized with expression:
decltype(auto) da5 = std::move(s);   //std::string&&
decltype(auto) da6 = s+s;            //std::string
decltype(auto) da7 = s[0];           //char&
decltype(auto) da8 = (s);            //std::string&
```

For the expressions, by rule the types are deduced as follows:

- Because `std::move(s)` is an *xvalue*, `da5` is an rvalue reference.
- Because `operator+` for strings returns a new temporary string by value (so it is a *prvalue*), `da6` is a plain value type.
- Because `s[0]` returns an lvalue reference to the first character, it is an *lvalue* and forces `da7` also to be an lvalue reference.
- Because `(s)` is an *lvalue*, `da8` is an lvalue reference. Yes, the parentheses make a difference here.

`decltype(auto)` cannot have additional qualifiers:

```
decltype(auto) da{s};              // OK
const decltype(auto)& da1{s};      // ERROR
decltype(auto)* da2{&s};           // ERROR
```

12.2.1 Return Type `decltype(auto)`

When using `decltype(auto)` as a return type, we use the rules of `decltype` as follows:

- If the expression returns/yields a plain value, then the value category is a *prvalue* and `decltype(auto)` deduces a value type.
- If the expression returns/yields an lvalue reference, then the value category is a *lvalue* and `decltype(auto)` deduces an lvalue reference.
- If the expression returns/yields an rvalue reference, then the value category is a *xvalue* and `decltype(auto)` deduces an rvalue reference.

That's exactly what we need for perfect returning: For a plain value we deduce a value and for a reference we deduce a reference of the same type and value category.

As a more general example consider a helper function of a framework that (after some initialization or logging) transparently calls a function as if we would call it directly:

generic/call.hpp

```
#include <utility>  // for forward<>()

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

Because we use universal references and `std::forward<>()`, we perfectly pass the given arguments to the function `f` to call. And because we use `decltype(auto)` as return type, we perfectly return the return value of `f()` to the caller of `call()`. For this reason we can call both functions that return by value and functions that return by reference. For example:

generic/call.cpp

```
#include "call.hpp"
```

```
#include <iostream>
#include <string>

std::string nextString()
{
  static long id{0};
  return "value" + std::to_string(++id);
}

std::ostream& print(std::ostream& strm, const std::string& val)
{
  return strm << "value: " << val;
}

int main()
{
  const auto& ref = call(nextString);    // call() returns temporary; ref extends lifetime
  call(print, std::cout, ref) << '\n';   // call() returns reference to stream
}
```

When calling

```
call(nextString)
```

`call()` calls the function `nextString()` without any arguments and returns a new string by value. This value is then passed by value back to the caller of `call()`.

When calling

```
call(print, std::cout, ref)
```

`call()` calls the function `print()` with `std::cout` and `ref` as perfectly forwarded arguments. The functions returns the passed stream back by reference, which is passed by reference back to the caller of `call()`.

12.2.2 Deferred Perfect Returning

If you want to perfectly return a value that was computed before we come to the `return` statement, declare a local object with `decltype(auto)`:

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

The type of `ret` is just the perfectly deduced type of `f()` and because we return just a name, the return type `decltype(auto)` just uses this type as return type of `call()`.

Don't use `auto&&` in this case to declare `ret`, because you would then always return by reference:

```
template<typename Func, typename... Args>
decltype(auto) call (Func f, Args&&... args)
{
  ...
  auto&& ret{f(std::forward<Args>(args)...)};
  ...
  return ret;   // fatal runtime error: returns a reference to a local object
}
```

Also, don't put additional parentheses around the whole return value in the return statement:

```
template<typename Func, typename... Args>
decltype(auto) call (Func f, Args&&... args)
{
  ...
  decltype(auto) ret{f(std::forward<Args>(args)...)};
  ...
  return (ret);   // fatal runtime error: always returns an lvalue reference
}
```

In this case the return type `decltype(auto)` switches to the rules of expressions and, because `ret` is an lvalue (an object with a name), always deduces an lvalue reference.

If you are used to put parentheses around names and expressions in return statement, just stop doing that. It was never necessary, but now when using `decltype(auto)` it might even be an error.

12.2.3 Perfect Forwarding and Returning with Lambdas

If your function to call is declared as a lambda, for perfect returning you have to change the return type `auto` of the lambda. A declaration such as:

```
[] (auto f, auto&&... args) {
  ...
}
```

stands for:

```
[] (auto f, auto&&... args) -> auto {
  ...
}
```

That means that we always return by value.

