

C++ Move Semantics

The Complete Guide

作者: Nicolai M. Josuttis

译者: 陈晓伟

本书概述

完整的介绍 C++ 移动语义。

C++11 添加的移动语义已经成为现代 C++ 的标志，也使语言变得复杂，即使经验丰富的开发者仍需要仔细处理。所以，一些编程书籍甚至不推荐对非常简单的类使用移动语义。因此，详细的解释 C++ 移动语义就变得刻不容缓。

本书会从基本原理开始来介绍移动语义，并解释移动语义的细节，使每个开发者都可以正确地使用移动语义。

你将学习到:

- 移动语义的起因和术语
- 如何隐式地获益于移动语义
- 如何显式地获益于移动语义
- 会遇到的问题，以及如何处理它们
- 所有的结果都取决于你的编程风格

重点在于所描述的特性，需要在实践中进行应用。示例和背景信息，有助于理解和改进简单类，甚至泛型库和框架的代码。

“我以为我理解了移动的语义，其实我真的不懂！我从你的书中学到了很多东西。” – Jonathan Boccaro

“这是我非常需要的书。” – Rob Bernstein

“有时候我觉得我对纠缠和量子隐形传态的理解，要比我对一些奇怪的 C++ 移动语义的理解要好。套用 Feynman 的话：如果你认为你理解了 C++ 的移动语义，那你就不理解 C++ 的移动语义。赶快阅读这本书吧。” – Victor Ciura

作者简介

Nicolai Josuttis (<http://www.josuttis.com>) 在编程界很有名，其发言和著作都很有权威，还是世界范围内畅销书的(共同)作者：

- 《The C++ Standard Library》
- 《C++ Templates》
- 《C++ Move Semantics》
- 《C++17》
- 《SOA in Practice》

同时也是一位富有创新精神的演讲者，曾在各种会议和活动中发言。还是独立的讲师，并且在 C++ 标准化方面有 20 多年的经验。

本书相关

- github 翻译地址：<https://github.com/xiaoweiChen/CPP-Move-Semantics>

目录

前言

目前为止，每当教授移动语义时，我都会说，“必须有人写一本关于移动的书，”通常的回答是：“那必须的的！你来吧！”。现在，我终于做到了。

和往常一样，写一本关于 C++ 的书时，我都会对要介绍的知识、要阐明的情况和要描述的结果感到惊讶。是时候出一本关于 Move 语义的书了，要覆盖了从 C++11 到 C++20 的所有版本。我在这个过程中学到了很多，相信你也一样。

本书就是实验

本书完成了实验的两个方面：

- 写一本有深度的书，会涉及复杂的核心语言特性，还没有核心语言专家的直接帮助。不过，我可以通过问问题来写这本书。
- 我自己在 Leanpub 上出版这本书，并按需印刷。这本书是逐步完成的，有了改进，就会发布一个新的版本。

好的方面是：

- 可以从经验丰富的编程人员那里了解语言特性——他们体会到某个特性可能造成的痛苦，并提出相关问题，以便能够激励和解释设计，以及其在实践中编程的结果。
- 当我还在写作时，可以阅读我的移动语义经验，并进行借鉴。
- 这本书和所有的读者都能从您的早期反馈中受益。

这意味着你也是实验的一部分。所以，请帮助我：对本书的缺陷、错误、未解释清楚的功能给予反馈，这样大家都能从这些改进中获益。

致谢

首先，我要感谢 C++ 社区，是你们使这本书成为可能。移动语义的功能的设计，有用的反馈，他们的好奇心是语言进化的基础。特别感谢那些告诉我和解释的所有问题的人们，以及给我的反馈，谢谢你们。

我要特别感谢每一个为这本书或审阅草稿并提供有价值的反馈的人。这些评论大大提高了这本书的质量，再次证明了好东西需要许多“聪明人”的投入。目前为止（这个列表还在增长），感谢你们：Javier Estrada, Howard Hinnant, Klaus Iglberger, Daniel Krugler, Marc Mutz, Aleksandr Solovev (alexolut), Peter Sommerlad 和 Tony Van Eerd。

此外，我还要感谢 C++ 社区和 C++ 标准委员会的每一个人。除了所有涉及添加新语言和库功能的工作外，这些专家还花了很多很多的时间和我解释和讨论他们的工作，他们很有耐心，也很有热情。

特别感谢 LaTeX 社区提供的文本系统，感谢 Frank Mittelbach 解决了我的 LATEX 问题。

最后，非常感谢本书校对，Tracey Duffy，她做了大量的工作，把我的“德式英语”翻译成地道的英语。

关于本书

这本书详解 C++ 的移动语义。从基本原理出发，解释了移动语义的所有特性和案例，这样就可以正确地理解和使用移动语义。

本书重点在于所描述的特性，需要在实践中进行应用。示例和背景信息，有助于理解和改进简单的类型，甚至是泛型库和框架的代码。

阅前须知

需要您熟悉 C++，应该熟悉类和引用的一般概念，并且应该能使用 C++ 标准库的 `iostream` 和容器等组件来编写 C++ 程序。还应该熟悉“现代 C++”的基本特性，比如：`auto` 或基于范围的 `for` 循环。

我的目标是让普通 C++ 程序员能够理解这些内容，他们不一定知道最新特性的所有细节。我将讨论基本特性，并在需要时回顾更细节的问题。

这也确保了专家和中级程序员都也可以阅读本书。

本书结构

这本书涵盖了 C++ 的移动语义，以及 C++20 的各个方面。这既适用于语言和库特性，也适用于日常应用程序编程的特性和（基础）库复杂实现的特性。

不同的章节是分组的，所以应该从头到尾的阅读。后面的章节通常依赖于前面章节介绍的特性，但交叉引用在特定的后续主题中也有帮助，并且会指出在何处引用前面介绍的特性。

本书包含以下部分：

- **Part I** 介绍了移动语义的基本特性（特别是非泛型代码）。
- **Part II** 介绍了泛型代码（特别是在模板和泛型 Lambda 中使用的）的移动语义的特性。
- **Part III** 包含在 C++ 标准库中移动语义的使用（也给出了一个在实践中如何使用移动语义的例子）。

如何阅读

不要畏惧这本书的页数。当你研究细节（比如实现模板）时，事情会变得非常复杂。对于基本的理解，本书的前三分之一（第一部分，特别是第 1-5 章）就足够了。

以我的经验，学习新东西的最好方法是看例子。因此，你会在书中找到很多例子。有些只是说明抽象概念的几行代码，有些会提供具体应用的完整程序。后一种示例将通过描述程序代码的文件的 C++ 注释来介绍。你可以在这本书的网站 <http://www.cppmove.com> 上找到这些文件。

实现方式

关于我编写代码和注释的方式，请注意以下提示。

初始化

我通常使用带大括号的现代初始化形式（C++11 中引入的统一初始化）：

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

这种形式的初始化称为大括号初始化，有以下优点：

- 可以与基本类型、类类型、聚合、枚举类型和 auto 一起使用
- 可以用于初始化带多个值的容器
- 可以检测截断错误（例如，用浮点值初始化 int）
- 不能与函数声明或调用混淆

如果大括号为空，则调用（子）对象的默认构造函数，并保证使用 0/false/nullptr 初始化基本数据类型。

错误描述

我会经常谈论编程错误。如果没有特殊提示，错误或注释为：

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

表示编译时错误。对应的代码不应该编译（使用符合标准的编译器）。

如果使用运行时错误，程序会编译通过，但不能正确运行或导致未定义的行为发生（因此，可能会执行预期的操作）。

代码简化

我试图用有用的例子来解释所有的特征。然而，为了专注于关键方面，我可能经常会跳过一部分代码。

- 大多数时候，使用省略号（“...”）来表示其他代码。注意，这里没有使用代码。如果看到带有代码字体的省略号，那么代码必须有这三个点作为语言特性（例如“typename...”）。
- 头文件中，我通常跳过预处理器保护。所有头文件应该有类似如下的内容：

```
1 #ifndef MYFILE_HPP  
2 #define MYFILE_HPP  
3 ...  
4 #endif // MYFILE_HPP  
5
```

因此，在项目中使用这些头文件时，请补全代码。

C++ 标准

不同的 C++ 标准定义了不同的 C++ 版本。

最初的 C++ 标准出版于 1998 年，随后在 2003 年通过勘误表进行了修订，对原来的标准进行了微小的修正。“旧 C++ 标准”即为 C++98 或 C++03 标准。

“现代 C++”始于 C++11，并在 C++14 和 C++17 中扩展。国际 C++ 标准委员会现在的目标是每三年发布一个新标准。显然，这就减少了进行大规模添加的时间，可以更快地为编程社区带来变化。

因此，开发更大的特性需要时间，而且可能涉及多个标准。下一个“更现代的 C++”已经在地平线上，如 C++20 所介绍的那样。同样，编程的方式可能会改变，但编译器需要一些时间来提供最新的语言特性。在撰写本书时，C++17 是主流编译器支持的最新标准。

幸运的是，在 C++11 和 C++14 中都介绍了移动语义的基本原理。因此，本书中的代码示例，通常可以在主流编译器的最新版本上编译。如果讨论 C++17 或 C++20 引入的特殊特性，我会进行说明。

示例代码的信息

你可以访问所有的示例程序，并从它的网站上找到更多关于这本书的信息，它的网址如下：

<http://www.cppmove.com>

反馈

欢迎消极的和积极的建设性意见，希望你会发现这是一本优秀的书。然而，有时我停止写作、审查和调整，以便“发布新版本”。因此，你会发现错误、不一致、可以改进的示例或缺失的主题。你们的反馈给了我解决这些问题的机会，我会通过这本书的网站告知所有读者这些变化，并改进后续的修订或版本。

联系我最好的方式是通过电子邮件。你可以在这本书的网站上找到电子邮件地址：

<http://www.cppmove.com>

如果你使用电子书，你可能想要确保有这本书的最新版本（记住它是逐步编写和出版的）。在提交报告之前，您还应该检查该书的 Web 站点以获取当前已知的勘误表。无论如何，在反馈意见时，请参考本版本的发布日期。目前出版日期为 2020 年 12 月 19 日（也可以在第二页，封面后的下一页找到）。

非常感谢！

1 Part I: 移动语义的基本特征

本书的这部分介绍了非泛型编程的移动语义的基本特性。对应用程序的编程有帮助，因此使用现代 C++ 的开发者都应该了解。

泛型编程的移动语义特性将在 Part II 部分中介绍。

1 移动语义的力量

本章通过一个简短的代码示例，演示了移动语义的基本原理。

1.1 设计移动语义的动机

为了了解移动语义的基本原则，使用小段代码为例，首先没有移动语义（例如，用一个旧的 C++ 编译器编译，仅支持 C++03），之后再用移动语义（用现代 C++ 编译器编译，支持 C++11 或更高的版本）。

1.1.1 C++03 的示例（使用移动语义之前）

假设我们有以下代码：

basics/motiv03.cpp

```
1 #include <string>
2 #include <vector>
3
4 std::vector<std::string> createAndInsert()
5 {
6     std::vector<std::string> coll; // create vector of strings
7     coll.reserve(3); // reserve memory for 3 elements
8     std::string s = "data"; // create string object
9
10    coll.push_back(s); // insert string object
11    coll.push_back(s+s); // insert temporary string
12    coll.push_back(s); // insert string
13
14    return coll; // return vector of strings
15 }
16
17 int main()
18 {
19     std::vector<std::string> v; // create empty vector of strings
20     ...
21     v = createAndInsert(); // assign returned vector of strings
22     ...
23 }
```

使用不支持移动语义的 C++ 编译器编译这段代码后，让我们看一下程序执行的各个步骤（检查堆栈和堆）。

- 首先，在 main() 中，我们创建了空的 vector v :

