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 **2020-12-19**.

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 **xi**

An Experiment . . . xi

Versions of This Book . . . xii

Acknowledgments . . . xiii

About This Book **xv**

What You Should Know Before Reading This Book . . . xv

Overall Structure of the Book . . . xv

How to Read This Book . . . xvi

The Way I Implement . . . xvi

The C++ Standards . . . xvii

Example Code and Additional Information . . . xviii

Feedback . . . xviii

Part I: Basic Features of Move Semantics **1**

1 The Power of Move Semantics **3**

1.1 Motivation for Move Semantics . . . 3

1.1.1 Example with C++03 (Before Move Semantics) . . . 3

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

1.2 Implementing Move Semantics . . . 18

1.2.1 Using the Copy Constructor . . . 19

1.2.2 Using the Move Constructor . . . 20

1.3 Copying as a Fallback . . . 21

1.4 Move Semantics for `const` Objects . . . 22

1.4.1 `const` Return Values . . . 23

1.5 Summary . . . 23

2 Core Features of Move Semantics 25

2.1 Rvalue References . . . 25
2.1.1 Rvalue References in Detail . . . 25
2.1.2 Rvalue References as Parameters . . . 26
2.2 `std::move()` . . . 27
2.2.1 Header File for `std::move()` . . . 27
2.2.2 Implementation of `std::move()` . . . 28
2.3 Moved-From Objects . . . 28
2.3.1 Valid but Unspecified State . . . 28
2.3.2 Reusing Moved-From Objects . . . 29
2.3.3 Move Assignments of Objects to Themselves . . . 30
2.4 Overloading by Different References . . . 30
2.4.1 `const` Rvalue References . . . 31
2.5 Passing by Value . . . 31
2.6 Summary . . . 32

3 Move Semantics in Classes 35

3.1 Move Semantics in Ordinary Classes . . . 35
3.1.1 When is Move Semantics Automatically Enabled in Classes? . . . 38
3.1.2 When Generated Move Operations Are Broken . . . 39
3.2 Implementing Special Copy/Move Member Functions . . . 40
3.2.1 Copy Constructor . . . 41
3.2.2 Move Constructor . . . 42
3.2.3 Copy Assignment Operator . . . 43
3.2.4 Move Assignment Operator . . . 44
3.2.5 Using the Special Copy/Move Member Functions . . . 46
3.3 Rules for Special Member Functions . . . 48
3.3.1 Special Member Functions . . . 49
3.3.2 By Default, We Have Copying and Moving . . . 50
3.3.3 Declared Copying Disables Moving (Fallback Enabled) . . . 50
3.3.4 Declared Moving Disables Copying . . . 51
3.3.5 Deleting Moving Makes No Sense . . . 52
3.3.6 Disabling Move Semantics with Enabled Copy Semantics . . . 53

3.3.7 Moving for Members with Disabled Move Semantics . . . 54
3.3.8 Exact Rules for Generated Special Member Functions . . . 54
3.4 The Rule of Five or Three . . . 57
3.5 Summary . . . 58

4 How to Benefit From Move Semantics 59
4.1 Avoid Objects with Names . . . 59
4.1.1 When You Cannot Avoid Using Names . . . 60
4.2 Avoid Unnecessary `std::move()` . . . 60
4.3 Initialize Members with Move Semantics . . . 61
4.3.1 Initialize Members the Classical Way . . . 61
4.3.2 Initialize Members via Moved Parameters Passed by Value . . . 63
4.3.3 Initialize Members via Rvalue References . . . 66
4.3.4 Compare the Different Approaches . . . 70
4.3.5 Summary for Member Initialization . . . 72
4.3.6 Should We Now Always Pass by Value and Move? . . . 73
4.4 Move Semantics in Class Hierarchies . . . 75
4.4.1 Implementing a Polymorphic Base Class . . . 75
4.4.2 Implementing a Polymorphic Derived Class . . . 77
4.5 Summary . . . 78

5 Overloading on Reference Qualifiers 79
5.1 Return Type of Getters . . . 79
5.1.1 Return by Value . . . 79
5.1.2 Return by Reference . . . 80
5.1.3 Using Move Semantics to Solve the Dilemma . . . 81
5.2 Overloading on Qualifiers . . . 83
5.3 When to Use Reference Qualifiers . . . 84
5.3.1 Reference Qualifiers for Assignment Operators . . . 84
5.3.2 Reference Qualifiers for Other Member Functions . . . 86
5.4 Summary . . . 87

6 Moved-From States 89
6.1 Required and Guaranteed States of Moved-From Objects . . . 89
6.1.1 Required States of Moved-From Objects . . . 90

6.1.2 Guaranteed States of Moved-From Objects . . . 91
6.1.3 Broken Invariants . . . 92
6.2 Destructible and Assignable . . . 93
6.2.1 Assignable and Destructible Moved-From Objects . . . 93
6.2.2 Non-Destructible Moved-From Objects . . . 94
6.3 Dealing with Broken Invariants . . . 97
6.3.1 Breaking Invariants Due to a Moved Value Member . . . 97
6.3.2 Breaking Invariants Due to Moved Consistent Value Members . . . 100
6.3.3 Breaking Invariants Due to Moved Pointer-Like Members . . . 102
6.4 Summary . . . 106

7 Move Semantics and `noexcept` 107
7.1 Move Constructors with and without `noexcept` . . . 107
7.1.1 Move Constructors without `noexcept` . . . 107
7.1.2 Move Constructors with `noexcept` . . . 110
7.1.3 Is `noexcept` Worth It? . . . 115
7.2 Details of `noexcept` Declarations . . . 116
7.2.1 Rules for Declaring Functions with `noexcept` . . . 116
7.2.2 `noexcept` for Special Member Functions . . . 117
7.3 `noexcept` Declarations in Class Hierarchies . . . 120
7.3.1 Checking for `noexcept` Move Constructors in Abstract Base Classes . . . 120
7.4 When and Where to Use `noexcept` . . . 122
7.5 Summary . . . 123

8 Value Categories 125
8.1 Value Categories . . . 125
8.1.1 History of Value Categories . . . 125
8.1.2 Value Categories Since C++11 . . . 127
8.1.3 Value Categories Since C++17 . . . 128
8.2 Special Rules for Value Categories . . . 130
8.2.1 Value Category of Functions . . . 130
8.2.2 Value Category of Data Members . . . 130
8.3 Impact of Value Categories When Binding References . . . 133
8.3.1 Overload Resolution with Rvalue References . . . 133
8.3.2 Overloading by Reference and Value . . . 134

8.4 When Lvalues become Rvalues . . . 135
8.5 When Rvalues become Lvalues . . . 135
8.6 Checking Value Categories with `decltype` . . . 136
8.6.1 Using `decltype` to Check the Type of Names . . . 136
8.6.2 Using `decltype` to Check the Value Category . . . 137
8.7 Summary . . . 138

Part II: Move Semantics in Generic Code 139

9 Perfect Forwarding 141
9.1 Motivation for Perfect Forwarding . . . 141
9.1.1 What we Need to Perfectly Forward Arguments . . . 141
9.2 Implementing Perfect Forwarding . . . 143
9.2.1 Universal (or Forwarding) References . . . 144
9.2.2 `std::forward<>()` . . . 145
9.2.3 The Effect of Perfect Forwarding . . . 146
9.3 Rvalue References versus Universal References . . . 147
9.3.1 Rvalue References of Actual Types . . . 148
9.3.2 Rvalue References of Function Template Parameters . . . 148
9.4 Overload Resolution with Universal References . . . 149
9.4.1 Fixing Overload Resolution with Universal References . . . 150
9.5 Perfect Forwarding in Lambdas . . . 151
9.6 Summary . . . 152

10 Tricky Details of Perfect Forwarding 153
10.1 Universal References as Non-Forwarding References . . . 153
10.1.1 Universal References and `const` . . . 153
10.1.2 Universal References in Detail . . . 156
10.1.3 Universal References of Specific Types . . . 157
10.2 Universal or Ordinary Rvalue Reference? . . . 159
10.2.1 Rvalue References of Members of Generic Types . . . 159
10.2.2 Rvalue References of Parameters in Class Templates . . . 160
10.2.3 Rvalue References of Parameters in Full Specializations . . . 161
10.3 How the Standard Specifies Perfect Forwarding . . . 163
10.3.1 Explicit Specification of Types for Universal References . . . 165

10.3.2 Conflicting Template Parameter Deduction with Universal References 166
10.3.3 Pure RValue References of Generic Types 167
10.4 Nasty Details of Perfect Forwarding 167
10.4.1 "Universal" versus "Forwarding" Reference 168
10.4.2 Why && for Both Ordinary Rvalues and Universal References? 169
10.5 Summary 169

11 Perfect Passing with `auto&&` 171
11.1 Default Perfect Passing 171
11.1.1 Default Perfect Passing in Detail 171
11.2 Universal References with `auto&&` 173
11.2.1 Type Deduction of `auto&&` 174
11.2.2 Perfectly Forwarding an `auto&&` Reference 175
11.3 `auto&&` as Non-Forwarding Reference 176
11.3.1 Universal References and the Range-Based `for` Loop 176
11.4 Perfect Forwarding in Lambdas 180
11.5 Using `auto&&` in C++20 Function Declarations 181
11.6 Summary 181

12 Perfect Returning with `decltype(auto)` 183
12.1 Perfect Returning 183
12.2 `decltype(auto)` 184
12.2.1 Return Type `decltype(auto)` 185
12.2.2 Deferred Perfect Returning 187
12.2.3 Perfect Forwarding and Returning with Lambdas 189
12.3 Summary 190

Part III: Move Semantics in the C++ Standard Library 191

13 Move-Only Types 193
13.1 Declaring and Using Move-Only Types 193
13.1.1 Declaring Move-Only Types 194
13.1.2 Using Move-Only Types 194
13.1.3 Passing Move-Only Objects as Arguments 195
13.1.4 Returning Move-Only Objects by Value 196

13.1.5 Moved-From States of Move-Only Objects . . . 196
13.2 Summary . . . 197

14 Moving Algorithms and Iterators 199
14.1 Moving Algorithms . . . 199
14.2 Removing Algorithms . . . 201
14.3 Move Iterators . . . 204
14.3.1 Move Iterators in Algorithms . . . 205
14.3.2 Move Iterators in Constructors and Member Functions . . . 207
14.4 Summary . . . 208

15 Move Semantics in Types of the C++ Standard Library 209
15.1 Move Semantics for Strings . . . 209
15.1.1 String Assignments and Capacity . . . 209
15.2 Move Semantics for Containers . . . 212
15.2.1 Basic Move Support for Containers as a Whole . . . 212
15.2.2 Insert and Emplace Functions . . . 214
15.2.3 Move Semantics for `std::array<>` . . . 216
15.3 Move Semantics for Vocabulary Types . . . 216
15.3.1 Move Semantics for Pairs . . . 217
15.3.2 Move Semantics for `std::optional<>` . . . 221
15.4 Move Semantics for Smart Pointers . . . 222
15.4.1 Move Semantics for `std::shared_ptr<>` . . . 222
15.4.2 Move Semantics for `std::unique_ptr<>` . . . 223
15.5 Move Semantics for IOStreams . . . 225
15.5.1 Moving IOStream Objects . . . 225
15.5.2 Using Temporary IOStreams . . . 226
15.6 Move Semantics for Multithreading . . . 227
15.6.1 `std::thread<>` and `std::jthread<>` . . . 227
15.6.2 Futures, Promises, and Packaged Tasks . . . 228
15.7 Summary . . . 230

Glossary 231

Index 235

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 we still discuss semantic details in the C++ standards committee.

Whenever I have taught what I have learned about C++ move semantics so far, I have 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 number of aspects to be taught, the situations to be clarified, and the consequences to be described. It really was time to write a book about all aspects of move semantics, covering all C++ versions from C++11 up to C++20. I learned a lot and I am sure you will too.

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.

This means that 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 a history of the major updates (newest first):

- **2020-10-27**: Several fixes (see errata)
- **2020-09-18**: Using lambdas with template parameters (since C++20)
- **2020-09-15**: Fix a bug with full specializations of universal references
- **2020-09-12**: `std::move()` for data members
- **2020-09-10**: Fixing deferred perfect returning
- **2020-09-09**: Proof reading done and final fixes to go into print
- **2020-09-06**: Helper trait `is_nothrow_movable` for abstract base classes
- **2020-09-06**: Value category of (references to) functions
- **2020-09-05**: Clarifications on `auto&&` including using it in C++20 functions
- **2020-08-29**: Move assignments of objects to themselves
- **2020-08-28**: Move semantics for `std::pair<>`, `std::optional<>`, threads, etc.
- **2020-08-28**: Unique pointers, IOStreams, and other move-only types
- **2020-08-26**: Several clarifications about universal references
- **2020-08-25**: Using `std::move()` when calling member functions
- **2020-08-20**: New chapter about move-only types
- **2020-08-19**: Discussion of `noexcept` details
- **2020-08-18**: New chapter about moving algorithms and iterators
- **2020-08-10**: Description of move semantics for shared pointers
- **2020-08-10**: Motivation for the reuse of moved-from objects and clarification of move assignments of objects to themselves
- **2020-08-09**: Recommendation not to return by value with `const`
- **2020-07-27**: Move semantics for strings and containers
- **2020-07-24**: General remarks on using reference qualifiers
- **2020-07-23**: Description of details of type deduction of universal references and reference collapsing for perfect forwarding
- **2020-07-07**: Description of perfect returning for lambdas
- **2020-06-21**: New chapter about perfect returning (with small fixes for perfect forwarding)
- **2020-06-12**: Improvements of the chapter about "invalid" states
- **2020-06-08**: New chapter about using `noexcept`
- **2020-06-06**: Improvements of the chapter about invalid states
- **2020-05-03**: Fixing of *move semantics in class hierarchies*
- **2020-04-29**: Fixing of move semantics for classes and invalid states after review
- **2020-04-25**: Fixing of code layout and missing figures in non-PDF versions
- **2020-04-22**: New chapter about moved-from states
- **2020-04-13**: Discussion of when you cannot avoid using `std::move()`
- **2020-02-01**: Discussion of when automatically generated move operations are broken
- **2020-01-19**: Constructors with universal references

- **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 their 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." Therefore, 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++ standards 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).

And finally, many thanks go to my proofreader, Tracey Duffy, who has done a tremendous job of converting my "German English" into native English.

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 as a programmer, you can understand and use move semantics correctly. The book is valuable for those who are just starting to learn about move semantics and is essential for those who 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. You should also be familiar with the basic features of "Modern C++," such as `auto` or the range-based `for` loop.

However, you do not have to be an expert. My goal is to make the content understandable for the average C++ programmer who does not necessarily know all the details of the latest features. 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 beginning to end. That is, later chapters usually rely on features introduced in earlier chapters. However, cross-references also help in specific subsequent topics, indicating where they refer to features and aspects introduced earlier.

The book therefore contains the following parts:

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

How to Read This Book

Do not be afraid by the number of pages in this book. 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 sufficient.

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 following advantages:

- It can be used with fundamental types, class types, aggregates, enumeration types, and `auto`
- It can be used to initialize containers with multiple values
- It can detect narrowing errors (e.g., initialization of an `int` by a floating-point value)
- It cannot 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 be taught, 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 do not use code font here. If you see an ellipsis with code font, code must have these three dots as a language feature (such as for "`typename...`").
- In header files I usually skip the preprocessor guards. All header files should have something like the following:

  ```
  #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++ standards 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.

The next "Even more Modern C++" is already on the horizon, as introduced 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 version supported 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 in 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 mention that explicitly.

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

If you use the ebook, you might want to ensure to have the latest version of this book available (remember it is written and published incrementally). You should also check the book's Web site for the currently known errata before submitting reports. In any case, refer to the publishing date of this version when giving feedback. The current publishing date is **2020-12-19** (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 benefits of move semantics using a short code example.

1.1 Motivation for Move Semantics

To understand the basic principles of move semantics, let us look at the execution of a small piece of code, first without move semantics (i.e., compiled with an old C++ compiler that supports only C++03) and then with move semantics (compiled with a modern C++ compiler that supports 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 us look at the individual steps of the program (inspecting both the stack and the heap) when we compile this program with a C++ compiler that does ***not*** support 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 that has 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"`:

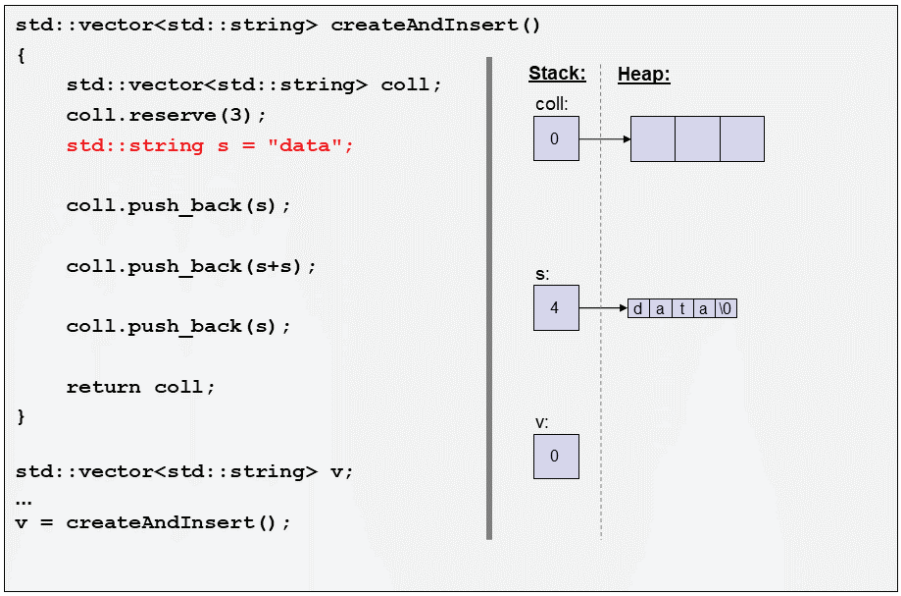
  ```
  std::string s = "data";
  ```

  A string is something like a vector with `char` elements. Essentially, we create an object on the stack with a member for the number of characters (having the value 4) and a pointer to the memory for the characters.[1]

  After this statement, the program 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] Internally, strings also store a terminating null character to avoid allocating memory when they are asked for a C string representation of their value with the member function `c_str()`.

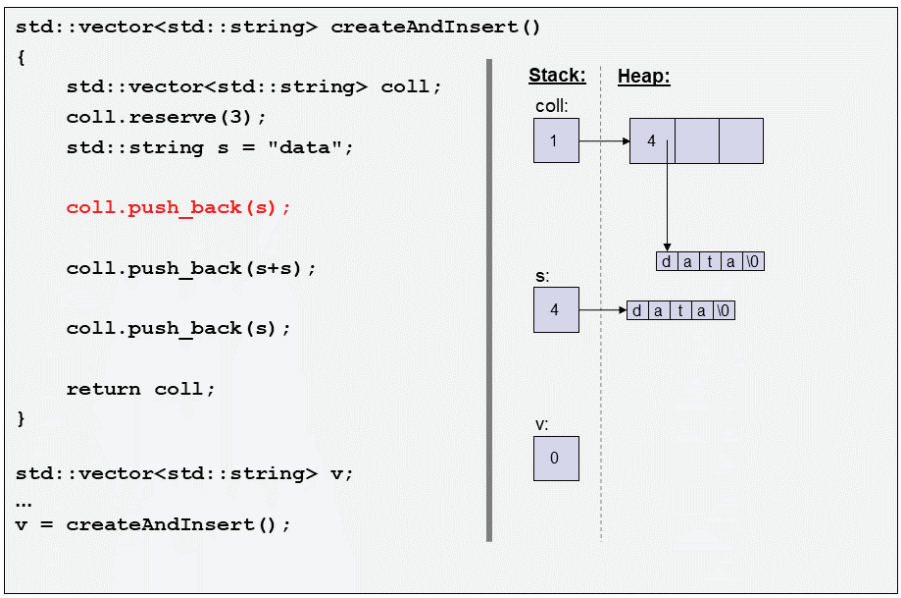
[2] With the *small string optimization (SSO)*, the string `s` might store its whole value on the stack provided the value is not too long. However, for the general case, let us assume that we do not have the small string optimization or the value of the string is long enough so that the small string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



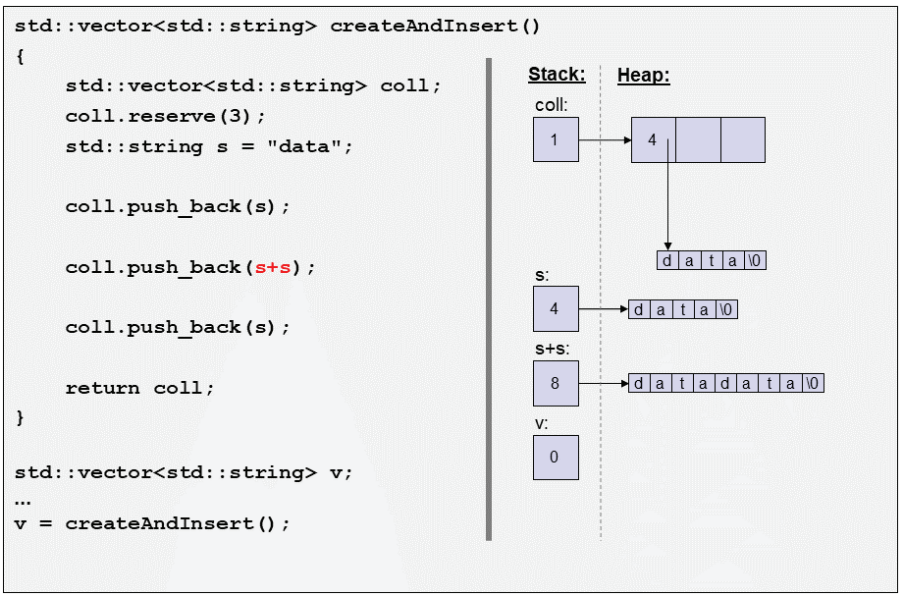
So far, we have nothing to optimize in this program. The current state is that we have two vectors, `v` and `coll`, and two strings, `s` and its copy, which is the first element in `coll`. They should all be separate objects with their own memory for the value, because modifying one of them should not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

- Let us now look at the next statement, which creates a new temporary string and again inserts it into the vector:

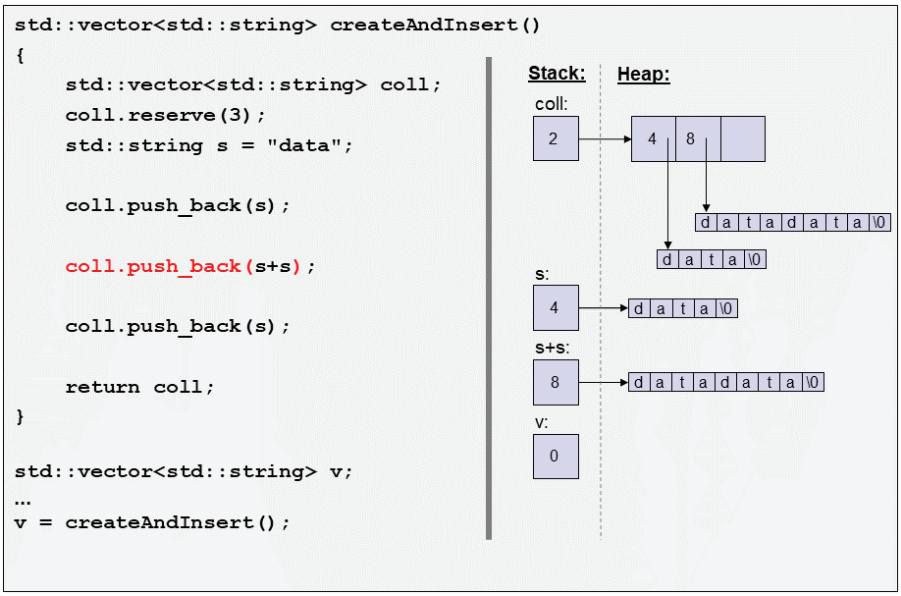
```
coll.push_back(s+s);
```

This statement is performed in three steps:

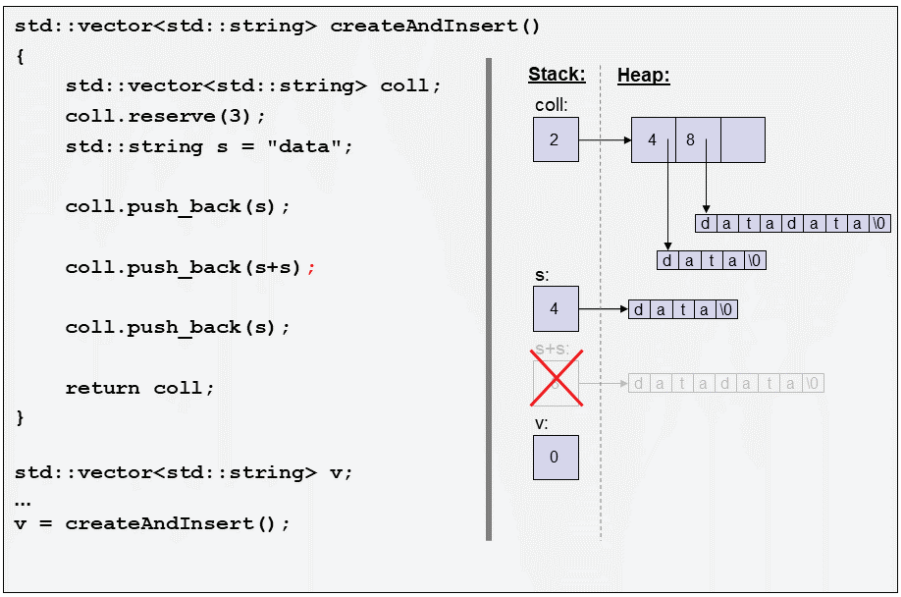
1. We create the temporary string `s+s`:



2. 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:



3. At the end of the statement, the temporary string `s+s` is destroyed because we no longer need it:

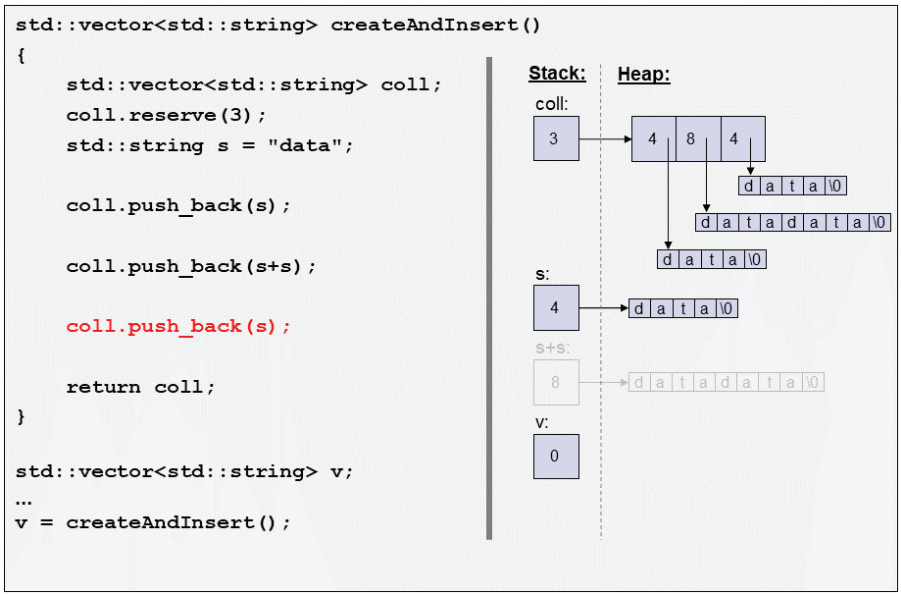


Here, we have the first moment where we generate code that is not performing well: we create a copy of a temporary string and destroy the source of the copy immediately afterwards, which means that we unnecessarily allocate and free memory that we could have just moved from the source to the copy.

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

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

Again, `coll` copies `s`:



This is also something to improve: because the value of `s` is no longer needed some optimization could use the memory of `s` as memory for the new element in the vector instead.

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

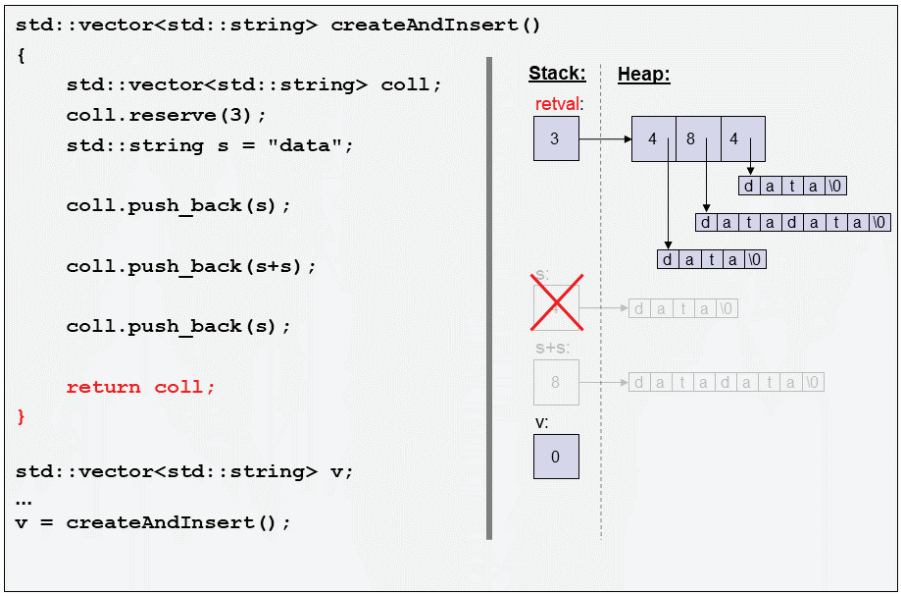
```
  return coll;
}
```

Here, the behavior of the program becomes a bit more complicated. We return by value (the return type is not a reference), which 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` is 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. However, that is 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 is not, either. It is simply up to the compiler whether the named return value optimization is performed.

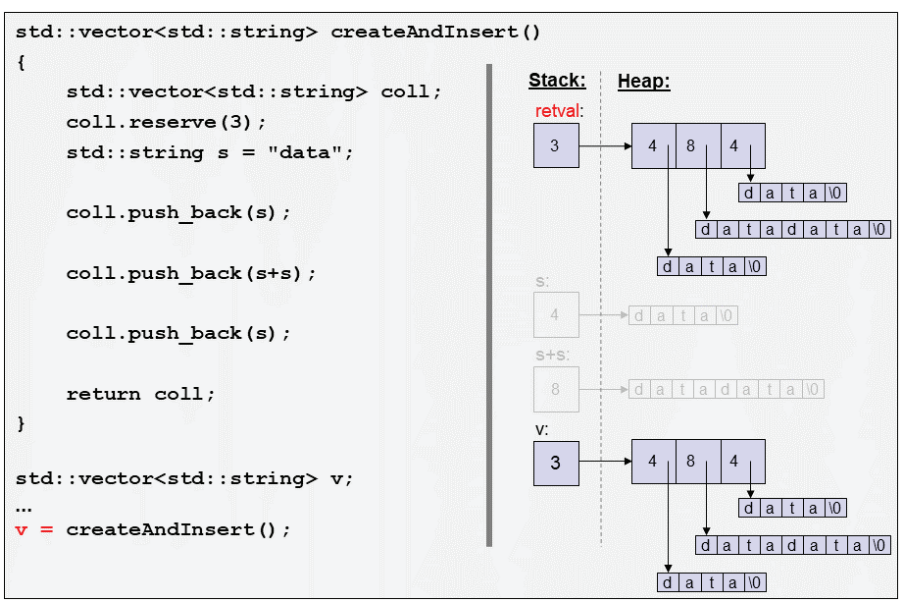
Let us assume that we have the named return value optimization. In that case, at the end of the return statement, `coll` now becomes the return value and the destructor of `s` is called, which frees the memory allocated when it was declared:



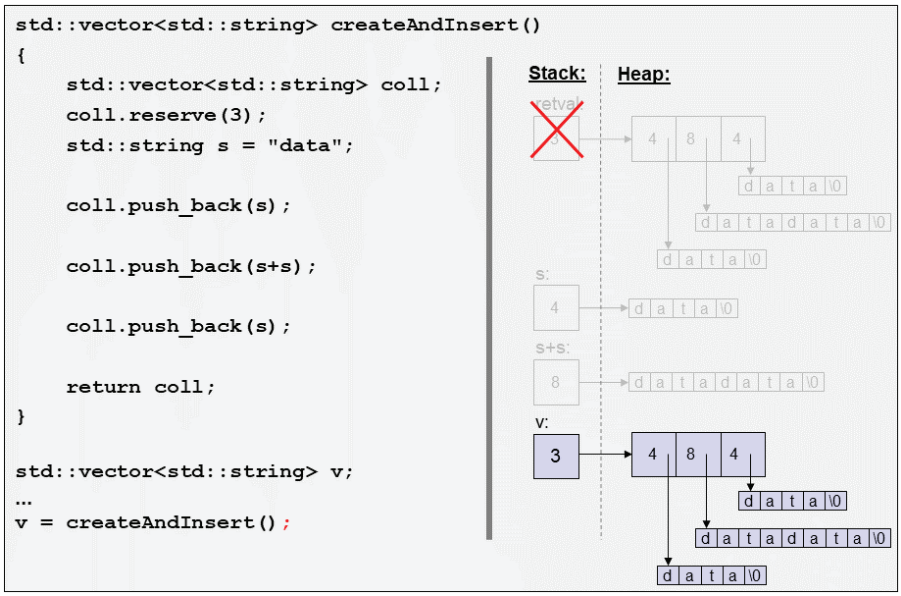
- Finally, we come to the assignment of the return value to `v`:

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

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



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



Again, we create a copy of a temporary object 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 ten times and released it six 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. In particular, instead of the last assignment, we could do the following:

- Pass the vector as an out parameter:

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

- Use `swap()`:

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

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

Since C++11, we have another option: compile and run the program with support for move semantics.

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

Let us now recompile the program with a modern C++ compiler (C++11 or later) that supports 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 small modification, though: we add a `std::move()` call when we insert the last element into `coll`. We will discuss this change when we come to this statement. Everything else is as before.

Again, let us 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`, which is placed on the stack with 0 elements:

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

- Then, we call

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

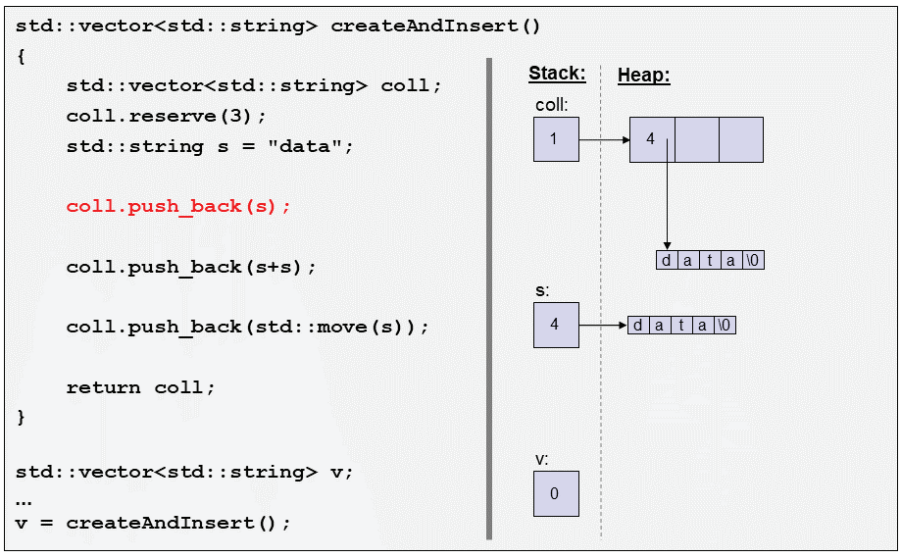
  where we create another empty vector `coll` on the stack and reserve uninitialized memory for three elements on the heap:

  ```
  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:



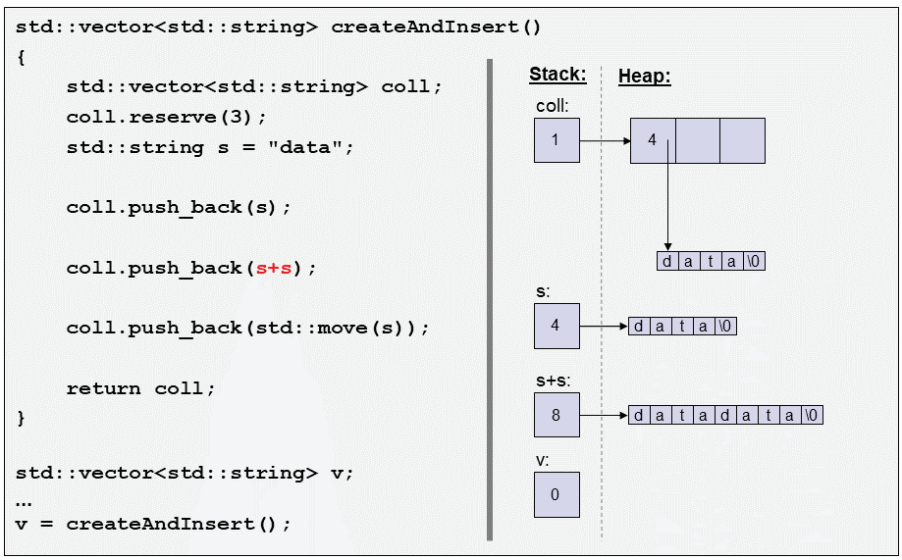
We have two vectors, `v` and `coll`, and two strings, `s` and its copy, which is the first element in `coll`. They should all be separate objects with their own memory for the value, because modifying one of them should not impact any of the other objects.

- This is where things change. First, let us look at the statement that creates a new temporary string and inserts it into the vector:

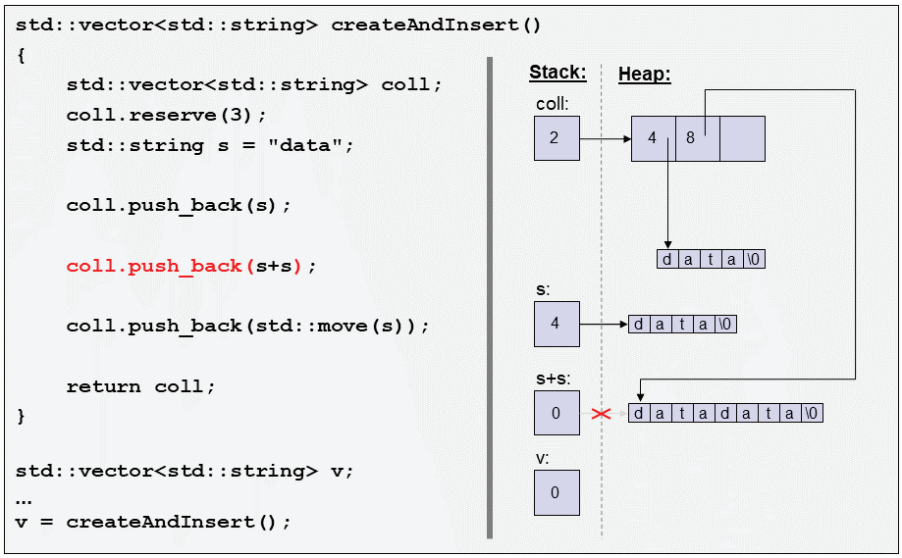
  ```
  coll.push_back(s+s);
  ```

  Again, this statement is performed in three steps:

  1. We create the temporary string `s+s`:

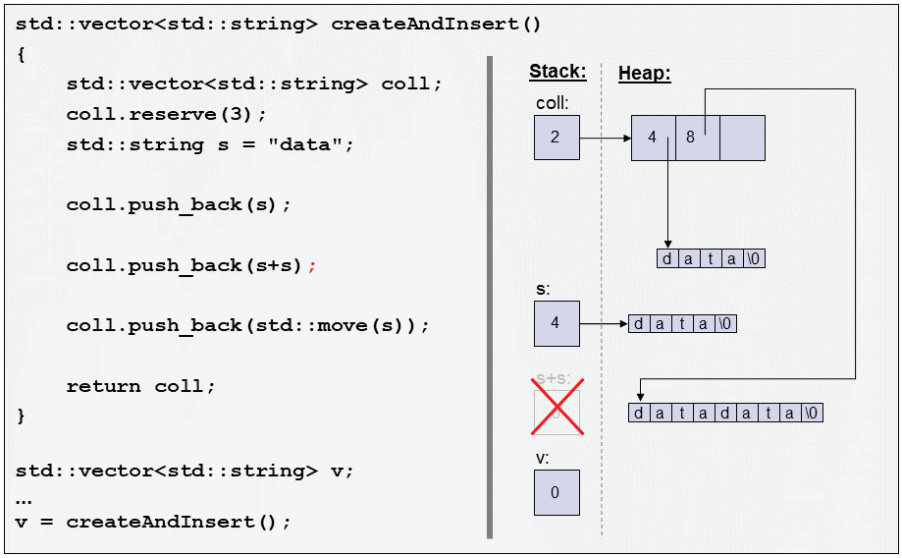


2. We insert this temporary string into the vector `coll`. However, here something different happens now: we steal the memory for the value from `s+s` and move it to the new element of `coll`.



This is possible because since C++11, we can implement special behavior for getting a value that 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. As we can see, the effect is an optimized implementation of copying a string where we no longer need the value: instead of creating an individual deep copy, we copy both the size and the pointer to the memory. However, 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 new value. Essentially, `s+s` is modified so that it gets the state of an empty string. The important point is that it no longer owns its memory. And that is important because we still have a third step in this statement.

3. At the end of the statement, the temporary string `s+s` is destroyed because we no longer need it. However, because the temporary string is no longer the owner of the initial memory, the destructor will not free this memory.



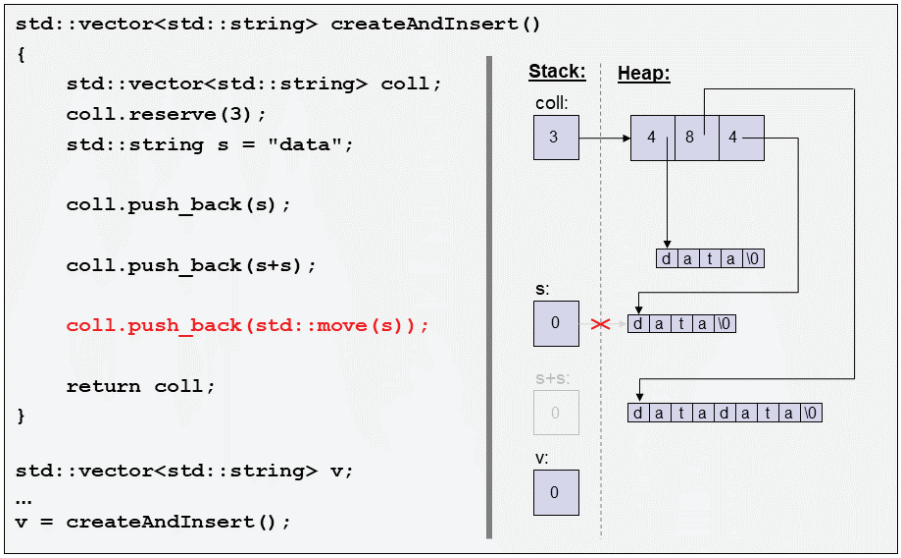
Essentially, we optimize the copying so that we move the ownership of the memory for the value of `s+s` to its copy in the vector.

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 steals the value from the source. It is not a technical move; it is a semantic move implemented by technically moving the memory for the value from the source string to its copy.

- The next statement is the 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`. However, 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 anything. It only says, "***I no longer need this value here***." It allows the implementation of the call to benefit from this mark by performing some optimization when copying the value, such as stealing the memory. Whether the value is moved is something the caller does not know.

– However, an optimization that steals the value has to ensure that the source object is still 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 do not know the value. It is like using a string parameter where you do not know which value was passed.

Note that it is also not guaranteed that the string either has its old value or is empty. The value it has is up to the implementers of the (library) function. In general, implementers can do with objects marked with `std::move()` whatever they like, provided they leave the object in a valid state. There are good reasons for this guarantee, which will be discussed later.

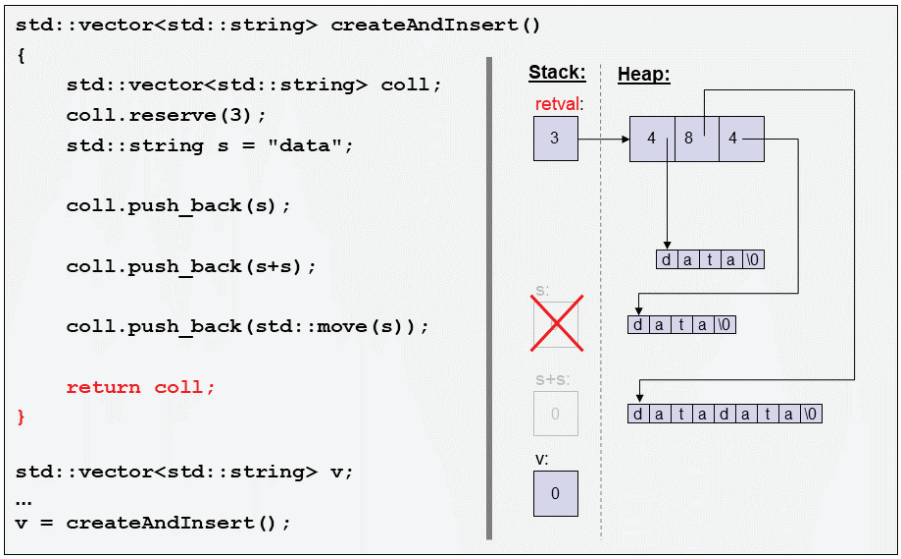
- Again, at the end of `createAndInsert()` we come to the return statement:

```
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 us assume that we have the named return value optimization. In that case, at the end of the return statement, `coll` now becomes 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`:

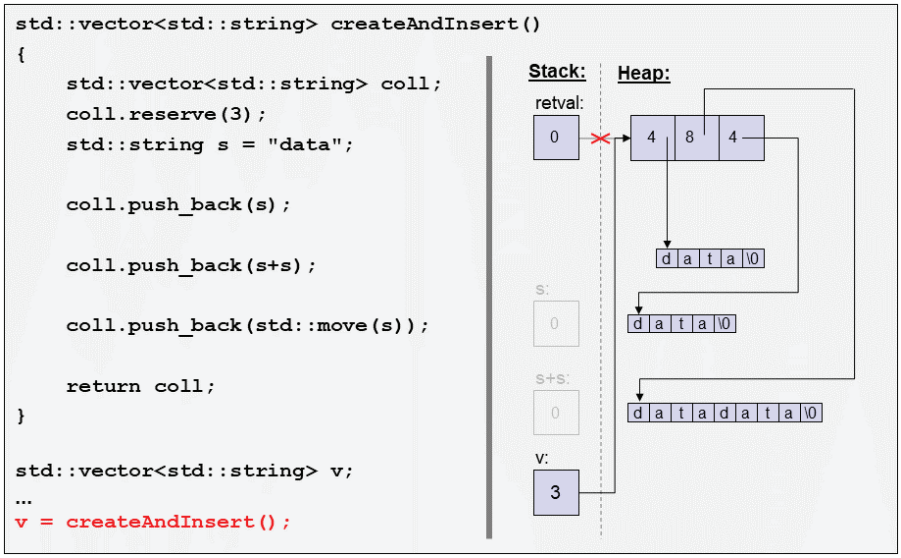


- So, finally, we come to the assignment of the return value to `v`:

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

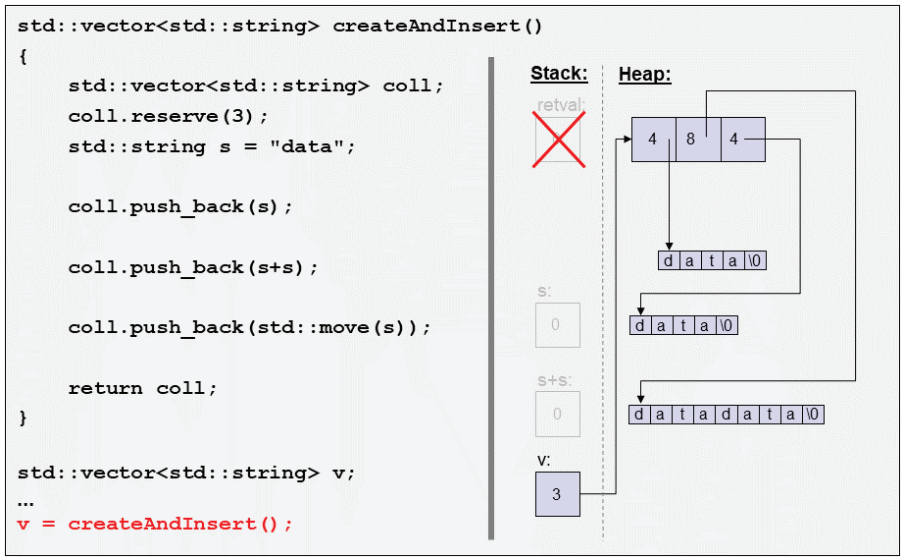
  Again, we can benefit from move semantics now because we have a situation we have already seen: we have to copy (here, assign) a value from a temporary return value that is about to die.

  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 enters into a valid state but we do not know its value.

However, 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 six allocations and releases of memory. All unnecessary memory allocations no longer took place:

- Allocations for inserting a temporary object into the collection
- Allocations for inserting a 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, meaning 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 much 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 us use the previous example to see where and how move semantics is implemented.

Before move semantics was implemented, class `std::vector<>` had only 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" that have 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 as follows:

- 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 receives the value of the passed argument but we can use an optimized implementation that 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. However, 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 or move an element. 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 us 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 us look at how this is implemented. The copy constructor of a very naive implementation of a string class would look like this:[4]

```
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} {                   // copy number of characters
        if (len > 0) {                      // if not empty
          data = new char[len+1];           // - allocate new memory
          memcpy(data, s.data, len+1);      // - and copy the characters
        }
    }
    ...
};
```

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

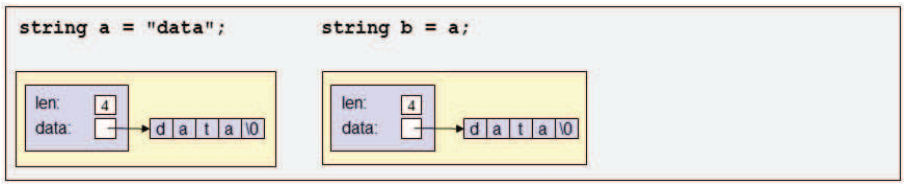
[3] The reason this is called an *rvalue reference* is discussed later in the chapter about value categories.

[4] The implementation of class `std::string` is more complicated because for its internal memory management it uses optimizations and allocators (helper objects that define the way to allocate memory).

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

after initializing the string `a` as follows:

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



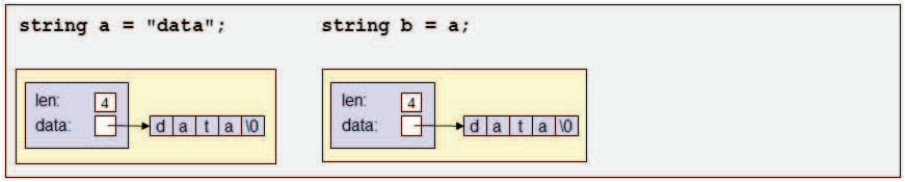
1.2.2 Using the Move Constructor

`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 the value is no longer needed. 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 number of characters and pointer to memory
        s.data = nullptr;             // release the memory for the source value
        s.len = 0;                    // and adjust number of characters accordingly
    }
    ...
};
```

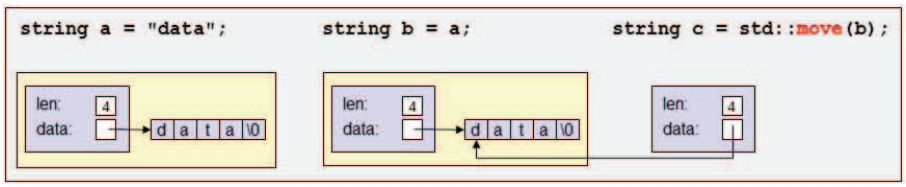
Given the situation from the copy constructor above:



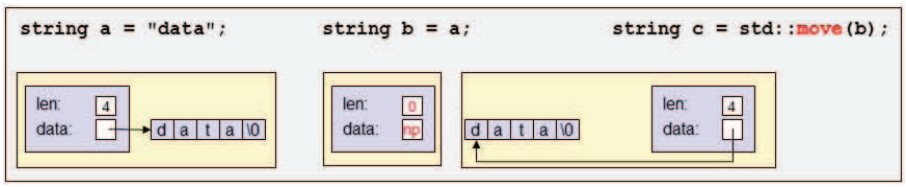
we can call this constructor for a string as follows:

```
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`, meaning that the new string gets ownership of the value of `b` (passed as `s`).



However, this is not enough, because the destructor of `b` would free the memory. Therefore, 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 subsequently 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 to improve performance here).

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 an rvalue reference is preferred. However, if no such function exists, the usual copy semantics is used. That way, we ensure that the caller does not 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 with only numeric members would be an example of that)

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` cannot be moved because any optimizing implementation requires that the passed argument can be modified. We cannot 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 that a `std::move()` for `const` objects essentially has no effect.

In principle, we could provide a special overload for this case by declaring a function taking a `const` rvalue reference. However, 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

The fact that `const` disables move semantics also has consequences for declaring return types. A `const` return value cannot be moved.

Therefore, since C++11, it is no longer good style to return by value with `const` (as some style guides 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, do not 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 the copying of objects, where we no longer need the value. 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 as movable. An object marked with `std::move()` is not (partially) destroyed (the destructor still will be called).
- 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. Usually, the implementer also has to ensure that the passed argument is in a valid state after the call.

- Moved-from objects of the C++ standard library are still valid objects, but you no longer know their value.
- Copy semantics is used as a fallback for move semantics (if copy semantics is supported). 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 used even if the object is explicitly marked with `std::move()`.
- Calling `std::move()` for a `const` object usually has no effect.
- If you return by value (not by reference), do not declare the return value as a whole to be `const`.

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 reference: rvalue references. Let us discuss what they are and how to use them.

2.1.1 Rvalue References in Detail

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

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

The name *rvalue reference* comes from the fact that these objects can usually refer only to *rvalues*, a *value category* for temporary objects that do not have a name and objects marked with `std::move()`.

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).

The syntax used to initialize the reference is irrelevant. You can use the equal sign, braces, or parentheses:

```
std::string s{"hello"};
...
std::string&& r1 = std::move(s);          // OK, rvalue reference to s
std::string&& r2{std::move(s)};           // OK, rvalue reference to s
std::string&& r3(std::move(s));           // OK, rvalue 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 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 receive 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, they will automatically switch to move semantics. This is the case when:

- We pass the value of a temporary object that will automatically be destroyed after the statement.
- We pass a non-`const` 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:

- The parameter can bind only to a temporary object that does not have a name or to an object marked with `std::move()`.
- According to the semantics of rvalue references:
  - The caller claims that it is no longer interested in the value. Therefore, you can modify the object the parameter refers to.
  - However, the caller might still be interested in using the object. Therefore, 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 should not. The recommended programming style is to no longer use an object 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 you use 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 foo1(const std::string& lr);  // binds to the passed object without modifying it
void foo1(std::string&& rv);       // binds to the passed object and might steal/modify the value
...
std::string s{"hello"};
...
foo1(s);                           // calls the first foo1(), s keeps its value
foo1(std::move(s));                // calls the second foo1(), s might lose its value
```

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

```
void foo2(const std::string& lr);  // binds to the passed object without modifying it
...                                // no other overload of foo2()
std::string s{"hello"};
...
foo2(s);                           // calls foo2(), s keeps its value
foo2(std::move(s));                // also calls foo2(), s keeps its value
```

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

```
void foo3(std::string&);           // modifies the passed argument
...
std::string s{"hello"};
...
foo3(s);                           // OK, calls foo3()
foo3(std::move(s));                // ERROR: no matching foo3() declared
```

Note that it does not make sense to mark a dying object with `std::move()`. In fact, this can even be counterproductive for optimizations.

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

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

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

Programs using `std::move()` usually compile without including this header file, because in practice almost all header files include `<utility>`. However, no standard header file is required to include `utility`. Therefore, when using `std::move()`, you should 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 achieve the same effect by calling `static_cast` manually as follows:

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

Therefore, we could also write:

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

Note that the `static_cast` does a bit more than only changing the type of the object here. It also enables the object to be passed to an rvalue reference (remember that passing objects with names to rvalue references is usually not allowed). 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. However, they should also be valid in the sense that they have a consistent state and all operations work as expected. The only thing you do not 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. However, 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 (potentially undefined behavior)
std::cout << s.front() << '\n';   // ERROR (potentially undefined behavior)
s = "new value";                  // OK
```

Although you do not 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 the declaration or implementation of move operations. The chapter *Moved-From States* will discuss this in detail.

2.3.2 Reusing Moved-From Objects

You might wonder why moved-from objects are still valid objects and are 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
}
```

Each time after we read a line into `row`, we use `std::move()` to move the value of `row` into the vector of all rows. 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 that swaps 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 the value of `a` into a temporary object to be able to move-assign the value of `b` afterwards. The moved-from object `b` then receives the value of `tmp`, which is the former value of `a`.

Code like this is used in sorting algorithms for example, 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. The algorithm might even 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), reused to get other values, and support all other operations objects their type supports without knowing the value. The chapter *Moved-From States* will discuss this in detail.

2.3.3 Move Assignments of Objects to Themselves

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 its value being known:

```
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 you have 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 call-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 that does not have 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 that does not have a name
  - An non-`const` object marked with `std::move()`

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

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 bind to other arguments than non-`const` lvalue references. Therefore, we had to introduce a new syntax and could not just implement move semantics as a different way of functions that modify passed arguments.

2.4.1 `const` Rvalue References

Technically, there is a fourth way of call-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 that does not have 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, stealing the value is allowed, but being `const`, we disable any modification of the passed argument. This is a contradiction in itself.

Nevertheless, creating objects with this behavior is quite easy: Simply 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 happen indirectly 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 (the C++ standard library class `std::optional<>` uses `const` rvalue references).

2.5 Passing by Value

If you declare a function to take 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 signals that it no longer needs the value of the passed argument (by using `std::move()` or passing a temporary object without a name), the parameter `str` is initialized with the value moved from the passed argument.

That means that with move semantics, call-by-value can suddenly become cheap if a temporary object is passed or the passed argument is marked with `std::move()`. Note that just 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 and 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. Passing the argument does not create a new string. Whether the value of the passed argument is stolen/modified 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.

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

2.6 Summary

- Rvalue references are declared with `&&` and no `const`.
- They can be initialized by temporary objects that do not have 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 call-by-value pretty cheap.
- `const` rvalue references are possible but implementing against them usually makes no sense.
- Moved-from objects should be in a *valid but unspecified* state. The C++ standard library guarantees that for its types. You can still (re)use them providing you do not make any assumptions about their value.

This page is intentionally left blank

Chapter 3

Move Semantics in Classes

This chapter shows 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 quite a 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 is 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). After the initialization of `c`, the first output is as follows:

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

We then insert this customer in a vector twice: we copy it once and we move it once:

```
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, do not 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 assigning a different value is not necessarily a way to improve performance, it is quite typical that implementations make both the string and the vector empty.

In any case, we can see that move semantics is automatically enabled for the 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 the following improvements:

- 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 operations generated 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 do something different when optimizing these operations. Therefore, the automatic generation of move operations is disabled when at least one of the following special member functions is 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. For example, if we 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 in that case.

Therefore: **do not implement or declare a destructor if there is not specific need** (a rule a surprising number of programmers do not follow).

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

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

Note that this means that move semantics is disabled only for members that are declared inside this base class. 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 this in detail.

3.1.2 When Generated Move Operations Are Broken

Note that generated move operations might introduce problems even though the generated copy operations work correctly. In particular, you have to be careful in the following situations:

- Members have restrictions on values
  - Values of members have restrictions
  - Values of members depend on each other
- Members with reference 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 the destructor of the object might even fail. For example, objects of the `Customer` class in this chapter might suddenly have an empty name even though we have assertions to avoid that. The chapter about moved-from states will discuss this in detail.

3.2 Implementing Special Copy/Move Member Functions

You can implement the special move member functions yourself. You do this in approximately the same way that a copy constructor and an assignment operator are implemented. The only difference is that the parameter is declared as a non-`const` rvalue reference and that inside the implementation, you have to specify where to optimize the usual copying with something better.

Let us look at a class that has 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)                  // noexcept declaration missing
   : 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) {  // noexcept declaration missing
    std::cout << "MOVEASSIGN " << cust.name << '\n';
    name = std::move(cust.name);
    values = std::move(cust.values);
    return *this;
  }
};
```

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

Note that both the move constructor and the move assignment operator should usually have a `noexcept` declaration when they are manually implemented, which is discussed in the chapter about move semantics and `noexcept`.