By explicitly declaring it to have the return type `decltype(auto)`, we enable perfect returning:

```
[] (auto f, auto&&... args) -> decltype(auto) {
  ...
  decltype(auto) r = f(std::forward<decltype(args)>(args)...);
  ...
```

```
  return r;   // OK returns perfectly due to decltype(auto)
};
```

Again, don't put additional parentheses around the whole return value in the return statement, because then `decltype` always deduces an lvalue reference:

```
auto lmbd2 = [] (auto f, auto&&... args) -> decltype(auto) {
  ...
  decltype(auto) r = f(std::forward<decltype(args)>(args)...);
  ...
  return (r);   // fatal runtime error: always returns an lvalue reference
};
```

12.3 Summary

- `decltype(auto)` is a placeholder type to deduce a value type from a value and a reference type from a reference.
- Use `decltype(auto)` to perfectly return a value in a generic function.
- In a `return` statement, never put parentheses around the return value/expression as a whole.

Part III

Move Semantics in the C++ Standard Library

After introducing all features of move semantics, this part of the book describes several applications of these features in the C++ standard library.

That way you get a good overview for the use of move semantics in practice.

This page is intentionally left blank

Chapter 13

Move Semantics in Types of the C++ Standard Library

This chapter discusses how the C++ standard library supports move semantics, including some tricky or even unexpected scenarios.

13.1 Move Semantics for Strings

We already had several examples in the book about how strings support move semantics. As a type that might allocate memory to hold its value, you can benefit from using move semantics instead of copy semantics.

Examples discussed are:

- Moving strings into a vector
- Implementing a move constructor for strings
- Initializing string members
- Reading strings in a loop and using them after a `move()`

13.1.1 String Assignments and Capacity

Note that the capacity of strings (the memory currently available for the value) usually does not shrink. However, the move operation might shrink or swap the capacity.

Consider the following example:

lib/stringmoveassign.cpp

```
#include <iostream>
#include <string>

int main()
{
  std::string s0;
```

```
  std::string s1{"short"};
  std::string s2{"a string with an extraordinary long value"};
  std::cout << "- s0 capa: " << s0.capacity() << " ('" << s0 << "')\n";
  std::cout << "  s1 capa: " << s1.capacity() << " ('" << s1 << "')\n";
  std::cout << "  s2 capa: " << s2.capacity() << " ('" << s2 << "')\n";

  std::string s3{std::move(s1)};
  std::string s4{std::move(s2)};
  std::cout << "- s1 capa: " << s1.capacity() << " ('" << s1 << "')\n";
  std::cout << "  s2 capa: " << s2.capacity() << " ('" << s2 << "')\n";
  std::cout << "  s3 capa: " << s3.capacity() << " ('" << s3 << "')\n";
  std::cout << "  s4 capa: " << s4.capacity() << " ('" << s4 << "')\n";

  std::string s5{"some pretty reasonable value"};
  std::cout << "- s4 capa: " << s4.capacity() << " ('" << s4 << "')\n";
  std::cout << "  s5 capa: " << s5.capacity() << " ('" << s5 << "')\n";

  s4 = std::move(s5);
  std::cout << "- s4 capa: " << s4.capacity() << " ('" << s4 << "')\n";
  std::cout << "  s5 capa: " << s5.capacity() << " ('" << s5 << "')\n";
}
```

Here, you can first see that usually even empty strings have capacity for some characters due to the *small string optimization (SSO)*, which usually reserves 15 or 22 bytes for the value in the string itself. Beyond the size for SSO, the string allocates the memory on the heap, which is at least the necessary size of the value (e.g., you might see here 41 or 47). For this reason, after

```
std::string s0;
std::string s1{"short"};
std::string s2{"a string with an extraordinary long value"};
```

we might get the following output:

```
- s0 capa: 15 ('')
  s1 capa: 15 ('short')
  s2 capa: 47 ('a string with an extraordinary long value')
```

or:

```
- s0 capa: 22 ('')
  s1 capa: 22 ('short')
  s2 capa: 47 ('a string with an extraordinary long value')
```

With the current implementations, a moved-from string is usually empty. This even applies when the value is stored in externally allocated memory (so that we have to copy all characters). After:

```
std::string s3{std::move(s1)};
std::string s4{std::move(s2)};
```

we usually get something like:

```
- s1 capa: 15 ('')
  s2 capa: 15 ('')
  s3 capa: 15 ('short')
  s4 capa: 47 ('a string with an extraordinary long value')
```

Note that it is not guaranteed that `s1` gets empty. Here as usual a moved-from string is in a *valid but unspecified state*, which means that `s1` could still have the value `"short"` or even any other value.