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

存放在堆栈上，元素数为 0，未为元素分配内存。

- 然后，调用

$v = createAndInsert();$

在堆栈上创建另一个 vector $coll$ ，并在堆上为三个元素预留内存：

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

因为分配的内存没有初始化，所以元素的数量仍然是 0。

- 然后，用“data” 初始化字符串：

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

字符串类似于具有 `char` 元素的 `vector`。其实是在堆栈上创建一个对象，其中一个成员表示字符的数量（值为 4），另一个指针表示字符的内存。

之后，程序的状态如下：堆栈上有三个对象：`v`、`coll` 和 `s`。其中 `coll` 和 `s` 已经在堆上分配了内存：

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

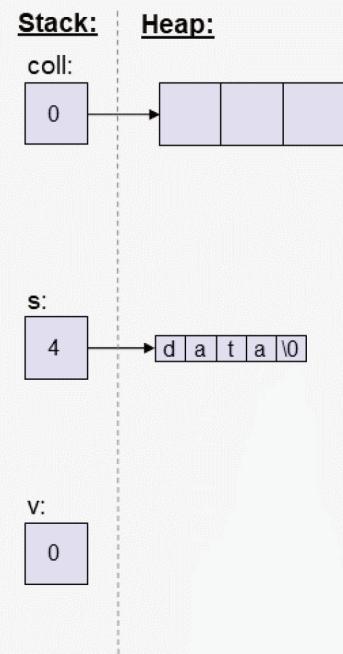
    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();
```



- 下一步是将字符串放到 `coll` 中：

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

C++ 标准库中的所有容器都具有语义，能创建副本。所以，`vector` 有了第一个元素，它是 `s` 的副本：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

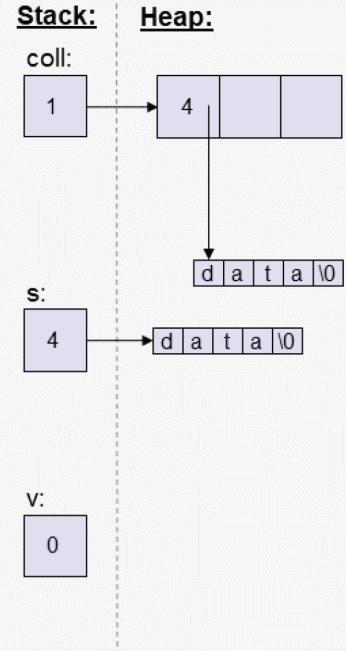
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



目前为止，我们还没有对这个程序进行优化。现在，我们有两个 `vector`, `v` 和 `coll`, 以及两个字符串, `s` 和它的副本, 这也是 `coll` 的第一个元素。它们都是独立的对象, 有自己的内存。

- 现在让我们看看下一句, 创建了一个新的临时字符串, 并再次将其插入 `vector` 中:

`coll.push_back(s+s);`

该语句分三个步骤执行:

1. 创建临时字符串 `s+s`:

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

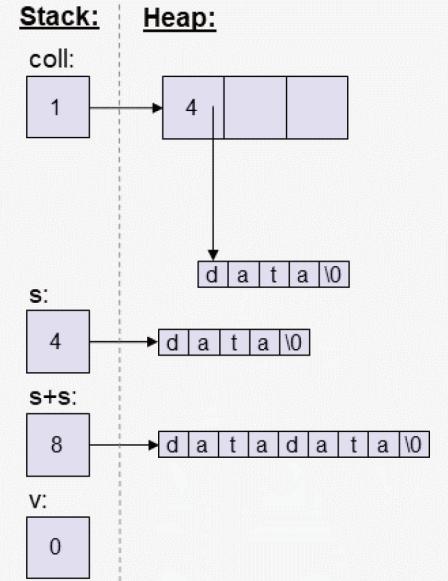
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



- 将这个临时字符串插入到 `coll` 中。容器创建副本，创建临时字符串的深层副本，包括分配内存：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

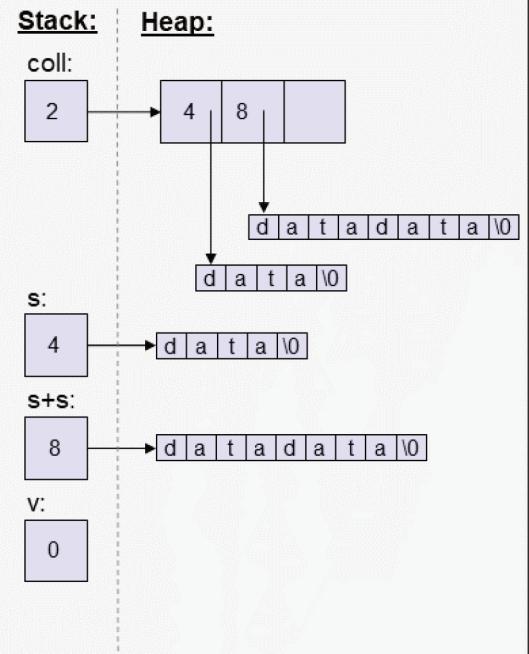
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



3. 末尾，临时字符串 $s+s$ 销毁:

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

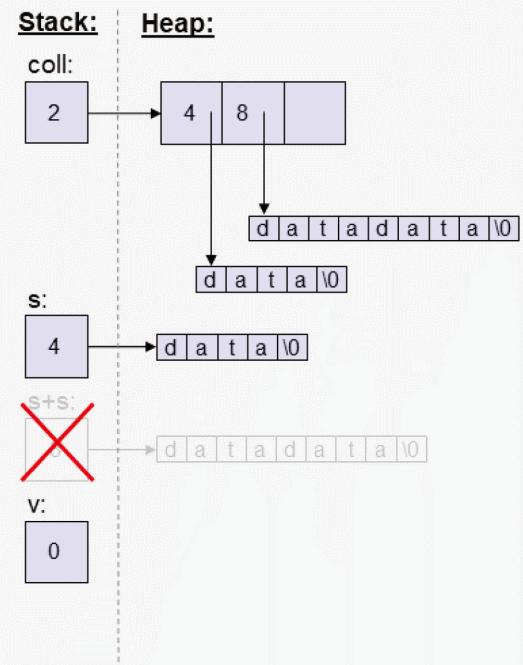
    coll.push_back(s);

    coll.push_back(s+s); // 错误：创建临时字符串的副本后销毁临时字符串

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();
```



这是我们第一次生成代码执行不大合适的地方：创建临时字符串的副本后销毁临时字符串。其实，可以不必分配和释放内存，直接使用原始的临时字符串对象。

- 下一行中，我们再将 `s` 插入 `coll` 中：

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

`coll` 继续拷贝 `s`:

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

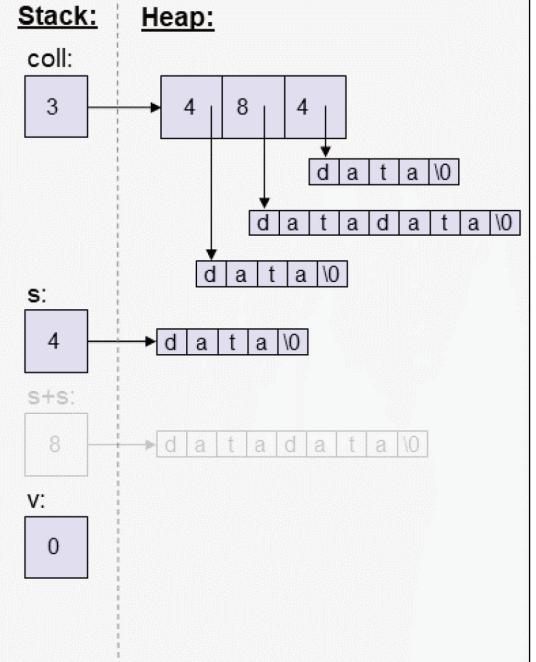
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



这也是需要优化：一些优化技巧可以使用 `s` 作为 `vector` 的新元素。

- 在 `createAndInsert()` 的末尾，使用 `return`:

```

1     return coll;
2 }
3

```

程序的行为变得有点复杂。按值返回（返回类型不是引用）的应该是 `coll` 的副本。创建副本意味着必须创建整个 `vector`，及其所有元素的副本。因此，必须为 `vector` 中的元素数组分配堆内存，并为每个字符串分配用于保存其值的值分配堆内存。这样，我们需要分配 4 次内存。由于 `coll` 被销毁，编译器可以执行命名返回值优化 (NRVO)。这意味着编译器可以生成代码，使 `coll` 只是用作返回值。

可以使用这种优化，不过这可能会改变程序的行为。如果在 `vector` 或 `string` 对象的复制构造函数中有 `print` 语句，则会不看 `print` 语句的再次输出，这意味着这种优化改变了程序的功能。然而，这是可以的，因为在 C++ 标准中允许这种优化，即使它有副作用。任何人都不应该知道这里有一个副本，是否执行命名返回值优化仅取决于编译器。

假设我们进行了命名返回值优化，在 `return` 语句的末尾，`coll` 现在成为返回值，并调用 `s` 的析构函数，释放声明 `s` 时分配的内存：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

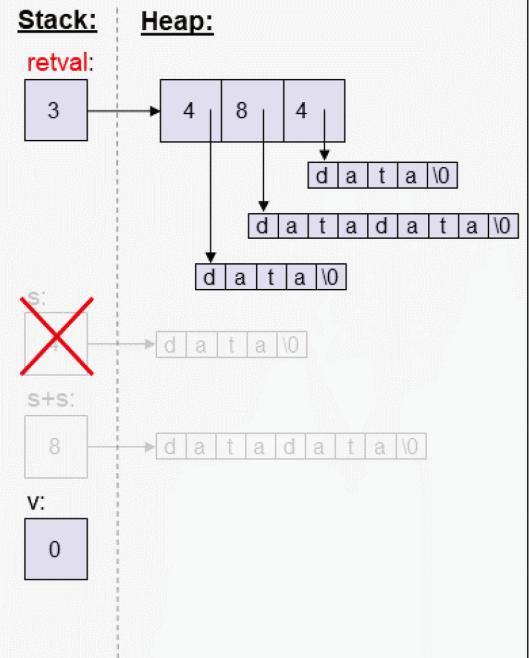
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



- 最后，将返回值赋给 *v*:

v = *createAndInsert()*;

这里可以改进的行为: 通常的赋值操作符的目标是赋予 *v* 与赋值源的值相同的值。一般来说, 任何值都不应该修改, 并且应该独立于赋值的对象。因此, 赋值操作符将创建了一个返回值的副本:

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

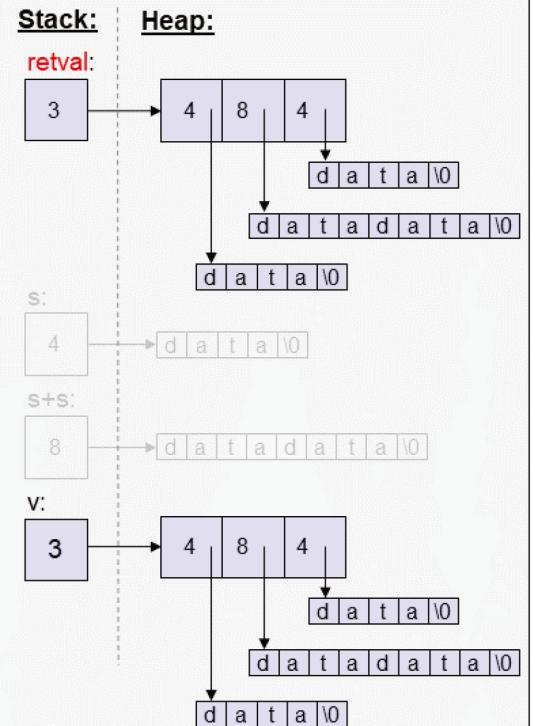
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



之后，就不再需要临时返回值了，可以销毁：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

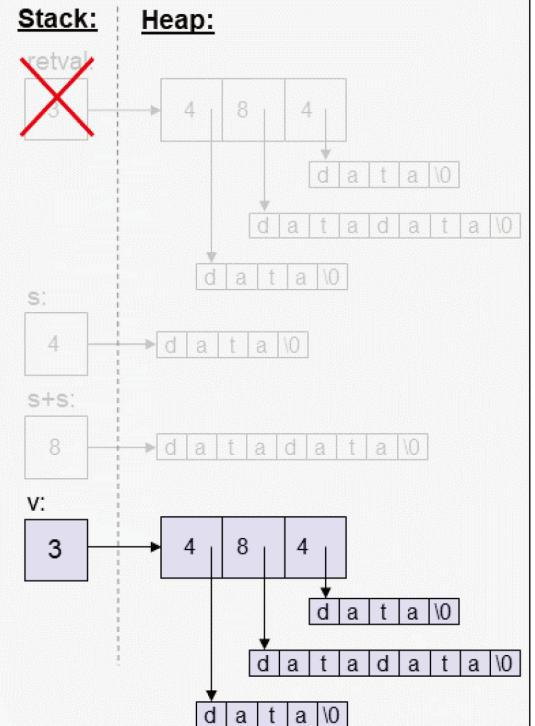
    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



我们创建了一个临时对象的副本，然后立即销毁该副本的源，再次分配和释放内存。这次适

用于四个分配，一个分配给 vector，其余分配给每个 string 元素。

对于 `main()` 中赋值后的状态，分配了 10 次内存，释放了 6 次。不必要的内存分配由以下原因引起：

- 将临时对象插入容器
- 将对象插入到不再需要该值的容器中
- 将一个临时 vector 的值赋于其他对象

我们可以或多或少地避免这些性能损失。我们可以这样做：

- 将 vector 作为 out 参数传递：

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

- 使用 `swap()`：

`createAndInsert().swap(v);`

但是，得到的代码看起来更难看懂（除非您在复杂的代码中看到一些美），而且在插入临时对象时没有真正的解决问题。

自 C++11 起，我们有了另一个选择：编译并运行支持移动语义的程序。

1.1.2 使用 C++11 的例子（使用移动语义）

现在让我们用一个支持移动语义的现代 C++ 编译器（C++11 或更高版本）重新编译这个程序：

[basics/motiv11.cpp](#)

```
1 #include <string>
2 #include <vector>
3
4 std::vector<std::string> createAndInsert()
5 {
6     std::vector<std::string> coll; // create vector of strings
7     coll.reserve(3); // reserve memory for 3 elements
8     std::string s = "data"; // create string object
9
10    coll.push_back(s); // insert string object
11    coll.push_back(s+s); // insert temporary string
12    coll.push_back(std::move(s)); // insert string (we no longer need the value of s)
13
14    return coll; // return vector of strings
15 }
16
17 int main()
18 {
19     std::vector<std::string> v; // create empty vector of strings
20     ...
21     v = createAndInsert(); // assign returned vector of strings
22     ...
23 }
```