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 statement printing 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)                  // noexcept declaration missing
   : 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, extended with the additional print statement.

The key difference to the copy constructor is to declare the parameter as a 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()`.[1] 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. Did we not declare the parameter `cust` to accept only 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. In particular, we might need the value multiple times and not lose it with its first use.

[1] You could also use `std::move(cust).name`, which is discussed later.

Therefore, the fact that move semantics is not passed through is a feature, not a bug. If we were to pass move semantics through, we would not be able to 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 uses of `str` implicitly had 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 an rvalue reference restricts what we can pass to this function but behaves just like 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 cannot 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)
}
```

Note that you should always implement the move constructor with a (conditional) `noexcept` specification to improve the performance of reallocations of a `vector` of `Customers`.

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. The only difference is the print statement at the beginning.

When implementing an assignment operator, you can (and maybe should) check for assignments of an object to itself. However, note that the default assignment operator generated does not do that and see the comments about this when discussing self-assignments with the move assignment operator.

You might also want to declare the assignment operator with reference qualifiers.

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) {  // noexcept declaration missing
    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 that takes 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. Finally, we return the object for further use.

Note that you should usually implement the move assignment operator with a `noexcept` specification.

You might also want to declare the move assignment operator with reference qualifiers.

Dealing with Move Assignments of Objects to Themselves

You might wonder whether you should check for assignments of an object to itself when implementing the move assignment operator. For example, a self move might happen as follows:

```
Customer c{"GNU's Not Unix"};
c = std::move(c);                       // move assigns c to itself
```

This looks easy to avoid but we might use references and pointers, and in that case, it is not necessarily that obvious that two objects are the same. Consider the following code as a still quite simple example:

```
std::vector<Customer> coll;
coll.emplace_back("GNU's Not Unix");    // coll has 1 element
coll[0] = std::move(coll.back());       // move assigns the only element to itself
```

As written, all types in the C++ standard library receive a valid but unspecified state when objects are moved to themselves. This means that by default, you might lose the values of your members and you might even have a more severe problem if your type does not work properly with members that have arbitrary values. In fact, the resulting state might be a problem in the following situations:

- Some values of a moved-from member are a problem
- The values of members depend on each other

The traditional/naive way to protect against self-assignments is to check whether both operands are identical (have the same address). You can also do this when implementing the move assignment operator:

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

Having a check like this is pretty cheap and would have the benefit that the object keeps its value on a self-move-assignment. However, note that in recursive data structures, it can also be a problem if you (move) assign a child object to a parent object. If we delete the old value of the parent (which owns the child) first before we assign the new value, you might then assign a deleted object. In this case, the objects are not identical, which means that comparing their addresses does not help.

In addition, consider the following from style guides:

- In his book *Effective C++* Scott Meyers says that self-assignments are usually not a problem when resource management is encapsulated properly by helper classes (but we did not have move semantics at that time). Furthermore, he points out that otherwise, you also have to ensure that the operator is exception-safe.
- The C++ Core Guidelines categorize this problem as a "*one-in-a-million* problem" that does not appear to be relevant in practice.[2]

It seems that in most cases, you can ignore self-move-assignments, which usually means that in your types, you give the same guarantee as the C++ standard library: after move assigning an object to itself, the object is in a valid but unspecified state.

[2] See http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self.

3.2.5 Using the Special Copy/Move Member Functions

Let us test the class `Customer` with a small program:

basics/customerimpl.cpp

```
#include "customerimpl.hpp"
#include <iostream>
#include <string>
#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 from time to time, a vector reallocates its internal memory (capacity) so that it is big enough for all elements. In this case, the vector grows from having memory for 1 to having memory for 2, 4, 8, and finally 16 elements. Therefore, 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. This has to do with the missing `noexcept` declarations, which we will discuss in the chapter about move semantics and `noexcept`.

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 growing by 50% each time).

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.

The Number of Copies We Saved

The output of this program again demonstrates the benefit of move semantics. Before move semantics support, we would only have copies when inserting and sorting elements. However, even in this small program that has 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, meaning that the number of saved copies might differ slightly.
- The only copies performed are caused 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 move semantics and `noexcept`.

3.3 Rules for Special Member Functions

Let us talk about special member functions and in particular, specify exactly when and how the special copy and move member functions are generated.

3.3.1 Special Member Functions

Let us first look briefly at the term *special member function*, because it is used in different ways. The C++ standard defines the following six 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, in many cases such as in *the rule of five*, we talk only about five these operations, because the default constructor is a little different to the other five (or three before C++11) operations. The other five operations are usually not declared and have more complex dependencies. So make sure you always know what is meant by *the* special member functions (I have tried to avoid this term so far).

Figure 3.1 gives an overview of when special member functions are automatically generated depending on which (other) constructors and special member functions are declared:[3]

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	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	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 declared automatically if no other constructor is user-declared.
- The special copy member functions and the destructor disable move support. The automatic generation of special move member functions is disabled (unless the moving operations are also declared). However,

[3] This table is adopted from Howard Hinnant with his kind permission.

still a request to move an object usually works because the copy member functions are used as a fallback (unless the special move member functions are explicitly deleted).

- The special move member functions disable the normal copying and assignment. The copying and other moving special member functions are deleted so that you can only move (assign) but not copy (assign) an object (unless the other operations are also declared).

Let us 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.

Assume 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:

Assume 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. The 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.

You might wonder why a declared destructor disables move semantics. The chapter about moved-from states discusses a concrete example of a a class with a destructor that causes generated move semantics not to work properly.

3.3.4 Declared Moving Disables Copying

If you have user-declared move semantics, you have disabled copy semantics. The copying special member functions are deleted.

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 to call the copy constructor and the copy assignment operator by declaring them with `=delete`.

Assume 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. However, this time, the consequences for the caller are more severe: the attempt to copy an object will no longer compile.

A class supporting move operations but not 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. In the C++ standard library there are a couple of move-only types (e.g., I/O stream classes, thread classes, and `std::unique_ptr<>`). 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 cannot move (you have disabled this operation; any fallback is not used) and cannot 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
```

You get the same effect by declaring copying special member functions as deleted and that is probably less confusing for other programmers.

Deleting the move operations and enabling the copy operations really 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
```

In this case, `=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 that supports copying but fails when moving is called does not make any sense. For the user of such a class, copying would sometimes work, sometimes not. As a guideline: never `=delete` the special move member functions.[4] If you want to disable both copying and moving, deleting the copying special member functions is enough.

3.3.6 Disabling Move Semantics with Enabled Copy Semantics

Based on what we have just discussed, we now know how to disable move semantics when copying still makes sense. Declaring the special move member functions as deleted is usually not the right way to do it because it disables the fallback mechanism.

The right way to disable move semantics while providing copy semantics is to declare one of the other special member functions (copy constructor, assignment operator, or destructor). I recommend that you default the copy constructor and the 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 functions 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, it is usually better to implement the special move member functions to fix any problem a default generated move operation has. The chapter about 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 will discuss later in detail. You have to declare the special copy member functions but cannot also declare the special move member functions (both deleting and defaulting would not work, implementing them makes the class unnecessarily complicated). Therefore, if you explicitly declare a copying special member function just to disable move semantics, add a big comment to ensure that this declaration is neither removed nor extended by a declaration of the special move member functions.

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

3.3.7 Moving for Members with Disabled Move Semantics

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

For example, assume 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 `cust`:

```
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, we will mention the corresponding guarantees here.

Copy Constructor

The copy constructor is automatically generated when all of the following apply:[5]

- No move constructor is user-declared
- No move assignment operator is user-declared

If generated (implicitly or with `=default`), the copy constructor has the following behavior:[6]

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

The generated copy constructor first passes the source object, to the best matching copy constructor of the base class(es). (remember that copy constructors are always called on a top-down basis). It prefers the copy constructor with the same declaration (usually declared `const&`), but if that is not available it might call the next best matching constructor (e.g., a copy constructor template). Afterwards, it copies all members of its class (again using the best match).

The generated copy constructor is declared as `noexcept` if all copy operations (the copy constructors of all base classes and the copy constructors of all members) give this guarantee.

Move Constructor

The move constructor is automatically generated when all of the following apply:

- No copy constructor is user-declared
- No copy assignment operator is user-declared
- No move assignment operator is user-declared
- No destructor is user-declared

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

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

The generated move constructor first passes the source object, marked with `std::move()` to pass through its move semantics, to the best matching move constructor of the base class(es). The best matching move constructor usually is the one with the same declaration (declared with `&&`). However, if that is not available it might call the next best matching constructor (e.g., a move constructor template or even a copy constructor). Afterwards, it moves all members of its class (again using the best match).

The generated move constructor is declared as `noexcept` if all called move/copy operations (the copy or move constructors of all base classes and the copy or move constructors of all members) give this guarantee.

---

[5] Since C++11, the copy constructor is deprecated if the copy assignment operator or the destructor is user-declared.

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

Copy Assignment Operator

The copy assignment operator is automatically generated when all of the following apply:[7]

- No move constructor is user-declared
- No move assignment operator is user-declared

If generated (implicitly or with `=default`), the copy assignment operator approximately has the following behavior:[8]

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

The generated copy assignment operator first calls the best matching assignment operator of the base class(es) for the passed source object (remember that assignment operators in contrast to copy constructors are *not* called on a top-down basis; they call the base class assignment operator(s) in the implementation). Afterwards it assigns all members of its class (again using the best match).

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

In addition, the generated copy assignment operator is declared as `noexcept` 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 all of the following apply:

- No copy constructor is user-declared
- No move constructor is user-declared
- No copy assignment operator is user-declared
- No destructor is user-declared

If generated (implicitly or with `=default`), the move assignment operator approximately has the following behavior:

```
MyClass& operator= (MyClass&& obj) noexcept-specifier {
  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 first calls the best matching move assignment operator of the base class(es) for the passed source object, marked with `std::move()` to pass through its move semantics. Afterwards it move assigns all members of its class (again using the best match).

---

[7] Since C++11, the copy assignment operator is deprecated if the copy constructor or the destructor is user-declared.

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

You might wonder why we still use members of the source object `obj` after the object was marked with `std::move()`:

```
Base::operator=(std::move(obj));   // - perform move assignments for base class members
```

However, in this case we mark the object for the specific context of a base class, which cannot see the members introduced in this class. Therefore, the derived members have a valid but unspecified state but we can still use the values of the new members.

The generated assignment operator also does not check for assignments of objects to themselves. Thus, in its default behavior, the operator will move assign each member to itself, which usually means that members receive a valid but unspecified value. If this is critical, you have to implement the operator yourself.

In addition, the generated move assignment operator is declared as `noexcept` 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 do not play 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.
- The **default constructor** (the "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.

However, note that these special member functions play a role when talking about moved-from states. The state of the default constructor is often the "natural" state of a moved-from object and a moved-from object always should be destructible.

3.4 The Rule of Five or Three

Whether and which special member functions are automatically generated depends on a combination of several of the rules just described. Many programmers do not know all these rules. Therefore, even before C++11, the usual guideline is to provide either none or all of *the* special member functions for copying, assignment, and destruction.

- Before C++11, this guideline was called the ***Rule of Three***: The guideline was to either declare all three (copy constructor, assignment operator, and destructor) or none of them.
- Since C++11, the rule has become the ***Rule of Five***, which is usually formulated as:[9] The guideline is to either declare all five (copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor) or none of them.

Here, *declaring* means:

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

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

That is, when one of these special member functions is either implemented or defaulted or deleted, you should implement or default or delete all four other special member functions.

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

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

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.

Therefore, I usually teach the ***Rule of Five or Three***:

- If you declare the copy constructor, move constructor, copy assignment operator, move assignment operator, or destructor, ***think very carefully*** about how to deal with the other special member functions.
- If you do not understand move semantics, think only about the copy constructor, copy assignment operator, and destructor if you are declaring one of them. If in doubt, disable move semantics by declaring the copying special member functions with `=default`.

3.5 Summary

- Move semantics is not passed through.
- For every class, the move constructor and move assignment operator are automatically generated (unless there is no way to do so).
- 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. You get move-only types (unless these special move member functions are deleted).
- Never `=delete` a special move member function.
- Do not declare a destructor if there is no specific need. There is no general need in classes derived from a polymorphic base class.

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.

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. The chapter also 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 for modern C++ programming.

4.1 Avoid Objects with Names

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

- We pass a temporary object that 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 `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
```

it would be better to program:

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

to automatically enable move semantics.

Of course, the advice to avoid objects with names might conflict with other important style guidelines such as readability and maintainability of source code. Instead of having a 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 Cannot Avoid Using Names

There are cases where you cannot 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 the same collection twice or calling two different functions storing the value somewhere.
- You have to deal with a parameter. The most common example of this is the following loop:

  ```
  // read and store line by line from myStream 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 with an explicit `std::move()`:

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

Remember that `std::move()` is just a `static_cast` to an rvalue reference. Therefore, `std::move(s)` is an expression that yields the type `std::string&&`. However, this no longer matches the return type and therefore disables the *return value optimization*, which usually allows the returned object to be used as a return value. For types where move semantics is not implemented, this might even force the copying of the return value instead of just using the returned object as the return value.

Therefore, if you return local objects by value, do not use `std::move()`:

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

Using `std::move()` when you already have a temporary object is at least redundant. For a function `createString()` that returns an object by value, you should just use the return value:

```
std::string s{createString()};                 // OK
```

instead of marking it with `std::move()` again:

```
std::string s{std::move(createString())};      // BAD: don't do this
```

Compilers might (have options to) warn about any counterproductive 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 us look at this with a 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

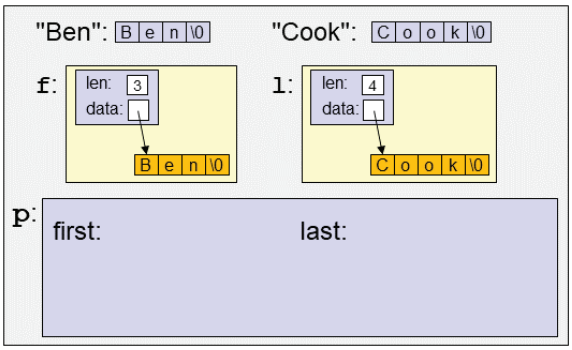
```
#include <string>

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 us look at what happens when we initialize an object of this class with two string literals:

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

The compiler finds out that the provided constructor can perform the initialization. However, the types of the parameters do not fit. Therefore, the compiler generates code to first create two temporary `std::strings`, which are initialized by the values of the two string literals, and binds the parameters `f` and `l` to them:

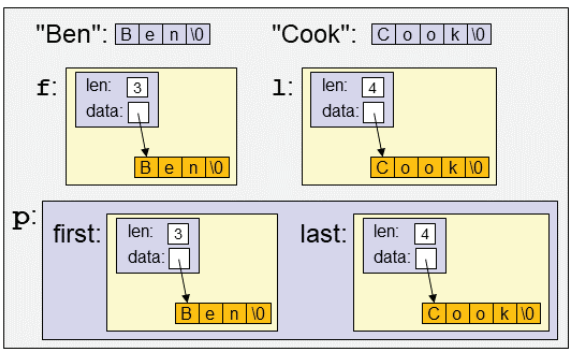


In general (if the *small string optimization (SSO)* is not available or the strings are too long), this means that code is generated that allocates memory for the value of each `std::string`.

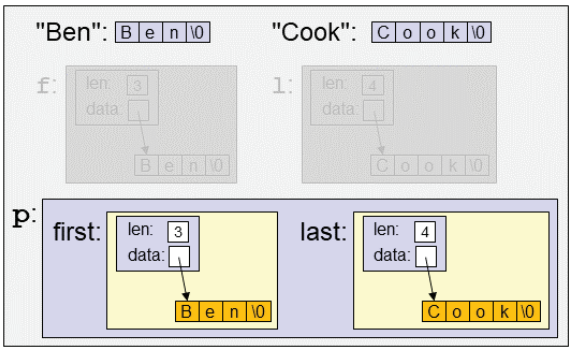
However, the temporary strings created are not used directly as members `first` or `last`. Instead, they are used to initialize these members. Unfortunately, move semantics is not used here for two reasons:

- The parameters `f` and `l`, are objects with names that exist for a longer period 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:



At the end of the constructor, the temporary strings are destroyed:



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

Using non-`const` Lvalue References?

You may wonder why we cannot simply use non-`const` lvalue references here:

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

However, 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
```

In general, a non-`const` lvalue reference does not bind to a temporary object. Therefore, 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 initialize members: the constructor takes each argument by value and moves it into the member:

basics/initmove.hpp

```
#include <string>

class Person {
 private:
```

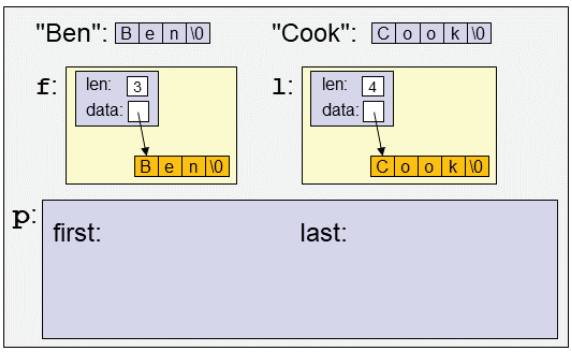
```
  std::string first;   // first name
  std::string last;    // last name
 public:
  Person(std::string f, std::string l)
   : first{std::move(f)}, last{std::move(l)} {
  }
  ...
};
```

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

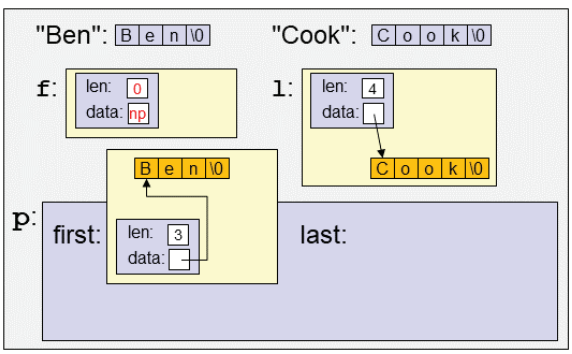
For example, if we pass two string literals:

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

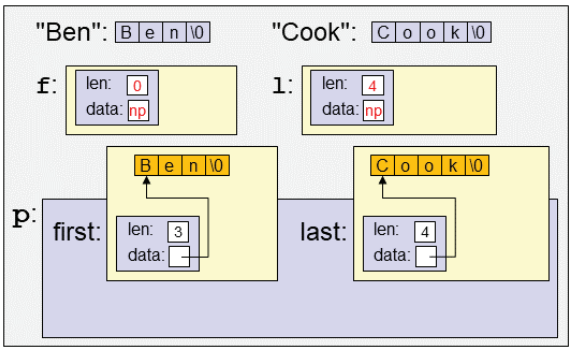
we first use them to initialize the parameters `f` and `l`:



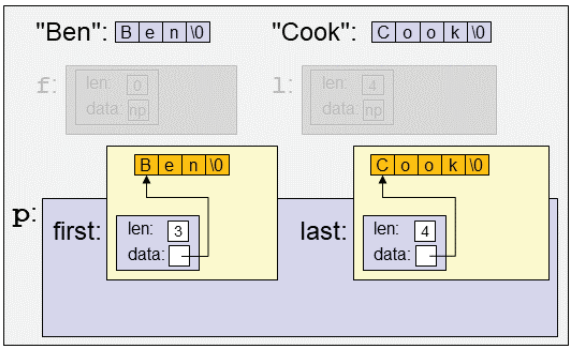
By using `std::move()`, we move the values of the parameters to the members. First, the member `first` steals the value from `f`:



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



Again, at the end of the constructor, the temporary strings are destroyed. This time it takes less time because the destructors of the strings no longer have to free allocated memory:



This way to initialize the members also works fine if we pass `std::strings`:

- If we pass two existing strings without marking them with `std::move()`, we copy the names to the parameters and move them to the members:

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

- If we pass two strings where the value is no longer needed, we do not need any allocation at all:

  ```
  std::string firstname{"Jane"};
  ...
  Person p{std::move(firstname),       // OK, move names via parameters to members
           getLastnameAsString()};
  ```

  In this case we move the passed strings twice: once to initialize the parameters `f` and `l` and once to move the values of `f` and `l` to the members.

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

4.3.3 Initialize Members via Rvalue References

There are even more ways to initialize the members of a `Person`, using multiple constructors.

Using Rvalue References

To support move semantics, we have already learned that we can declare a parameter as a non-`const` rvalue reference. This allows the parameter to steal the value from a passed temporary object or an object marked with `std::move()`.

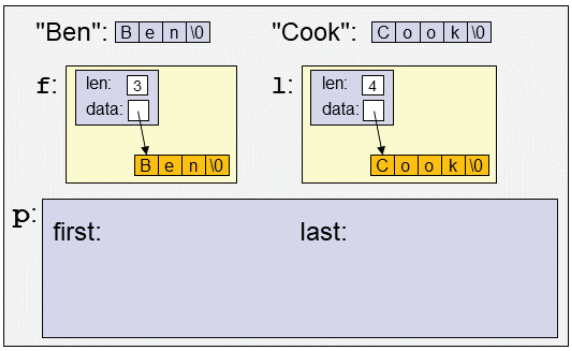
Consider we declare the constructor as follows:

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

This initialization would also work for our passed string literals:

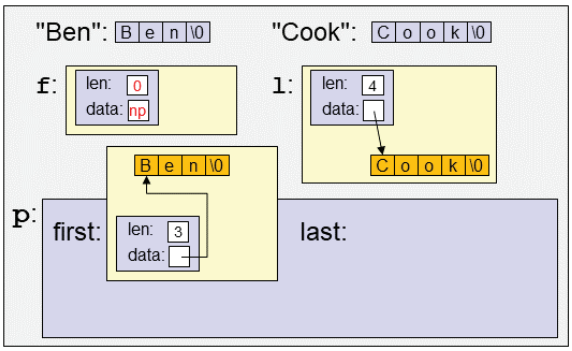
```
Person p{"Ben", "Cook"};
```

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

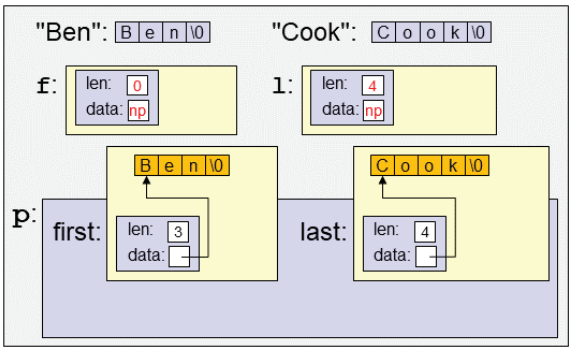


And because we have non-`const` references, we can modify them. In this case, we mark them with `std::move()` so that the initialization of the members can steal the values.

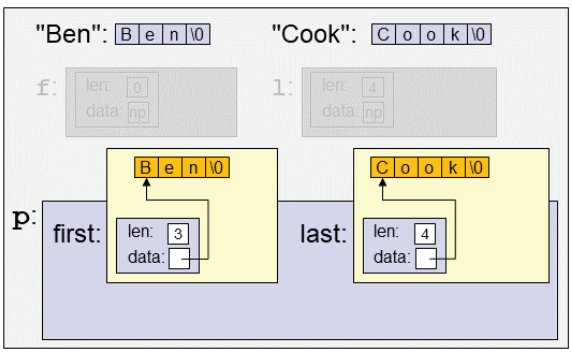
First, the member `first` steals the value from `f`:



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



Again, at the end of the constructor, the temporary strings are destroyed without the need to free allocated memory:



However, this constructor does not work in all cases.

Overloading for Rvalue and Lvalue References

While functions that take rvalue references work fine when we pass temporary objects to them (here, we pass temporary `std::string`s created from string literals), they also have restrictions: we cannot pass named objects that still need the value afterwards. Therefore, is we have only a constructor that takes rvalue references, we cannot 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 a traditional constructor, which is declared with a `const` lvalue reference. However, we might also pass a string literal and an existing string. Therefore, in total, we need four 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 that take 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} {
  }
  ...
};
```

The benefit of this solution is that we reduce the number of moves. If we pass a string literal, we initialize the members directly with the passed pointer instead of creating a `std::string` and moving its value to the member.

4.3.4 Compare the Different Approaches

With the introduction of new ways of initializing members, 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 is usually performance. Therefore, let us measure how long it takes to initialize `Persons` with three kinds of arguments: passing string literals, passing existing strings, and 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 performing the three initializations above with strings of a significant length.

Now we combine the different definitions of the 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 the class `Person`.

The effect of running this code on three different platforms with three different compilers with significant optimization activated are as follows:

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

If we benefit from the *small string optimization (SSO)*, by using quite short strings, which means that we do not allocate any memory at all (and move semantics should be no significant help), the numbers are quite

close. Nevertheless, you can still measure a slight drawback using the traditional constructor that takes a `const` lvalue reference. However, this definitely requires that good optimization is activated.

Besides trying the different test programs, you can use *basics/initperf.cpp* as one combined program that performs all these different measurements on your favorite platform.

However, you might have very expensive members that do 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:
  ...
};
```

In such a situation, we get a problem with the approach of taking an initial argument of 10000 `doubles` 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 with such members), you should use move semantics on of the following alternatives:

- switch from taking the parameter by lvalue reference to taking it by value and move it into the member
- Overload the constructors for move semantics

The first option allows us to have only one constructor so that the code is easier to maintain. However, this does result in more move operations than necessary. Therefore, if move operations might take significant time, you are better to 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, it is better to 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 of 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. In this case, this strategy pays off. If we already have a value, which we update or modify, using this approach would be counterproductive.

A simple example would be setters. Consider the following implementation of the 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) {          // take by value
    first = std::move(s);                     // and move
  }
  ...
};
```

Assume we use the class as follows:

```
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 four allocations (provided we do not have SSO).

Now consider that we implement the setter in the traditional way 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) {  // take by reference
    first = s;                               // and assign
  }
  ...
};
```

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

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) {  // take by lvalue reference
    first = s;                               // and assign
  }
  void setFirstname(std::string&& s) {       // take by rvalue reference
    first = std::move(s);                    // and move assign
  }
  ...
};
```

However, even this approach might be counterproductive, 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 for setting 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 need new memory for it anyway). When modifying an existing value, this policy might be counterproductive.

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 application:

```
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) {          // take by value
    values.push_back(std::move(s));       // and move into the collection
  }
  ...
};
```

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. However, there are a few additional aspects to consider.

4.4.1 Implementing a Polymorphic Base Class

A polymorphic base class usually introduces the `virtual` member functions you can call for all objects of derived classes. For example:

```
class GeoObj {
 public:
  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 the problem of slicing
};
```

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 does not 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 draw() const = 0;      // pure virtual function (introducing the API)
  ...
  virtual ~GeoObj() = default;        // disables move semantics for name
  ...     // other special member functions due to the problem of slicing
};
```

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

Dealing with Slicing

However, there is the problem of slicing. Consider the following code using a reference of the base class `GeoObj` for objects of the derived class `Circle`:

```
Circle c1{...}, c2{...};

GeoObj& geoRef{c1};
geoRef = c2;          // OOPS: uses GeoObj::operator=() and assigns no Circle members
```

Because we call the assignment operator for a `GeoObj` and the operator is not `virtual`, the compiler calls `GeoObj::operator=()`, which does not deal with any member of any derived class. Even declaring the assignment operator with `virtual` would not help, because an operator of the derived class does not override the assignment operator of the base class (the parameter types for the second operand differ).

To avoid this problem, you should disable the use of the assignment operator in polymorphic class hierarchies. Furthermore, if the class is not abstract, you should also avoid having public copy constructors to disable implicit type conversions to the base class. Therefore, 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 draw() const = 0;      // pure virtual function (introducing the API)
    ...
    virtual ~GeoObj() = default;        // would disable move semantics for name
  protected:
    // enable copy and move semantics (callable only for derived classes):
    GeoObj(const GeoObj&) = default;
    GeoObj(GeoObj&&) = default;
    // disable assignment operator (due to the problem of slicing):
    GeoObj& operator= (GeoObj&&) = delete;
    GeoObj& operator= (const GeoObj&) = delete;
};
```

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

4.4.2 Implementing a Polymorphic Derived Class

A polymorphic derived class might look as follows (see *poly/polygon.hpp* for the complete header file):[1]

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

Usually, in a polymorphic derived classes there is no need to declare a special member function. Especially, there is no need to declare a virtual destructor again (unless you have to implement it). Declaring a destructor again (whether or not it is `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 ~Polygon() = default;   // OOPS: don't do that because it disables move semantics
};
```

[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`.

However, without declaring the destructor, move semantics works for both `Polygon` members, `name` and `points`. Consider the following program:

poly/polygon.cpp

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

int main()
{
  Polygon p0{"Poly1", {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();
}
```

This program has the following output:

```
polygon '' over
polygon 'Poly1' over (1,1) (1,9) (9,9) (9,1)
polygon 'Poly1' over (1,1) (1,9) (9,9) (9,1)
```

For both members, `name` and `points`, the values were moved from `p0` to `p2`.

Note that if you have to implement the move constructor in class `Polygon`, you need special care to provide the right `noexcept` condition.

4.5 Summary

- Avoid objects with names.
- Avoid unnecessary `std::move()`. Especially do not use it when returning a local object.
- Constructors that initialize members from parameters, for which move operations are cheap, should take the argument by value and move it to the member.
- Constructors that initialize members from parameters, for which move operations take a significant amount of 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. However, do not take by value and move to update/modify existing values.
- Do not declare a `virtual` destructor in derived classes (unless you have to implement it).

Chapter 5

Overloading on Reference Qualifiers

This chapter discusses overloading member functions for different reference qualifiers. As we will see, this will answer the common community question about 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 had the following alternatives:

- Return by value
- Return by lvalue reference

Let us discuss these alternatives briefly.