Move assigning a different string value might shrink the capacity. The last two steps of the example program essentially perform:

```
std::string s4{"a string with an extraordinary long value"};
std::string s5{"some pretty reasonable value"};
s4 = std::move(s5);
```

You can find the following outputs in practice:

- Swapping memory:

```
- s4 capa: 41 ('a string with an extraordinary long value')
  s5 capa: 28 ('some pretty reasonable value')
- s4 capa: 28 ('some pretty reasonable value')
  s5 capa: 41 ('')
```

- Moving memory (after freeing the old memory):

```
- s4 capa: 47 ('a string with an extraordinary long value')
  s5 capa: 31 ('some pretty reasonable value')
- s4 capa: 31 ('some pretty reasonable value')
  s5 capa: 15 ('')
```

As you can see, capacity of `s4` usually shrinks, sometimes to the capacity of the destination, sometimes to the minimum capacity for empty strings. However, neither is guaranteed.

13.2 Move Semantics for Containers

We already had several examples in the book about how containers, such as `std::vector<>`, support move semantics. As types that usually allocate memory to hold their values, you can benefit from using move semantics instead of copy semantics. However, `std::array<>` does not allocate memory on the heap, so that special rules for `std::array<>` apply.

In the motivating example of move semantics, we have seen the following support for move semantics:

- By overloading `push_back()`, the C++ standard supports move semantics for inserting new elements.
- By providing a move constructor and a move assignment operator, containers make copying temporary objects (such as return values) cheap.

That is typical. All containers support move semantics when

- copying them,
- assigning them,
- inserting elements.

But there is more to say.

13.2.1 Basic Move Support of Containers as a Whole

All containers define a move constructor and move assignment operator to support the move semantics for unnamed temporary objects and objects marked with `std::move()`.

For example, class `std::list<>` is declared as follows:

```
template<typename T, typename Allocator = allocator<T>>
class list {
  public:
    ...
    list(const list&);                      // copy constructor
    list(list&&);                           // move constructor
    list& operator=(const list&);           // copy assignment
    list& operator=(list&&) noexcept(...);  // move assignment
    ...
};
```

This makes returning/passing a container by value and assigning the return value cheap. For example:

```
std::list<std::string> createAndInsert()
{
  std::list<std::string> coll;
  ...
  return coll;          // move constructor if not optimized away
}

std::list<std::string> v;
...
v = createAndInsert();  // move assignment
```

However, note that we have additional requirements and guarantees that apply to the move constructor and move assignment operator of all containers except `std::array<>`. They essentially mean that moved-from containers are usually empty.

Container Guarantees for Move Constructors

For the move constructor:

ContainerType `cont1{...};`
ContainerType `cont2{std::move(cont1)};`

the C++ standard specifies constant complexity. That means that the duration of a move does not depend on the number of elements.

With this guarantee, implementers have no other option but to steal the memory of elements as a whole from the source object `cont1` to the destination object `cont2`, leaving the source object `cont1` in an initial/empty state.

You might argue that the move constructor could in addition create a new value in the source object, but that doesn't make much sense because it only makes the operation slower.

For vectors, it is even not allowed to create a new value, because the move constructor guarantees there not to throw:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
  public:
    ...
    vector(const vector&);                    // copy constructor
    vector(vector&&) noexcept;                // move constructor
    ...
};
```

So, for vectors it is more or less required that after calling the move constructor, the moved-from object is empty. For the other containers (except `std::array<>`), it is not strictly required but usually makes no sense to implement anything else.

Container Guarantees for Move Assignment Operators

For the move assignment operator:

```
ContainerType cont1{...}, cont2{...};
cont2 = std::move(cont1);
```

the C++ standard guarantees that this operation either overwrites or destroys each element of the destination object `cont2`. That way it is guaranteed that all resources, that the elements of the destination container `dest2` own on entry, are released. As a consequence, there are only two ways to implement a move assignment:

- Destroy the old elements and move the whole contents of the source to the destination (i.e., move the pointer to the memory of the source to the destination)
- Move element-by-element from the source `cont1` to the destination `cont2` and destroy all remaining element not overwritten in the destination

Both ways require linear complexity, which is therefore specified. Note that just swapping the contents of the source and the destination is not valid.

However, since C++17 all containers guarantee not to throw when the memory is interchangeable.[1] For example:

```
template<typename T, typename Allocator = allocator<T>>
class list {
  public:
    ...
    list& operator=(list&&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    ...
};
```

[1] See http://wg21.link/n4258.

That `noexcept` guarantee rules out the second option to implement the move assignment as an element-by-element move. The reason is that in general move operations might throw. Only the implementation destroying old elements does not throw.[2] So, when the memory is interchangeable, we have to use the first option to implement the move assignment operator.

That means that, if the memory is interchangeable (which is especially true if the default standard allocator is used), it is essentially more or less required that the moved-from container `cont1` is empty after a move assignment. This applies to all containers except `std::array<>`.

After a move assignment to itself a container has an unspecified value.

13.2.2 Insert and Emplace Functions

All containers support moving a new element into the container.

For example, vectors support move semantics, by having two different implementations of `push_back()`:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
  public:
    ...
    // insert a copy of elem:
    void push_back (const T& elem);

    // insert elem, when the value of elem is no longer needed:
    void push_back (T&& elem);
    ...
};
```

The `push_back()` function for rvalues forwards the passed element with `std::move()` so that the move constructor of the element type is called instead of the copy constructor.

In the same way, all containers have corresponding overloads. For example:

```
template<typename Key, typename T, typename Compare = less<Key>,
         typename Allocator = allocator<pair<const Key, T>>>
class map {
  public:
    ...
    pair<iterator, bool> insert(const value_type& x);
    pair<iterator, bool> insert(value_type&& x);
    ...
};
```

[2] Destructors can technically throw, but this breaks all basic guarantees of the C++ standard library. That's why all destructors are `noexcept` by default, so that leaving a destructor with an exception will usually `std::terminate()` the program.

Emplace Functions

Since C++11, containers also provide emplace functions (such as `emplace_back()` for vectors). Instead of passing a single argument of the element type (or convertible to the element type), you can pass multiple arguments to initialize a new element directly in the container. That way you save a copy or move.

Note that even then containers can benefit from move semantics, by supporting move semantics for the initial arguments of constructors.

Function like `emplace_back()` use perfect forwarding to avoid creating copies of the passed arguments. For example, for vectors the `emplace_back()` member function is defined as follows:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
  public:
    ...
    // insert a new element with perfectly forwarded arguments:
    template<typename... Args>
    constexpr T& emplace_back(Args&&... args) {
      ...
      // call the constructor with the perfectly forwarded arguments:
      place_element_in_memory(T(std::forward<Args>(args)...));
      ...
    }
    ...
};
```

Internally, the vector initializes the new element with the perfectly forwarded arguments.

13.2.3 Move Semantics for `std::array<>`

`std::array<>` is the only container that does not allocate memory on the heap. In fact, it is implemented as a templified C data structure with an array member:

```
template<typename T, size_t N>
struct array {
  T elems[N];
  ...
};
```

Therefore we can't implement move operations internally moving pointers to memory.

As a consequence, `std::array<>` has a couple of different guarantees:

- The move constructor has linear complexity because it has to move element by element.
- The move assignment operator might always throw because it has to move assign element by element.

So, in principle there is no difference between copying or moving an array of numeric values:

```
std::array<double, 1000> arr;
...
auto arr2{arr};              // copies all double elements/values
auto arr3{std::move(arr)};   // still copies all double elements/values
```

For all other containers the latter would only move internal pointers to the new object and would therefore be a significantly cheaper operation.

But moving an array is still better than copying if moving the elements is cheaper than copying them. For example:

```
std::array<std::string, 1000> arr;
...
auto arr2{arr};              // copies string-by-string
auto arr3{std::move(arr)};   // moves string-by-string
```

If the strings allocate heap memory (have a significant size if the small string optimization (SSO) is used), moving the array of strings usually is significantly faster.

You can see this with the program *lib/contmove.cpp* checking the difference between copying and moving an array and a vector of different element types (`double`, a small string, and a large string). Note that on your platform there might still be small performance differences between copying and moving an array of `doubles` or small strings, because slightly different code with different optimizations is generated.

13.3 Move Semantics for Smart Pointers

While raw pointers don't benefit from move semantics (their address values are always copies), smart pointers can benefit from move semantics.

- Shared pointers (`std::shared_ptr<>` support move semantics, which is helpful, because moving a shared pointer is significantly cheaper than copying one.
- Unique pointers (`std::unique_ptr<>` even need move semantics, because copying a unique pointer is not possible.

13.3.1 Move Semantics for `std::shared_ptr<>`

Shared pointers have the concept of shared ownership. Multiple shared pointers can "own" the same object and when the last owner is destroyed (or gets a new value), a "deleter" of the owned object is called.

For example:

```
{
  std::shared_ptr<int> sp1;                  // init shared pointer not owning anything
  {
    auto sp2{std::make_shared<int>(42)};     // init shared pointer owning new int with value 42
    ...
    sp1 = sp2;                               // sp1 and sp2 now share ownership
    ...
    *sp2 = 77;                               // modify value via sp2
    ...
  }                                          // sp2 destroyed, sp1 is the only owner
  std::cout << *sp1 << '\n';                 // use modified value via sp1
}                                            // last owner destroyed so that delete is called
```

In this example the assignment operator copies the ownership of the `int`, so that afterwards both shared pointers own the object. However, note that copying the ownership is a pretty expensive operation. The reason is that a counter has to track the number of owners:

- each time we copy a shared pointer, the owner counter is incremented
- each time we destroy a shared pointer or assign a new value, the owner counter is decremented

And modifying the value of the counter is expensive, because the modification is an atomic operation to avoid problems when multiple threads deal with shared pointers that own the same object.

For that reason is is for example significantly cheaper to iterate over a collection of shared pointers by reference:

```
std::vector<std::shared_ptr<...>> coll;
...
for (auto sp : coll) {          // expensive
  ...
}
...
for (const auto& sp : coll) {   // cheap
  ...
}
```

Regarding move semantics, it is therefore better to move shared pointers instead of copying them. For example, instead of implementing this:

```
std::shared_ptr<int> lastPtr;             // init shared pointer not owning anything
while (...) {
  auto ptr{std::make_shared<int>(getValue())};   // init shared pointer owning new int value
  ...
  lastPtr = ptr;                          // expensive (note: ptr no longer used)
}                                         // ptr destroyed, lastPtr is the only owner
```

you should better implement this:

```
std::shared_ptr<int> lastPtr;             // init shared pointer not owning anything
while (...) {
  auto ptr{std::make_shared<int>(getValue())};   // init shared pointer owning new int value
  ...
  lastPtr = std::move(ptr);               // cheap
}                                         // ptr destroyed, lastPtr is the only owner
```

As a consequence, objects having shared pointer members lose the ownership of the resource these members point to when the objects are moved. This is good for the performance, but it might create invalid moved-from states. You should therefore double check the state of moved-from objects that have members of type `std::shared_ptr<>`.

13.3.2 Move Semantics for `std::unique_ptr<>`

13.4 Move Semantics for Pairs

******* This chapter/section is at work *******

13.4.1 `std::make_pair()`

******* This chapter/section is at work *******

13.5 Move Semantics for Containers

******* This chapter/section is at work *******

13.6 `std::optional<>`

******* This chapter/section is at work *******

A type using `const` rvalue references.

13.7 Move Iterators

******* This chapter/section is at work *******

```
std::list<std::string> src;
...
std::vector<std::string> v1{src.begin(), src.end()};          // copy strings into v1