这里有一个小修改：将最后一个元素插入 `coll` 时，添加了 `std::move()` 的调用，我们将在讨论这个语句时聊聊这个更改。其他都和之前一样。

让我们再次通过检查堆栈和堆来了解程序的各个步骤。

- 首先，在 `main()` 中创建空 `vector v`，包含 0 个元素：

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

- 之后，调用：

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

在堆栈上创建另一个空 `vector coll`，并保留三个元素未初始化的内存：

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

```
coll.reserve(3);
```

- 然后，我们创建用“`data`”初始化的字符串 `s`，并再次将其插入 `coll` 中：

```
std::string s = "data"; coll.push_back(s);
```

目前为止，还没有什么需要优化的，程序状态与 C++03 相同：

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

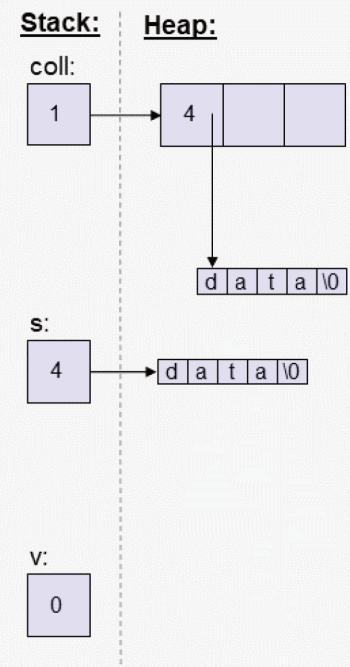
    coll.push_back(s);

    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();
```



我们有两个 `vector`，`v` 和 `coll`，还有两个字符串，`s` 和它的副本，也就是 `coll` 的第一个元素。它们都是独立的对象，有自己的内存。

- 首先，看看创建一个新的临时字符串并将其插入 `vector` 中的语句：

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

同样，该语句分三个步骤执行：

1. 创建临时字符串 `s+s`:

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

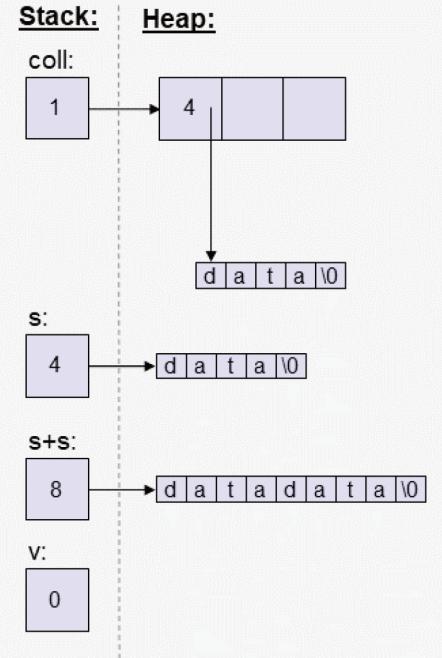
    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



- 将这个临时字符串插入到 `coll` 中。然而，这里发生了一些不同的事情：我们从 `s+s` 中窃取了内存，并将其移动到 `coll` 的新元素中。

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

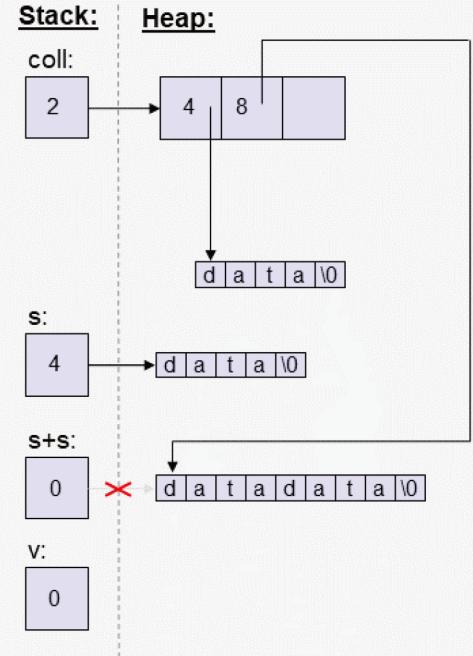
    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



自 C++11 起，我们可以获取不再需要的值。因为编译器知道在执行 `push_back()` 调用之后，临时对象 `s+s` 将被销毁。因此，调用的是 `push_back()` 的另一个实现，其是对

不再需要值的字符串进行复制的优化：我们复制了大小和指向内存的指针，而不是创建拷贝。然而，这种浅复制是不够的；还修改了临时对象 `s+s`，将其大小设置为 0，并将 `nullptr` 赋值为新值。其实，`s+s` 被修改了，以便呈现空字符串的状态。重要的是，它不再拥有自己的内存。

- 语句的结尾，临时字符串 `s+s` 被销毁。但由于临时字符串不再是初始内存的所有者，析构函数不会释放该内存。

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

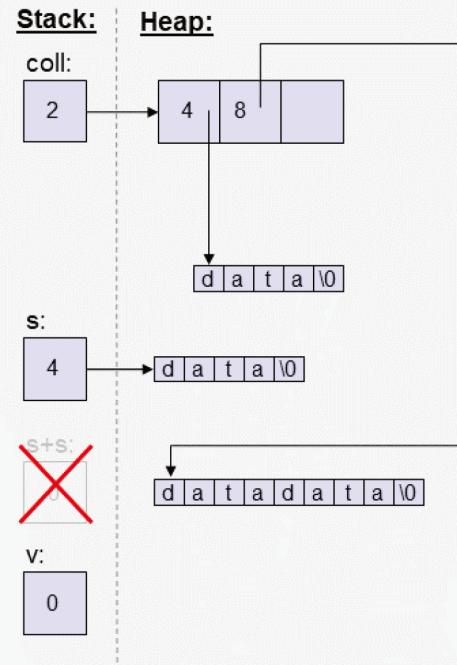
    coll.push_back(s);

    coll.push_back(s+s); // Optimized copy

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();
```



我们优化了复制过程，将 `s+s` 的内存所有权转移到它在 `vector` 中的副本。

这一切都是通过编译器自动完成的，编译器可以对即将死亡的对象发出信号，可以使用新实现来复制一个从源端窃取值的字符串值。这不是一个技巧，这是一种语义上的移动，通过从底层技术上将值的内存从源字符串移动到它的副本实现。

- 下一条语句是我们为 C++11 版本修改的语句。将 `s` 插入到 `coll` 中，但是通过对要插入的字符串 `s` 调用 `std::move()`，语句发生了改变：

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

如果没有 `std::move()`，就会像第一次调用 `push_back()` 一样：`vector` 会创建传递的字符串 `s` 的副本。这次，我们用 `std::move()` 标记了 `s`，这意味着“这里不再需要这个值”。因此，这里会使用另一个 `push_back()` 实现，会在传递临时对象 `s+s` 时使用。第三个元素通过将这个值的内存所有权，从 `s` 转移到副本中，从而来窃取这个值：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

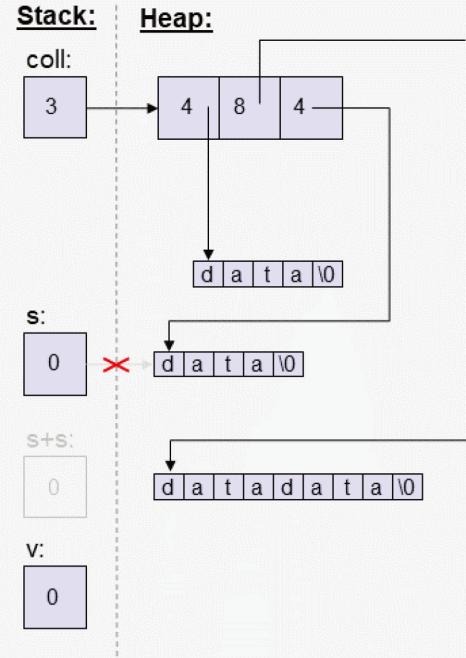
    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



关于移动语义，请注意以下两点：

- `std::move(s)` 在这里只表示 `s` 是可移动的，只表示不再需要这个值，允许实现通过在复制值时进行一些优化（比如偷取内存）从而获益。调用者并不知道值是否进行了移动。
- 然而，窃取值的优化必须确保源对象仍然处于有效状态。被移动的对象既不会部分销毁，也不会完全销毁。C++ 标准库为它的类型进行了规定：标记为 `std::move()` 的对象执行操作后，该对象处于有效，但未定义的状态。

以上，就是在这条语句执行完成后会发生的事

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

这里保证 `s` 仍然是有效的字符串。这就像使用不知道传递了哪个值的字符串参数一样。

注意，它也不能保证字符串要么有旧值，要么为空，由运行时库实现者决定。通常，实现者可以对 `std::move()` 操作的对象做任何事，只要保持对象的有效状态。保证的理由，稍后讨论。

- `createAndInsert()` 最后的 `return` 语句为：

```

1   return coll;
2 }
3

```

是否生成具有指定返回值优化的代码取决于编译器，`coll` 只是成为了返回值。但是，如果不使用这种优化，`return` 语句仍然是从即将失效的源创建对象的情况。如果没有使用指定的返回值优化，仍然会使用移动语义，返回值会从 `coll` 中窃取值。最坏的情况下，必须从源文件复制成员的大小、容量和指向内存的指针（总共 12 或 24 个字节）到返回值中，并在源文件中为这些成员赋值。

我们假设进行了返回值优化。在 return 语句中，*coll* 成为返回值，并调用 *s* 的析构函数，不再需要释放任何内存，因为内存移到了 *coll* 的第三个元素中：

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

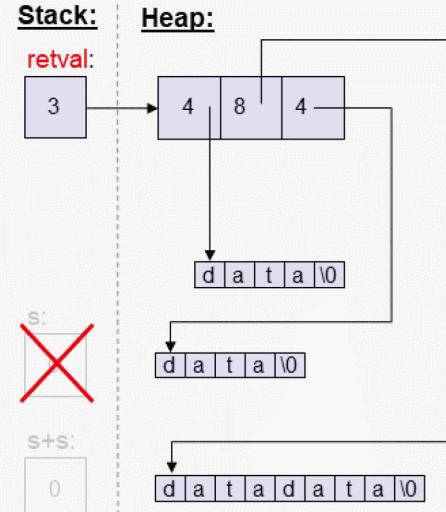
    coll.push_back(s);

    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();
```



- 最后，来给 *v* 赋值：

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

同样可以从移动语义中获益：必须从一个即将销毁的临时返回值中复制（这里是赋值）一个值。

现在，移动语义允许从源 vector 转移值的赋值操作符：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

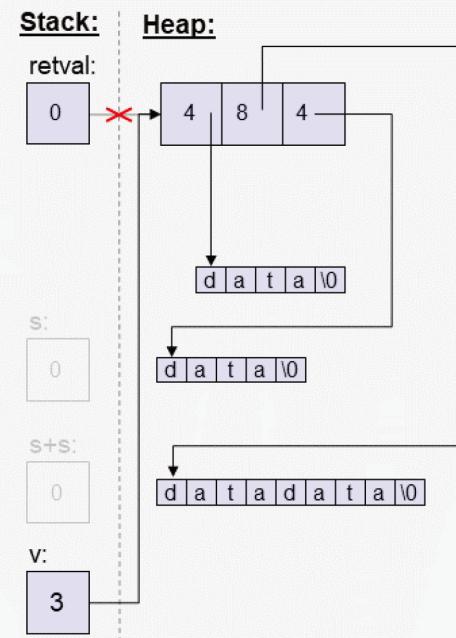
    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



同样，临时对象不会（部分地）销毁。它进入了某个状态，但我们不知道它的值。
但在赋值之后，语句的末尾会销毁（修改过的）临时返回值：

```

std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s = "data";

    coll.push_back(s);

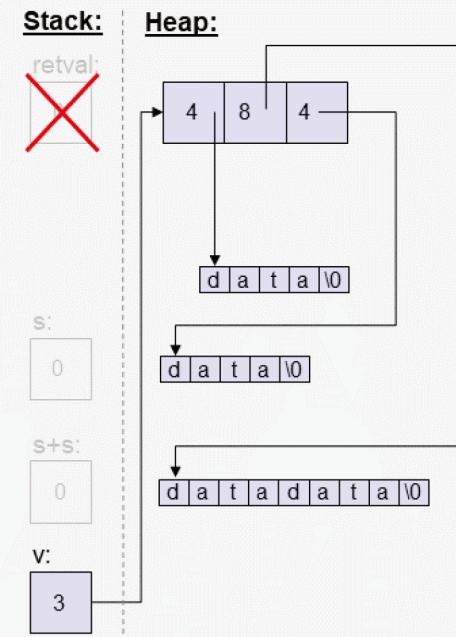
    coll.push_back(s+s);