5.1.1 Return by Value

A getter returning by value would look like this (remember: do not 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 approach with an approach that returns a reference, you can see that the version that returns the string by value has a performance overhead of a factor between 2 and 100 (provided the names have a significant length so that SSO does not help). Giving access to a member that is an image or a collection of thousands of elements might be even worse. In that case, getters often return by (`const`) reference to improve the performance.

5.1.2 Return by Reference

A getter returning by reference would look as follows:

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

This is faster but somewhat unsafe because the caller has to ensure that the object the returned reference refers to lives long enough. In fact, there is a lifetime risk that you use the return value of the getter longer than the object you call the getter for.

One way of falling 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 that returns a temporary object that we refer to with our getter. However, the range-based `for` loop is defined so that the code above is equivalent to the following:

```
reference range = returnPersonByValue().getName();
// OOPS: returned temporary object 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[1] 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 of what they refer to. However, in this case, `range` does not refer to the `Person` returned by `returnPersonByValue()`; `range` refers to the return value of `getName()`, which is a *reference* to a returned `Person`. Thus, `range` extends the lifetime of the reference but not of the temporary object that the reference refers to. Therefore, with the end of the first statement, the returned temporary object is destroyed and we use a reference to the name of a destroyed object when we iterate over the characters of the name.

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 were to return the name by value. In that case, `range` would extend the lifetime of a copy of the name so that we can use the name until the end of the lifetime of `range`.

5.1.3 Using Move Semantics to Solve the Dilemma

With move semantics, we now have a way to solve the dilemma. We can return by reference if it is safe to do so 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 with different reference qualifiers in the same way as when overloading a function for `&&` and `const&` parameters:

- 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 that we have marked with `std::move()`).

[1] The exact type of the reference is `auto&&` for reasons we discuss later

- 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 a temporary object. Because the temporary objects are about to die, the getter can move out the member `name` as the return value, which means that we do not have to allocate new memory for the return value; we steal the value from the member.

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

`std::move()` for Calling Member Functions

Note that this feature means that it might be worth using `std::move()` even when calling member functions. For example:

```
void foo()
{
  Person p{...};
  ...
  coll.push_back(p.getName());              // calls getName() const&
  ...
  coll.push_back(std::move(p).getName());   // calls getName() &&   (OK, p no longer used)
}
```

Using `std::move()` when we call `getName()` will improve the performance of this program. Instead of returning a reference to a `const std::string`, which we can only copy, the return value is the moved name of `p` returned as a non-`const string` so that `push_back()` can use move semantics to move it into `coll`.

As usual, after this call, `p` is in a *valid but unspecified state*.

For an example where this feature is used in the C++ standard library, see class `std::optional<>`.

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 with 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();               // calls foo() &
  C{}.foo();             // calls foo() &&
  std::move(x).foo();    // calls foo() &&

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

This program demonstrates all reference qualifiers that are possible and when they are called. Usually, we have only two or three of these overloads, such as using `&&` and `const&` (and `&`) for getters.

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

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

5.3 When to Use Reference Qualifiers

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

Although we do have this feature, it is not used as much as it could be. In particular, we could (and should) use it to ensure that operations that modify objects are not called for temporary objects that are about to die.[2]

5.3.1 Reference Qualifiers for Assignment Operators

One example of better use of reference qualifiers would mean modifying the implementation of assignment operators. As proposed in http://wg21.link/n2819, it could be better to declare the assignment operator with reference qualifiers wherever we can.[3]

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

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

For example, the assignment operators 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 enables accidental assignment of a new value to a temporary string:

```
std::string getString();

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

Consider we declare the assignment operators with reference qualifiers instead:

```
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:

```
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 bugs like the following:

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

if (getValue() = 0) {     // OOPS: compiles although = is used instead of ==
  ...
}
```

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

Note that all of these proposals to fix the C++ standard accordingly have so fare been rejected. The main reason was concerns about backward compatibility. However, when implementing your own class, you can use this improvement 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 this 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 used when references to objects are returned. That way, we can reduce the risk of accessing a member of a destroyed temporary object.

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, the following overloads would be better:

```
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. However, you could and should provide these overloads for your types. In that case, do not forget that the implementations for rvalue references should move out expensive members.

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.
- It can make sense to mark objects with `std::move()` even when calling member functions.
- Use reference qualifiers in assignment operators.

This page is intentionally left blank

Chapter 6

Moved-From States

Although move semantics as generated or naively implemented usually works fine, we should at least have a look at the possible case of a move operation bringing objects into a state that is not supported by the C++ standard library or breaks invariants of the type.

In this chapter, we clarify the definition of an "invalid" state 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 has not been called yet and will still be called when the lifetime of the moved-from object ends. Therefore, the destructor has to at least run smoothly.

However, the C++ standard library guarantees more for its moved-from types. Moved-from objects are in a "valid but unspecified state." This means that you can use a moved-from object just like any object of its type where you do not know its value. It is like using a non-`const` reference parameter of the type without having any idea about the value of the passed object. By knowing that we can do more than just destroy moved-from objects, we can, for example, use move semantics to implement sorting and mutating algorithms.

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

- What are the **requirements** for using moved-from objects safely with the C++ standard library?
- What **guarantees** should you 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 any function, the requirements it formulates for the passed types and objects also apply to any moved-from objects passed or used internally.

Basically, you always have to be able to destroy 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 operation). 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`.

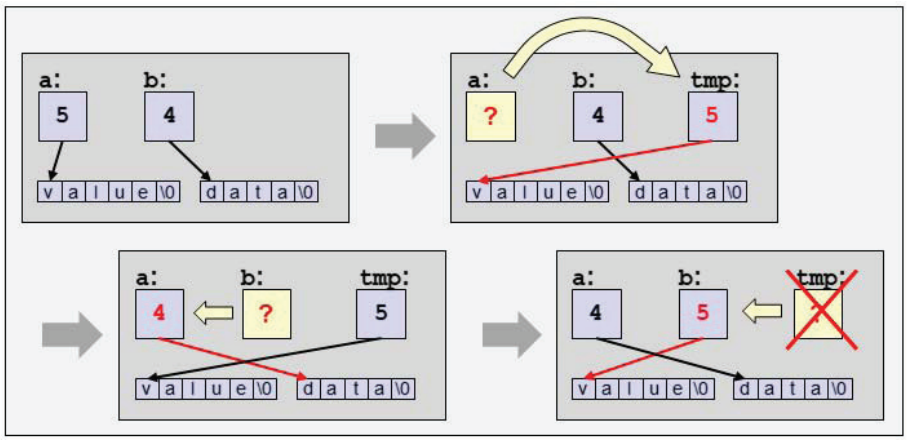


Figure 6.1. Moved-from objects in swap functions

Note that we might also have self-swapping by passing the same object to both parameters. In that case, we might even assign an object with a moved-from state to itself. This all should work (provided a moved-from object still can have any value).

So, for moved-from objects we have the same basic requirements that usually apply to all objects:

- We have to be able to destroy moved-from objects.
- We have to be able to assign a new value to moved-from objects.
- We should be able to copy, move, or assign a moved-from objects to another object.

Moved-from objects should also be able to deal with additional requirements particular operations have. For example, to sort objects we have to support calling `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 does not 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]

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

6.1.2 Guaranteed States of Moved-From Objects

The guarantees for moved-from objects define which code is well-defined when you use them. In general, the designer of a class decides which guarantees are given. However, because we always destroy an object at the end of its lifetime. the minimum guarantee you always have to 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 basic operations such as copying, assigning objects of the same type are often enough. However, users usually expect that you can also deal with all other ways to assign a new value to your object. 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"
s.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 statements 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 that 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 do not 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 use it only 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 that the users of your type are aware of these guarantees. Usually, they expect that your types give the same guarantee as the C++ library, so that you can use a moved-from object just like any other object of the type without knowing its value.

6.1.3 Broken Invariants

The C++ standard library defines what all moved-from objects being in a "valid but unspecified state" means 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 is in a state that means that its invariants are not broken. You can use the object like a non-const reference parameter without knowing 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

Furthermore, 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 place that character at the end of the value (without 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 intended 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 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 needs 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 always need 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. However, we decided to give the full guarantee.

Ideally, a moved-from state that does not support all operations should be detectable. The objects should know this state and provide a member function to check for this state. Moved-from objects might also refuse to execute operations not supported in this state. However, corresponding checks might cost performance in the general case.

In the C++ standard library, some types provide APIs to check whether objects are in a moved-from state. For example, `std::futures` have a member function `valid()` that returns `false` for moved-from objects. But the interfaces to check for moved-from states vary.

Pretty often the moved-from state is the default constructed state, which means that the moved-from state is parts of the invariants anyway. In any case, make sure users of your types know about what is well-defined and what is not.

6.2 Destructible and Assignable

Let us discuss how to ensure that moved-from objects fulfill the basic requirements for supporting 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 work just 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 (that has the unspecified state).

Because we can and should usually expect for the members that moved-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 have usually implemented an assignment operator or a destructor to do more than just assign or destroy 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 one reason why move semantics is disabled by default when a destructor is implemented.

```
    }
  }
};
```

So far, the class does not support move semantics because we have a user-declared destructor. You also cannot copy `Tasks` objects because copying `std::threads` is disabled. However, 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 the following example:

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`. Therefore, 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 that represent a running thread. Therefore, `numThreads` is just copied, which means 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 cannot always avoid this problem, because you may always explicitly have to do something with a subset of the elements an object contains when the object is 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()`)
- Implementing the move operations yourself

- Disabling move semantics (which would be the behavior here without declaring special move member functions)

According to the *Rule of Zero*, you should encapsulate error-prone resource management in a helper type so that application programmers have to implement *zero* special member functions. In this case, you might use a helper class (template) that provides both members (the `std::array` and a member for the actual number of elements used) and a correct implementations of the move operations.

The whole problem is also a side effect of a design mistake of class `std::thread`. The type does not follow the RAII principle. For all `std::threads` with running threads, you have to call `join()` (or `detach()`) before their destructor is called; otherwise the destructor of a thread throws. Since C++20, you could and should use class `std::jthread` instead, which will automatically call `join()` if the object still represents a running thread.

6.3 Dealing with Broken Invariants

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

Fortunately, this is only a problem if move semantics is explicitly requested, because temporary objects are destroyed immediately anyway. However, an explicit request to move might not only be caused by marking an object with `std::move()`. You can create moved-from objects with:

- `std::move()` for an object
- Moving algorithms (`std::move()` and `std::move_backward()`)
- Algorithms that "remove" elements by moving "non-removed" elements to the front (e.g., `std::remove()`, `std::remove_if()`, `std::unique()`)
- Move iterators

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 that does not break the invariants.
- Disable move semantics.
- Relax the invariants that define all possible moved-from states also as valid. In particular, this might mean that member functions and functions that use the objects have to be implemented differently to deal with the new possible states.
- Document and provide a member function to check for the state of "broken invariants" so that users of the type do not use an object of this type after it has been marked with `std::move()` (or only use a limited set of operations).

Let us look at some examples 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 relates to objects where the moved-from state of a member is a problem in itself because the state is valid but should not happen.

Consider 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 that represents a valid card. This would mean that we probably do not 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 {
    return value;
  }
};
```

In this class, the generated special move member functions create an invalid state that breaks 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 do not use `std::move()` or another moving operation 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;
...                                 // initialize deck
Card c{std::move(deck[0])};         // deck[0] has invalid state
deck[0] = Card{"ace-of-hearts"};    // deck[0] is valid again
```

However, printing the value of a moved-from `card` might fail:

```
std::vector<Card> deck;
...                                 // initialize 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-");              // find position of substring (no check)
  std::cout << val.substr(0, pos) << ' '
            << val.substr(pos+4) << '\n';   // OOPS: possible core dump
}
```

[4] Thanks to Tony Van Eerd for this example.

This code might fail at runtime because for a moved-from card, there it is no longer a guarantee that the value contains `"-of-"`. In that case, `find()` initializes `pos` with `std::string::npos`, which throws an exception of type `std::out_of_range` 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 for fixing 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, you can then 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"` OK)? And how do you ensure that objects with the default value perform well without allocating memory?

- Internally allow the new state but disallow calling `getValue()` or other member functions.

  You can 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 and raise an assertion or an exception.

- Extend the invariant by introducing a new state that a `Card` might have no value.

  This means that you have to implement the moving special member functions because you have to ensure that for a moved-from object, the member `value` is in this state.

  Usually, a moved-from state is equivalent to a default-constructed state. Therefore, this is also an opportunity to provide a default constructor. Ideally, you might also provide a member function, checking for this state.

  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-");              // find position of substring
  If (pos != std::string::npos) {           // check whether it exists
    std::cout << val.substr(0, pos) << ' '
              << val.substr(pos+4) << '\n';
  }
```

```
    else {
      std::cout << "no value\n";
    }
  }
```

Alternatively, `getValue()` might return a `std::optional<std::string>` (available since C++17). It seems that there is no obvious perfect solution. You have to think about what each of these fixes mean 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 use.

Note that this class worked fine before C++11, where move semantics was not supported (which might mean the first option is the best one). Thus, C++11 might introduce states for classes that were not possible when the class was implemented. It is a rare case but it does mean that the introduction of move semantics could break existing code.

See class `Email` for another example of a class where we internally mark the moved-from state to handle it separately and make this state visible after a "removing" algorithm that leaves elements in a moved-from state.

6.3.2 Breaking Invariants Due to Moved Consistent Value Members

The second reason for move operations to break invariants relates to objects where two members have to be consistent but 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, it is more often the case 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 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`.

Consider 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();
}
```

This program typically has the following output (do not 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 such as the one for objects of the C++ standard library, which states 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 on 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. For example, if it searches for the first or last digit, you will definitely find one. For a moved-from string, which is typically empty, this is no longer the case. Therefore, code not double-checking whether 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. However, if you follow the rule of the C++ standard library, you should leave your moved-from object in a valid state, which might mean that you introduce a possible state that represents "I do not have 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 in 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 was not supported. The invariant is broken when switching to C++11 or later and moved-from objects are used.

6.3.3 Breaking Invariants Due to Moved Pointer-Like Members

The third reason for move operations breaking invariants relates to objects with members that have pointer-like 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 an int value
    }
};
```

[5] Thanks to Geoffrey Romer and Herb Sutter for providing the idea for this example in an email discussion of the C++ standard library working group.

The objects of this class receive an initial integral value that can be shared with copies of these objects. 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 do not 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 the ownership away from the original object. This means that the moved-from state of a `SharedInt` brings the member `sp` into 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 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 that you 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 that 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 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 generally, 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 in debug mode:

```
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 do not 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 would always 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;
    // special "value" for moved-from objects:
    inline static std::shared_ptr<int> movedFromValue{std::make_shared<int>(0)};

  public:
    explicit SharedInt(int val)
     : sp{std::make_shared<int>(val)} {
    }

    std::string asString() const {
      return std::to_string(*sp);    // OOPS: unconditional deref
    }

    // fix moving special member functions:
    SharedInt (SharedInt&& si)
     : sp{std::move(si.sp)} {
        si.sp = movedFromValue;
    }
    SharedInt& operator= (SharedInt&& si) noexcept {
      if (this != &si) {
        sp = std::move(si.sp);
        si.sp = movedFromValue;
      }
      return *this;
    }

    // enable copying (deleted with user-declared move operations):
    SharedInt (const SharedInt&) = default;
```

[6] Note that `inline static` members are only supported since C++17. Before C++17, you had to define the member in a separate CPP file to respect the *one definition rule* (ODR).

```
    SharedInt& operator= (const SharedInt&) = default;
};
```

Note that we have to `=default` the copying special member functions because these are deleted when we have user-declared special move member functions.

6.4 Summary

- For each class, clarify the state of moved-from objects. You have to ensure that they are at least destructible (which is usually the case without implementing special member functions). However, users of your class might expect/require more.
- The requirements of functions of the C++ standard library also apply to moved-from objects.
- Generated special move member functions might bring moved-from objects into a state such that a class invariant is broken. This might happen especially if:
  - Classes have no default constructor with a determinate value (and therefore no natural moved-from state)
  - Values of members have restrictions (such as assertions)
  - Values of members depend on each other
  - Members with pointer-like semantics are used (pointers, smart pointers, etc.)
- If the moved-from state breaks invariants or invalidates operations, you should fix this by using 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 complete for C++11, we detected a problem: vector reallocations could not 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 us introduce the topic with 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 a copy and a move constructor to make calls of these constructors visible:

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 us create and initialize a vector of Persons and insert a Person later when it exists:

basics/person.cpp

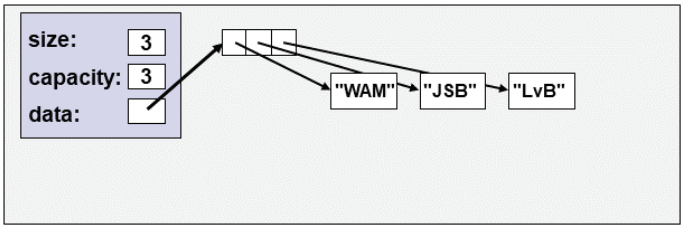
```
#include "person.hpp"
#include <iostream>
#include <vector>

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:

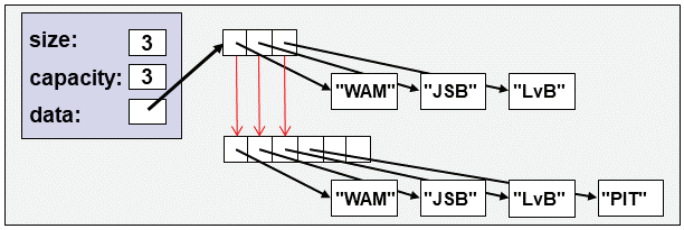
```
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
MOVE Pjotr Iljitsch Tschaikowski
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
```

We first copy the initial values into the vector (because the std::initializer_list<> constructor of a container takes the passed arguments by value). As a result, the vector typically allocates memory for three elements (in the figure, I use shortcuts for the string values):

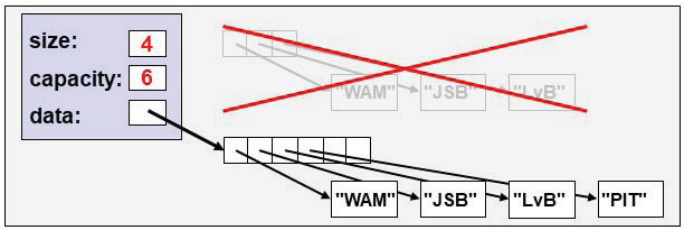


The important thing here is what happens next: We insert a fourth element with `push_back()`, which has the following consequences:

- Because the vector needs more memory internally, it allocates the new memory (for say, six elements), moves in the fourth string (we create a temporary `Person` and move it into the vector with `push_back()`), but also **copies** the existing elements to the new memory:



- At the end of this operation the vector destroys the old elements, releases the old memory for theses elements, and updates its members:



The question is, why does the vector not use the move constructor to move the elements from the old to the new memory?

Strong Exception Safety Guarantee

The reason that vector reallocation does not use move semantics is the *strong exception handling guarantee* we give for `push_back()`: When an exception is thrown in the middle of the reallocation of the vector the C++ standard library guarantees to roll back the vector to its previous state. That is, `push_back()` gives a kind of a transactional guarantee: either it succeeds or it has no effect.

The C++ standard was able to give this guarantee in C++98 and C++03 because there C++ could only copy the elements. If something goes wrong while copying the elements, the source objects are still available. Internal code that handles the exception simply destroys the copies created so far and release the new

memory to bring the vector back to its previous state (the C++ standard library assumes and requires that destructors do not throw; otherwise, it would not be able to roll back).

Reallocation is a perfect place for move semantics because we move elements from one location to the other. Therefore, since C++11, we want to use move semantics here. However, then we are in trouble: if an exception is thrown during the reallocation, we might not be able to roll back. The elements in the new memory have already stolen the values of the elements in the old memory. Therefore, destroying the new elements would not be enough; we have to move them back. But how do we know that moving them back does not fail?

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, this state may need memory, which means that the move might throw if we are out of memory (e.g., node-based containers of Visual C++ are implemented that way).

We also cannot 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.

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`

Therefore, 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 Persons in the same way as before:

basics/personmove.cpp

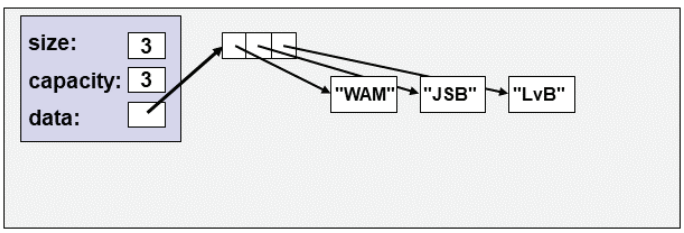
```
#include "personmove.hpp"
#include <iostream>
#include <vector>

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");
}
```

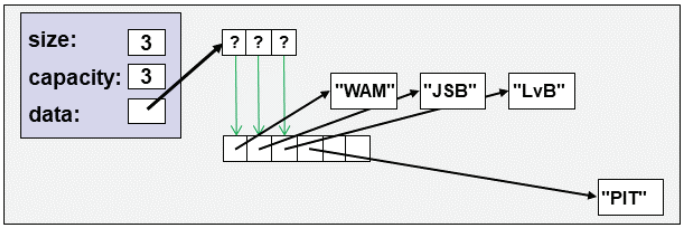
we get the following output:

```
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
```

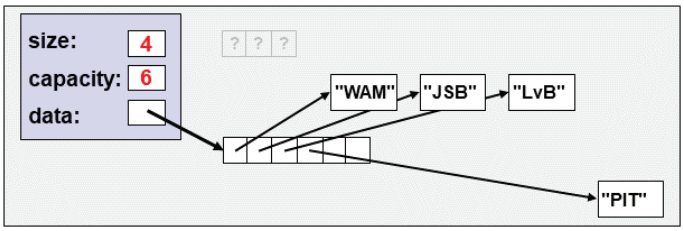
Again, we first copy the initial values into the vector



However, as the last three rows of the output visualize, the vector now uses the move constructor to move its elements to the new reallocated memory:



This means that at the end, the vector 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, at runtime the program will call `std::terminate()`, which usually calls `std::abort()` to signal an abnormal end of the program (and often creates a core dump).

Therefore, we should give the guarantee not to throw if the string member and the output operation do 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_v<std::string>
                  && 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`. In this case, we require two things to give the guarantee:

- With `std::is_nothrow_move_constructible_v<std::string>` (before C++20, you had to use `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 that tells us whether all the corresponding operations to perform the passed expression 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; therefore, 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 do not 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 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;
    ...
};
```

In this case we declare that the default move constructor should be generated:

```
class Person {
    ...
    // force default generated move constructor:
    Person(Person&& p) = default;
    ...
};
```

This means that we print only when we copy. When the generated move constructor is used, we only see that we do not perform a copy.

Now let us use our usual program to print 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 see only 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 do not 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 do not have to implement the move constructor, you do not 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:[1]

```
static_assert(std::is_nothrow_move_constructible_v<Person>);
```

or up to C++17:

```
static_assert(std::is_nothrow_move_constructible<Person>::value, "");
```

[1] For abstract base classes, you have to use a different type trait.

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 adapted 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 that wraps 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 the preprocessor can be replace with nothing or `noexcept` (e.g., compile with `-DNOEXCEPT=noexcept`). We then 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 a significant level of optimization is activated). That is, a reallocation (typically forced by inserting a new element) might take about 20 instead of 200 milliseconds. This means 180 milliseconds less in which we cannot use the vector for anything. This can be a huge benefit.[2]

7.2 Details of `noexcept` Declarations

As seen, `noexcept` was introduced to allow specification of conditional guarantees not to throw. In general, knowing at compile time that a function cannot throw can improve code and optimizations because you do not have to deal with possible clean-ups due to exceptions raised. Note that if the `noexcept` guarantee is violated, the program calls `std::terminate()`, which usually calls `std::abort()` to cause an "abnormal program termination" (e.g., a core dump).

7.2.1 Rules for Declaring Functions with `noexcept`

A couple of rules apply when declaring `noexcept` conditions:

- The `noexcept` condition must be a compile-time expression that yields a value convertible to `bool`.
- You cannot overload functions that have only different `noexcept` conditions.
- In class hierarchies, a `noexcept` condition is part of the specified interface. Overwriting a base class function that is `noexcept` with a function that is not `noexcept` is an error (but not the other way around).

For example:

```
class Base {
 public:
  ...
  virtual void foo(int) noexcept;
  virtual void foo(int);            // ERROR: overload on different noexcept clause only
  virtual void bar(int);
};

class Derived : public Base {
 public:
  ...
```

[2] Note that there is a bug in Visual C++ 2015 that means that move is always used on reallocation and we do not 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).

```
  virtual void foo(int) override;    // ERROR: override giving up the noexcept guarantee
  virtual void bar(int) noexcept;    // OK (here we also guarantee not to throw)
};
```

However, for non-virtual functions, derived-class members can hide base-class members with a different noexcept declaration:

```
class Base {
 public:
  ...
  void foo(int) noexcept;
};

class Derived : public Base {
 public:
  ...
  void foo(int);                     // OK, hiding instead of overriding
};
```

Conditions follow the same rules after being evaluated at compile time. For example, consider the following class hierarchy:

```
class Base {
 public:
  virtual void func() noexcept(sizeof(int) < 8);     // might throw if sizeof(int) >= 8
};

class Derived : public Base {
 public:
  void func() noexcept(sizeof(int) < 4) override;    // might throw if sizeof(int) >= 4
};
```

In this case, we will get a compile-time error if the size of int is 4, because then the base class guarantees not to throw when func() is called, while the derived class no longer gives this guarantee for func(). When the size of int is less than 4, both are noexcept, which is fine. When the size of int is at least 8, both are not noexcept, which is also fine. Therefore, derived classes should only restrict exception guarantees further.

7.2.2 noexcept for Special Member Functions

noexcept conditions may be automatically generated for special member functions.

noexcept for Copying and Moving Special Member Functions

By rule, a noexcept condition is generated when special member functions are generated but not implemented.[3] In that case, the operations guarantee not to throw if the corresponding operations called for all bases classes and non-static members guarantee not to throw.

For example:

basics/specialnoexcept.cpp

```
#include <iostream>
#include <type_traits>

class B
{
  std::string s;
};

int main()
{
  std::cout << std::boolalpha;
  std::cout << std::is_nothrow_default_constructible<B>::value << '\n';
  std::cout << std::is_nothrow_copy_constructible<B>::value << '\n';
  std::cout << std::is_nothrow_move_constructible<B>::value << '\n';
  std::cout << std::is_nothrow_copy_assignable<B>::value << '\n';
  std::cout << std::is_nothrow_move_assignable<B>::value << '\n';
}
```

The output of the program is as follows:

```
true
false
true
false
true
```

The generated copy constructor and copy assignment operator might throw because copying a std::string might throw. However, the generated default constructor, move constructor, and move assignment operator guarantee not to throw because the default constructor, move constructor, and move assignment operator of class std::string guarantee not to throw.[4]

Note that the noexcept condition is even generated when these special member functions are user-declared with =default. Thus, we have the same effect if we declare class B as follows:

[3] This also applies to the default constructor, the "pseudo" special member functions such as a defaulted operator< or operator== (available since C++20), and even to all inherited constructors.

[4] For the move assignment operator of strings, the noexcept guarantee is only given if the strings use the same or an interchangeable allocator.

```
class B
{
  std::string s;
 public:
  B(const B&) = default;                // noexcept condition automatically generated
  B(B&&) = default;                     // noexcept condition automatically generated
  B& operator= (const B&) = default;    // noexcept condition automatically generated
  B& operator= (B&&) = default;         // noexcept condition automatically generated
};
```

When you have a defaulted special member function you can explicitly specify a different noexcept guarantee than the generated one. For example:

```
class C
{
  ...
 public:
  C(const C&) noexcept = default;       // guarantees not to throw (OK since C++20)
  C(C&&) noexcept(false) = default;     // specifies that it might throw (OK since C++20)
  ...
};
```

Before C++20, if the generated and specified noexcept condition contradict, the defined function was deleted.

noexcept for Destructors

By rule, destructors always guarantee not to throw by default. This applies to both generated and implemented destructors.

For example:

```
class B
{
  std::string s;
 public:
  ...
  ~B() {                                // automatically always declared as ~B() noexcept
    ...
  }
};
```

With noexcept(false), you can declare them without this guarantee, but that usually never makes any sense because several guarantees of the C++ standard library are based on the fact that destructors never throw.

7.3 noexcept Declarations in Class Hierarchies

We saw that especially when we have to implement a move constructor, we should declare it with a noexcept guarantee. In general, following the rules of the C++ standard, we should declare it not to throw when all base classes and all member types do not throw on a move assignment.

The general pattern would be as follows:

```
class Base {
  ...
};

class Drv : public Base {
  MemType member;
  ...
  // move constructor:
  Drv(Drv&&) noexcept(std::is_nothrow_move_constructible_v<Base> &&
                      std::is_nothrow_move_constructible_v<MemType>);
};
```

Here, the move constructor of class Drv guarantees not to throw if the base class Base and the member type *MemType* give this guarantee.

The move assignment operator might use the same pattern but note that the move assignment operator should be deleted in polymorphic types anyway, which means that there is usually no need to implement them in derived classes.

7.3.1 Checking for noexcept Move Constructors in Abstract Base Classes

Note that the type trait std::is_nothrow_move_constructible<> does not always work as expected. For abstract base classes, it always yields false because it also checks whether you can create an object of this type with the move constructor, which is not possible for abstract types.

Therefore, a declaration "*I guarantee not to throw if the abstract base class guarantees not to throw*" that works in all cases cannot be formulated using the standard type traits. Usually, you simply (have to) know whether the base class move constructor might throw.

To enable you to check for a class whether its move constructor guarantees not to throw, you can implement the following helper type trait (here, implementing it for C++20).[5] However, note that you have to provide an implementation for each pure virtual function:

poly/isnothrowmovable.hpp

```
// type trait to check whether a base class guarantees not to throw
// in the move constructor (even if the constructor is not callable)
#ifndef IS_NOTHROW_MOVABLE_HPP
#define IS_NOTHROW_MOVABLE_HPP

#include <type_traits>
```

[5] Thanks to Daniel Krügler for providing this solution.