std::vector<std::string> v2{make_move_iterator(src.begin()),  // move strings into v2
                            make_move_iterator(src.end())};
// src still has the same number of elements but with unspecified value
```

13.8 Summary

- ...

This page is intentionally left blank

Chapter 14

Move-Only Types

14.1 Move Semantics for Unique Pointers

******* This chapter/section is at work *******

14.2 Move Semantics for I/O Streams

******* This chapter/section is at work *******

14.3 Move Semantics for Threads

******* This chapter/section is at work *******

14.4 Summary

- ...

This page is intentionally left blank

Glossary

This glossary provides a short definition of the most important non-trivial technical terms used in this book.

C

CPP file
A file in which *definitions* of variables and non-inline functions are located. Most of the executable (as opposed to declarative) code of a program is normally placed in CPP files. They are named *CPP* files because they are usually named with the suffix `.cpp`. But for historical reasons the suffix also might be `.C`, `.c`, `.cc`, or `.cxx`. See also *header file* and *translation unit*.

F

forwarding reference
One of two terms for rvalue references of the form `T&&` where `T` is a deducible template parameter. Special rules that differ from ordinary rvalue references apply (see *perfect forwarding*). The term was introduced by C++17 as replacement for *universal reference*, because the primary use of such a reference is to forward objects. However, note that it does not automatically forward. That is, the term does not describe what it *is* but for what it is typically used for.

full specialization
An alternative definition for a (*primary*) template that no longer depends on any template parameter.

G

glvalue
A category of expressions that produces a location for a stored value (generalized localizable value). A glvalue can be an *lvalue* or an *xvalue*. The chapter about *value categories* describes details.

H

header file
A file meant to become part of a translation unit through a `#include` directive. Such files often contain *declarations* of variables and functions that are referred to from more than one translation unit, as well as *definitions* of types, inline functions, templates, constants, and macros. They are usually named with a suffix like `.hpp`, `.h`, `.H`, `.hh`, or `.hxx`. They are also called *include files*. See also *CPP file* and *translation unit*.

I

include file
See *header file*.

incomplete type
A class that is declared but not defined, an array of unknown size, an enumeration type without the underlying type defined, `void` (optionally with `const` and/or `volatile`), or an array of incomplete element type.

L

lvalue
A category of expressions that produces a location for a stored value that is not assumed to be movable (i.e., *glvalues* that are no *xvalues*). Typical examples are expressions denoting named objects (variables or members) and string literals. The chapter about *value categories* describes details.

N

named return value optimization (NRVO)
A feature that allows us to optimize away the creation of a return value from a named object in a `return` statement. When we already have a local object, which we return in a function by value, the compiler can use the object directly as return value instead of copying it.
Note that the *named return value optimization* differs from the *return value optimization (RVO)*, which is the optimization for returning an object created on the fly in the return statement. The named return value optimization is optional in all versions of C++. If the compiler does not generate corresponding code, move semantics is used to create the return value.

P

prvalue
A category of expressions that perform initializations. Prvalues can be assumed to designate pure mathematical value such as `1` or `true` and temporaries (especially values returned by value). Any *rvalue* before C++11 is a *prvalue* in C++11. The chapter about *value categories* describes details.

R

return value optimization (RVO)
A feature that allows us to optimize away the creation of a return value from a temporary object created in a `return` statement. When we create the return value in the return statement on the fly and return it by value, the compiler can use the created object directly as return value instead of copying it.
Note that the *return value optimization* differs from the *named return value optimization (NRVO)*, which is the optimization for returning a named local object. The *return value optimization* was optional before C++17 but is mandatory since C++17 (*named return value optimization* is still optional).

rvalue
A category of expressions that are not *lvalues*. An rvalue can be a *prvalue* (such as a temporary) or an *xvalue* (e.g., an *lvalue* marked with `std::move()`). What was called a *rvalue* before C++11 is called a *prvalue* in C++11. The chapter about *value categories* describes details.

S

small/short string optimization (SSO)
An approach to save allocating memory for short strings by always reserving memory for a certain number of characters. A typical value in standard library implementations is to always reserve 16 or 24 bytes of memory so that the string can have 15 or 23 characters (plus 1 byte for the null terminator) without allocating memory. This makes all strings objects larger but usually saves a lot of running time, because in practice, strings are often shorter than 16 or 24 characters and allocating memory on the heap is a pretty expensive operation.

T

translation unit
A *CPP* file with all the header files and standard library headers it includes using `#include` directives, minus the program text that is excluded by conditional compilation directives such as `#if`. For simplicity, it can also be thought of as the result of preprocessing a *CPP* file. See *CPP file* and *header file*.

U

universal reference
One of two terms for rvalue references of the form `T&&` where `T` is a deducible template parameter. Special rules that differ from ordinary rvalue references apply (see *perfect forwarding*). The term was coined by Scott Meyers as a common term for both *lvalue reference* and *rvalue reference*. Because “universal” was, well, too universal, the C++17 standard introduced the term *forwarding reference* instead.

V

value category
A classification of expressions. The traditional value categories *lvalues* and *rvalues* were inherited from C. C++11 introduced alternative categories: *glvalues* (generalized lvalues), whose evaluation identifies stored objects, and *prvalues* (pure rvalues), whose evaluation initialize objects. Additional categories subdivide *glvalues* into *lvalues* (localizable values) and *xvalues* (eXpiring values). In addition, in C++11 *rvalues* serve as a general category for both *xvalues* and *prvalues* (before C++11, *rvalues* were what *prvalues* are in C++11). The chapter about *value categories* describes details.

variadic template
A template with a template parameter that represents an arbitrary number of types or values.

X

xvalue
A category of expressions that produce a location for a stored object that can be assumed to be no longer needed. A typical example is an *lvalue* marked with `std::move()`. The chapter about *value categories* describes details.

Index

J

L

M

N

O

P

R

S

T

U

V

X