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

    return coll;
}

std::vector<std::string> v;
...
v = createAndInsert();

```



最后，和之前使用移动语义时一样，但有一些事情发生了改变：节省了 6 个内存分配和释放。
所有不必要的内存分配不再发生：

- 将临时对象移动插入容器
- 当使用 `std::move()` 表示不再需要该值时，用于将对象移动插入容器
- 临时 vector 及其元素的分配方式

第二种情况下，优化是在我们的帮助下完成的。通过添加 `std::move()`，说明不再需要 `s` 的值。所有其他优化都生效了，因为编译器知道对象即将销毁，所以编译器也可以使用移动语义的优化实现。

这意味着返回字符串向量，并将其赋值给现有向量不再有性能问题。我们可以像使用整型一样使用字符串向量，从而获得更好的性能。在实践中，用移动语义重新编译代码可以将速度提高 10% 到 40%(取决于现有代码的难易程度)。

1.2 实现移动语义

用前面的例子来了解移动语义是如何实现的。

实现移动语义之前，`std::vector<T>` 只有一个 `push_back()` 的实现 (这里简化了 `vector` 的声明):

```

1 template<typename T>
2 class vector {
3     public:
4     ...
5     // insert a copy of elem:
6     void push_back (const T& elem);
7     ...
8 };

```

传递参数给 `push_back()` 的方法只有一种：将其绑定到 `const` 引用上。`push_back()` 的实现方式是在不修改参数的情况下，创建参数的副本。

C++11 开始，`push_back()` 有了重载：

```

1 template<typename T>
2 class vector {
3     public:
4     ...
5     // insert a copy of elem:
6     void push_back (const T& elem);
7     // insert elem when the value of elem is no longer needed:
8     void push_back (T&& elem);
9     ...
10 };

```

第二个 `push_back()` 使用了为移动语义引入的新语法。使用两个 `&` 声明实参，不使用 `const`，这样的参数称为**右值引用**。只有一个 `&` 的“普通引用”现在称为**左值引用**。在这两个调用中，都传递了通过引用插入的值。但是，区别如下：

- 使用 `push_back(const T&)`，可以保证不修改传递的值。
当调用者仍然需要传递的值时，将调用此函数。
- 通过 `push_back(T&&)`，可以修改传递的实参 (因此它不是 `const`) 来“窃取”值。语义上的含义是，新元素接收传递的值，但可以使用优化实现，将值移动到 `vector` 中。

当调用者不再需要传递的值时，将调用此函数。实现必须确保传递的参数仍然处于有效状态，但值可以更改。因此，在调用此参数之后，调用者仍然可以使用传递的参数，只要调用者不对其值做任何操作。

但 *vector* 不知道如何复制或移动元素。在确保 *vector* 有足够的内存容纳新元素之后，*vector* 将工作委托给元素的类型。

在本例中，元素是字符串。如果我们复制或移动传递的字符串会发生什么。

1.2.1 使用复制构造函数

用于传统复制语义的 *push_back(const T&)* 调用 *string* 类的复制构造函数，该构造函数初始化 *vector* 中的新元素。一个非常简单的 *string* 类实现的复制构造函数，看起来是这样的：

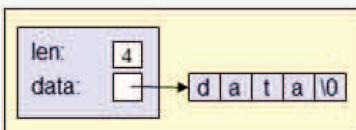
```
1 class string {
2     private:
3         int len; // current number of characters
4         char* data; // dynamic array of characters
5     public:
6         // copy constructor: create a full copy of s:
7         string (const string& s)
8             : len(s.len) { // copy number of characters
9                 if (len > 0) { // if not empty
10                     data = new char[len+1]; // - allocate new memory
11                     memcpy(data, s.data, len+1); // - and copy the characters
12                 }
13             }
14             ...
15 };
```

假设我们对值为"data" 的字符串使用这个复制构造函数：

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

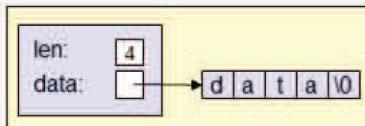
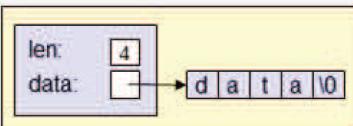
初始化字符串 a 后，如下所示：

```
string a = "data";
```



上面的复制构造函数将复制成员 *len* 以获取字符数，为数据指针分配新的内存，并将源 *a*(作为 *s* 传递) 中的所有字符复制到新字符串：

```
string a = "data";           string b = a;
```



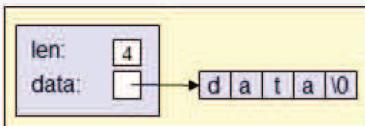
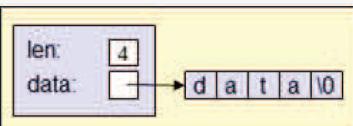
1.2.2 使用移动构造函数

对于移动语义, `push_back(T&&)` 调用相应的构造函数, 即移动构造函数。从现有字符串创建新字符串的构造函数, 和移动语义一样, 构造函数使用非 `const` 右值引用 (`&&`) 作为形参:

```
1 class string {
2     private:
3         int len; // current number of characters
4         char* data; // dynamic array of characters
5     public:
6     ...
7     // move constructor: initialize the new string from s (stealing the value):
8     string (string&& s)
9     : len{s.len}, data{s.data} { // copy number of characters and pointer to memory
10        s.data = nullptr; // release the memory for the source value
11        s.len = 0; // and adjust number of characters accordingly
12    }
13    ...
14};
```

给定上述拷贝构造函数的情况:

```
string a = "data";           string b = a;
```

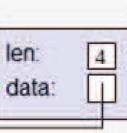
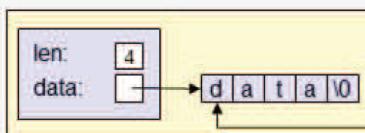
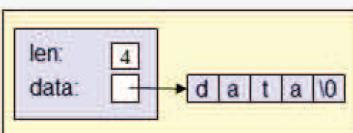


我们可以这样调用 `string` 对象的构造函数:

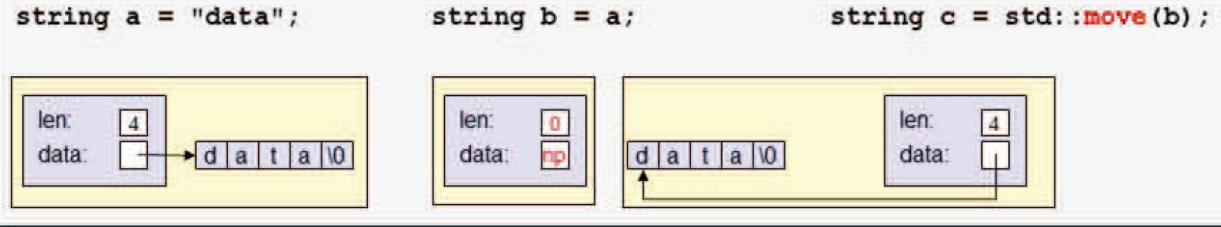
```
1 std::string c = std::move(b); // init c with the value of b (no longer needing its
2 value here)
```

移动构造函数首先复制成员 `len` 和 `data`, 这意味着新字符串获得 `b` 值的所有权 (作为 `s` 传递)。

```
string a = "data";           string b = a;           string c = std::move(b);
```



这是不够的，因为 *b* 的析构函数将释放内存。因此，我们还可以修改源字符串，使其失去内存的所有权，并使其表示空字符串：



结果是，*c* 现在有了 *b* 的值，而且 *b* 是空字符串。注意，唯一的保证是 *b* 随后处于有效但未定义的状态。根据 C++ 库中移动构造函数的实现方式，可能不为空（但通常是空的，因为这是提高性能的最简单和最好的方法）。

1.3 复制是一种应急方式

可以通过移动语义，移动临时对象或用 *std::move()* 标记的对象，可以通过“窃取”源值来优化值的复制（采用非 *const* 右值引用）。但是，如果没有针对移动语义的优化版本，则使用复制。

例如，像 *vector* 这样的容器类缺少 *push_back()* 的重载：

```
1 template<typename T>
2 class MyVector {
3     public:
4     ...
5     void push_back (const T& elem); // insert a copy of elem
6     ... // no other push_back() declared
7 };
```

仍然可以传递一个临时对象或使用 *std::move()*：

```
1 MyVector<std::string> coll;
2 std::string s{"data"};
3 ...
4 coll.push_back(std::move(s)); // OK, uses copy semantics
```

对于临时对象或使用 *std::move()* 的对象，首选是将形参声明为右值引用的函数。如果不存在这样的函数，则使用复制语义。这样，就可以确保调用者不需要知道是否存在优化。优化有可能没有，因为：

- 函数/类是在移动语义支持之前实现，或者没有考虑支持移动语义
- 没有什么需要优化的（只有数字成员的类就是一个例子）

对于泛型代码，如果不再需要一个对象的值，可以用 *std::move()*。即使没有移动语义的支持，相应的代码也会编译通过。

出于同样的原因，甚至可以用 *std::move()* 来标记基本数据类型的对象，如 *int*（或指针）。仍然会使用复制的值（地址）方式：

```

1 std::vector<int> coll;
2 int x{42};
3 ...
4 coll.push_back(std::move(x)); // OK, but copies x (std::move() has no effect)

```

1.4 *const* 对象的移动语义

最后，不能移动用 *const* 声明的对象。因为任何优化实现都要求可以修改传递的实参，如果不允许修改，就不能窃取。

使用 *push_back()* 的重载：

```

1 template<typename T>
2 class vector {
3     public:
4     ...
5     // insert a copy of elem:
6     void push_back (const T& elem);
7     // insert elem when the value of elem is no longer needed:
8     void push_back (T&& elem);
9     ...
10 };

```

对 *const* 对象使用的函数是具有 *const&* 形参的 *push_back()* 重载：

```

1 std::vector<std::string> coll;
2 const std::string s{"data"};
3 ...
4 coll.push_back(std::move(s)); // OK, calls push_back(const std::string&)

```

const 对象的 *std::move()* 没起作用。

原则上，可以通过声明带有 *const* 右值引用的函数进行重载，但在语义上没有意义。同样，*const* 左值引用会作为处理这种情况的备选。

1.4.1 *const* 返回值

const 禁用移动语义对声明返回类型也有影响。*const* 返回值不能移动。

因此，从 C++11 开始，用 *const* 返回值就不再是好的方式了（正如过去的一些风格指南所推荐的那样）。例如：

```

1 const std::string getValue();
2
3 std::vector<std::string> coll;
4 ...
5 coll.push_back(getValue()); // copies (because the return value is const)

```

当按值返回时，不要将整个返回值声明为 *const*。仅在声明部分返回类型时使用 *const*（例如：返回的引用或指针所指向的对象）：

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

1.5 总结

- 移动语义允许对对象的复制进行优化，它可以隐式使用（用于未命名的临时对象或局部返回值），也可以显式使用（通过 `std::move()`）。
- `std::move()` 表示不再需要这个值，它将对象标记为可移动的。标记为 `std::move()` 的对象不会（部分地）销毁（析构函数仍然会调用）。
- 通过使用非 `const` 右值引用（例如 `std::string&&`）声明函数，可以定义一个接口，调用者在接口中从语义上声明不再需要传递的值。函数可以通过“窃取”这个信息来进行优化，或者对传递的参数做任何修改。通常，实现者还必须确保传递的参数在调用后处于有效状态。
- 移动的 C++ 标准库的对象仍然是有效的对象，但其值为未定义。
- 拷贝语义用作移动语义的备选（如果拷贝语义支持的话）。如果没有采用右值引用的实现，则使用任何采用普通 `const` 左值引用的实现（如：`const std::string&`）。即使对象被显式地标记为 `std::move()`，也会使用备选方式。
- 对 `const` 对象调用 `std::move()` 通常没有效果。
- 如果按值（而不是按引用）返回，不要将返回值声明为 `const`。

2 移动语义的核心

了解了第一个使用移动语义的例子之后，本章会对移动语义的基本特征进行讨论。

2.1 右值引用

为了支持移动语义，C++ 引入了一种新的引用类型：右值引用。我们讨论一下这是什么，以及如何使用。

2.1.1 细节部分

右值引用使用两个 & 号声明。与普通引用一样，右值引用了一个存在的对象，该对象作为初始值传递。但根据语义，右值引用只能引用没有名称的临时对象，或使用 `std::move()` 的对象：

```
1 std :: string returnStringByValue(); // forward declaration
2 ...
3 std :: string s{ "hello" };
4 ...
5 std :: string&& r1{s}; // ERROR
6 std :: string&& r2{std :: move(s)}; // OK
7 std :: string&& r3{returnStringByValue()}; // OK, extends lifetime of return value
```

右值引用来自这样一个事实：对象通常只能引用右值，这是类别，用于没有名称的临时对象和使用 `std::move()` 的对象。

与成功初始化返回值引用一样，引用将返回值的生命周期延长到引用的生命周期结束（普通的 `const` 左值引用已经具有此行为）。

用于初始化引用的语法无关紧要。使用等号、大括号或圆括号都可以：

```
1 std :: string s{ "hello" };
2 ...
3 std :: string&& r1 = std :: move(s); // OK, rvalue reference to s
4 std :: string&& r2{std :: move(s)}; // OK, rvalue reference to s
5 std :: string&& r3(std :: move(s)); // OK, rvalue reference to s
```

所有这些引用都具有这样的语义：“只要对象的状态有效，就可以窃取/修改引用的对象。”编译器不会检查这些语义，因此可以对该类型的任何非 `const` 对象那样修改右值引用，也可能不做修改。如果对一个对象有一个右值引用，该对象可能会收到一个不同的值（可能是也可能不是一个默认构造对象的值），或者保留原始值。

移动语义允许我们使用不再需要的值进行优化。如果编译器自动检测到从生命周期结束的对象中获取值，将自动切换到移动语义：

- 传递一个临时对象的值，该对象将在语句执行后自动撤销。
- 传递一个使用 `std::move()` 的非 `const` 对象。

2.1.2 作为参数的右值引用

将形参声明为右值引用时，具有的行为和语义：

- 形参只能绑定到一个没有名称的临时对象，或者绑定到一个使用 `std::move()` 的对象。

- 根据右值引用的语义:

- 调用者不再对值感兴趣。因此，可以修改参数所引用的对象。
- 但是，调用者可能仍然对使用对象感兴趣。因此，任何修改都应该使引用的对象保持有效状态。

例如:

```

1 void foo(std::string&& rv); // takes only objects where we no longer need the value
2 ...
3 std::string s{"hello"};
4 ...
5 foo(s); // ERROR
6 foo(std::move(s)); // OK, value of s might change
7 foo(returnStringByValue()); // OK

```

可以在通过 `std::move()` 传递命名对象后使用，但通常不这样做。推荐的编程方式是不在 `std::move()` 后面使用对象:

```

1 void foo(std::string&& rv); // takes only objects where we no longer need the value
2 ...
3 std::string s{"hello"};
4 ...
5 foo(std::move(s)); // OK, value of s might change
6 std::cout << s << '\n'; // OOPS, you don't know which value is printed
7 foo(std::move(s)); // OOPS, you don't know which value is passed
8 s = "hello again"; // OK, but rarely done
9 foo(std::move(s)); // OK, value of s might change

```

对于标记为“OOPS”的两行，只要不对 `s` 的当前值有期望，技术上是可以调用的。因此，打印是可以的。

2.2 `std::move()`

如果有一个对象，当使用的时候，生存期没有结束，可以用 `std::move()` 标记它，表示“在这里不再需要这个值。”`std::move()` 不进行移动，只在使用表达式的上下文中设置了一个临时标记:

```

1 void foo1(const std::string& lr); // binds to the passed object without modifying it
2 void foo1(std::string&& rv); // binds to the passed object and might steal/modify
     the value
3 ...
4 std::string s{"hello"};
5 ...
6 foo1(s); // calls the first foo1(), s keeps its value
7 foo1(std::move(s)); // calls the second foo1(), s might lose its value

```

带有 `std::move()` 标记的对象仍然可以传递给接受普通 `const` 左值引用的函数:

```

1 void foo2(const std::string& lr); // binds to the passed object without modifying it
2 ... // no other overload of foo2()
3 std::string s{"hello"};

```

```
4 ...
5 foo2(s); // calls foo2(), s keeps its value
6 foo2(std::move(s)); // also calls foo2(), s keeps its value
```

注意，用 `std::move()` 标记的对象不能传递给非 `const` 左值引用：

```
1 void foo3(std::string&); // modifies the passed argument
2 ...
3 std::string s{"hello"};
4 ...
5 foo3(s); // OK, calls foo3()
6 foo3(std::move(s)); // ERROR: no matching foo3() declared
```

注意，用 `std::move()` 标记马上会销毁的对象是没有意义的。事实上，这甚至会对优化产生反效果。

2.2.1 `std::move()` 的头文件

`std::move()` 定义为 C++ 标准库中的一个函数。因此，使用时需要包含头文件 `<utility>`：

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

使用 `std::move()` 的程序在编译时通常不包含这个头文件，因为几乎所有的头文件都包含了 `<utility>`。但是，非标准头文件需要包含该头文件。因此，当使用 `std::move()` 时，应该显式包含 `<utility>` 以使程序可移植。

2.2.2 实现 `std::move()`

`std::move()` 只不过是对右值引用的 `static_cast`。可以手动调用 `static_cast` 来达到相同的效果，如下所示：

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

因此，我们也可以这样写：

```
1 std::string s;
2 ...
3 foo(static_cast<std::string&>(s)); // same effect as foo(std::move(s))
```

注意，`static_cast` 所做的不仅仅是改变对象的类型。还允许将对象传递给右值引用（记住，通常不允许将具有名称的对象传递给右值引用）。我们将在关于价值类别的章节中详细讨论。

2.3 移动的对象

在 `std::move()` 之后，移动的对象不会（部分）销毁。它们仍然是有效的对象，至少会为其调用析构函数。然而，它们也是有效的，因为它们具有一致的状态，所有操作都按照预期工作，但不知道的是它们的值。这就像使用类型的参数，而不知道传递了哪个值。

2.3.1 有效但未定义的状态

C++ 标准库保证移动的对象处于有效但未定义的状态。

考虑如下代码:

```
1 std :: string s;
2 ...
3 coll.push_back(std :: move(s));
```

用 `std::move()` 传递 `s` 之后, 可以获取字符数量, 打印相应的值, 甚至赋一个新值。如果不先检查字符数量, 就不能打印第一个字符或任何其他字符:

```
1 foo(std :: move(s)); // keeps s in a valid but unclear state
2
3 std :: cout << s << '\n'; // OK (don't know which value is written)
4 std :: cout << s.size() << '\n'; // OK (writes current number of characters)
5 std :: cout << s[0] << '\n'; // ERROR (potentially undefined behavior)
6 std :: cout << s.front() << '\n'; // ERROR (potentially undefined behavior)
7 s = "new value"; // OK
```

尽管不知道具体的值, 但该字符串处于一致状态。例如, `s.size()` 将返回字符数, 可以遍历所有有效的索引:

```
1 foo(std :: move(s)); // keeps s in a valid but unclear state
2
3 for (int i = 0; i < s.size(); ++i) {
4     std :: cout << s[i]; // OK
5 }
```

对于用户定义类型, 还应该确保移动的对象处于有效状态, 有时需要声明或实现移动操作。“移动后的状态”这一章将对此进行详细讨论。

2.3.2 重用移动的对象

您可能想知道为什么已移动的对象仍然是有效的对象, 并且没有(部分)销毁。原因是使用了移动语义, 在这些应用中再次使用已移动的对象是有意义的。

例如, 考虑从流中逐行读取字符串并将其移动到 `vector` 对象中的代码:

```
1 std :: vector<std :: string> allRows;
2 std :: string row;
3 while (std :: getline(myStream, row)) { // read next line into row
4     allRows.push_back(std :: move(row)); // and move it to somewhere
5 }
```

每次将一行读入行后, 使用 `std::move()` 将 `row` 的值移动到所有行的向量中。然后, `std::getline()` 再次使用已移动的对象行来读入下一行。

第二个例子, 考虑一个交换两个值的泛型函数:

```
1 template<typename T>
2 void swap(T& a, T& b)
3 {
4     T tmp{std :: move(a)};
5     a = std :: move(b); // assign new value to moved-from a
6     b = std :: move(tmp); // assign new value to moved-from b
```

我们将 *a* 的值移动到一个临时对象中，以便之后能够移动并赋 *b* 的值。然后，被移动的对象 *b* 接收到 *tmp* 的值，这是 *a* 之前的值。

例如，在排序算法中移动不同元素的值，使它们处于有序状态。给已移动的对象赋新值总是发生在那。该算法甚至可以对这种移动对象的方式使用排序。

通常，已移动的对象是可以销毁的有效对象（析构函数不应该失败），重用以获得其他值，并支持类型的所有操作对象，而不需要知道具体值。“已移动的状态”这一章将对此进行详细讨论。

2.3.3 将对象赋值给自己

已移动的对象处于有效但未定义状态的规则，通常也适用于直接或间接自移动的对象。

例如，下面的语句之后，对象 *x* 通常在不知道其值的情况下是有效的：

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

同样，C++ 标准库保证对用户定义类型的实例对象通常也提供这种保证，但有时必须来修复默认生成的移动状态。

2.4 通过引用进行重载

引入右值引用之后，有三种主要的引用调用方式：

- ***void foo(const std::string& arg)***

将实参作为 *const* 左值引用。

只有对传递的参数的读访问权。如果类型适合，可以把所有东西都传递给用这种方式声明的函数：

- 可修改的命名对象
- *const* 命名对象
- 没有名称的临时对象
- 用 *std::move()* 标记的对象

语义上的意思是给 *foo()* 传递的实参读访问权。这个参数就是我们所说的 *in* 参数。

- ***void foo(std::string& arg)***

将实参作为非 *const* 左值引用。

对传递的参数具有写访问权限。即使类型适合，也不能再将所有内容传递给以这种方式声明的函数：

- 可修改的命名对象

对于所有其他参数，则不进行编译。

语义上的意思是，让 *foo()* 对传递的参数具有读/写访问权限。参数就是输出或输入/输出参数。

- ***void foo(std::string&& arg)***

将实参作为非 *const* 右值引用。

对传递的参数具有写访问权限。但是，仍然对可以传递的内容有限制：

- 没有名称的临时对象
- 用 `std::move()` 标记的对象

语义上的意思是给 `foo()` 对传递的参数的写访问权来窃取值。它是一个 `in` 参数，附加的约束是调用者不再需要这个值。

右值引用绑定到非 `const` 左值引用之外的其他实参。因此，必须引入新的语法，不能仅仅将移动语义作为修改传递参数的函数的一种不同方式来实现。

2.4.1 `const` 右值引用

从技术上讲，还有第四种调用引用的方式：

- `void foo(const std::string&& arg)`

将实参作为 `const` 右值引用。

对传递的参数具有读访问权，这里的限制有：

- 没有名称的临时对象
- 用 `std::move()` 标记的 `const` 或非 `const` 对象

然而，这种情况没有意义。作为右值引用，允许窃取值，但作为 `const`，则禁止对传递的实参进行任何修改，这本身就是矛盾的。

尽管如此，创建具有这种行为的对象是非常容易的：只需用 `std::move()` 标记一个 `const` 对象：

```
1 const std::string s{ "data" };
2 ...
3 foo(std::move(s)); // would call a function declared as const rvalue reference
```

当声明函数返回带有 `const` 的值时，这种情况可能会间接发生：

```
1 const std::string getValue();
2 ...
3 foo(getValue()); // would call a function declared as const rvalue reference
```

这种情况通常由用于 `const` 左值引用重载。可能会进行特定实现，但通常没有意义（C++ 标准库类 `std::optional<T>` 使用 `const` 右值引用）。

2.5 按值传递

当声明一个函数以值作为参数，会（自动）使用移动语义。

例如：

```
1 void foo(std::string str); // takes the object by value
2 ...
3 std::string s{ "hello" };
4 ...
5 foo(s); // calls foo(), str becomes a copy of s
6 foo(std::move(s)); // calls foo(), s is moved to str
7 foo(returnStringByValue()); // calls foo(), return value is moved to str
```

如果调用者发出信号表示不再需要传递的实参的值 (通过 `std::move()` 或传递一个没有名称的临时对象), 形参 `str` 将用从传递的实参中窃取值进行初始化。

这意味着, 如果使用移动语义传递了一个临时对象或传递的参数标记为 `std::move()`, 那么按值调用会突然变得便利起来。就像按值返回本地对象一样, 这个移动也可以优化掉。如果没有优化掉, 那么这个调用现在肯定是高效的 (如果有更高效的移动语义的话)。

注意以下差异:

```
1 void fooByVal(std::string str); // takes the object by value
2 void fooByRRef(std::string&& str); // takes the object by rvalue reference
3 ...
4 std::string s1{"hello"}, s2{"hello"};
5 ...
6 fooByVal(std::move(s1)); // s1 is moved
7 fooByRRef(std::move(s2)); // s2 might be moved
```

这里, 我们比较两个函数: 一个以值作为字符串, 另一个以字符串作为右值引用。在这两种情况下, 都传递了带有 `std::move()` 的字符串。

- 按值获取字符串的函数将使用移动语义, 因为新字符串是用传递的参数的值创建的。
- 通过右值引用获取字符串的函数可以使用移动语义, 传递参数不会创建新字符串。是否窃取/修改传递的参数值取决于函数的实现。

因此:

- 声明为支持移动语义的函数可能不使用移动语义。
- 通过值声明接受参数的函数将使用移动语义。

移动语义的效果并不能保证优化发生和优化效果。我们知道的是, 传递的对象随后处于有效但未定义的状态。

2.6 总结

- 右值引用使用 `&&` 声明, 没有 `const`。
- 可以由没有名称的临时对象或用 `std::move()` 标记的非 `const` 对象初始化。
- 右值引用延长了由返回值的对象的生命周期。
- `std::move()` 对应的是右值引用类型的 `static_cast`。这允许我们将命名对象传递给右值引用。
- `std::move()` 标记的对象也可以通过 `const` 左值引用, 传递给接受实参但不接受非 `const` 左值引用的函数。
- 标记为 `std::move()` 的对象也可以传递给按值接受参数的函数。在这种情况下, 移动语义用于初始化参数, 这可以使按值调用非常高效。
- `const` 右值引用是可能的, 但针对它们的实现通常没有意义。
- 已移动的对象处于有效但未定义的状态, C++ 标准库为其类型保证了这一点。您仍然可以(重新) 使用它们, 只要您不对它们的价值做任何期望。

3 类中的移动语义

本章展示了如何在类中使用移动语义。展示了普通类如何从移动语义中获益，以及如何显式地在类中实现移动操作。

3.1 普通类中移动语义

假设有一个相当简单的类，其中的成员类型的移动语义可以让其不一样：

basics/customer.hpp

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4 #include <cassert>
5
6 class Customer {
7 private:
8     std::string name; // name of the customer
9     std::vector<int> values; // some values of the customer
10 public:
11     Customer(const std::string& n)
12         : name{n} {
13         assert(!name.empty());
14     }
15
16     std::string getName() const {
17         return name;
18     }
19
20     void addValue(int val) {
21         values.push_back(val);
22     }
23
24     friend std::ostream& operator<< (std::ostream& strm, const Customer& cust) {
25         strm << '[' << cust.name << ": ";
26         for (int val : cust.values) {
27             strm << val << ' ';
28         }
29         strm << ']';
30         return strm;
31     }
32 };
```