```
template<typename Base>
struct Wrapper : Base {
  using Base::Base;
  // implement all possibly wrapped pure virtual functions:
  void print() const {}
  ...
};

template<typename T>
static constexpr inline bool is_nothrow_movable_v
  = std::is_nothrow_move_constructible_v<Wrapper<T>>;

#endif // IS_NOTHROW_MOVABLE_HPP
```

You can now check even for abstract base classes whether the move constructor is noexcept. The following program demonstrates the different behavior of the standard and the user-defined type trait:

poly/isnothrowmovable.cpp

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

class Base {
  std::string id;
  ...
 public:
  virtual void print() const = 0;   // pure virtual function (forces abstract base class)
  ...
  virtual ~Base() = default;
 protected:
  // protected copy and move semantics (also forces abstract base class):
  Base(const Base&) = default;
  Base(Base&&) = default;
  // disable assignment operator (due to the problem of slicing):
  Base& operator= (Base&&) = delete;
  Base& operator= (const Base&) = delete;
};

int main()
{
  std::cout << std::boolalpha;
  std::cout << "std::is_nothrow_move_constructible_v<Base>: "
            << std::is_nothrow_move_constructible_v<Base> << '\n';
  std::cout << "is_nothrow_movable_v<Base>:                 "
            << is_nothrow_movable_v<Base> << '\n';
}
```

The program has the following output:

```
std::is_nothrow_move_constructible<Base>: false
is_nothrow_movable<Base>:                 true
```

Therefore, if you have to implement the move constructor in a class derived from an abstract base class, you could use such a helper type trait to declare the move constructor as follows:

```
class Drv : public Base {
  MemType member;
  ...
  // move constructor:
  Drv(Drv&&) noexcept(is_nothrow_movable_v<Base> &&
                      is_nothrow_movable_v<MemType>);
};
```

To not having the need to implement all pure virtual functions, a compiler supported type trait is missing in the C++ standard.

7.4 When and Where to Use noexcept

As seen, noexcept was introduced in C++11 as a response to the problem that we cannot use move semantics when reallocating elements in a vector.

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 then call other functions where we also need the noexcept guarantee), but we might also help the compiler to optimize code because no code has to be generated to handle exceptions. 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). Therefore, 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 that has 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 should use the implicitly supplied (non-throwing) exception specification.
- Library functions designed for compatibility with C code may be marked as unconditionally noexcept.

The following examples demonstrate what the first item means:

- 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` even though when used correctly 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.

The second item is the guideline you should follow when implementing move semantics. We propose that you use a conditional `noexcept` declaration only 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 many implementation details.

With `http://wg21.link/p0884`, we added that wrapping types should also 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, any destructor is always implicitly declared as `noexcept`.

The consequences for your code are as follows:

- You should implement the move constructor, the move assignment operator, and a `swap()` function with a (conditional) `noexcept`.
- For all other functions, mark them with an unconditional `noexcept` only if you know that they cannot throw.
- Destructors do not need a `noexcept` specification.

7.5 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 might just want to mark them with an unconditional `noexcept` if they never throw.
- Destructors are always declared with `noexcept` (even when implemented).

This page is intentionally left blank

Chapter 8

Value Categories

This chapter introduces the formal terminology and rules for move semantics. We formally introduce value categories such as lvalue, rvalue, prvalue, and xvalue and discuss their role when binding references to objects. This allows us to also discuss details of the rule that move semantics is not automatically passed through, as well as a very subtle behavior of `decltype` when it is called for expressions.

This chapter is the most complicated chapter of the book. You will probably see facts and features that are tricky and for some people hard to believe. Come back to it later, whenever you read about value categories, binding references to objects, and `decltype` again.

8.1 Value Categories

To compile an expression or statement it does not only matter whether the involved types fit. For example, you cannot assign an `int` to an `int` when on the left-hand side of the assignment an `int` literal is used:

```
int i = 42;

i = 77;     // OK
77 = i;     // ERROR
```

For this reason, each expression in a C++ program has a *value category*. Besides the type, the value category is essential to decide what you can do with an expression.

However, value categories have changed over time in C++.

8.1.1 History of Value Categories

Historically (taken from Kernighan&Ritchie C, *K&R* C), we had only the value categories *lvalue* and *rvalue*. The terms came from what was allowed in an assignment:

- An ***lvalue*** could occur on the ***l****eft-hand* side of an assignment
- An ***rvalue*** could occur only on the ***r****ight-hand* side of an assignment

According to this definition, when you use an `int` object/variable you use an lvalue, but when you use an `int` literal you use an rvalue:

```
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-hand side of an assignment
```

However, these categories were important not only for assignments. They were used generally to specify whether and where an expression can be used. For example:

```
int x;              // x is an lvalue when used in an expression

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 became more complicated with ANSI-C because an `x` declared as `const int` could not stand on the left-hand side of an assignment but could still be used in several other places where only an lvalue could be used:

```
const int c = 42;       // Is c an lvalue or rvalue?

c = 42;                 // now an ERROR (so that c should no longer be an lvalue)
const int* p1 = &c;     // still OK (so that c should still be an lvalue)
```

The decision in C was that `c` declared as `const int` 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 could not do anymore was to have a `const` object on the left-hand side of an assignment.

As a consequence, in ANSI-C, the meaning of the ***l*** changed to ***l****ocator value*. An ***lvalue*** is now an object that has a specified location in the program (so that you can take the address, for example). In the same way, an ***rvalue*** can now be considered just a ***r****eadable value*.

C++98 adopted these definitions of value categories. However, with the introduction of move semantics, the question arose as to which value category an object marked with `std::move()` should have, because objects of a class marked with `std::move()` should follow the following rules:

```
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
```

With the exception of fundamental data types, an object marked with `std::move()` should still behave like an lvalue by allowing you to modify its value. On the other hand, there are restrictions such as that you should not be able to take the address.

A new category *xvalue* ("eXpiring value") was therefore introduced to specify the rules for objects explicitly marked as *I no longer need the value here* (mainly objects marked with `std::move()`). However, most of the rules for former rvalues also apply to xvalues. Therefore, the former primary value category *rvalue* became a composite value category that now represents both new primary value categories *prvalue* (for everything that was an rvalue before) and *xvalue*. 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.

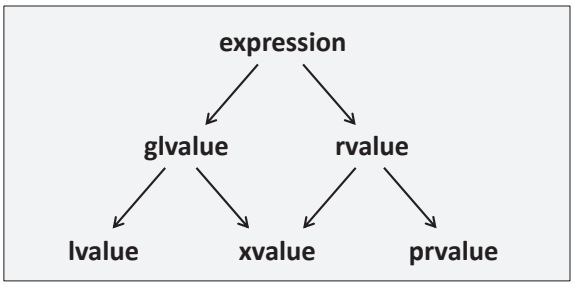


Figure 8.1. Value categories since C++11

We have the following primary categories:

- ***lvalue*** ("locator value")
- ***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 a plain value member of an rvalue)
- An expression that is just a string literal (e.g., `"hello"`)
- The return value of a function if it is declared to return an lvalue reference (return type *Type*`&`)
- Any reference to a function, even when marked with `std::move()` (see below)
- 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 built-in literal that is not a string literal (e.g., `42`, `true`, or `nullptr`)
- The return type of a function if it is declared to return 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 object with `std::move()`
- A cast to an rvalue reference of an object type (not a function type)
- The returned value of a function if it is declared to return an rvalue reference (return type *Type*`&&`)
- A non-`static` value member of an rvalue (see below)

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*.
- All objects marked with `std::move()` and their value 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 in 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 that 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 an 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.

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 a (long-living) object whose resources/values are no longer needed.

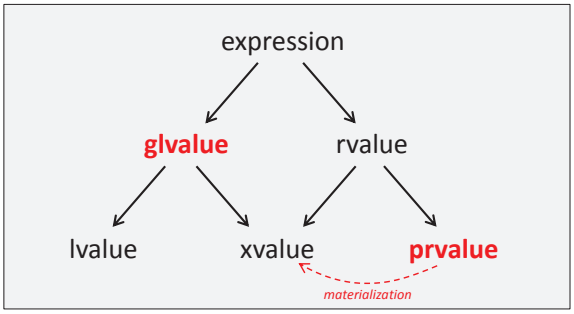


Figure 8.2. Value categories since C++17

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 was not possible without either copy or move support. However, since C++17, we can pass prvalues around by value as long as we do not 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 is used 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* that designates the temporary object. Therefore, 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 an 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 that designates 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 Special Rules for Value Categories

We have special rules for the value category of functions and members that have an impact on move semantics.

8.2.1 Value Category of Functions

A special rule in the C++ standard states that all expressions that are references to functions are lvalues.[1] For example:

```
void f(int) {
}

void(&fref1)(int) = f;     // fref1 is an lvalue
void(&&fref2)(int) = f;    // fref2 is also an lvalue

auto& ar = std::move(f);   // OK: ar is lvalue of type void(&)(int)
```

In contrast to types of objects, we can bind a non-`const` lvalue reference to a function marked with `std::move()` because a function marked with `std::move()` is still an lvalue.

8.2.2 Value Category of Data Members

If you use data members of objects (e.g., when using members `first` and `second` of a `std::pair<>`), special rules apply.

In general, the value categories of data members are as follows:

- Data members of lvalues are lvalues.
- Reference and `static` data members of rvalues are lvalues.
- Plain data members of rvalues are xvalues.

[1] Thanks to Daniel Krügler for pointing this out.

This rule reflects that reference or `static` members are not really part of an object. If you no longer need the value of an object, this applies also to the plain data members of the object. However, the values of members that refer to somewhere else or are `static` might still be used by other objects.

For example:

```
std::pair<std::string, std::string&> foo();   // note: member second is reference

std::vector<std::string> coll;
...
coll.push_back(foo().first);                  // moves because first is an xvalue here
coll.push_back(foo().second);                 // copies because second is an lvalue here
```

You need `std::move()` to move the member `second` here:

```
coll.push_back(std::move(foo().second));      // moves
```

If you have an lvalue (an object with a name), you have two options to mark a member with `std::move()`:

- **`std::move(`***obj***`)`**`.`***member*
- **`std::move(`***obj*`.`*member*`)`**

Because `std::move()` means "*I no longer need this value here*" it looks like you should mark the *obj* if you no longer need the value of the object and mark *member* if you no longer need the value of the member. However, the situation is a bit more complicated.

`std::move()` for Plain Data Members

If the member is neither `static` nor a reference, by rule, `std::move()` always converts members to xvalues so that move semantics can be used.

Consider we have declared the following:

```
std::vector<std::string> coll;
std::pair<std::string, std::string> sp;
```

The following code moves first the member `first` and then the member `second` into `coll`:

```
sp = ... ;
coll.push_back(std::move(sp.first));      // move string first into coll
coll.push_back(std::move(sp.second));     // move string second into coll
```

However, the following code has the same effect:

```
sp = ... ;
coll.push_back(std::move(sp).first);      // move string first into coll
coll.push_back(std::move(sp).second);     // move string second into coll
```

It looks a bit strange that we still use *obj* after marking it with `std::move()`, but in this case we know which part of the object may be moved so that we can still use a different part. Therefore, I would prefer to mark the member with `std::move()` when I, for example, have to implement a move constructor.

`std::move()` for Reference or `static` Members

If the members are references or `static`, different rules apply:[2] A reference or `static` member of an rvalue is still an lvalue. Again, this rule reflects that the value of such a member is not really part of the object. Saying "*I no longer need the value of the object*" should not imply "*I no longer need the value (of a member) that is not part of the object*."

Therefore, it makes a difference how you use `std::move()` if you have reference or `static` members:

- Using `std::move()` for the ***object*** has no effect:

```
struct S {
  static std::string statString;   // static member
  std::string& refString;          // reference member
};

S obj;
...
coll.push_back(std::move(obj).statString);   // copies statString
coll.push_back(std::move(obj).refString);    // copies refString
```

- Using `std::move()` for the ***members*** has the usual effect:

```
struct S {
  static std::string statString;
  std::string& refString;
};
S obj;
...
coll.push_back(std::move(obj.statString));   // moves statString
coll.push_back(std::move(obj.refString));    // moves refString
```

  Whether such a move is useful is a different question. Stealing the value of a static member or referenced member means that you modify a value outside the object you use. It might make sense, but it could also be surprising and dangerous. Usually, a type `S` should better protect access to theses members.

In generic code, you might not know whether members are static or references. Therefore, using the approach to mark the object with `std::move()` is less dangerous, even though it looks weird:

```
coll.push_back(std::move(obj).mem1);   // move value, copy reference/static
coll.push_back(std::move(obj).mem2);   // move value, copy reference/static
```

In the same way, `std::forward<>()`, which we introduce later, can be used to perfectly forward members of objects. See *basics/members.cpp* for a complete example.

[2] Thanks to Andrey Upadyshev for pointing this out.

8.3 Impact of Value Categories When Binding References

Value categories play an important role when we bind references to objects. For example, in C++98/C++03, they define that you can assign or pass an rvalue (a temporary object without a 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
```

8.3.1 Overload Resolution with Rvalue References

Let us 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 the 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 usually about to die)
void f(const X&&);   // no clear semantic meaning
```

The numbers list the priority for overload resolution so that you can see which function is called when multiple overloads are provided. The smaller the number, the higher the priority (priority 1 means that this is tried first).

Note that you can only pass rvalues (prvalues, such as temporary objects without a name) or xvalues (objects marked with `std::move()`) to rvalue references. That is where their name comes from.

You can usually ignore the last column of the table because `const` rvalue references do not make much sense semantically, meaning that we get the following rules:

Call	f(X&)	f(const X&)	f(X&&)	f(const X&&)
f(v)	1	2	no	no
f(c)	no	1	no	no
f(X{})	no	3	1	2
f(move(v))	no	3	1	2
f(move(c))	no	2	no	1

Table 8.1. Rules for binding references

- A non-`const` lvalue reference takes only non-`const` lvalues.
- An rvalue reference takes only 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 introduce universal/forwarding references.

There we will also learn that sometimes, you can pass an lvalue to an rvalue reference (when a template parameter is used). Be aware that not every declaration with `&&` follows the same rules. The rules here apply if we have a ***type*** (or type alias) declared with `&&`.

8.3.2 Overloading by Reference and Value

We can declare functions by both reference and value parameters: For example:

```
void f(X);          // call-by-value
void f(const X&);   // call-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 call-by-value and call-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.

Therefore, you should usually only take an argument either by value or by reference (with as many reference overloads as you think are useful) but never both.

8.4 When Lvalues become Rvalues

As we have 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 seems to work. 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. Therefore, passing them to an rvalue reference does not compile. However, there is a hidden operation involved, because the type of the argument (array of six 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 that does not have a name.

Therefore, 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
```

8.5 When Rvalues become Lvalues

Let us now look at the implementation of a function that declares the parameter as an rvalue reference:

```
void rvFunc(std::string&& str) {
  ...
}
```

As we have 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 are dealing with an object that has a name. This means that we use `str` as an lvalue. We can do only what we are allowed to do with an lvalue.

This means that we cannot directly call our own function recursively:

```
void rvFunc(std::string&& str) {
  rvFunc(str);                   // ERROR: passing an lvalue to an rvalue reference
}
```

We have to to mark `str` 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** that we have already discussed. Again, note that this is a feature, not a bug. If we passed move semantics through, we would not be able to use an object that was passed with move semantics twice, because the first time we use it it would lose its value. Alternatively, we would need a feature that temporarily disables 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 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, with a cast to an rvalue reference 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 an 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 is just how we define the rules of move semantics and value categories in the C++ standard. Take it as it is. Fortunately, compilers 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.

8.6 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. However, it can also be used to determine the value category of an expression.

8.6.1 Using `decltype` to Check the Type of Names

In a function that takes 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 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).

8.6.2 Using `decltype` to Check the Value Category

So far, we have passed only 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` also yields the value category according to the following conventions:

- For a **prvalue** it just yields its value type: *type*
- For an **lvalue** it yields its type as an lvalue reference: *type*`&`
- For an **xvalue** it yields its type as an 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 placed inside parentheses, which is an expression and no longer just a name, `decltype` yields its type and its 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 this with the former implementation of this function not using additional parentheses. Here, `decltype` of `(str)` yields `std::string&` because `str` is an lvalue of type `std::string`.

The fact that for `decltype`, it makes a difference when we put additional parentheses around a passed name, will also have significant consequences when we later discuss `decltype(auto)`.

Check for a Value Category Inside Code

In general, you can now check for a specific value category inside code as follows:

- `!std::is_reference_v<decltype((`*expr*`))>`
  checks whether *expr* is a **prvalue**.
- `std::is_lvalue_reference_v<decltype((`*expr*`))>`
  checks whether *expr* is an **lvalue**.
- `std::is_rvalue_reference_v<decltype((`*expr*`))>`
  checks whether *expr* is an **xvalue**.
- `!std::is_lvalue_reference_v<decltype((`*expr*`))>`
  checks whether *expr* is an **rvalue**.

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*.

Before C++20, you have to skip the suffix `_v` and append `::value` instead.

8.7 Summary

- Any expression in a C++ program belongs to 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()`)
- Whether a call or operation in C++ is valid depends on both the type and the value category.
- Rvalue references of types can only bind to rvalues (prvalues or xvalues).
- 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.
- Functions and references to functions are always lvalues.
- For rvalues (temporary objects or objects marked with `std::move()`), plain value members have move semantics but reference or `static` members have not.
- `decltype` can either check for the declared type of a passed name or for the type and the value category of a passed expression.

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. In particular, we discuss *universal references* (also called *forwarding references*) and (perfect) forwarding.

The following chapters then discuss tricky details of universal references and perfect forwarding 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 have already learned 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 forward its move semantics to another function.

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 these functions:

```
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 that we want to call `foo()` for the same arguments indirectly via a helper function `callFoo()`. That helper function would also need 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 an 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 (that bind to objects that do not have move semantics) are just passed as they are.
- Arguments declared as rvalue references (that bind to objects that have move semantics) are passed with `std::move()`.

This allows us to forward move semantics perfectly: for any argument that is passed with move semantics we keep move semantics; but we do not add move semantics when we get an argument that does not have it.

Only with this implementation is the use of `callFoo()` to call `foo()` 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, which means that we need `std::move()` to pass it as an rvalue again. However, we cannot 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.

For *perfect forwarding* in generic code, we would always need all these overloads for each parameter. To support all combinations, this means having 9 overloads for 2 generic arguments and 27 overloads for 3 generic arguments.

Therefore, C++11 introduced a special way to *perfectly forward* given arguments without any overloads but still keeping the type and the 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 is usually fine but there might be cases where 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).

With this, generic code would need even 16 and 64 overloads for two or three parameters, unless the language provides a special feature for perfect forwarding.

9.2 Implementing Perfect Forwarding

To avoid overloading functions for parameters with different value categories, C++ introduced a special mechanism for *perfect forwarding*. You need three things:

1. Take the call parameter as a pure rvalue reference (declared with `&&` but without `const` or `volatile`).
2. The type of the parameter has to be a template parameter of the function.
3. When forwarding the parameter to another function, use a helper function called `std::forward<>()`, which is declared in `<utility>`.

You have to implement a function that perfectly forwards an argument as follows:

```
template<typename T>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));   // equivalent to foo(std::move(arg)) for passed rvalues
}
```

`std::forward<>()` is effectively a conditional `std::move()` so that we get the same behavior as the three (or four) overloads of `callFoo()` above:

- If we pass an rvalue to `arg`, we have the same effect as calling `foo(std::move(arg))`
- If we pass an lvalue to `arg`, we have the same effect as calling `foo(arg)`

In 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));
}
```

We can also apply `std::forward<>()` 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 do not call `forward<>()` once for all arguments; we call it for each argument individually. Therefore, we have to place the ellipsis ("`...`") behind the end of the `forward()` expression instead of directly behind `args`.

However, what exactly is happening here is pretty tricky and needs a 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. However, that is not the case. An **rvalue reference** (not qualified with `const` or `volatile`) **of a function template parameter** does not follow the rules of ordinary rvalue references. It is a different thing.

Two Terms: Universal and Forwarding Reference

Such a reference is called a ***universal reference***. Unfortunately, there is also another term for it that is mainly used in the C++ standard: ***forwarding reference***. There is no difference between these two terms, it is just that we have a historical mess here with two established terms meaning the same thing.

Both terms describe basic aspects of universal/forwarding references:

- They can universally bind to objects of all types (`const` and non-`const`) and value categories.
- They are usually used to forward arguments; but note that this is not the only use (one reason for me to prefer the term *universal reference*).

Universal References Bind To All Value Categories

The important feature of universal references is that they can bind to objects and expressions 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 (whether we have an rvalue or an lvalue) 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 a generic rvalue reference that is qualified with `const` (or `volatile`) is ***not*** a universal reference. You can only pass rvalues:

```
template<typename T>
void callFoo(const T&& arg);   // arg is not a universal/forwarding reference

const X c;
callFoo(c);                    // ERROR: c is not an rvalue
callFoo(std::move(c));         // OK, arg is const X&&
```

Note also that I did not talk about the type of `T` yet. I will explain later what type is deduced as `T` with universal references.

Later, we discuss the corresponding example using lambdas.

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
}
```

Like `std::move()`, `std::forward<>()` is also defined as a function template in the header file `<utility>`.

The expression `std::forward<T>(arg)` is essentially implemented as follows:

- If an rvalue of type `T` is passed to the function, the expression is equivalent to `std::move(arg)`.
- If an lvalue of type `T` is passed to the function, the expression is equivalent to `arg`.

That is, `std::forward<>()` is a `std::move()` only for passed rvalues.

Just like for `std::move()`, the semantic meaning of `std::forward<>()` is *I no longer need this value here*, with the additional benefit that we preserve the type (including constness) and the value category of the object the passed universal reference binds to. You can argue that you only say conditionally that you no longer need the value, but because you do not know whether `std::forward<>()` becomes a `std::move()`, it would be an error to assume that the object has its value afterwards. So again, all you should assume is that after using `std::forward<>()`, the object is usually valid but you do not know its value.

`std::forward<>()` for Calling Member Functions

Note that you might even use `std::forward<>()` for universal references when calling member functions. Remember that a member function might have specific overloads for move semantics using reference qualifiers. In that case, `std::forward<>()` can be used to call the member function if you no longer need the value of the object afterwards.

For example, assume we have overloaded getters to improve the performance when returning the name of a temporary person:

```
class Person
{
 private:
  std::string name;
 public:
  ...
  void print() const {
    std::cout << "print()\n";
  }

  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
  }
};
```

In a function that takes a universal reference, we might then benefit from using `std::forward<>()` as follows:

```
template<typename T>
void foo(T&& x)
{
  x.print();                        // OK, no need to forward the passed value category

  x.getName();                      // calls getName() const&
  std::forward<T>(x).getName();     // calls getName() && for rvalues (OK, no longer need x)
}
```

After using `std::forward<>()`, `x` is in a *valid but unspecified state*. Whenever you use `std::forward<>()`, make sure you do no longer use (the value of) the object.

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 foo(arg), so it calls foo(X&)
callFoo(c);                 // OK, expands to foo(arg), so it calls foo(const X&)
callFoo(X{});               // OK, expands to foo(std::move(arg)), so it calls foo(X&&)
callFoo(std::move(v));      // OK, expands to foo(std::move(arg)), so it calls foo(X&&)
callFoo(std::move(c));      // OK, expands to foo(std::move(arg)), so it calls foo(const X&)
```

Whatever we pass to `callFoo()` becomes an lvalue (because the parameter `arg` is an object with a name). However, the type of `arg` depends on what we pass:

- If we pass an lvalue, `arg` is an lvalue reference (`X&`, when we pass a non-`const` `X`, or `const X&`, when we pass a `const X`).
- If we pass an rvalue (an unnamed temporary object or 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 real type name or the name of 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 us discuss the differences in detail.

[1] We will discuss the drawbacks of having the same syntax and why no different syntax was introduced later.

9.3.1 Rvalue References of Actual Types

If we have an ordinary rvalue reference to a name that is not a template parameter of the function called (or if the reference is declare with `const` or `volatile`), 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() (no need to 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
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 does not 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 here because it is equivalent to `std::move()` when called for rvalue references).

9.3.2 Rvalue References of Function Template Parameters

If we have a non-`const`/`volatile` rvalue reference to a function template parameter, we can pass objects of *all* value categories. The passed argument may or may not be `const`:

```
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)
}

std::vector<std::string> v;
const std::vector<std::string> 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()`:

- The type of `arg` may or may not be `const` now.
- In this case, it usually does not 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 pass all arguments to `foo()` with move semantics, passing the non-`const` lvalue `v` as an xvalue).

9.4 Overload Resolution with Universal References

We have already 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`, a non-`const` variable `v` thereof, and a `const` variable `c` thereof:

```
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 (usually long-living object)
void f(X&&);          // can steal value (object usually about to die)
void f(const X&&);    // contradicting 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	no	no	2
`f(c)`	no	1	no	no	2
`f(X{})`	no	4	1	3	2
`f(move(v))`	no	4	1	3	2
`f(move(c))`	no	3	no	1	2

Table 9.1. Rules for binding all references

Again, the numbers list the priority for overload resolution (the smallest numbers have the highest priority) 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 than just instantiating the function template for the exact type.

9.4.1 Fixing Overload Resolution with Universal References

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 that takes 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

Therefore, you should be very careful when implementing a constructor that has 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
```

Therefore, it is better to avoid implementing generic constructors that declare the first parameter as a universal reference and can be called for one argument of an arbitrary type.

The other option is to constrain the constructor in a way that it is disabled if the passed is (convertible to) the type of the class. The effect is that the copy or move constructor has to be used instead. 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 needed 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";
  }
};
```

9.5 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 usually declare the universal references with `auto&&` as explained in the chapter about `auto&&`.

However, since C++20, you can also use template parameters in lambdas:[2]

```
auto callFoo = []<typename T>(T&& arg) {  // OK, universal reference since C++20
  foo(std::forward<T>(arg));              // perfectly forward arg
};
```

9.6 Summary

- Declarations with two ampersands (*name*`&&`) can be two different things:
  - If *name* is not a function template parameter it is an ordinary rvalue reference, binding only to rvalues.
  - If *name* is a function template parameter it is a *universal reference*, binding to all value categories.
- A *universal reference* (called *forwarding reference* in the C++ standard) is a reference that can universally 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 its parameter is an rvalue.
- It might make sense to mark objects with `std::forward<>()` even when calling member functions.
- Universal references are the second-best option of all overload resolutions.
- Avoid implementing generic constructors for one universal reference (or constrain them for specific types).

[2] Some compilers support this before C++20 in non-pedantic mode.

Chapter 10

Tricky Details of Perfect Forwarding

After introducing (perfect) forwarding and universal/forwarding references in general in the previous chapter, this chapter discusses tricky details of perfect forwarding and universal/forwarding references, such as:

- Non-forwarding universal references
- When exactly we have a universal reference
- Type deduction for universal references

The following chapters then discuss 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 (also called *forwarding references*) that have nothing to do with forwarding because there can be other benefits of being able to bind to all objects while still knowing the value category and/or whether the object is `const`.

In this chapter, we discuss some initial examples. After introducing a second kind of universal reference, `auto&&`, we then 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 objects of any value category and still preserve whether or not it is `const`. The only other reference that binds to all objects, `const&`, loses the information about whether the passed argument is `const` or not.

Forwarding Constness

This means that if we want to avoid overloading but want to have different behavior for `const` and non-`const` arguments and support all value categories, we have to use universal references.

Consider the following example:

generic/universalconst.cpp

```
#include <iostream>
#include <string>

void iterate(std::string::iterator beg, std::string::iterator end)
{
  std::cout << "do some non-const stuff with the passed range\n";
}

void iterate(std::string::const_iterator beg, std::string::const_iterator end)
{
  std::cout << "do some const stuff with the passed range\n";
}

template<typename T>
void process(T&& coll)
{
  iterate(coll.begin(), coll.end());
}

int main()
{
  std::string v{"v"};
  const std::string c{"c"};

  process(v);                 // coll binds to a non-const lvalue, iterators passed
  process(c);                 // coll binds to a const lvalue, const_iterators passed
  process(std::string{"t"}); // coll binds to a non-const prvalue, iterators passed
  process(std::move(v));      // coll binds to a non-const xvalue, iterators passed
  process(std::move(c));      // coll binds to a const xvalue , const_iterators passed
}
```

The program has the following output:

```
do some non-const stuff with the passed range
do some const stuff with the passed range
do some non-const stuff with the passed range
do some non-const stuff with the passed range
do some const stuff with the passed range
```

In the program, we declare `process()` with a parameter as a universal reference to a collection (container, string, etc.). As usual for universal references, we can pass strings of all possible value categories.

Inside `process()`, we can still have different behavior depending on whether or not the argument is `const`. In this case, we call `begin()` and `end()` for the passed collection to pass them as a half-open range to a function that iterates over all elements:

```
template<typename T>
void process(T&& coll)
{
  iterate(coll.begin(), coll.end());
}
```

However, we have called different implementations of the function `iterate()`: one for `iterators` (able to modify the elements), and one for `const_iterators` (not able to modify the elements). Because universal references preserve whether a `const` object was passed, we call the `iterate()` that matches the constness of the passed collection.

Note that we do not use (perfect) forwarding inside `process()`. We just want to refer universally to both `const` and non-`const` objects. We might even use `coll` after iterating over all elements.

You might argue that we should know whether we modify elements when iterating over them. However, assume that this generic function also allows us to pass the operation we call for each element. It might read or modify the elements and for that, `const` correctness is important.

Note that using `std::forward<>()` here is at least questionable:

```
template<typename T>
void process(T&& coll)
{
  iterate(std::forward<T>(coll).begin(), std::forward<T>(coll).end());  // ???
}
```

Claiming at two different locations that the same object is not longer needed is a source of trouble because then both locations might interpret this as reason to steal the value from the object. Therefore, the location that steals last might not use the right value. You should never use `std::move()` or `std::forward<>()` twice for the same object (unless the object was reinitialized before the second use).

Using `std::forward<>()` only once here is also a source of trouble because we have no guaranteed evaluation order for arguments of function calls:

```
template<typename T>
void process(T&& coll)
{
  iterate(coll.begin(), std::forward<T>(coll).end());   // ???
}
```

In this particular example, using `std::forward<>()` once or twice might work because `begin()` and `end()` do not steal/modify the value from the passed object. However, in general, it is an error unless you know exactly how this information is used.

10.1.2 Universal References in Detail

The previous example demonstrates that declaring an argument as a universal reference has more use than just perfectly forwarding it to another function. Let us analyze the mechanisms a little further.

Consider the following declaration:

```
template<typename T>
void foo(T&& arg) {
  ...
}
```

When we declare `arg` we have a reference that can bind universally to all types and value categories. For a non-`const` object `v` and a `const` object `c` the type `T` and the type of `arg` is deduced as follows:

	`T`	**`arg`**
`foo(`**`v`**`)`	*Type*`&`	*Type*`&`
`foo(`**`c`**`)`	`const` *Type*`&`	`const` *Type*`&`
`foo(`*Type*`{})`	*Type*	*Type*`&&`
`foo(`**`move(v)`**`)`	*Type*	*Type*`&&`
`foo(`**`move(c)`**`)`	`const` *Type*	`const` *Type*`&&`

That means that the information about the passed argument is split as follows:

- `arg` still knows the value and its type including whether or not it is `const`. It is an lvalue reference if an lvalue was passed and an rvalue reference if an rvalue was passed. does also know whether an lvalue or an rvalue was passed.
- `T` knows the type and has some information about the value category of the passed argument (whether an lvalue or rvalue was passed). According to a specific template type deduction rule, if an lvalue was passed, `T` is an lvalue reference; otherwise, `T` is not a reference.

Calling `std::forward<T>(arg)` brings all information together again to restore both constness and value category to perfectly forward the passed argument with its current value. However, if you do not need the value category of the passed argument for perfect forwarding, you do not need `std::forward<>()`.

Constness Dependent Code

If you only have to know whether the passed argument was `const`, you can use both `arg` and `T`. For example:

```
template<typename T>
void foo(T&& arg)
{
  if constexpr(std::is_const_v<std::remove_reference_t<T>>) {
    ...     // passed argument is const
  }
  else {
    ...     // passed argument is not const
  }
}
```

Here, we use `if constexpr` (a compile-time `if`, introduced with C++17) to do different things depending on whether the passed argument is `const`.

Note that it is important to use `std::remove_reference<>` to remove the referenceness of `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 do not know yet:

- `std::remove_reference_t<T>` (available since C++14) is a shortcut for `typename std::remove_reference<T>::type`
- `std::is_const_v<T>` (available since C++17) is a shortcut for `std::is_const<T>::value`.

Value Category Dependent Code

By using `T` we can get/check the information about the passed value category (at least whether an lvalue or an rvalue was passed). For example:

```
template<typename T>
void foo(T&& arg)
{
  if constexpr(std::is_lvalue_reference_t<T>) {
    ...    // passed argument is lvalue (has no move semantics)
  }
  else {
    ...    // passed argument is rvalue (has move semantics)
  }
}
```

Sometimes checks like this are necessary (e.g., to deal with sub-objects differently depending on whether an lvalue or rvalue was passed).

10.1.3 Universal References of Specific Types

The fact that ordinary rvalue references and universal references share the same syntax creates multiple problems. It is not only that it is not easy to find out what we have; there is also the problem that you cannot declare a universal reference of a specific type.

For example, consider a function declared to take universal references:

```
template<typename T>
void processString(T&& arg);
```

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 constrain this function to only take strings (both `const` and non-`const` without losing this information), we cannot do that easily.

Since C++20, constraining a universal reference to specific types is possible with the keyword `requires`. However, you have to decide whether and which kind of type conversions you support:

- When the type must match (no implicit conversions allowed):

```
template<typename T>
requires std::is_same_v<std::remove_cvref_t<T>, std::string>
void processString(T&&) {
  ...
}
```

- When implicit conversions should be allowed:

```
template<typename T>
requires std::is_convertible_v<T, std::string>
void processString(T&&) {
  ...
}
```

- When even explicit conversions should be allowed:

```
template<typename T>
requires std::is_constructible_v<std::string, T>
void processString(T&&) {
  ...
}
```

Usually, `std::is_convertible` is what you want because it fits with the usual rules of function calls.

Note that `std::is_convertible` and `std::is_constructible` have the opposite order for source and destination type of the conversion.

See *generic/universaltype.cpp* for a complete example with all cases.

Before C++20, you need the `enable_if<>` type trait instead of `requires` and the shortcuts with the `_v` and `_t` suffix for type traits might not have been supported. For example, the following code supports all types that are implicitly convertible to `std::string` since C++11:

```
template<typename T,
         typename = typename std::enable_if<
                                  std::is_convertible<T, std::string>::value
                                           >::type>
void processString(T&& args);
```

To restrict to type `std::string` in C++11, we need:

```
template<typename T,
         typename = typename std::enable_if<
                                  std::is_same<typename std::decay<T>::type,
                                               std::string
                                              >::value
                                           >::type>
void processString(T&& arg);
```

Here, the type trait `std::decay<>` is used to remove both referenceness and constness from type `T` (the type trait `std::remove_cvref<>` is only provided since C++20).

In a similar way, we can constrain generic code to let universal references bind to rvalues only. However, none of this would be necessary if we had a specific syntax for universal references. For example:

```
void processString(std::string&&& arg);   // assume &&& declares a universal reference
```

We do not have this specific syntax, however, 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 some interesting corner cases for the question of whether an object is an ordinary rvalue reference or a universal reference.

10.2.1 Rvalue References of Members of Generic Types

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);   // not a 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 not a universal reference
  insert(coll, std::move(str));   // OK
  ...
}
```

10.2.2 Rvalue References of Parameters in Class Templates

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 not a universal reference
  ...
  void foo(T&& arg);   // arg is not a 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 not a 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

An rvalue reference to a parameter of a full specialization of a function template is ***not*** a universal reference.

For example:

```
template<typename T>            // primary template
void foo(T&& arg);              // - arg is a universal reference
...
template<>                      // full specialization (for rvalues only)
void foo(std::string&& arg);    // - arg is not a universal reference
```

For lvalues of type `std::string` still the primary template will be called. To specialize all cases of the primary template for `std::string`s you have to provide a second full specialization:

```
template<>                      // full specialization (for lvalues)
void foo(std::string& arg);
```

Here is a complete example:

generic/universalspec.cpp

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

// primary template taking a universal reference:
template<typename Coll, typename T>
void insert(Coll& coll, T&& arg)
{
  std::cout << "primary template for type T called\n";
  coll.push_back(arg);
}

// full specialization for rvalues of type std::string:
template<>
void insert(std::vector<std::string>& coll, std::string&& arg)
{
  std::cout << "full specialization for type std::string&& called\n";
  coll.push_back(arg);
}

// full specialization for lvalues of type std::string:
template<>
void insert(std::vector<std::string>& coll, const std::string& arg)
{
  std::cout << "full specialization for type const std::string& called\n";
  coll.push_back(arg);
}
```

```
int main()
{
  std::vector<std::string> coll;
  ...
  insert(coll, std::string{"prvalue"});   // calls full specialization for rvalues
  std::string str{"lvalue"};
  insert(coll, str);                      // calls full specialization for lvalues
  insert(coll, std::move(str));           // calls full specialization for rvalues
  ...
}
```

Note that you have to declare/define full specializations of member function templates outside the class definition:

```
template<typename T>
class Cont {
  ...
  // primary template:
  template<typename U>
  void insert(U&& v) {  // universal reference
    coll.push_back(std::forward<U>(v));
  }
  ...
};

// full specializations for Cont<T>::insert<>():
// - have to be outside the class
// - need specializations for rvalues and lvalues

template<>
 template<>
void Cont<std::string>::insert<>(std::string&& v)
{
  coll.push_back(std::move(v));
}

template<>
 template<>
void Cont<std::string>::insert<>(const std::string& v)
{
  coll.push_back(v);
}
```

10.3 How the Standard Specifies Perfect Forwarding

To finally understand all rules of perfect forwarding, let us see how the rules are formally specified in the C++ standard.

Again, consider you have the following declaration:

```
template<typename T>
void f(T&& arg)              // arg is universal/forwarding reference
{
  g(std::forward<T>(arg));   // 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 arg is declared as MyType&&
f(std::move(v));             // T is deduced as MyType, so arg is declared as MyType&&
```

However, for universal references, there is a special rule, if lvalues are passed (see the section [temp.deduct.call] of the C++ standard):[1]

> 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 that in this case:

- If the type of the parameter `arg` is declared with `&&` and not declared with `const` or `volatile`
- and an lvalue is passed
- then `T` is deduced as `T&` instead.

In our example, that means:

```
template<typename T>
void f(T&& arg);             // arg is a 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 `arg` declared as `T&&`. So, if `T` is `T&` instead, what does that mean? Here, the *reference collapsing* rule of C++ (see the section [dcl.ref] of the C++ standard) gives an answer:

- *Type*`&` `&` becomes *Type*`&`
- *Type*`&` `&&` becomes *Type*`&`
- *Type*`&&` `&` becomes *Type*`&`
- *Type*`&&` `&&` becomes *Type*`&&`

[1] Since C++17, the C++ standard uses the term *forwarding reference* in this definition.

That means:

```
MyType v;
const MyType c;

f(v);   // T is deduced as MyType& and arg has this type
f(c);   // T is deduced as const MyType& and arg has this type
```

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 passed type parameter:

  ```
  static_cast<T&&>(t)
  ```

  Here, the reference collapsing rules apply again:
  - If type `T` is an lvalue reference, `T&&` is still an lvalue reference (`&&` has no effect). Therefore, we cast `arg` to an lvalue reference, which means that `arg` has no move semantics.
  - However, if `T` is an rvalue reference (or not a reference at all), `T&&` is (still) an rvalue reference. Therefore, we cast `arg` to an rvalue reference and that way we change the value category to an xvalue, which is the effect of `std::move()`.

In total, therefore, we get:

```
template<typename T>
void f(T&& arg)               // arg is a universal/forwarding reference
{
  g(std::forward<T>(arg));    // perfectly forward (move() only for passed rvalues)
}

MyType v;
const MyType c;

f(v);                 // T and arg are MyType&, forward() has no effect in this case
f(c);                 // T and arg are const MyType&, forward() has no effect in this case
f(MyType{});          // T is MyType, arg is MyType&&, forward() is equivalent to move()
f(std::move(v));      // T is MyType, arg is MyType&&, forward() is equivalent to move()
```

Note that string literals are lvalues so that we deduce `T` and `arg` for them as follows:

```
f("hi");              // lvalue passed, so T and arg have type const char(&)[3]
f(std::move("hi"));   // xvalue passed, so T is deduced as const char[3]
                      //                and arg has type const char(&&)[3]
```

Remember also that references to functions are always lvalues, Therefore, T is always deduced as an lvalue reference if we pass a reference to a function to a universal reference:

```
void func(int) {
}

f(func);              // lvalue passed to f(), so T and arg have type void(&)(int)
f(std::move(func));   // lvalue passed to f(), so T and arg have type void(&)(int)
```

10.3.1 Explicit Specification of Types for Universal References

When a universal/forwarding reference is declared, you can also explicitly specify the type of the template parameter instead of deducing it. However, remember that your parameter is declared as T&&. Therefore, you have the following behavior:

```
template<typename T>
void f(T&& arg)               // arg is universal/forwarding reference
{
  g(std::forward<T>(arg));    // perfectly forward (move() only for passed rvalues)
}

f<std::string>(...);          // arg is a raw rvalue reference binding to rvalues only
f<std::string&>(...);         // arg is an lvalue reference binding to non-const lvalues only
f<const std::string&>(...);   // arg is a const lvalue reference binding to everything
f<std::string&&>(...);        // arg is a raw rvalue reference binding to rvalues only
```

Thus, with an explicit specification the universal reference no longer acts as a universal reference. As a caller, you specify which specific kind of reference you get.

Therefore, to pass an lvalue, where you still need the value, make sure you specify the template parameter as lvalue reference. Otherwise, the code will not compile:

```
template<typename T>
void f(T&& arg)                    // arg is universal/forwarding reference
{
  g(std::forward<T>(arg));         // perfectly forward (move() only for passed rvalues)
}

std::string s;
...
f<std::string>(s);                 // ERROR: cannot bind rvalue reference to lvalue
f<std::string&>(s);                // OK, does not move and forward s
f<std::string>(std::move(s));      // OK, does move and forward s
f<std::string&&>(std::move(s));    // OK, does move and forward s
```

The last two calls are equivalent.

These rules also apply if you use the C++20 feature to declare ordinary functions with `auto&&`:

```
void f(auto&& arg) {
  g(std::forward<decltype(arg)>(arg));
}
```

10.3.2 Conflicting Template Parameter Deduction with Universal References

The special rule for deducing template parameters of universal references (deducing the type as an 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 the parameter `coll`, `T` it is deduced as `std::string`.
- However, according to the special rule for universal references, parameter `elem` forces to deduce `T` as `std::string&`.

Therefore, the compiler raises an ambiguity error.

There are two ways to solve this problem:

- You can use the type trait `std::remove_reference<>`:

```
template<typename T>
void insert(std::vector<std::remove_reference_t<T>>& vec, T&& elem)
{
  vec.push_back(std::forward<T>(elem));
}

std::vector<std::string> coll;
std::string s;
...
insert(coll, s);    // OK, with T deduced as std::string& vec now binds to coll
```

- You can 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));
}
```

10.3.3 Pure RValue References of Generic Types

With the special rule for deducing template parameters of universal references (deducing the type as an lvalue reference when lvalues are passed), we can constrain generic reference parameters to only bind to rvalues:[2]

```
template<typename T>
requires (!std::is_lvalue_reference_v<T>)   // bind to rvalues only
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));
}
```

Before C++20, the type trait `std::enable_if<>` had to be used again:

```
template<typename T,
         typename
           = typename std::enable_if<!std::is_lvalue_reference<T>::value
                                     >::type>
void callFoo(T&& arg) {
  foo(std::forward<T>(arg));
}
```

10.4 Nasty Details of Perfect Forwarding

Whenever something new and complex like move semantics is invented, people make mistakes. Making mistakes is human, and over a long period we cannot really avoid them because we do not know all applications and extensions of move semantics in the future in advance. In the C++ standard we also took a few decisions about perfect forwarding and universal references that (looking back at least) are mistakes.

[2] Thanks to Herb Sutter to coming up with this topic.

10.4.1 "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 did not introduce any special term for universal references; it simply introduced special rules for rvalue references of function templates parameters.

In 2012, when describing the behavior of these references, Scott Meyers, one of the key authors of the C++ community, introduced the term *universal reference* for them. To be consistent with the terms *rvalue reference* and *lvalue reference*, which describe what the references bind to, the intention was to introduce *universal reference* as a description that this kind of reference can "universally" bind to all value categories.

Unfortunately, a couple of years later, the C++ standards committee decided to introduce a different term with C++17: *forwarding reference*. The official reason is that these references are not that "universal" because they cannot bind to everything and their main purpose is to perfectly forward parameters (see `http://wg21.link/n4164`).

Although forwarding is a major use case for these references, as we have seen, forwarding is not the only use case. Another very important use case is just to bind a reference, universally, to any object just to keep the information about its value category and/or whether it is `const`. For example:

- We might have to use the same object (whatever it is) twice without losing the information whether the object is `const`. One typical example is to call `begin()` and `end()` for a passed collection (as we have seen before and is also used later in the implementation of the range-based `for` loop).
- We might have to refer to objects of any value category without declaring it to be `const` (an example is given later where we use a universal reference to call the range-based `for` loop).

The question is, why did the C++ standards committee not just adopt the term already established in the C++ community? Well, the argument was that the term is confusing, but we have more confusing terms in the C++ standard and the confusion of now having two terms for the same thing is certainly far worse.[3]

In any case, it was the deliberate decision of the C++ standards committee to come up with a different term. There is no doubt that *universal reference* was good enough (we have far worse terms in the C++ standard); however, not adopting this term and using another or better term was even more important for a critical mass of people in the C++ standards committee.

I still prefer to use *universal reference* (the books should at least be consistent; the standard often uses its own terminology only valid for experts anyway). However, for clarification, I often (have to) use *universal/-forwarding reference* to deal with this mess.

For you, this means that whenever you hear the term *forwarding reference*, you have to translate it to *universal reference* (or vice versa). And if in doubt, use *universal/forwarding reference*.

[3] There is a rumor of another reason for coming up with a different term: some people in the C++ standards committee may not have liked 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++ standards committee we are human, after all.

10.4.2 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 did we not introduce a specific syntax for universal references?

An alternative proposal, for example, might have been (and is sometimes discussed as a possible fix):

- 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 constraining universal references to concrete types, we would then just have to declare

```
void processString(std::string&&& arg);   // assume &&& declares a universal reference
```

instead of

```
template<typename T>
requires std::is_convertible_v<T, std::string>
void processString(T&& arg);
```

or even:

```
template<typename T,
         typename =
           typename std::enable_if<std::is_convertible<T, std::string>::value
                                   >::type>
void processString(T&& args);
```

There is an important lesson to be learned here: it is better to have a ridiculous but clear syntax than to have a cool but confusing mess.

10.5 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.
- You can use universal references to implement special handling for passed arguments with move semantics even without using `std::forward<>()`.
- To have a universal reference of a specific type, you need concepts/requirements (since C++20) or some template tricks (up to C++17).
- Only rvalue references of function template parameters are universal references. Rvalue references of class template parameters, members of template parameters, and full specializations are ordinary rvalue references, you can only bind to rvalues.

- When specifying the types of universal references explicitly, they act no longer as universal references. Use *Type&* to be able to pass lvalues then.
- The C++ standards committee introduced *forwarding reference* as a "better" term for *universal reference*. Unfortunately, the term *forwarding reference* restricts the purpose of universal references to a common specific use case and creates the unnecessary confusion of having two terms for the same thing. Therefore, use *universal/forwarding reference* to avoid even more confusion.

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 dealing with return values perfectly. In this chapter, we discuss how to forward return values perfectly to somewhere else. For this purpose, we introduce `auto&&` as another way to declare a *universal reference* (again also called *forwarding reference*). However, 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

We often 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 pass a return value to another function directly, the value is passed perfectly, keeping its type and value category. You do not 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, which means 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, which means 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(std::string&)
```

- 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, which means that the return value is perfectly forwarded with the best match for an rvalue reference, allowing process() to 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 a prvalue, which means that the return value is perfectly forwarded with the best match for an rvalue reference, allowing process() to 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 a declaration for const&&, that overload would be taken.

Therefore: do not mark values returned by value with const and do not mark returned non-const rvalue references with const.

11.2 Universal References with auto&&

However, in generic code, how can you program passing a return value *later* but still keeping its type and value category?

The answer is that again, you need a universal/forwarding reference, which, however, is not declared as a parameter. For this purpose, auto&& was introduced.

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 Type Deduction of auto&&

If you declare something with auto&&, you also declare a *universal reference*. You define a reference that binds to all value categories and the type of this reference preserves the type and the value category of its initial value.

If you declare

```
auto&& ref{...};          // ref is a universal/forwarding reference
```

the type ref is deduced according to the same type deduction rules as universal references being function template parameters:

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

By rule, the type of ref (which is the type auto&& or T&&) is

- An lvalue reference if we refer to an lvalue
- An rvalue reference if we refer to an rvalue

For example:

```
// forward declarations:
std::string retByValue();
std::string& retByRef();
std::string&& retByRefRef();
const std::string& retByConstRef();
const std::string&& retByConstRefRef();

// deduced auto&& types:
std::string s;
```

```
auto&& r1{s};                    // std::string&
auto&& r2{std::move(s)};         // std::string&&

auto&& r3{retByValue()};         // std::string&&
auto&& r4{retByRef()};           // std::string&
auto&& r5{retByRefRef()};        // std::string&&
auto&& r6{retByConstRef()};      // const std::string&
auto&& r7{retByConstRefRef()};   // const std::string&&
```

Whenever we use an rvalue (prvalue or xvalue) to initialize a reference declared with `auto&&`, we declare an rvalue reference. For example, this is the case when we bind the reference to an object marked with `std::move()` or to a returned plain value or rvalue reference.

However, when we use an lvalue to initialize a reference declared with `auto&&`, we declare an lvalue reference. For example, this is the case when we bind the reference to an object that has a name or to the return value of a function returning an lvalue reference.

Because string literals are lvalues (with a character array as type) we also get an lvalue reference when we bind the universal reference to a string literal:

```
auto&& r8{"hello"};              // const char(&)[6]
```

Because references to functions are always lvalues we also get an lvalue reference when we bind the universal reference to a function:

```
std::string foo(int);            // forward declaration

auto&& r9{foo};                  // lvalue of type std::string(&)(int)
```

11.2.2 Perfectly Forwarding an `auto&&` Reference

The same way we can perfectly forward the value passed to a universal reference as a function template parameter:

```
template<typename T>
void callFoo(T&& ref) {
  foo(std::forward<T>(ref));     // becomes foo(std::move(ref)) 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 rvalues
```

Remember that the expression `std::forward<decltype(ref)>(ref)` is essentially implemented as follows:

- If the passed type `decltype(ref)` is an lvalue reference, which it is if `ref` was initialized with a returned lvalue reference, the expression casts `ref` to an lvalue reference, which means that `ref` is passed without move semantics.

- If the passed type decltype(ref) is an rvalue reference, which it is if ref was initialized with a returned plain value or rvalue reference, the expression casts ref to an rvalue reference, which is the effect of std::move(ref).

Therefore, if we initialize the universal reference with the return value of a function:

```
auto&& ret{compute(t)};     // initialize a universal reference with the return value
```

the expression

```
process(std::forward<decltype(ret)>(ret));   // perfectly forward the return value
```

expands to

```
process(std::move(ret));
```

if and only if compute() returned an rvalue (such as a temporary object or an rvalue reference).

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 and still preserving its value category and the information about 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 `range` as a universal reference is that we want to be able to bind it to *every* range so that we can use it twice (once to ask for its beginning, and once to ask for its end) without creating a copy or losing the information about 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` for all these cases:

- With `auto`, we would create a copy of the range (which takes time and disables modification of the elements).
- With `auto&`, we would disable initialization of 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 such as the following:

```
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. This also applies to rvalue references. However, we do not 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()`. This means that we extend the lifetime of the reference but not of the return value of `createString()`. Therefore, the loop iterates over strings that were already destroyed.

Using the Range-Based `for` Loop

A universal reference can make sense even when *calling* the range-based `for` loop.

To modify the elements while iterating over them you have to use a non-`const` reference. Consider a function template that assigns a passed value to all elements of a passed collection:

```
template<typename Coll, typename T>
void assign(Coll& coll, const T& value) {
  for (auto& elem : coll) {
    elem = value;
  }
}
```

It looks like it usually works for all container and element types (where assignments are supported):

```
std::vector<int> coll1{0, 8, 15};
...
assign(coll1, 42);    // OK

std::vector<std::string> coll2{"hello", "world"};
...
assign(coll2, "ok");  // OK
```

However, there is a case where it suddenly does not work:

```
std::vector<bool> collB{false, true, false};
...
assign(collB, true);  // ERROR: cannot bind non-const lvalue reference to an rvalue
```

What happened? Well, let us look at the code the range-based `for` loop expands to here:

```
std::vector<bool> coll{false, true, false};
...
{
  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>` are not objects of type `bool`; they are single bits. The way this is implemented is that for `std::vector<bool>` the type of a reference to an element is *not* a reference to the element type. The implementation of class `std::vector<bool>` is a partial specialization of the implementation of the primary template `std::vector<T>` where references to elements are objects of a proxy class that you can use like a reference:

```
namespace std {
  template<...>
  class vector<bool, ...> {
   public:
    ...
    class reference {
      ...
    };
    ...
  };
}
```

A value of this class `std::vector<bool>::reference` is returned when dereferencing an iterator. Therefore, in the expanded code of the range-based `for` loop, the statement

```
auto& elem = *pos;
```

tries to bind a non-`const` lvalue reference to a temporary object (prvalue), which is not allowed.

However, there is a solution to the problem: use a universal reference when calling the range-based `for` loop instead:

```
template<typename Coll, typename T>
void assign(Coll& coll, const T& value) {
  for (auto&& elem : coll) {  // note: universal reference support proxy element types
    elem = value;
  }
}
```

Because a universal reference can bind to any object of any value category (even to a prvalue), using this generic code with `vector<bool>` now compiles:

```
std::vector<bool> collB{false, true, false};
...
assign(collB, true);  // OK (universal reference used to bind to an element)
```

We have thus 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.

Consider the example, we already had with a function templates:

```
auto callFoo = [](auto&& arg) {           // arg is a universal/forwarding reference
  foo(std::forward<decltype(arg)>(arg));  // perfectly forward arg
};

std::string s{"BLM"};
callFoo(s);                     // OK, arg is std::string&
callFoo(std::move(s));          // OK, arg is std::string&&
```

Note that since C++20, you can avoid using `decltype(arg)` by declaring the lambda with template parameters.

The following generic lambda uses this to perfectly forwards a variadic number of 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 allow their use as functions). The definition above expands to a compiler-defined class (*closure type*) with universal references defined as template parameters:

```
class NameDefinedByCompiler {
    ...
  public:
    template<typename... Args>
    auto operator() (Args&&... args) const {
      ...
      foo(std::forward<decltype(args)>(args)...);
    }
};
```

Again, note that a generic rvalue reference that is qualified with `const` (or `volatile`) is ***not*** a universal reference. This also applies to lambdas. If a parameter is declared with `const auto&&`, you can only pass rvalues:

```
auto callFoo = [](const auto&& arg) {  // arg is not a universal reference
  ...
};

const std::string cs{"BLM"};
callFoo(cs);                           // ERROR: s is not an rvalue
callFoo(std::move(cs));                // OK, arg is const std::string&&
```

Inside this lambda, it would be enough to use `std::move()` to perfectly forward the passed argument.

11.5 Using auto&& in C++20 Function Declarations

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.

The following definition:

```
void callFoo(auto&& val) {
  foo(std::forward<decltype(arg)>(arg));
}
```

is equivalent to the following:

```
template<typename T>
void callFoo(T&& val) {
  foo(std::forward<decltype(arg)>(arg));
}
```

This means that you can also explicitly specify the type of the parameter here. However, in the same way as for ordinary function templates, you have to qualify the template parameter with a type that fits the value category of your passed argument:

```
std::string s;
...
callFoo<std::string>(s);                // ERROR: cannot bind rvalue reference to lvalue
callFoo<std::string&>(s);               // OK, does not move and forward s
callFoo<std::string>(std::move(s));     // OK, does move and forward s
callFoo<std::string&&>(std::move(s));   // OK, does move and forward s
```

11.6 Summary

- Do not return by value with const (otherwise you disable move semantics for return values).
- Do not mark returned non-const rvalue references with const.
- auto&& can be used to declare a universal reference that is not a parameter. Just like 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
- Use std::forward<decltype(*ref*)>(*ref*) to perfectly forward a universal reference *ref* that is declared with auto&&.
- 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 parameters that are universal references in a function template.

This page is intentionally left blank

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 returning values perfectly. This chapter 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 that we then return to the caller. The question is, how do we perfectly return the value but still keep 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 do not 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, for example, we give access to the element of a container (consider `foo()` as 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 issue a warning 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

but still keep both the type and the value category of what we return.

C++14 introduced a new placeholder type for this 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, both the type and the value category are retained.

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 an lvalue reference: *type*&
  - For an **xvalue**, it yields its type as an 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` to also be an lvalue reference.
- Because `(s)` is an *lvalue*, `da8` is an lvalue reference. Yes, the parentheses make a difference here.

In contrast to `auto&&`, which is always a reference, `decltype(auto)` is sometimes just a value (if initialized with the name of an object of a value type or with a prvalue expression).

Note that `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 an *lvalue* and `decltype(auto)` deduces an lvalue reference.
- If the expression returns/yields an rvalue reference, then the value category is an *xvalue* and `decltype(auto)` deduces an rvalue reference.

That is 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 were to 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)...);
}
```

The function declares `args` as a variadic number of *universal references* (also called *forwarding references*). With `std::forward<>()`, it perfectly forwards these given arguments to the function `f` passed as first argument. Because we use `decltype(auto)` as the return type, we perfectly return the return value of `f()`

to the caller of `call()`. Therefore, 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()
{
  return "Let's dance";
}

std::ostream& print(std::ostream& strm, const std::string& val)
{
  return strm << "value: " << val;
}

std::string&& returnArg(std::string&& arg)
{
  return std::move(arg);
}

int main()
{
  auto s = call(nextString);                    // call() returns temporary object

  auto&& ref = call(returnArg, std::move(s));   // call() returns rvalue reference to s
  std::cout << "s:   " << s << '\n';
  std::cout << "ref: " << ref << '\n';

  auto str = std::move(ref);                    // move value from s and ref to str
  std::cout << "s: "   << s << '\n';
  std::cout << "ref: " << ref << '\n';
  std::cout << "str: " << str << '\n';

  call(print, std::cout, str) << '\n';          // call() returns reference to std::cout
}
```

When calling

```
auto s = call(nextString);
```

the function call() calls the function nextString() without any arguments and returns its return value perfectly to initialize s.

When calling

```
auto&& ref = call(returnArg, std::move(s));
```

the function call() calls the function returnArg() with s marked with std::move(). returnArg() returns back the passed argument as an rvalue reference, which then call() perfectly returns to the caller to initialize ref. str still has its value and ref refers to it:

```
s:   Let's dance
ref: Let's dance
```

With

```
auto str = std::move(ref);
```

we move the value of both s and ref to str, which means we get the following situation:

```
s:
ref:
str: Let's dance
```

When calling

```
call(print, std::cout, ref) << '\n';
```

the function call() calls the function print() with std::cout and ref as perfectly forwarded arguments. print() returns the passed stream back as an lvalue reference, which is then returned perfectly to the caller of call().

12.2.2 Deferred Perfect Returning

To perfectly return a value that was computed earlier, we have to declare a local object with decltype(auto) and return it with std::move() if and only if it is an rvalue reference. For example:[1]

```
template<typename Func, typename... Args>
decltype(auto) call(Func f, Args&&... args)
{
  decltype(auto) ret{f(std::forward<Args>(args)...)};
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
    return std::move(ret);   // move xvalue returned by f() to the caller
  }
  else {
    return ret;              // return the plain value or the lvalue reference
  }
}
```

[1] Thanks to Matthias Kretz for providing this solution.

In this function, the type of ret is just the perfectly deduced type of f(). By using if constexpr (a compile-time if available since C++17), we use the two ways of decltype(auto) and decltype to deduce a type as follows:

- If ret is declared as an rvalue reference, decltype(auto) uses the *expression* std::move(ret), which is an xvalue, to deduce an rvalue reference. So we move the value returned by f() to the caller of this function.
- If ret is declared as a plain value or lvalue reference, decltype(auto) uses the type of the *name* ret, which is then also a value type or an lvalue reference type.