这个类有两个(可能)开销很大的成员，一个是字符串 *name*，一个是整型 vector *values*:

```
1 class Customer {
2 private:
3     std::string name; // name of the customer
4     std::vector<int> values; // some values of the customer
5     ...
```

```
6 };
```

复制这两个成员的成本很高

- 要复制 `name`, 我们必须为字符串的字符分配内存 (除非 `name` 很短, 并且使用小字符串优化 (SSO) 实现字符串)。
- 要复制这些值, 必须为 `vector` 的元素分配内存。

如果有 `string` 类型的 `vector` 或者其他大型元素类型, 那么开销会更大。例如, `string` 类型 `vector` 的副本必须为元素的动态数组和每个元素所需的内存分配内存。

好消息是, 这样的类通常自动支持移动语义。自 C++11 以来, 编译器通常会生成移动构造函数和移动赋值操作符 (类似于自动生成复制构造函数和复制赋值操作符)。

这样会有以下效果:

- 按值返回本地 `Customer` 将使用移动语义 (如果它没有优化掉的话)。
- 通过值传递未命名的 `Customer` 对象将使用移动语义 (如果没有优化掉的话)。
- 按值传递临时的 `Customer` 对象 (例如, 由另一个函数返回) 将使用移动语义 (如果它没有优化掉的话)。
- 通过值传递一个标有 `std::move()` 的 `Customer` 对象将使用移动语义 (如果它没有被优化掉的话)。

例如:

`basics/customer1.cpp`

```
1 #include "customer.hpp"
2 #include <iostream>
3 #include <random>
4
5 #include <utility> // for std::move()
6
7 int main()
8 {
9     // create a customer with some initial values:
10    Customer c{"Wolfgang Amadeus Mozart"};
11    for (int val : {0, 8, 15}) {
12        c.addValue(val);
13    }
14    std::cout << "c: " << c << '\n'; // print value of initialized c
15
16    // insert the customer twice into a collection of customers:
17    std::vector<Customer> customers;
18    customers.push_back(c); // copy into the vector
19    customers.push_back(std::move(c)); // move into the vector
20    std::cout << "c: " << c << '\n'; // print value of moved-from c
21
22    // print all customers in the collection:
23    std::cout << "customers:\n";
24    for (const Customer& cust : customers) {
```

```
25     std::cout << " " << cust << '\n';
26 }
27 }
```

这里，创建并初始化一个 *customer* *c*(为了避免 SSO，我们使用一个相当长的名称)。初始化 *c* 后，第一个输出如下：

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

然后将这个 *customer* 插入到 vector 中两次：复制一次，移动一次：

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

然后，*c* 的输出：

```
c: [: ]
```

第二次调用 *push_back()* 时，名称和值都移到了 vector 的第二个元素中。但是，已移动的对象处于有效但未定义的状态。因此，第二个输出可以有任何值 *name* 和 *values*：

- 可能仍然有相同的值：

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

- 可能有完全不同的值：

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

然而，移动语义是为了优化性能而提供，而分配不同的值并不一定是提高性能，所以在实现中通常把字符串和 vector 都设为空。

任何情况下，我们都可以看到为 *Customer* 类自动启用了移动语义。出于同样的原因，可以保证以下代码的高效：

```
1 Customer createCustomer()
2 {
3     Customer c{ ... };
4     ...
5     return c; // uses move semantics if not optimized away
6 }
7
8 std::vector<Customer> customers;
9 ...
10 customers.push_back(createCustomer()); // uses move semantics
```

详见 basics/customer2.cpp 的示例。

自从 C++11 以来，类中能使用移动语义的成员，就会使用移动语义。这些类有：

- 从不需要的源创建新对象，则使用成员的移动构造函数:

```

1 Customer c1{ ... }
2 ...
3 Customer c2{ std::move(c1) }; // move members of c1 to members of c2
4

```

- 移动赋值操作符如果从不再需要该的源赋值。

```

1 Customer c1{ ... }, c2{ ... };
2 ...
3 c2 = std::move(c1); // move assign members of c1 to members of c2
4

```

注意，通过显式实现以下改进，这样的类可以从移动语义获益:

- 初始化成员时使用移动语义
- 使用移动语义使获取函数既安全又快速

3.1.1 什么时候使用自启动移动语义的类?

编译器可以自动生成特殊的移动成员函数(移动构造函数和移动赋值操作符)。然而，也有一些限制。约束是编译器必须假定生成操作是正确的。正确的做法是优化正常的复制行为：移动成员，而不是复制成员。

如果类改变了复制或赋值的常规行为，那么在优化这些操作时可能也必须做一些不同的事情。因此，当用户声明以下至少一个特殊成员函数时，将禁用移动操作的自动生成:

- 复制构造函数
- 复制赋值运算符
- 另一个移动操作
- 析构函数

注意，是“用户声明”。任何形式的复制构造函数、复制赋值操作符或析构函数的显式声明都禁用移动语义。例如，如果实现了一个什么也不做的析构函数，就禁用了移动语义:

```

1 class Customer {
2 ...
3 ~Customer() { // automatic move semantics is disabled
4 }
5 };

```

即使下面的声明也足以禁用移动语义:

```

1 class Customer {
2 ...
3 ~Customer() = default; // automatic move semantics is disabled
4 };

```

用户声明析构函数的行为为默认的，因此禁用了移动语义。通常，这种情况下，复制语义将作为一种退阶选择。

因此没有特定的需要，就不要实现或声明析构函数（很多程序员都没有遵循这一规则）。

这也意味着默认情况下，多态基类禁用了移动语义：

```
1 class Base {
2 ...
3     virtual ~Base() { // automatic move semantics is disabled
4 }
5 };
```

注意，这意味着移动语义只对在这个基类中声明的成员禁用。对于派生类的成员，移动语义仍然会自动生成（如果派生类没有显式声明特殊的成员函数）。在讨论类层次中的移动语义时会讨论了这一点。

3.1.2 生成的移动操作有负面影响的情况

请注意，生成的移动操作可能会引入问题，即使生成的复制操作工作正确。特别是在以下情况，必须小心：

- 对成员变量有限制
 - 值有限制
 - 值相互依赖
- 使用了引用语义的成员（指针，智能指针，…）
- 对象没有默认构造

可能出现的问题是，已移动的对象可能不再有效：可能破坏不变量，或者对象的析构函数失败。例如，本章中的 *Customer* 类的对象可能会有空名称（即使我们有断言来避免这种情况）。关于已移动的那一章将对此进行详细讨论。

3.2 实现复制/移动函数

可以自己实现特殊的移动成员函数。这与实现复制构造函数和赋值操作符的方式大致相同。唯一的区别是形参需要声明为非 *const* 右值引用，并且在实现内部必须指定在何处优化复制操作。

让我们看一个类，同时实现了特殊的复制和移动成员函数，用于在对象复制和移动时：

[basics/customerimpl.hpp](#)

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4 #include <cassert>
5
6 class Customer {
7     private:
8         std::string name; // name of the customer
9         std::vector<int> values; // some values of the customer
10    public:
11        Customer(const std::string& n)
12            : name{n} {
```

```

13     assert (!name.empty());
14 }
15
16 std::string getName() const {
17     return name;
18 }
19
20 void addValue(int val) {
21     values.push_back(val);
22 }
23
24 friend std::ostream& operator<< (std::ostream& strm, const Customer& cust) {
25     strm << '[' << cust.name << ":" ;
26     for (int val : cust.values) {
27         strm << val << ',';
28     }
29     strm << ']';
30     return strm;
31 }
32
33 // copy constructor (copy all members):
34 Customer(const Customer& cust)
35 : name{cust.name}, values{cust.values} {
36     std::cout << "COPY " << cust.name << '\n';
37 }
38
39 // move constructor (move all members):
40 Customer(Customer&& cust) // noexcept declaration missing
41 : name{std::move(cust.name)}, values{std::move(cust.values)} {
42     std::cout << "MOVE " << name << '\n';
43 }
44
45 // copy assignment (assign all members):
46 Customer& operator= (const Customer& cust) {
47     std::cout << "COPYASSIGN " << cust.name << '\n';
48     name = cust.name;
49     values = cust.values;
50     return *this;
51 }
52
53 // move assignment (move all members):
54 Customer& operator= (Customer&& cust) { // noexcept declaration missing
55     std::cout << "MOVEASSIGN " << cust.name << '\n';
56     name = std::move(cust.name);
57     values = std::move(cust.values);
58     return *this;
59 }
60 };

```

让我们详细看看特殊的拷贝/移动成员函数的实现。

请注意，手动实现移动构造函数和移动赋值操作符时，通常都应该有 noexcept 声明，这将在移动语义和 noexcept 章节中讨论。

3.2.1 复制构造函数

复制构造函数的实现如下：

```
1 class Customer {
2 private:
3     std::string name; // name of the customer
4     std::vector<int> values; // some values of the customer
5
6 public:
7     ...
8     // copy constructor (copy all members):
9     Customer(const Customer& cust)
10    : name{cust.name}, values{cust.values} {
11        std::cout << "COPY " << cust.name << '\n';
12    }
13    ...
14};
```

自动生成的复制构造函数，只复制所有成员。实现中，我们只添加了一条打印特定 *customer* 复制时的语句。

3.2.2 移动构造函数

移动构造函数的实现如下：

```
1 class Customer {
2 private:
3     std::string name; // name of the customer
4     std::vector<int> values; // some values of the customer
5 public:
6     ...
7     // move constructor (move all members):
8     Customer(Customer&& cust) // noexcept declaration missing
9     : name{std::move(cust.name)}, values{std::move(cust.values)} {
10        std::cout << "MOVE " << name << '\n';
11    }
12    ...
13};
```

同样，这也是默认生成的移动构造函数所做的，并使用附加的打印语句进行扩展。

与复制构造函数的区别是将形参声明为非 *const* 右值引用然后移动成员。

注意，这里非常重要的一点：移动语义没有传递。当使用 *cust* 的成员初始化成员时，必须用 *std::move()*。没有这个，就只能复制它们（移动构造函数的性能与复制构造函数相同）。

您可能想知道为什么没有传递移动语义。我们没有声明参数 *cust* 只接受带有移动语义的对象

吗？但请注意这里的语义：当调用者不再需要该值时，将调用该函数。在移动构造函数内部，现在有了要处理的值，必须决定在哪里需要它，需要多长时间。特别是，可能多次需要这个值，并且不会在第一次使用时丢失。

因此，移动语义没有传递是一个特性，而不是一个 bug。如果要传递移动语义，就不能使用两次传递了移动语义的对象。

例如：

```
1 void insertTwice(std::vector<std::string>& coll, std::string&& str)
2 {
3     coll.push_back(str); // copy str into coll
4     coll.push_back(std::move(str)); // move str into coll
5 }
```

如果 *str* 的所有使用都隐式地具有移动语义，则 *str* 的值将在第一次 *push_back()* 调用时移走。

这里需要学习的是，将形参声明为右值引用限制了可以传递给该函数的内容，但其行为与任何其他该类型的非 *const* 对象一样。必须再次指定何时何地不再需要这个值。有关更多细节，后续正式讨论当右值变成左值时会继续。

另一个注意事项：当复制构造函数的打印语句打印传递的客户的名称时：

```
1 Customer(const Customer& cust)
2 : name{cust.name}, values{cust.values} {
3     std::cout << "COPY " << cust.name << '\n'; // cust.name still there
4 }
```

移动构造函数不能使用 *custom.name*，因为在构造函数初始化时，该值可能已经移走了。必须使用新对象的成员：

```
1 Customer(Customer&& cust)
2 : name{std::move(cust.name)}, values{std::move(cust.values)} {
3     std::cout << "MOVE " << name << '\n'; // have to use name (cust.name moved away)
4 }
```

注意，应该始终使用 noexcept 规范来实现移动构造函数，以提高 *Customer* vector 重新分配时的性能。

3.2.3 复制赋值操作符

复制赋值操作符的实现如下：

```
1 class Customer {
2     private:
3         std::string name; // name of the customer
4         std::vector<int> values; // some values of the customer
5     public:
6     ...
7     // copy assignment (assign all members):
8     Customer& operator=(const Customer& cust) {
9         std::cout << "COPYASSIGN " << cust.name << '\n';
10        name = cust.name;
```

```
11     values = cust.values;
12     return *this;
13 }
14 ...
15 };
```

与自动生成的复制赋值操作符一样，只需对所有成员进行赋值。唯一的区别是开头的打印语句。

实现赋值操作符时，可以（也许应该）检查对象对自身的赋值。但请注意，生成的默认赋值操作符不会这样做，在讨论使用移动赋值操作符进行自我赋值时，请查看关于这一点的解释。

您可能还希望使用引用限定符声明赋值操作符。

3.2.4 移动赋值运算符

移动赋值操作符的实现如下：

```
1 class Customer {
2     private:
3     std::string name; // name of the customer
4     std::vector<int> values; // some values of the customer
5     public:
6     ...
7     // move assignment (steal all members):
8     Customer& operator=(Customer&& cust) { // noexcept declaration missing
9         std::cout << "MOVEASSIGN " << cust.name << '\n';
10        name = std::move(cust.name);
11        values = std::move(cust.values);
12        return *this;
13    }
14 };
```

同样，必须将移动构造函数声明为接受非 *const* 右值引用的函数。然后在主体中，必须实现如何改进通常的复制，因为我们可以从源对象中窃取值。

本例中，我们做了自动生成的移动赋值操作符所做的事情：对函数体中的成员进行移动赋值，而不是复制赋值。此外，添加了打印语句。最后，返回对象以供进一步使用。

注意，通常应该使用 noexcept 规范来实现移动赋值操作符。

您可能还希望使用引用限定符声明移动赋值操作符。

处理对象自身的移动分配

可能想知道在实现移动赋值操作符时，是否应该检查对象对自身的赋值。例如，自移动可能会发生如下情况：

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

这看起来很容易避免，可以使用引用和指针。这种情况下，两个对象是否相同并不明显。下面的代码是一个非常简单的例子：

```
1 std::vector<Customer> coll;
```

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

当对象自移动时, C++ 标准库中的所有类型都接收一个有效但未定义的状态。默认情况下, 可能会丢失成员的值, 如果类型不能正确地处理具有任意值的成员, 甚至可能会遇到更严重的问题。事实上, 在以下情况下, 可能产生有问题的状态:

- 已移动成员的一些值有问题
- 成员的值相互依赖

防止自赋值的传统/简单方法是检查两个操作数是否相同(具有相同的地址)。在实现移动赋值操作符也可以这样做:

```
1 Customer& operator=(Customer&& cust) { // noexcept declaration missing
2     std::cout << "MOVEASSIGN " << cust.name << '\n';
3     if (this != &cust) { // move assignment to myself?
4         name = std::move(cust.name);
5         values = std::move(cust.values);
6     }
7     return *this;
8 }
```

使用这样的检查非常廉价, 它的好处是对象将其值保存。但请注意, 在递归数据结构中, 如果(移动)子对象分配给父对象, 也可能有问题。如果在赋给新值之前先删除父对象(拥有子对象)的旧值, 可能会赋给一个已删除的对象。这种情况下, 对象并不相同, 所以比较它们的地址没有什么用。

此外, 考虑以下编程风格:

- Scott Meyers 在《Effective C++》一书中说, 当使用助手类正确封装资源管理时(那时还没有移动语义), 自赋值通常不是问题。此外, 他指出, 还必须确保操作符的异常安全。
- 《C++ 核心指南》将这个问题归类为“百万分之一的问题”, 在实践中似乎并不相关。

似乎在大多数情况下, 可以忽略自移动赋值, 在你定义的类型中, 提供了与 C++ 标准库相同的保证: 移动将一个对象赋值给它自己之后, 该对象处于一个有效但未定义的状态。

3.2.5 使用特殊的复制/移动成员函数

让我们用一个小程序来测试 *Customer* 类:

[basics/customerimpl.cpp](#)

```
1 #include "customerimpl.hpp"
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <algorithm>
6
7 int main()
8 {
9     std::vector<Customer> coll;
```

```
10 for (int i=0; i<12; ++i) {  
11     coll.push_back(Customer{"TestCustomer " + std::to_string(i-5)});  
12 }  
13  
14 std::cout << "----- sort():\n";  
15 std::sort(coll.begin(), coll.end(),  
16           [] (const Customer& c1, const Customer& c2) {  
17               return c1.getName() < c2.getName();  
18           });  
19 }
```

初始化

第一部分是有 12 个 *Customer* 的向量的初始化:

```
1 std::vector<Customer> coll;  
2 for (int i=0; i<12; ++i) {  
3     coll.push_back(Customer{"TestCustomer " + std::to_string(i-5)});  
4 }
```

这个初始化可能有以下输出:

```
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
```

每次插入新对象时，都会创建一个临时对象并将其移动到向量中。因此，对于每个元素，我们都有一个 MOVE。

另外，有几份副本上有 COPY 的标记。出现副本的原因：vector 会不时重新分配其内部内存（容量），以便它足够大，可以容纳所有元素。在这种情况下，vector 从有存储 1 个元素的内存增长到有存储 2 个、4 个、8 个，最后是 16 个元素。因此，必须先复制 1 2 4，然后是 vector 中已经存在的 8 个元素。在循环之前使用 `col.reserve(20)` 可以避免这些副本的出现。但是，您可能想知道为什么这里不使用移动。这与缺少的 noexcept 声明有关，我们将在关于移动语义和 noexcept 的章节中讨论。

注意，向 vector 增加容量的确切策略是特定于实现的。因此，当实现增长不同时，输出可能会有所不同（例如，每次增长 50%）。

排序

接下来，我们按名称对所有元素进行排序：

```
1 std::sort( coll.begin(), coll.end(),
2             [] (const Customer& c1, const Customer& c2) {
3                 return c1.getName() < c2.getName();
4             });

```

排序可能出现以下输出：

```
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
```

整个排序只有元素移动，有时是创建一个新的临时对象 (MOVE)，有时是分配值到不同的位置 (MOVEASSIGN)。

同样，输出取决于 `sort()` 的确切实现，这是特定于实现的。

保存的副本数量

这个程序的输出再次证明了移动语义的好处。在支持移动语义前，我们只在插入和排序元素时有副本。然而，即使在这个有 12 个元素的小程序中，移动语义将 44 个拷贝转换成廉价的移动。对于 1 万名客户，我们将节省超过 15 万份拷贝。请注意，每个 *Customer* 副本都意味着最多分配 2 个内存，稍后将进行相应的释放。

请再次注意：

- `std::sort()` 是特定于实现的，保存的副本数量可能不同。
- 执行的唯一副本是由于重新分配 `vector` 用于存储元素的内存造成的。它们也应该使用移动操作，这将在关于移动语义的章节中讨论。

3.3 特殊成员函数的规则

让我们来讨论一下特殊成员函数，特别是确切地指定何时以及如何生成特殊的复制和移动成员函数。

3.3.1 特殊的成员函数

首先简单地看看特殊成员函数，因为它以不同的方式使用。C++ 标准将以下 6 种操作定义为特殊成员函数：

- 默认构造函数
- 复制构造函数
- 复制赋值运算符
- 移动构造函数 (C++11)
- 移动赋值操作符 (C++11)
- 析构函数

然而，在许多情况下，我们只讨论其中 5 个操作，因为默认构造函数与其他 5 个（或 C++11 之前的 3 个）操作略有不同。其他 5 个操作通常没有声明，并且具有更复杂的依赖关系。因此，请了解特殊成员函数的含义（到目前为止，我一直试图避免使用这个术语）。

图 3.1 概述了什么时候会根据声明的构造函数和特殊成员函数，自动生成特殊成员函数：

图 3.1 自动生成特殊成员函数的规则

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

可以在这个表格中看到一些基本的规则:

- 只有在没有用户声明其他构造函数的情况下，才会自动声明默认构造函数。
- 特殊的复制成员函数和析构函数禁用了移动支持。自动生成特殊的移动成员函数是禁用的（除非也声明了移动操作）。但是，移动对象的请求通常是有效的，因为复制成员函数用作后备（除非特殊的移动成员函数显式删除）。
- 特殊的移动成员函数禁用了正常的复制和赋值。复制和其他移动特殊成员函数将删除，这样只能移动（分配）对象，而不能复制（分配）对象（除非还声明了其他操作）。

3.3.2 默认情况下，有复制和移动

在默认编译情况下，为类生成复制和移动特殊成员函数。

假设类 *Person* 的声明如下:

```

1 class Person {
2 ...
3 public:
4 ...
5 // NO copy constructor/assignment declared
6
7 // NO move constructor/assignment declared
8
9 // NO destructor declared
10};

```

这种情况下，*Person* 可以复制和移动:

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

```
2
3 Person p{ "Tina" , "Fox" };
4 coll.push_back(p); // OK, copies p
5 coll.push_back(std::move(p)); // OK, moves p
```

3.3.3 已声明的复制禁用移动 (启用退阶)

在声明复制特殊成员函数 (或析构函数) 时, 禁用了移动特殊成员函数的自动生成:

假设 *Person* 的声明如下:

```
1 class Person {
2 ...
3 public:
4 ...
5 // copy constructor/assignment declared:
6 Person(const Person&) = default;
7 Person& operator=(const Person&) = default;
8
9 // NO move constructor/assignment declared
10};
```

因为退阶机制是有效的, 所以复制和移动 *Person* 可以编译, 但移动是作为复制执行的:

```
1 std::vector<Person> coll;
2
3 Person p{ "Tina" , "Fox" };
4 coll.push_back(p); // OK, copies p
5 coll.push_back(std::move(p)); // OK, copies p
```

因此, 用 *=default* 声明一个特殊成员函数与根本不声明它是不同的。复制构造函数和复制赋值操作符是用户声明的, 它禁用了移动构造和移动赋值, 以便移动到副本。声明的析构函数具有相同的效果。

您可能想知道为什么声明的析构函数会禁用移动语义。关于已移动状态那一章讨论了具体的例子, 一个类的析构函数导致生成的移动语义不能正常工作。

3.3.4 声明移动, 禁用复制

如果声明了的移动语义, 就禁用了复制语义。删除复制特殊成员函数。

换句话说, 如果移动构造函数或移动赋值操作符显式声明 (使用 *=default* 实现、生成, 或使用 *=delete* 禁用), 那么通过使用 *=delete* 声明, 就禁用了调用复制构造函数和复制赋值操作符。

假设 *Person* 的声明如下:

```
1 class Person {
2 ...
3 public:
4 ...
5 // NO copy constructor declared
6
7 // move constructor/assignment declared:
```

```

8 Person (Person&&) = default;
9 Person& operator=(Person&&) = default;
10 };

```

本例中，我们有一个只移动类型。Person 可以移动，但不能复制：

```

1 std :: vector<Person> coll;
2
3 Person p{ "Tina" , "Fox" };
4 coll.push_back(p); // ERROR: copying disabled
5 coll.push_back(std :: move(p)); // OK, moves p
6
7 coll.push_back(Person{ "Ben" , "Cook" }); // OK, moves temporary person into coll

```

同样，使用 `=default` 声明特殊成员函数与根本不声明不同。调用的后果更加严重：复制对象的尝试将无法编译。

支持移动操作但不支持复制操作的类是有意义的。可以使用这种只移动类型传递资源的所有权或句柄，而无需共享或复制。在 C++ 标准库中也有一些只移动的类型（例如，输入输出流、线程和 `std::unique_ptr<>`），详细信息请参见关于仅移动类型的章节。

3.3.5 删除移动没有意义

出于同样的原因，如果将移动构造函数声明为 `delete`，则不能移动（已禁用此操作，没有使用任何退阶）和不能复制（因为声明的移动构造函数禁用了复制操作）：

```

1 class Person {
2 public:
3 ...
4 // NO copy constructor declared
5 // move constructor/assignment declared as deleted:
6 Person (Person&&) = delete;
7 Person& operator=(Person&&) = delete;
8 ...
9 };
10
11 Person p{ "Tina" , "Fox" };
12 coll.push_back(p); // ERROR: copying disabled
13 coll.push_back(std :: move(p)); // ERROR: moving disabled

```

通过声明复制被删除的特殊成员函数，可以获得相同的效果，这可能不会让其他程序员感到困惑。

删除移动操作和启用复制操作真的没有意义：

```

1 class Person {
2 public:
3 ...
4 // copy constructor explicitly declared:
5 Person(const Person& p) = default;
6 Person& operator=(const Person&) = default;
7

```

```

8 // move constructor/assignment declared as deleted:
9 Person (Person&&) = delete;
10 Person& operator=(Person&&) = delete;
11 ...
12 };
13
14 Person p{ "Tina" , "Fox" };
15 coll.push_back(p); // OK: copying enabled
16 coll.push_back(std::move(p)); // ERROR: moving disabled

```

这种情况下，`=delete` 禁用了退阶机制（因此也与根本不声明它不同）。编译器找到该声明并报告该调用为错误。

支持复制但在调用移动时失败的类型没有任何意义。对于此类的用户，复制有时可行，有时不行。作为指导原则：永不 `=delete` 特殊的移动成员函数。如果想要同时禁用复制和移动，删除复制的特殊成员函数就足够了。

3.3.6 禁用移动语义与启用复制语义

基于我们刚才讨论的内容，现在知道了当复制仍然有意义时如何禁用移动语义。将特殊的移动成员函数声明为已删除通常不是正确的做法，因为它禁用了退阶机制。

提供复制语义的同时，禁用移动语义的正确方法是，声明另一个特殊成员函数（复制构造函数、赋值操作符或析构函数）。我建议你使用默认复制构造函数和赋值操作符（声明其中一个就足够了，但可能会造成不必要的混淆）：