Other solutions do not always work:

- The following would do the right thing even before C++20, but we have a performance issue:

```
decltype(auto) call(...)
{
  decltype(auto) ret{f(...)};
  ...
  return static_cast<decltype(ret)>(ret);   // perfect return but unnecessary copy
}
```

  The fact that we always use a static_cast<> might disable move semantics and copy elision. For plain values, it is like having an unnecessary std::move() in the return statement.

- Simply returning ret would not always work:

```
decltype(auto) call(...)
{
  decltype(auto) ret{f(...)};
  ...
  return ret;                                // may be an ERROR
}
```

  The return type of call() would be correct. However, if f() returns an rvalue reference, we cannot return the lvalue ret because a non-const revalue reference does not bind to an lvalue.

- Using auto&& to declare ret would not work because you would then always return by reference:

```
decltype(auto) call(...)
{
  auto&& ret{f(...)};
  ...
  return ret;   // fatal runtime error: returns a reference to a local object
}
```

Do ***never*** put additional parentheses around a returned name when using decltype(auto):

```
decltype(auto) call(...)
{
  decltype(auto) ret{f(...)};
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
    return std::move(ret);   // move value returned by f() to the caller
  }
```

```
  else {
    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 always deduces an lvalue reference because `ret` is an lvalue (an object with a name),

If you are used to putting parentheses around names and expressions in return statements, stop doing that. It was never necessary, but now it might even be an error when using `decltype(auto)`.

12.2.3 Perfect Forwarding and Returning with Lambdas

If a lambda should return perfectly, you have to change its return type. A declaration such as:

```
[] (auto f, auto&&... args) {
  ...
}
```

stands for:

```
[] (auto f, auto&&... args) -> auto {
  ...
}
```

This means that by default, lambdas always return by value.

By explicitly declaring the lambda with the return type `decltype(auto)`, we enable perfect returning:

```
[] (auto f, auto&&... args) -> decltype(auto) {
  ...
  return f(std::forward<decltype(args)>(args)...);
};
```

For deferred perfect returning, we need the same trick as introduced before: we have to use `std::move()` if we return an rvalue reference:

```
[] (auto f, auto&&... args) -> decltype(auto) {
  decltype(auto) ret = f(std::forward<decltype(args)>(args)...);
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
    return std::move(ret);   // move value returned by f() to the caller
  }
  else {
    return ret;              // return the value or the lvalue reference
  }
};
```

Again, do not put additional parentheses around a returned name because then `decltype(auto)` always deduces an lvalue reference:

```
[] (auto f, auto&&... args) -> decltype(auto) {
  ...
  decltype(auto) ret = f(std::forward<decltype(args)>(args)...);
  ...
  if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
    return std::move(ret);   // move value returned by f() to the caller
  }
  else {
    return (ret);   // FATAL RUNTIME ERROR: always returns an lvalue reference
  }
};
```

12.3 Summary

- `decltype(auto)` is a placeholder type for deducing 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.

This gives you a good overview of the use of move semantics in practice.

This page is intentionally left blank

Chapter 13

Move-Only Types

One major application of move semantics is *move-only* types. These are types where objects represent a value or the "own" a resource for which copying does not make any sense. However, it should still be possible to pass the value or the ownership around (e.g., passing it as an argument, returning it, or storing it in a container).

Examples of move-only types in the C++ standard library are:

- IOStreams
- Threads
- Unique pointers

In all these examples, an object represents a resource (an opened stream, a running thread, or an allocated object). The object "owns" the resource in the sense that the destructor of the object will release the resource (close the stream, end or wait for the end of the thread, release the allocated object).

You can pass the ownership around but copying of the resource is not supported. Copying might not make sense for semantic reasons (what is a copy of an opened file, what is a copy of a running thread?) or for technical reasons (if we have two owners of the same resource, we have to synchronize the access or deal with potential consequences).

Thus, move-only types simplify the management of unique resources. The type system is used to disable copying but we can still pass these resources around. There is always only one location at a time where the the resource is stored/managed. However, we can still refer to these resources with pointers or references.

13.1 Declaring and Using Move-Only Types

Move-only types and objects have some common aspects when declared or used.

13.1.1 Declaring Move-Only Types

Move-only types have copying disabled. Usually, the copy constructor and the copy assignment operator are deleted, while the move constructor and move assignment operator are defaulted or implemented:

For example:

```
class MoveOnly {
 public:
  // constructors:
  MoveOnly();
  ...
  // copying disabled:
  MoveOnly(const MoveOnly&) = delete;
  MoveOnly& operator= (const MoveOnly&) = delete;
  // moving enabled:
  MoveOnly(MoveOnly&&) noexcept;
  MoveOnly& operator= (MoveOnly&&) noexcept;
};
```

By rule, it would be enough to declare the moving special member function (because declaring special move members marks the copying members as deleted). However, explicitly marking the copying special member function with `=delete` makes the intention more clear.

13.1.2 Using Move-Only Types

With a declaration like the one above, you can create and move but not copy objects. For example:

```
std::vector<MoveOnly> coll;
...
coll.push_back(MoveOnly{});       // OK, creates a temporary object, which is moved into coll
...
MoveOnly mo;
coll.push_back(mo);               // ERROR: can't copy mo into coll
coll.push_back(std::move(mo));    // OK, moves mo into coll
```

To move the value of a move-only element out of the container, simply use `std::move()` for a reference to the element. For example:

```
mo = std::move(coll[0]);          // move assign first element (still there with moved-from state)
```

However, remember that after this call, the element is still in the container with a moved-from state.

Moving out all elements is also possible in loops:

```
for (auto& elem : coll) {             // note: non-const reference
  coll2.push_back(std::move(elem));   // move element to coll2
}
```

Again: the elements are still in the container with their moved-from state.

For move-only types, a couple of operations that are usually valid are not possible:

- You cannot use `std::initializer_lists` because they are usually passed by value, which requires copying of the elements:

```
std::vector<MoveOnly> coll{ MoveOnly{}, ... };  // ERROR
```

- You can only iterate by reference over all move-only elements of a container:

```
std::vector<MoveOnly> coll;
...
for (const auto& elem : coll) { // OK
  ...
}
...
for (auto elem : coll) {        // ERROR: can't copy move-only elements
  ...
}
```

See *lib/moveonly.cpp* for a complete program with all example statements from this section.

13.1.3 Passing Move-Only Objects as Arguments

You can pass and return move-only objects by value provided move semantics is used:

```
void sink(MoveOnly arg);        // sink() takes ownership of the passed argument

sink(MoveOnly{});               // OK, moves temporary objects to arg
MoveOnly mo;
sink(mo);                       // ERROR: can't copy mo to arg
sink(std::move(mo));            // OK, moves mo to arg because passed by value
```

Semantically, you pass the ownership of the associated resource here to the function. However, note that this is only the case if the argument is taken by value.

The `sink()` function can also be declared to take a move-only object by (rvalue or universal) reference. In that case, you still have to pass an lvalue with `std::move()`. However, you then do not know here whether the ownership of the passed resource is taken by `sink()`.

```
void sink(MoveOnly&& arg);      // sink() might take ownership of the passed argument

MoveOnly mo;
sink(mo);                       // ERROR: can't pass lvalue mo to arg
sink(std::move(mo));            // OK, might move mo to something inside sink()
```

There was a discussion between Scott Meyers and Herb Sutter (two of the leading authors of the C++ community) about how a sink function for a move-only type should be declared. Herb's position was to take the argument by value, Scott's position was to take it by rvalue reference.

As far as I know, they later agreed that it would be better to take the argument by rvalue reference. However, the real answer is that for your code, this should not matter. The usual rule for `std::move()` should also apply here: if you pass a move-only object with `std::move()`, you *might or might not* lose its value. If it is important for you to give up ownership (because you want to ensure that the file is closed,

the thread has stopped, or the associated resource was released), ensure this directly after the call with a corresponding statement. For example:

```
MoveOnly mo;

foo(std::move(mo));                 // might move ownership
// ensure mo's resource is longer acquired/owned/open/running:
mo.close();                         // or mo.reset() or mo.release() or so
```

Move-only objects usually have such a functions but the names differ (e.g., in the C++ standard library, it is called `close()` for streams, `join()` for threads, or `reset()` for unique pointers). These functions usually bring the objects into a default constructed state.

13.1.4 Returning Move-Only Objects by Value

You can also implement source functions that returns a (new) move-only object by value, which means passing the ownership to the caller of the function.

If you return a local object that way, move semantics is automatically used:

```
MoveOnly source()
{
  MoveOnly mo;
  ...
  return mo;              // moves mo to the caller
}

MoveOnly m{source()};   // takes ownership of the associated value/resource
```

It is only if you have non-local data that you might need a `std::move()` in the return statement (e.g., when moving out the value of a member in a member function).

13.1.5 Moved-From States of Move-Only Objects

The moved-from state of move-only objects is usually that they no longer own their resource. This is a defined state, which the user of the type should be able to double-check. However, sometimes, move operations just swap the internal data so that a move assignment assigns another resource to the moved-from object (e.g., the stream classes do this).

The C++ standard library uses different ways and names to check for a "moved-from" state that no longer owns a resource. A positive check (*Is there (still) a resource?*) might, for example, look as follows:

- `if(s.is_open())` for a file stream `s`
- `if(up)` for a unique pointer `up`
- `if(t.joinable())` for a thread `t`
- `if(f.valid())` for a `std::future` `f`

13.2 Summary

- Move-only types allow us to move "owned" resources around without being able to copy them. The copying special member functions are deleted.
- You cannot use move-only types in `std::initializer_lists`.
- You cannot iterate by value over collections of move-only types.
- If you pass a move-only object to a sink function and want to ensure that you have lost ownership (file closed, memory freed, etc.), explicitly release the resource directly afterwards.

This page is intentionally left blank

Chapter 14

Moving Algorithms and Iterators

The C++ standard library has special support for moving elements while iterating over them. For this purpose, special algorithms and special iterators (*move iterators*) were introduced with C++11. This chapter discusses how to use them.

14.1 Moving Algorithms

The C++ standard library provides a few algorithms that move elements. These algorithms are:

- `std::move()`, moving elements to another range or backwards in the same range (do not confuse this algorithm with the `std::move()` to mark an object that you no longer need its value)
  You specify the beginning of the destination range and the elements are moved from the beginning to the end of the source range.
- `std::move_backward()`, moving elements to another range or forward in the same range
  You specify the end of the destination range and the elements are moved from the end to the beginning of the source range.

These algorithms are the counterparts of the `std::copy()` and `std::copy_backward()` algorithms using move semantics. And yes, we have another overload for `std::move()` taking multiple parameters (three iterators and, since C++17, an optional parallel execution policy).

The effect of these algorithms is a move assignment to the destination range calling `std::move(`*elem*`)` for each element while iterating over them.

Consider the following example:

lib/movealgo.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>

template<typename T>
```

```
void print(const std::string& name, const T& coll)
{
  std::cout << name << " (" << coll.size() << " elems): ";
  for (const auto& elem : coll) {
    std::cout << " '" << elem << "'";
  }
  std::cout << "\n";
}

int main(int argc, char** argv)
{
  std::list<std::string> coll1 { "love", "is", "all", "you", "need" };
  std::vector<std::string> coll2;

  // ensure coll2 has enough elements to overwrite their values:
  coll2.resize(coll1.size());

  // print out size and values:
  print("coll1", coll1);
  print("coll2", coll2);

  // move assign the values from coll1 to coll2
  // - not changing any size
  std::move(coll1.begin(), coll1.end(),          // source range
            coll2.begin());                      // destination range

  // print out size and values:
  print("coll1", coll1);
  print("coll2", coll2);

  // move assign the first three values inside coll2 to the end
  // - not changing any size
  std::move_backward(coll2.begin(), coll2.begin()+3,  // source range
                     coll2.end());                    // destination range

  // print out size and values:
  print("coll1", coll1);
  print("coll2", coll2);
}
```

When calling the `std::move()` algorithm, we move assign all values from the source container to the destination container:

```
// move assign the values from coll1 to coll2
// - not changing any size
```

```
std::move(coll1.begin(), coll1.end(),          // source range
          coll2.begin());                      // destination range
```

As usual for overwriting algorithms, the destination container must already have enough elements that are overwritten (otherwise you have undefined behavior). The number of elements does not change (neither in the source nor in the destination range). However, the elements of the source range get a moved-from state. Thus, we do not know the values of the strings in the source range after this call (unless we have specified behavior for moved-from objects such as for move-only types).

When calling the `std::move_backward()` algorithm in the example, we then move assign the first three elements to the end of the same collection:

```
// move assign the first three values inside coll2 to the end
// - not changing any size
std::move_backward(coll2.begin(), coll2.begin()+3,  // source range
                   coll2.end());                    // destination range
```

Again, elements where the value was moved away get a moved-from state unless another value was moved there. As a result, we do not know the value of the first two elements anymore (the value of the third element was overwritten with the moved-assigned value of the first element). After this call, all we know is that the last three elements of `coll2` are `"love"`, `"is"`, and `"all"`.

Therefore, the output of the whole program is something like this (`?` signaling that we do not know the value):

```
coll1 (5 elems):  'love' 'is' 'all' 'you' 'need'
coll2 (5 elems):  '' '' '' '' ''
coll1 (5 elems):  '?' '?' '?' '?' '?'
coll2 (5 elems):  'love' 'is' 'all' 'you' 'need'
coll1 (5 elems):  '?' '?' '?' '?' '?'
coll2 (5 elems):  '?' '?' 'love' 'is' 'all'
```

A moved-from string is often empty but this is not guaranteed. In practice, I even found this output on one platform:

```
coll1 (5 elems):  'love' 'is' 'all' 'you' 'need'
coll2 (5 elems):  '' '' '' '' ''
coll1 (5 elems):  '' '' '' '' ''
coll2 (5 elems):  'love' 'is' 'all' 'you' 'need'
coll1 (5 elems):  '' '' '' '' ''
coll2 (5 elems):  'need' 'you' 'love' 'is' 'all'
```

As usual: beware when using the moved-from elements.

14.2 Removing Algorithms

By design, C++ algorithms use iterators to deal with the elements of containers and ranges. However, like pointers operating on arrays, iterators can only read and write values; they cannot insert or remove elements. Therefore, "removing" algorithms do not really remove elements; they only move the values of all elements that are not removed to the front of the processed range and return the new end.

For example, given you have a sequence of the following integer values:

```
1 2 3 4 5 4 3 2 1
```

calling the algorithm `std::remove()` with the value 2 to remove all elements with the value 2 modifies the sequence as follows:

```
1 3 4 5 4 3 1 2 1
```

All elements that do not have the value 2 are moved to the front and as the new end (the position behind the "last" element), the position of the 2 is returned.

If possible, these algorithms move. That is, they leave elements where the value was moved-away in a moved-from state. In this case, if the elements were strings, the last 2 would have been untouched, but the last 1 would be moved forward before the 2 so that the last element gets a moved-from state.

Therefore, these algorithms can also leave elements in a moved-from state. In fact, the following algorithms can create moved-from states:

- `std::`**`remove()`** and `std::`**`remove_if()`**
- `std::`**`unique()`**

Let us look at a full example. Consider a class where we can see whether an element has a moved-from state:

lib/email.hpp

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

// class for email addresses
// - asserts that each email address has a @
// - except when in a moved-from state
class Email {
 private:
  std::string value;        // email address
  bool movedFrom{false};    // special moved-from state
 public:
  Email(const std::string& val)
   : value{val} {
      assert(value.find('@') != std::string::npos);
  }
  Email(const char* val)    // enable implicit conversions for string literals
   : Email{std::string(val)} {
  }

  std::string getValue() const {
      assert(!movedFrom);  // or throw
      return value;
  }
```

```
  ...

  // implement move operations to signal a moved-from state:
  Email(Email&& e) noexcept
   : value{std::move(e.value)}, movedFrom{e.movedFrom} {
      e.movedFrom = true;
  }
  Email& operator=(Email&& e) noexcept {
      value = std::move(e.value);
      movedFrom = e.movedFrom;
      e.movedFrom = true;
      return *this;
  }
  // enable copying:
  Email(const Email&) = default;
  Email& operator=(const Email&) = default;

  // print out the current state (even if it is a moved-from state):
  friend std::ostream& operator<< (std::ostream& strm, const Email& e) {
    return strm << (e.movedFrom ? "MOVED-FROM" : e.value);
  }
};
```

By implementing the move constructor and move assignment operator, we set a member `movedFrom`, which we evaluate in the output operator.

Now let us put some elements of this class into a vector and remove some elements using the algorithm `std::remove_if()`. In this case, we remove all email addresses ending with `".de"`:

lib/removeif.cpp

```
#include "email.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<Email> coll{ "tomdomain.de", "jillcompany.com",
                           "sarahdomain.de", "hanacompany.com" };

  // remove all email addresses ending with ".de":
  auto newEnd = std::remove_if(coll.begin(), coll.end(),
                               [] (const Email& e) {
                                 auto&& val = e.getValue();
                                 return val.size() > 2 &&
```

```
                                        val.substr(val.size()-3) == ".de";
                                });

  // print elements up to the new end:
  std::cout << "remaining elements:\n";
  for (auto pos = coll.begin(); pos != newEnd; ++pos) {
     std::cout << "  '" << *pos << "'\n";
  }

  // print all elements in the container:
  std::cout << "all elements:\n";
  for (const auto& elem : coll) {
    std::cout << "  '" << elem << "'\n";
  }
}
```

The output of the program is as follows:

```
remaining elements:
  "jill@company.com"
  "hana@company.com"
all elements:
  "jill@company.com"
  "hana@company.com"
  "sarah@domain.de"
  "MOVED-FROM"
```

After "removing" the elements ending with `".de"`, the new end is the position behind the second element. However, in the container, you still have the third element, which was not moved at all, and the fourth element that has a new moved-from state because it was move-assigned to the second element (which was moved to the first element before).

Be careful when using these moved-from objects, as discussed in the chapter about moved-from states.

14.3 Move Iterators

By using *move iterators* (also introduced with C++11), you can use move semantics even in other algorithms and in general wherever input ranges are taken (e.g., in constructors).

However, be careful when using these iterators. While iterating over elements of a container or range, each access to an element uses `std::move()`. This might be significantly faster but it leaves the element in a valid but unspecified state. You should not use an element twice.

14.3.1 Move Iterators in Algorithms

Using move iterators in algorithms usually only makes sense when the algorithm guarantees to use each element only once. Therefore, the algorithm should:

- Require the *input iterator* category for the source and the *output iterator* category for the destination
- Or guarantee to use each element only once (e.g., as specified for the `std::for_each()` algorithm)

For algorithms with callables, allowing the specification the detailed functionality, the elements are passed to the callable with `std::move()`. Inside the callable you can decide how to deal with them:

- Take the argument by value to always move/steal the value or resource
- Take the argument by rvalue/universal reference to decide which value/resource to move/steal

For example:

lib/foreachmove.cpp

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

template<typename T>
void print(const std::string& name, const T& coll)
{
  std::cout << name << " (" << coll.size() << " elems): ";
  for (const auto& elem : coll) {
    std::cout << " \"" << elem << "\"";
  }
  std::cout << "\n";
}

void process(std::string s)   // gets moved value from rvalues
{
  std::cout << "- process(" << s << ")\n";
  ...
}

int main()
{
  std::vector<std::string> coll{"don't", "vote", "for", "liars"};
  print("coll", coll);

  // move away only the elements processed:
  std::for_each(std::make_move_iterator(coll.begin()),
                std::make_move_iterator(coll.end()),
                [] (auto&& elem) {
                  if (elem.size() != 4) {
```

```
                    process(std::move(elem));
                  }
                });

  print("coll", coll);
}
```

In this program, we move all elements that have a size of 4 to a helper function `process()`:

```
// move away only the elements processed:
std::for_each(std::make_move_iterator(coll.begin()),
              std::make_move_iterator(coll.end()),
              [] (auto&& elem) {
                if (elem.size() != 4) {
                  process(std::move(elem));
                }
              });
```

To do this, we take the element marked, which the move iterators marked with `std::move()`, by universal reference and pass it with `std::move()` to `process()`. Because `process()` takes the argument by value, the value is essentially moved away.

As a result, all elements that do not have a size of 4 become moved-from objects in the container `coll`. Therefore, the output of this program is as follows (`?` signaling that we do not know the value):

```
coll (4 elems):  "don't" "vote" "for" "liars"
- process(don't)
- process(for)
- process(liars)
coll (4 elems):  "?" "vote" "?" "?"
```

Usually, the last line is as follows:

```
coll (4 elems):  "" "vote" "" ""
```

As you can see, a helper function `std::make_move_iterator()` is used so that you do not have to specify the element type when declaring the iterator. Since C++17, *class template argument deduction (CTAD)* enables simply declaring the type `std::move_iterator` directly without the need to specify the element type:

```
std::for_each(std::move_iterator{coll.begin()},
              std::move_iterator{coll.end()},
              [] (auto&& elem) {
                if (elem.size() != 4) {
                  process(std::move(elem));
                }
              });
```

14.3.2 Move Iterators in Constructors and Member Functions

You can also use move iterators wherever an algorithm that reads elements once is used. A useful scenario might be to move elements of a source container to another container (of the same or a different kind). For example:

lib/moveitor.cpp

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

template<typename T>
void print(const std::string& name, const T& coll)
{
  std::cout << name << " (" << coll.size() << " elems): ";
  for (const auto& elem : coll) {
    std::cout << " \"" << elem << "\"";
  }
  std::cout << "\n";
}

int main()
{
  std::list<std::string> src{"don't", "vote", "for", "liars"};

  // move all elements from the list to the vector:
  std::vector<std::string> vec{std::make_move_iterator(src.begin()),
                               std::make_move_iterator(src.end())};

  print("src", src);
  print("vec", vec);
}
```

The program has the following output: (? signaling that we do not know the value):

```
src (4 elems):  "?" "?" "?" "?"
vec (4 elems):  "don't" "vote" "for" "liars"
```

Note again that the number of elements in the source container did not change. We moved all elements to the initialized new container. Therefore, the elements in the source range are in a moved-from state afterwards and we do not know their values.

14.4 Summary

- The C++ standard library provides special algorithms to move multiple elements, called `std::move()` (same name as the `std::move()` to mark a single object as movable) and `std::move_backward()`. They leave elements in a moved-from state.
- Removing algorithms can leave elements in a moved-from state.
- Move iterators allow us to use move semantics when iterating over elements. You can use these iterators in algorithms or other places such as constructors where ranges are used to initialize/set values. However, ensure that the iterators use each element only once.

Chapter 15

Move Semantics in Types of the C++ Standard Library

This chapter discusses the most important or remarkable applications of move semantics in the C++ standard library.

It might help you to better understand basic types such as strings and containers and give you some insights into tricky use of move semantics.

15.1 Move Semantics for Strings

Strings are objects that might allocate memory to hold their values. For that reason, you can benefit from using move semantics for them.

We have already had several examples in the book about how strings support move semantics:

- 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()`

In this section, we will present the impact of move semantics on string in more detail.

15.1.1 String Assignments and Capacity

The capacity of strings (the memory currently available for the value) does usually not shrink. Only move operations, `swap()`, or `shrink_to_fit()` might shrink 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 extraordinarily 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{"quite a 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";
}
```

In this program, 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 has at least size necessary to store the value. Therefore, after

```
std::string s0;
std::string s1{"short"};
std::string s2{"a string with an extraordinarily long value"};
```

we might get something like one of the following outputs:

- Platform A:

```
- s0 capa: 15 ('')
  s1 capa: 15 ('short')
  s2 capa: 43 ('a string with an extraordinarily long value')
```

- Platform B:

```
- s0 capa: 22 ('')
  s1 capa: 22 ('short')
  s2 capa: 47 ('a string with an extraordinarily long value')
```

Here, both platforms seem to support SSO (for up to 15 or 22 `chars`) and the second platforms allocates memory for 47 `chars` when only memory for 43 `chars` is needed.

On all platforms, a moved-from string is usually empty. This applies even when the value is not 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:

- Platform A:

```
- s1 capa: 15 ('')
  s2 capa: 15 ('')
  s3 capa: 15 ('short')
  s4 capa: 43 ('a string with an extraordinarily long value')
```

- Platform B:

```
- s1 capa: 22 ('')
  s2 capa: 22 ('')
  s3 capa: 22 ('short')
  s4 capa: 47 ('a string with an extraordinarily long value')
```

However, note that there is no guarantee that `s1` becomes empty. As usual, the C++ standard library guarantees only that 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 extraordinarily long value"};
std::string s5{"quite a reasonable value"};
s4 = std::move(s5);
```

You can find the following outputs in practice:

- A platform swapping memory:

```
- s4 capa: 43 ('a string with an extraordinarily long value')
  s5 capa: 24 ('quite a reasonable value')
- s4 capa: 24 ('quite a reasonable value')
  s5 capa: 43 ('')
```

- A platform moving memory (after freeing the old memory):

```
- s4 capa: 47 ('a string with an extraordinarily long value')
  s5 capa: 31 ('quite a reasonable value')
- s4 capa: 31 ('quite a reasonable value')
  s5 capa: 22 ('')
```

As you can see, the capacity of `s4` usually shrinks, sometimes to the capacity of the destination, sometimes to the minimum capacity for empty strings. However, neither is guaranteed.

15.2 Move Semantics for Containers

Containers are usually objects that have to allocate memory to hold their elements. For that reason, you can benefit from using move semantics for them. However, there is one exception: `std::array<>` does not allocate memory on the heap, which means that for `std::array<>` special rules apply.

We have already had several examples in the book about how containers support move semantics. In the initial example motivating move semantics, we saw 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 doing the following:

- Copying the containers
- Assigning the containers
- Inserting elements into the container

However, there is more to say.

15.2.1 Basic Move Support for Containers as a Whole

All containers define a move constructor and move assignment operator to support 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<>`. These requirements and guarantees 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)};   // move the container
```

the C++ standard specifies constant complexity. This 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 also create a new value in the source object but that does not make much sense because it only makes the operation slower.

For vectors, providing a value in a moved-from object is even indirectly forbidden because the move constructor of `std::vector<>` guarantees never to throw:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
  public:
    ...
    vector(const vector&);              // copy constructor
    vector(vector&&) noexcept;          // move constructor
    ...
};
```

In summary, effectively we have the following guarantees for containers after using them as source in a move constructor:

- For vectors, it is essentially required that the moved-from container is empty.
- For the other containers (except `std::array<>`), being empty is not strictly required but it 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);               // move assign the container
```

the C++ standard guarantees that this operation either overwrites or destroys each element of the destination object `cont2`. This guarantees 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 from the source to the destination).

- Move element by element from the source `cont1` to the destination `cont2` and destroy all remaining elements not overwritten in the destination.

Both ways require linear complexity, which is therefore specified. However, that with this definition, just swapping the contents of the source and the destination is not allowed.

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);
    ...
};
```

This `noexcept` guarantee for the assignment operator rules out the second option to implement the move assignment as an element-by-element move. This is because in general, move operations might throw. Only the implementation that destroys old elements does not throw.[2] Therefore, when the memory is interchangeable, we have to use the first option to implement the move assignment operator.

In summary, effectively we have the following guarantees for containers after using them as source in a move assignment:

- If the memory is interchangeable (which is especially true if the default standard allocator is used), it is essentially required that the moved-from container is empty. This applies to all containers except `std::array<>`.
- Otherwise, the moved-from container is in a valid but unspecified state.

However, note that after a move assignment to itself, a container always has an unspecified but valid state.

15.2.2 Insert and Emplace Functions

All containers support moving a new element into the container.

Insert Functions

For example, vectors support move semantics by having two different implementations of `push_back()`:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
  public:
    ...
```

[1] See http://wg21.link/n4258.

[2] Destructors can technically throw but this breaks basic guarantees of the C++ standard library. That is why all destructors are `noexcept` by default, meaning that leaving a destructor with an exception will usually `std::terminate()` the program.

```
    // 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);
    ...
};
```

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.

Functions like `emplace_back()` use perfect forwarding to avoid creating copies of the passed arguments. For example, for `std::vector<>`, 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.

15.2.3 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 cannot implement move operations in a way that they move pointers to internal 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.

Therefore, 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.

However, 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 (i.e., they have a significant size if the small string optimization (SSO) is used), moving the array of strings is usually significantly faster.

You can see this with the program *lib/contmove.cpp*, which checks 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.

15.3 Move Semantics for Vocabulary Types

The C++ standard library provides a couple of vocabulary types for handling one or multiple values with value semantics (objects hold and copy their value autonomously and as a whole).

In principle, they all provide move semantics. However, some of them need special discussion and remarks.

15.3.1 Move Semantics for Pairs

`std::pair<>` is a good example of both the benefits of move semantics and the complexity it might introduce. In principle, we have only a generic data structure with two members (defined in namespace `std`):

```
template<typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
  ...
};
```

However, to support move semantics (and a couple of other tricky corner cases, such as reference members), we have the following declaration (here, I show the C++14 version, which I made a bit more readable):

```
template<typename T1, typename T2>
struct pair {
  // types of each member:
  using first_type = T1;     // same as: typedef T1 first_type
  using second_type = T2;
  // the members:
  T1 first;
  T2 second;

  // constructors:
  constexpr pair();
  constexpr pair(const T1& x, const T2& y);
  template<typename U, typename V> constexpr pair(U&& x, V&& y);
  pair(const pair&) = default;
  pair(pair&&) = default;
  template<typename U, typename V> constexpr pair(const pair<U, V>& p);
  template<typename U, typename V> constexpr pair(pair<U, V>&& p);
  template<typename... Args1, typename... Args2>
    pair(piecewise_construct_t, tuple<Args1...> first_args,
                                tuple<Args2...> second_args);
  // assignments:
  pair& operator=(const pair& p);
  pair& operator=(pair&& p) noexcept(...);
  template<typename U, typename V> pair& operator=(const pair<U, V>& p);
  template<typename U, typename V> pair& operator=(pair<U, V>&& p);

  // other:
  void swap(pair& p) noexcept(...);
};
```

As you can see, the class supports move semantics. We have a defaulted move constructor and an implemented move assignment operator (which has a corresponding `noexcept` condition not to throw if both member types guarantee not to throw):[3]

```
template<typename T1, typename T2>
struct pair {
  ...
  pair(pair&&) = default;
  ...
  pair& operator=(pair&& p) noexcept(...);
  ...
};
```

Therefore, the following code:

```
std::pair<std::string, std::string> p1{"some value", "some other value"};
auto p2{p1};
auto p3{std::move(p1)};

std::cout << "p1: " << p1.first << '/' << p1.second << '\n';
std::cout << "p2: " << p2.first << '/' << p2.second << '\n';
std::cout << "p3: " << p3.first << '/' << p3.second << '\n';
```

has the following output:

```
p1: /
p2: some value/some other value
p3: some value/some other value
```

However, type `std::pair<>` also supports perfect forwarding by dealing with universal/forwarding references:

```
template<typename T1, typename T2>
struct pair {
  ...
  template<typename U, typename V> constexpr pair(U&& x, V&& y);
  ...
};
```

Therefore, we can use move semantics when initializing a pair. For example:

```
int val = 42;
std::string s1{"value of s1"};
std::pair<std::string, std::string> p4{std::to_string(val), std::move(s1)};

std::cout << "s1: " << s1 << '\n';
std::cout << "p4: " << p4.first << '/' << p4.second << '\n';
```

[3] The assignment operator cannot be defaulted because of special handling that is necessary when member types are references.

has the following output:

```
s1:
p4: 42/value of s1
```

The corresponding member template is implemented as you would expect for universal/forwarding references:

```
template<typename U, typename V>
constexpr pair::pair(U&& x, V&& y)
 : first(std::forward<U>(x)), second(std::forward<V>(y)) {
}
```

Note that the universal/forwarding references also mean that we can create and assign pairs of different types provided a corresponding type conversion is defined. For example:

```
std::pair<const char*, std::string> p5{"answer", "is 42"};
auto p6{std::move(p5)};

std::cout << "p5: " << p5.first << '/' << p5.second << '\n';
std::cout << "p6: " << p6.first << '/' << p6.second << '\n';
```

has the following output:

```
p5: answer/
p6: answer/is 42
```

When initializing `p6`, we convert the `first` member of `p5` (declared as `const char*`) to a `std::string`, while using move semantics when using the `second` member of `p5` to initialize the second member of `p6`.

Finally, note that `std::pair<>` supports members that have reference types. In this case, special rules apply when using `std::move()` for these members. See *basics/members.cpp* for a complete example.

`std::make_pair()`

`std::pair<>` comes with a convenience function template `std::make_pair<>()` for creating pairs without having to specify the types of the members:

```
auto p{std::make_pair(42, "hello")};  // creates std::pair<int, const char*>
```

`std::make_pair<>()` is a good example to demonstrate one additional thing you have to take into account when using move semantics with rvalue and universal/forwarding references. Its declaration has changed through the different versions of the C++ standard:

- In the first C++ standard, C++98, `make_pair<>()` was declared inside namespace `std` using call-by-reference to avoid unnecessary copying:

  ```
  template<typename T1, typename T2>
  pair<T1,T2> make_pair (const T1& a, const T2& b)
  {
    return pair<T1,T2>(a,b);
  }
  ```

  This, however, almost immediately caused significant problems when using pairs of string literals or raw arrays. For example, when `"hello"` was passed as the second argument, the type of the corresponding

parameter `b` became a reference to an array of `const chars` (`const char(&)[6]`). Therefore, type `char[6]` was deduced as type `T2` and used as type of the second member. However, initializing an array member with an array is not possible because you cannot copy arrays.

In this case the *decayed* type should be used as member type, which is the type you get when you would pass the argument by value (`const char*` for string literals).

- As a consequence, with C++03, the function definition was changed to use call-by-value:[4]

```
template<typename T1, typename T2>
pair<T1,T2> make_pair (T1 a, T2 b)
{
  return pair<T1,T2>(a,b);
}
```

As you can read in the rationale for the issue resolution, "*it appeared that this was a much smaller change to the standard than the other two suggestions, and any efficiency concerns were more than offset by the advantages of the solution*."

- With C++11, `make_pair()` had to support move semantics, which meant that the arguments had to become universal/forwarding references. Again, we have the problem that for references the type of the arguments does not decay. Therefore, the definition changed as follows:

```
template<typename T1, typename T2>
constexpr pair<typename decay<T1>::type, typename decay<T2>::type>
make_pair (T1&& a, T2&& b)
{
   return pair<typename decay<T1>::type,
               typename decay<T2>::type>(forward<T1>(a),
                                         forward<T2>(b));
}
```

which, since C++14, can be written as follows:

```
template<typename T1, typename T2>
constexpr pair<decay_t<T1>, decay_t<T2>>
make_pair (T1&& a, T2&& b)
{
   return pair<decay_t<T1>, decay_t<T2>>(forward<T1>(a), forward<T2>(b));
}
```

The real implementation is even more complex since C++11: to support `std::ref()` and `std::cref()`, the function also unwraps instances of `std::reference_wrapper<>`.

The C++ standard library perfectly forwards passed arguments in many places in a similar way, often combined with using `std::decay<>`.

[4] See the C++ standard library issue http://wg21.link/lwg181 for details.

15.3.2 Move Semantics for `std::optional<>`

`std::optional<>` is a value type available since C++17 that extends the possible values of a *contained type* by the value "*there is no value*." That avoids the need to mark one specific value of the type to have this semantics (such as the value 0 for pointers).

Optional objects also support move semantics. If you move the object as a whole, the state is copied and the *contained object* (if there is one) is moved. As a result, a moved-from object still has the same state but any value becomes unspecified.

However, you can also move a value into or out of the contained object. For example:

```
std::optional<std::string> os;
std::string s = "a very very very long string";
os = std::move(s);                  // OK, moves
std::string s2 = *os;               // OK, copies
std::string s3 = std::move(*os);    // OK, moves
```

Note that after the last call, `os` still has a string value but as usual for moved-from objects, the value is unspecified. Thus, you can use it as long as you do not make any assumption about its value. You can even assign a new string value there.

Note also that some overloads ensure that temporary optionals are moved. Consider a function that returns an optional string:

```
std::optional<std::string> func();
```

In this case, the following is well-defined to move the value:

```
std::string s4 = func().value();    // OK, moves
std::string s5 = *func();           // OK, moves
```

This behavior is possible by using reference qualifiers that provide rvalue overloads for the corresponding member functions:

```
namespace std {
  template<typename T>
  class optional {
    ...
    constexpr T& operator*() &;
    constexpr const T& operator*() const&;
    constexpr T&& operator*() &&;
    constexpr const T&& operator*() const&&;

    constexpr T& value() &;
    constexpr const T& value() const&;
    constexpr T&& value() &&;
    constexpr const T&& value() const&&;
  };
}
```

By using reference qualifiers, the class can return the moved value when the operation is called for rvalues (temporary objects or objects marked with `std::move()`):

```
std::vector<std::string> coll;
std::optional<std::string> optStr;
...
coll.push_back(std::move(optStr).value());   // OK, moves from member into coll
```

Note that class `std::optional<>` is one of the rare places in the C++ standard library where `const` rvalue references are used. The reason is that `std::optional<>` is a wrapper type that wants to ensure that the operations do the right thing even when `const` objects are marked with `std::move()` and the contained type provides special behavior for `const` rvalue references.

15.4 Move Semantics for Smart Pointers

While raw pointers do not 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 support only move semantics because copying a unique pointer is not possible.

15.4.1 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" for the owned object is called.

For example:

```
{
  std::shared_ptr<int> sp1;                 // init shared pointer that does not own anything
  {
    auto sp2{std::make_shared<int>(42)};    // init shared pointer that owns new int
    ...
    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`, which means that afterwards, both shared pointers own the object. However, note that copying the ownership is a pretty expensive operation. This is because 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

Furthermore, 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.

Therefore, it is 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
  ...
}
```

With regard to 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 that does not own anything
while (...) {
  auto ptr{std::make_shared<int>(getValue())};   // init shared pointer that owns new int
  ...
  lastPtr = ptr;                           // expensive (note: ptr no longer used)
}                                          // ptr destroyed, lastPtr is the only owner
```

it would be better to implement this:

```
std::shared_ptr<int> lastPtr;              // init shared pointer that does not own anything
while (...) {
  auto ptr{std::make_shared<int>(getValue())};   // init shared pointer that owns new int
  ...
  lastPtr = std::move(ptr);                // cheap
}                                          // ptr destroyed, lastPtr is the only owner
```

As a consequence, objects with shared pointer members lose the ownership of the resource these members point to when the objects are moved. This is good for 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<>`.

15.4.2 Move Semantics for `std::unique_ptr<>`

The class template `std::unique_ptr<>` implements the concept of exclusive ownership. The type system ensures that there can be only one owner of an object at any one time. The trick is to use the type system to disable any attempt to copy a unique pointer. Because this check is done at compile time this approach does not introduce any significant performance overhead at runtime.

You can create a single unique pointer in a source function and return it to the caller:

lib/uniqueptr1.cpp

```
#include <iostream>
#include <string>
#include <memory>
```

```
std::unique_ptr<std::string> source()
{
   static long id{0};

   // create string with new and let ptr own it:
   auto ptr = std::make_unique<std::string>("obj" + std::to_string(++id));
   ...
   return ptr;          // transfer ownership to caller
}

int main()
{
   std::unique_ptr<std::string> p;
   for (int i = 0; i < 10; ++i) {
      p = source();     // p gets ownership of the returned object
                        // (previously returned object of source() is deleted)
      std::cout << *p << '\n';
      ...
   }
} // last-owned object of p is deleted
```

As usual for moved-from types, to pass ownership to a sink function you can only pass it with `std::move()`:

```
std::vector<std::unique_ptr<std::string>> coll;
std::unique_ptr<std::string> up;
...
coll.push_back(up);              // ERROR: copying disabled
coll.push_back(std::move(up));   // OK, moves ownership into new element of coll
```

If you pass a unique pointer to a potential sink function that takes the argument by reference, you do not know whether the ownership was moved. Whether the ownership was moved depends on the implementation of the function. In that case, you can double-check the state with `operator bool()`:

```
std::unique_ptr<std::string> up;
...
sink(std::move(up));   // might move ownership to sink()
if (up) {              // does it still have the ownership?
  ...
}
```

Alternatively, you might ensure that the ownership is gone (resource released):

```
std::unique_ptr<std::string> up;
...
sink(std::move(up));   // might move ownership to sink()
up.reset();            // ensure ownership is gone (resource deleted)
```

15.5 Move Semantics for IOStreams

IOStreams were introduced with C++98 as an abstraction for something you can read from or write to (standard I/O, files, and even strings). It was an early design decision that it is not possible to copy these objects (what would it mean to copy an object representing an open file, having two handles for the same files or copying the file, and how do you synchronize access?).

However, since C++11, move semantics allows us to move IOStream objects around and to use temporary streams.

15.5.1 Moving IOStream Objects

Consider the following example:

lib/outfile.cpp

```
#include <iostream>
#include <fstream>
#include <stream>

std::ofstream openToWrite(const std::string& name)
{
  std::ofstream file(name);           // open a file to write to
  if (!file) {
    std::cerr << "can't open file '" << name << "'\n";
    std::exit(EXIT_FAILURE);
  }
  return file;                        // return ownership (open file)
}

void storeData(std::ofstream fstrm)   // takes ownership of file (but this might change)
{
  fstrm << 42 << '\n';
}                                     // closes the file

int main()
{
  auto outFile{openToWrite("iostream.tmp")};  // open file
  storeData(std::move(outFile));              // store data

  // better ensure that the file is closed:
  if (outFile.is_open()) {
    outFile.close();
  }
}
```

Here, the function `openToWrite()` opens and returns an output file stream:

```
std::ofstream openToWrite(const std::string& name)
{
  std::ofstream file(name);              // open a file to write to
  ...
  return file;                           // return ownership (open file)
}
```

We use the return value to initialize `outFile` and pass it to `storeData()`:

```
auto outFile{openToWrite("iostream.tmp")};   // open file
storeData(std::move(outFile));               // store data
```

Because `storeData()` takes the argument by value, it takes the ownership of the open file. Therefore, at the end of `storeData()`, the file is closed:

```
void storeData(std::ofstream fstrm)    // takes ownership of file (but this might change)
{
  ...
}                                      // closes the file
```

However, `storeData()` might also take the argument by reference, which means that it does not necessarily take the ownership. In that case, you might want to double-check the state of the passed argument `outFile` afterwards:

```
// better ensure that the file is closed:
if (!outFile.is_closed()) {
  outFile.close();
}
```

Calling `outFile.close()` is usually enough but would set the failbit of the file stream if it was already closed.

15.5.2 Using Temporary IOStreams

Since C++11, the IOStreams library also provides function overloads to take rvalue references, which allows us to take temporary objects. For example:

```
std::string s = "hello, world";
std::ofstream("fstream1.tmp") << s << '\n';          // OK since C++11
```

You can even write a string literal to a stream that way (this compiled before C++11 but wrote the address of the string literals using `operator<<(const void*)`):

```
std::ofstream("fstream1.tmp") << "hello, world\n";   // correct since C++11
                                                    // (wrote address before)
```

In the same way, you can parse a given string using a temporary string stream:

```
std::string name, firstname, lastname;
...
name = "Tina Turner";
std::istringstream{name} >> firstname >> lastname;   // OK since C++11
```

Finally, you can use `std::getline()` to parse the first line from a temporary stream:

```
std::string multiLineString, firstLine;
...
std::getline(std::stringstream{multiLineString},        // read from temporary string stream
             firstLine);
```

15.6 Move Semantics for Multithreading

The C++ standard library has a couple of special types that represent running threads or are used to synchronize them. Some of them you can neither copy nor move (e.g., atomic types or condition variables). However, some of them are move-only types. Let us look at some details of them from the perspective of move semantics.

15.6.1 `std::thread<>` and `std::jthread<>`

Objects representing running threads (`std::thread` or since C++20 `std::jthread`)[5] can be passed around but not copied. That allows us to put them in containers.

Consider the following example:

lib/thread.cpp

```
#include <iostream>
#include <thread>
#include <vector>

void doThis(const std::string& arg) {
  std::cout << "doThis(): " << arg << '\n';
}
void doThat(const std::string& arg) {
  std::cout << "doThat(): " << arg << '\n';
}

int main()
{
  std::vector<std::thread> threads;    // better std::jthread since C++20

  // start a couple of threads:
  std::string someArg{"Black Lives Matter"};
  threads.push_back(std::thread{doThis, someArg});
```

[5] `std::jthread` fixes some flaws of class `std::thread` by becoming a RAII type (no further need to call `join()` or `detach()` for the thread before the destructor is called) and supporting a cooperative mechanism to signal cancellation of the thread. It is always better to use `std::jthread` instead of `std::thread`.

```
  threads.push_back(std::thread{doThat, std::move(someArg)});
  ...

  // wait for all threads to end:
  for (auto& t : threads) {
    t.join();
  }
}
```

Here, we start two threads and put them into a collection of all running threads:

```
std::vector<std::thread> threads;    // better std::jthread since C++20

std::string someArg{"black lives matter"};
threads.push_back(std::thread{doThis, someArg});
threads.push_back(std::thread{doThat, std::move(someArg)});
```

We start a thread by declaring an object of type `std::thread` (or better, `std::jthread` since C++20) and move the temporary object into `threads`. Behind the argument for the callable (function, lambda, function object), we can pass additional arguments that are passed to the callable when the thread starts. By default, the constructor of the thread class copies these arguments. With `std::move()`, we switch to move semantics here. Therefore, `doThis()` gets a copy of the passed string `someArg`, while `doThat()` gets the moved value of the string.

At the end, we wait for the end of all threads (which is necessary to avoid a core dump when `std::thread` is used):

```
// wait for all threads to end:
for (auto& t : threads) {
  t.join();
}
```

15.6.2 Futures, Promises, and Packaged Tasks

For some helper types for synchronizing the exchange of a (return) value between two threads, also move-only types are used (futures, promises, and packaged tasks are move-only).

Consider the following example:

lib/future.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <thread>
#include <future>

void getValue(std::promise<std::string> p)
{
```

```
  try {
    std::string ret{"vote"};
    ...
    // store result:
    p.set_value_at_thread_exit(std::move(ret));
  }
  catch(...) {
    // store exception:
    p.set_exception_at_thread_exit(std::current_exception());
  }
}

int main()
{
  std::vector<std::future<std::string>> results;

  // create promise and future to deal with outcome of the thread started:
  std::promise<std::string> p;
  std::future<std::string> f{p.get_future()};
  results.push_back(std::move(f));

  // start thread and move the promise to it:
  std::thread t{getValue, move(p)};
  t.detach();  // would not be necessary for std::jthread
  ...

  // wait for all threads to end:
  for (auto& fut : results) {
    std::cout << fut.get() << '\n';
  }
}
```

Here, in `main()`, we first create a promise to be able to send an outcome (value or exception) to the associated future that can read it:

```
std::promise<std::string> p;
std::future<std::string> f{p.get_future()};
```

The future is moved into a collection of handles for futures:

```
results.push_back(std::move(f));
```

We could also pass the result of `p.get_future()` directly to `push_back()`:

```
results.push_back(p.get_future());
```

The promise is moved to the thread started:

```
std::thread t{getValue, move(p)};
```

The thread takes it as an argument for the passed callable `getValue()`, which takes the passed promise by value:

```
void getValue(std::promise<std::string> p)
{
  ...
}
```

This way, `getValue()` receives ownership of the promise and destroys this source of the communication mechanism at the end of the thread.

`getValue()` could also take the promise by reference because the thread started holds the moved value of the promise until it ends.

15.7 Summary

We have learned a lot of individual lessons from the types discussed in this chapter but let me summarize here a couple of the more general aspects of using move semantics with types of the C++ standard library.

- Moved-from standard strings are usually empty but that is not guaranteed.
- Moved-from standard containers (except `std::array<>`) are usually empty if no special allocator is used. For vectors, this is (indirectly) guaranteed; for the other containers, it is indirectly guaranteed that all elements are moved away or destroyed and inserting new members would make no sense.
- Move assignments can change the capacity of strings and vectors.
- To support decaying when passing values to universal/forwarding references (often necessary to deduce the same type for string literals of different length), use the type trait `std::decay<>`.
- Generic wrapper types should use reference qualifiers to overload member functions that access the wrapped objects. This might even mean using overloads for `const` rvalue references.
- Avoid copying of shared pointers (e.g., by passing them by value).
- Use `std::jthread` (available since C++20) instead of `std::thread`.

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 usually have the suffix `.cpp`. However, 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 a 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 rather 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 produce 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 usually have a suffix such as `.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, struct, or unscoped enumeration type that is declared but not defined, an array of unknown size, `void` (optionally with `const` and/or `volatile`), or an array of an incomplete element type.

L

lvalue
A category of expressions that produce a location for a stored value that is not assumed to be movable (i.e., *glvalues* that are not *xvalues*). Typical examples are expressions that denote named objects, string literals, and (references to) functions. 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. If we already have a local object that we return in a function by value, the compiler can use the object directly as a return value instead of copying or moving 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 that is created in the `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 temporary object directly as a return value instead of copying or moving 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 object without name) or an *xvalue* (e.g., an *lvalue* marked with `std::move()`). What was called an *rvalue* before C++11 is called a *prvalue* since 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 string 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 quite an 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 initializes objects. Additional categories subdivide *glvalues* into *lvalues* (localizable values) and *xvalues* (eXpiring values). In addition, since C++11, *rvalues* serve as a general category for both *xvalues* and *prvalues* (before C++11, *rvalues* were what *prvalues* are since 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

`&` 30
    `const` 30
`&&` 30
    `const` 31
`...` xvii

A

about the book xv
abstract base class 75
    check for `noexcept` move constructor 120
algorithm
    `move()` 199
    `move_backward()` 199
    `remove()` 201
    `unique()` 201
    with `move_iterator` 205, 207
alternative syntax
    for universal references 169
`array<>`
    move semantics 216
`assert()`
    invalid moved-from state 97
assignment
    move to itself 30
    self-move 45
assignment operator
    and move semantics 38
    copy assignment **56**
    implementing copying 43
    implementing moving 44
    move assignment **56**
    with reference qualifier 84
`auto&`
    for functions 130
`auto&&` **173**
    for functions 175
    in range-based `for` loop 176
    `string` literals 175
    type deduction 174

B

binding references 133
brace initialization xvi

C

C++03 3
C++11 11
C++14 11
C++98 3
call-by-reference 30
    decay 219
call-by-value 31
    versus call-by-reference 195
class
    broken move 39
    consistent members 100
    invalid 94
    member with disabled move semantics 54

moved-from members 97
moved-from state 89
move semantics 35
pointer members 102
reference members 102
class hierarchies 75
and `noexcept` 120
check for `noexcept` move constructor of base class 120
collapsing rule for references 163
compiler options 61
consistent members 100
`const`
and `std::move()` 22
check for universal reference 156
return value 23
rvalue reference 31, 221
universal reference 153
`const&` 30
`const&&` 31
constructor
copy **55**
implementing copying 41
implementing moving 42
member initialization 61
move **55**
with universal reference 150
container
emplace functions 215
inserting 214
move assignment guarantees 213
move constructor guarantees 213
move-only elements 194
move semantics 212
copy as a fallback 21
copy assignment
and `noexcept` 118
deleted 194
implementation 43
copy assignment operator **56**
and move semantics 38
generated 56
copy constructor **55**
and move semantics 38
and `noexcept` 118
deleted 194
for strings 19
generated 55
implementation 41
not best match 150
CPP file **231**
curly braces xvi
cv-unqualified 163

D

data member
and `std::move()` 131
value category 131
`decay<>`
for references 219
`decltype` 136
check value category 137
`decltype(auto)` **184**
deferred returning 187
lambda 189
deduction
of template parameters 163, 166
of type `auto&&` 174
`=default` 39
move constructor 51
default constructor 57
deferred perfect returning 187
in lambdas 189
`=delete`
move constructor 52
destructor 57
and move semantics 38, 93
and `noexcept` 119
disable
move semantics 53, 104
universal reference 151

E

ellipsis xvii
email to the author xviii
`emplace_back()` 214
implementation 215
emplace functions 215
`enable_if<>` 151

for universal reference 158
ERROR xvii
exception
noexcept 107
strong guarantee 109
explicit specification
universal references 165
expression
check value category 137
decltype 137

F

fallback copying 21
for loop
and references 80
universal references 176
forward<>() 145
calling member functions 146
header file 145
versus move() 164
forwarding
details 153
perfect 141
forwarding reference 141, **231**, see universal reference
alternative syntax 169
auto&& 173
check for constness 156
check for value category 157
const 144
explicit specification 165
in constructor 150
not forwarding 153, 176
of specific type 157
overload resolution 149
rvalue reference parameter 144
to rvalues only 167
type deduction 163
vs. universal reference 168
fstream 225
temporary 226
full specialization **231**
function
as universal reference 165
auto& 130
auto&& 175
value category 130
function template
full specialization 161
future 228

G

gcc
warnings on move() 61
getline() 29, 60
temporary stream 226
getter 79
glossary **231**
glvalue 127, **231**
guarantees of moved-from objects 89
guards xvii

H

header file **232**
for std::forward<>() 145
for std::move() 27
guards xvii
Herb Sutter
on move-only types 195
hierarchies of classes 75

I

if constexpr
check for constness 156
check for value category 157
ifstream 225
temporary 226
implicit conversions 135
include file **232**
incomplete type **232**
inheritance
and noexcept 116
initialization xvi
initialize members 61
initializer_list<>
move-only types 194
insert() 214

inserting functions 214
invalid state 94, 97
invariant 92
IOStreams
  move semantics 225
is_nothrow_movable<> 120
is_nothrow_move_constructible<> 112, 120
istream 225
  temporary 226
iterator
  move_iterator 204

J

jthread 97, 227

L

lambda
  decltype(auto) 189
  deferred prefect returning 189
  perfect forwarding 151, 180
  perfect returning 189
  return type 189
  template parameters 151
lvalue 125, **232**
  check for 138

M

make_move_iterator() 204
make_pair() 219
materialization 129
member
  and std::move() 130
  for moved-from state 202
  initialization 61
  invalid moved-from state 97
  pointers 102
  references 102
  value category 130
  with disabled move semantics 54
member function
  called with std::forward<>() 146
  called with std::move() 82
  with reference qualifier 86
member type
  reference 159, 160
Meyers
  on move-only types 195
motivation 3
move() 27
  and const 22
  as static_cast 28
  calling member functions 82
  change value category 135
  compiler warnings 61
  for functions 130
  for members 130
  header file 27
  impossible to avoid 60
  in return statement 81
  moved-from objects 28
  reuse the object 29
  self-move 45
  to itself 30
  valid but unspecified state 28, 92
  versus forward<>() 164
  with disabled move semantics 54
move() algorithm 199
move assignment
  and noexcept 118
  broken 39
  container guarantees 213
  generated 56
  implementation 44
  operator **56**
  self-move 45
  to itself 30
move_backward() algorithm 199
move constructor **55**
  and noexcept 118
  broken 39
  container guarantees 213
  =default 51
  =delete 52
  for strings 20
  generated 55
  implementation 42
  noexcept 107, 110

not best match 150
moved-from members 97
moved-from objects 28
guarantees 89
invalid 89
requirements 89
reuse 29
moved-from state 89
member 202
move-only types 196
move iterator 204
move-only
moved-from state 196
threads 227
move-only type **193**
declaration 194
in containers 194
`initializer_list<>` 194
move-only types
sink function 195
move semantics
and destructor 93
and `emplace_back()` 214, 215
disable 53, 104
for containers 212
for getters 79
for IOStreams 225
for smart pointers 222
for std::array<> 216
for std::`shared_ptr<>` 222
for std::`unique_ptr<>` 223
for strings 209
initialization 70
performance on initialization 70

N

name
and `decltype` 136
named objects 59
named return value optimization (NRVO) **232**
`noexcept` 107
and inheritance 116
and special member functions 117
check for move constructor of base class 120
move constructor 110
usage 122
violation 112
NRVO 232

O

`ofstream` 225
temporary 226
`optional<>` 221
`ostream` 225
temporary 226
overloading
by reference and by value 134
on reference qualifiers 79
references 30
overload resolution
with rvalue references 133
with universal references 149

P

packaged task 228
`pair<>` 217
`make_pair()` 219
parameter
by value 31
perfect forwarding 141
rvalue reference 26
partially formed 90
pass-by-reference 30
value categories 133
pass-by-value 31
versus pass-by-reference 73, 195
passing
perfect 171
pass through move semantics 42
perfect forwarding 141
`auto&&` 175
details 153
`emplace_back()` 215
lambdas 151, 180
std::pair<> 218
variadic template 143

perfect passing 171
perfect returning 183
  deferred 187
  lambdas 189
performance
  of initialization with move semantics 70
pointer
  as member 102
  move semantics 222
polymorphic classes 75
preprocessor guards error xvii
promise 228
proxy type 179
prvalue 127, **232**
  check for 138
`push_back()` 18, 214

R

range-based `for` loop
  and references 80
  universal references 176
reallocation
  `noexcept` 107
recursive call
  with rvalue reference 135
reference
  and value categories 133
  as member 102
  binding 133
  decay 219
  forwarding 144
  overloading 30
  overload resolution 133, 149
  proxy type 179
  rvalue reference 25
  universal 144
reference collapsing rule 163
reference member
  and `std::move()` 132
  value category 132
reference qualifier 79
  for assignment operator 84
  for member function 86
reference to function
  as universal reference 165
  `auto&` 130
  `auto&&` 175
  value category 130
`remove()` algorithm 201
requirements of moved-from objects 89
`requires` 151
resolution
  with rvalue references 133
  with universal references 149
return
  by reference 80
  by value 79
  `const` 23
  overload 81
return perfectly 183
return value optimization (RVO) **233**
reuse moved objects 29
rule of five 57
rule of five or three 58
rule of three 57
rule of zero 97
runtime error xvii
rvalue 125, **233**
  check for 138
  passed to generic code 167
rvalue reference 25
  and member types 159, 160
  as parameter 26
  `const` 31, 221
  decay 219
  for member initialization 66
  for move-only types 195
  generic 144
  in full specializations 161
  name 133
  overload resolution 133
  recursive call 135
  term 133
  value category 135
RVO 233

S

Scott Meyers

on move-only types 195
self-move 30, 45
shared_ptr<>
as member 102
invalid moved-from state 102
move semantics 222
sink function
for move-only types 195
slicing 76
small/short string optimization 210
small/short string optimization (SSO) **233**
smart pointer
as member 102
move semantics 222
sorting algorithms 29
special member function 48, 49
specific type
universal reference 157
SSO 210, 233
state
invalid 89, 97
invariant 92
partially formed 90
static_assert() 114
static_cast<>
change value category 136
std::move() 28
static member
and std::move() 132
value category 132
std::forward<>() 145, see forward<>()
std::move() 27, see move()
and const 22
as static_cast 28
calling member functions 82
change value category 135
compiler warnings 61
for data members 131, 132
for functions 130
for members 130
for reference members 132
for static members 132
header file 27
impossible to avoid 60
reuse the object 29
unnecessary 60
stream
temporary 226
string
copy constructor 19
move constructor 20
move semantics 209
string literal
as universal reference 164
auto&& 175
string stream
temporary 226
strong exception safety guarantee 109
Sutter
on move-only types 195
swap() 29, 90
noexcept 122
syntax
for universal references 169

T

template
explicit specification and universal references 165
template parameter
for lambdas 151
template parameter deduction
for universal references 166
temploid 160
temporary
stream 226
terminology **231**
thread 94, 227
translation unit **233**
twice processing a value 60
type
universal reference 157
type deduction
of type auto&& 174
type trait
std::decay<> 219
suffix _t 157
suffix _v 157

U

uniform initialization xvi
unique() algorithm 201
unique_ptr<>
    move semantics 223
universal reference 141, **233**
    alternative syntax 169
    and member types 159, 160
    auto&& 173
    check for constness 156
    check for value category 157
    const 144, 153
    explicit specification 165
    for move-only types 195
    in constructor 150
    in full specializations 161
    in range-based for loop 176
    not forwarding 153, 176
    of specific type 157
    overload resolution 149
    rvalue reference parameter 144
    template parameter deduction 166
    to references to functions 165
    to rvalues only 167
    to string literals 164
    type deduction 163
    vs. forwarding reference 168
unnecessary std::move() 60
use of move semantics 59

V

valid but unspecified state 28, 89, **92**
value category 125, **234**
    change with static_cast<> 136
    change with std::move() 135
    check 137, 138
    check for universal reference 157
    decltype 137
    history 125
    implicit conversions 135
    of members 130
    of references to functions 130
    rvalue reference 135
variadic template **234**
    perfect forwarding 143
vector
    move assignment guarantees 213
    move constructor guarantees 213
    move-only elements 194
    push_back() 18
    reallocation and noexcept 107
virtual functions 75
vocabulary types 216

W

wide contract 122

X

xvalue 127, **234**
    check for 138