```

1 class Customer {
2 ...
3 public:
4 ...
5 Customer(const Customer&) = default; // disable move semantics
6 Customer& operator=(const Customer&) = default; // disable move semantics
7 };

```

因为没有发现生成的特殊的移动成员函数，所以现在会复制临时的 `Customer`，甚至标记为 `std::move()` 的 `Customer` 对象：

```

1 std::vector<Customer> customers;
2 ...
3 customers.push_back(createCustomer()); // OK, falls back to copying
4 customers.push_back(std::move(customers[0])); // OK, falls back to copying

```

但是，通常最好是实现特殊的移动成员函数，来修复默认生成的移动操作所存在的任何问题。关于迁移状态的章节将讨论实践中的例子。

请注意，只有声明特殊的复制成员函数才会违反“5 规则”，我们将在后面详细讨论这个规则。必须声明特殊的复制成员函数，但也不能声明特殊的移动成员函数（删除和默认都不能工作，实现它们会使类变得复杂）。因此，如果显式地声明一个复制特殊成员函数只是为了禁用移动语义，请添加一个注释，以确保该声明不会被特殊移动成员函数的声明删除或扩展。

3.3.7 为禁用移动语义的成员进行移动

如果某个类型的移动语义不可用或已被删除，则这不会影响具有此类型成员的类的移动语义的生成。生成的默认移动构造函数和赋值操作符逐个成员决定是复制还是移动它。如果移动不可能（即使移动操作被删除），则生成一个副本（如果可能的话）。

例如，以下类：

```
1 class Customer {
2 ...
3 public:
4 ...
5 Customer(const Customer&) = default; // copying calls enabled
6 Customer& operator=(const Customer&) = default; // copying calls enabled
7 Customer(Customer&&) = delete; // moving calls disabled
8 Customer& operator=(Customer&&) = delete; // moving calls disabled
9 };
```

如果这个类被另一个类中的成员使用：

```
1 class Invoice {
2     std::string id;
3     Customer cust;
4 public:
5     ... // no special member functions
6 };
```

生成的移动构造函数将移动 *id* 字符串，但复制客户的 *cust*：

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

3.3.8 生成的特殊成员函数的规则

现在我们可以总结特殊成员函数的新规则（何时生成它们以及它们的行为方式）。

例如，假设有以下派生类：

```
1 class MyClass : public Base
2 {
3 private:
4     MyType value;
5     ...
6 };
```

这里缺少 noexcept，我们将在后面关于 noexcept 的章节中介绍它，但我们会在这里提到相应的保证。

复制构造函数

当应用以下所有内容时，将自动生成复制构造函数：

- 用户没有声明移动构造函数

- 用户没有声明移动赋值操作符

如果生成 (隐式或使用 `=default`)，复制构造函数具有以下行为:

```

1 MyClass( const MyClass& obj) noexcept-specifier
2 : Base(obj), value(obj.value) {
3 }
```

生成的复制构造函数，首先将源对象传递给基类的最佳匹配的复制构造函数。(记住，复制构造函数总是在自顶向下的基础上调用)。它倾向于使用具有相同声明的复制构造函数 (通常声明为 `const&`)。如果不可用，可能会调用下一个最佳匹配的构造函数 (例如，复制构造函数模板)。然后，复制类的所有成员 (同样使用最佳匹配)。

生成的复制构造函数声明为 `noexcept`，除非所有复制操作 (所有基类的复制构造函数和所有成员的复制构造函数) 都有这个保证。

移动构造函数

移动构造函数在应用以下所有条件时自动生成:

- 用户没有声明复制构造函数
- 用户没有声明复制赋值操作符
- 用户没有声明移动赋值操作符
- 用户没有声明析构函数

如果生成 (隐式或使用 `=default`)，移动构造函数具有以下行为:

```

1 MyClass( MyClass&& obj) noexcept-specifier
2 : Base(std::move(obj)), value(std::move(obj.value)) {
3 }
```

生成的移动构造函数首先传递源对象 (标记为 `std::move()` 来传递移动语义)，给基类最佳匹配的移动构造函数。最佳匹配的移动构造函数通常是具有相同声明 (用 `&&` 声明) 的构造函数。然而，如果这是不可用的，可能会调用下一个最佳匹配的构造函数 (例如，移动构造函数模板，甚至复制构造函数)。然后，移动类的所有成员 (同样使用最佳匹配)。

生成的移动构造函数声明为 `noexcept`，除非所有调用的移动/复制操作 (所有基类的复制或移动构造函数，以及所有成员的复制或移动构造函数) 都保证了这一点。

复制赋值运算符

复制赋值操作符在应用以下所有条件时自动生成:

- 用户没有声明移动构造函数
- 用户没有声明移动赋值操作符

如果生成 (隐式或使用 `=default`)，复制赋值操作符大致具有以下行为:

```

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

生成的复制赋值操作符，首先对传递的源对象调用基类的最佳匹配赋值操作符（记住，与复制构造函数相比，赋值操作符不是自上而下调用的；它们在实现中调用基类赋值操作符）。然后，为其类的成员进行分配（同样使用最佳匹配）。

注意，生成的赋值操作符不会检查对象对自赋值。如果这很重要，必须自己实现操作符。

此外，生成的复制赋值操作符声明为 noexcept，除非所有赋值操作（基类成员的赋值和新成员的赋值）都给出了这个保证。

移动赋值操作符

移动赋值操作符在以下所有条件都适用时自动生成：

- 用户没有声明复制构造函数
- 用户没有声明移动构造函数
- 用户没有声明复制赋值操作符
- 用户没有声明析构函数

如果生成（隐式或使用 =default），移动赋值操作符大致具有以下行为：

```

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

生成的移动赋值操作符，首先为传递的源对象调用基类的最佳匹配的移动赋值操作符，用 `std::move()` 标记来传递移动语义。然后它移动分配它的类的所有成员（同样使用最佳匹配）。

你可能想知道，为什么在源对象标记为 `std::move()` 之后，我们仍然使用对象 `obj` 的成员：

```

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

本例中，将对象标记为基类的特定上下文，基类不能看到在该类中引入的成员。因此，派生成员具有有效但未定义的状态，但仍然可以使用新成员的值。

生成的赋值操作符也不会检查对象对自赋值。因此，在默认行为中，操作符将把每个成员都赋值给自身，这通常意味着成员接收一个有效但未定义的值。如果这很重要，需要自己实现操作符。

此外，生成的移动赋值操作符被声明为 noexcept，除非所有调用的赋值操作（基类成员的赋值和新成员的赋值）都保证了这一点。

其他特殊成员函数

其他特殊成员函数对移动语义没有那么重要的作用：

- 析构函数对于移动语义没有什么特别之处，除了声明禁用了移动操作的自动生成。
- 如果没有声明其他构造函数，默认构造函数（“不那么特殊”的特殊成员函数）仍然会自动生成。也就是说，移动构造函数的声明禁用了默认构造函数的生成。

但请注意，当谈到已移动状态时，这些特殊的成员函数发挥了作用。默认构造函数的状态通常是已移动对象的“自然”状态，而已移动对象是可销毁的。

3.4 三五法则

是否自动生成或自动生成哪个特殊成员函数取决于刚才的规则，许多开发者并不知道这些规则。因此，即使在 C++11 之前，指导方针是不提供任何或所有用于复制、赋值和销毁的特殊成员函数。

- C++11 之前，这条原则称为“3 法则”：要么声明全部三种（复制构造函数、赋值操作符和析构函数），要么一个都不声明。
- 从 C++11 开始，该规则就变成了“5 法则”，通常的表述方式是：要么声明所有 5 种（复制构造函数、移动构造函数、复制赋值操作符、移动赋值操作符和析构函数），要么一个都不声明。

在这里，声明的意思是：

- 要么实现 ({}...{})
- 或者声明为默认 (=default)
- 或声明为已删除 (=delete)

当其中一个特殊成员函数被实现、默认或删除时，应该同时实现、默认或删除所有其他四个特殊成员函数。

但是，您应该注意这条规则。我建议更多地把它作为一个指导方针，当其中一个特殊成员函数是用户声明的时候，仔细考虑这 5 个特殊成员函数。

为了只启用复制语义，应该使用默认复制特殊成员函数，而不声明特殊的移动成员函数（删除和默认特殊移动成员函数是行不通的，实现它们会使类变得复杂）。如果生成的移动语义创建了无效状态，则建议使用此选项，我们将在关于无效已移动状态的章节中讨论这一点。

当应用 5 规则时，有时开发者使用它来为新的移动操作添加声明，却不理解这意味着什么。开发者只是用 =default 来声明移动操作，因为已经实现了复制操作，所以他们希望遵循 5 规则。

因此，我通常教授的“5 法则”或“3 法则”是：

- 如果声明了复制构造函数、移动构造函数、复制赋值操作符、移动赋值操作符或析构函数，则必须仔细考虑如何处理其他特殊成员函数。
- 如果不理解移动语义，只考虑复制构造函数、复制赋值操作符和析构函数（如果声明其中之一）。如果有疑问，可以使用 =default 声明复制特殊成员函数来禁用移动语义。

3.5 总结

- 移动语义不可传递。
- 对于每个类，移动构造函数和移动赋值操作符都是自动生成的（除非没有办法这样做）。
- 用户声明复制构造函数、复制赋值操作符或析构函数将禁用类中对移动语义的自动支持。这不会影响派生类中的支持。
- 用户声明一个移动构造函数或移动赋值操作符将禁用类中对复制语义的自动支持，将得到的是只移动类型（除非这些特殊的移动成员函数删除）。
- 永远不要 =delete 特殊的移动成员函数。

- 如果没有特定的需要，不要声明析构函数。从多态基类派生的类也一样，没有特殊需要，不要声明析构函数。

4 如何从移动语义中获益

大多数时候，开发者可以从自动移动语义中获益。然而，即使作为一个普通的应用开发者，也可以从移动语义中获益更多。

本章讨论如何从普通应用程序代码、类和类层次结构中的移动语义中获益。本章还介绍了相应的新准则。

注意，即使在引入移动语义几年之后，这些功能在社区中还是不太为人知晓。我写这本书的一个原因，是让现代 C++ 编程成为艺术。

4.1 避免命名对象

移动语义允许对不再需要的值进行优化。如果编译器自动检测到从一个生命周期结束的对象中，将自动切换到移动语义。情况是这样的：

- 传递一个临时对象，该对象将在语句之后自动销毁。
- 按值返回一个局部对象。

另外，可以通过 `std::move()` 标记对象来强制使用移动语义。

让编译器来做这些工作更容易，所以移动语义的第一个建议：避免有名称的对象。

例如：

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

可以改成：

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

就会自动启用移动语义。

当然，这个建议可能会与其他指导原则冲突，比如：源代码的可读性和可维护性。与其使用一个复杂的语句，不如使用多个语句。这种情况下，如果不再需要一个对象（并且知道复制对象可能会花费大量时间），应该使用 `std::move()`：

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

4.1.1 当你不能避免使用名称

有些情况下，需要为命名对象使用 `std::move()`。最典型的例子是：

- 必须多次使用一个对象。例如，可能会获得值，以便在函数或循环中处理它两次：

```
1 std::string str{getData()};
2 ...
3 coll1.push_back(str); // copy (still need the value of str)
4 coll2.push_back(std::move(str)); // move (no longer need the value of str)
5
```

当在同一个集合中插入值两次或调用两个不同的函数将值存储在某处时，情况也相同。

- 处理参数。最常见的例子是下面的循环：

```

1 // read and store line by line from myStream in coll
2 std::string line;
3 while (std::getline(myStream, line)) {
4     coll.push_back(std::move(line)); // move (no longer need the value of line)
5 }
6

```

4.2 避免不必要的 `std::move()`

如果支持的话，按值返回本地对象会自动使用移动语义。然而，为了安全起见，开发者可能会尝试使用显式 `std::move()` 强制执行：

```

1 std::string foo()
2 {
3     std::string s;
4     ...
5     return std::move(s); // BAD: don't do this
6 }

```

请记住，`std::move()` 只是对右值引用的 `static_cast`。因此，`std::move(s)` 会产生 `std::string&&` 类型的表达式。然而，这不再匹配返回类型，因此禁用了返回值优化，这通常允许返回对象作为返回值使用。对于没有实现移动语义的类型，这甚至会强制复制返回值，而不是仅仅使用返回的对象作为返回值。

因此，如果按值返回局部对象时，不要使用 `std::move()`：

```

1 std::string foo()
2 {
3     std::string s;
4     ...
5     return s; // best performance (return value optimization or move)
6 }

```

临时对象时使用 `std::move()` 是多余的。对于按值返回对象的 `createString()` 函数，应该只使用返回值：

```

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

```

而不是用 `std::move()` 再次标记：

```

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

```

编译器可能（可以选择）对任何适得其反或不必要的 `std::move()` 使用发出警告。例如，gcc 有选项-Wpessimizing-move（通过-Wall 启用）和-Wredundancy-move（通过-Wextra 启用）。

在某些应用程序中，返回语句中的 `std::move()` 是合适的。一个是移出成员的值，另一个是返回带有移动语义的参数。

4.3 用移动语义初始化成员

即使在具有移动语义的类型成员 (如字符串成员或容器) 的类中，也可以受益于移动语义。

4.3.1 用经典方法初始化成员

一个有两个 string 成员的类，可以在构造函数中初始化。这样的类通常会这样实现：

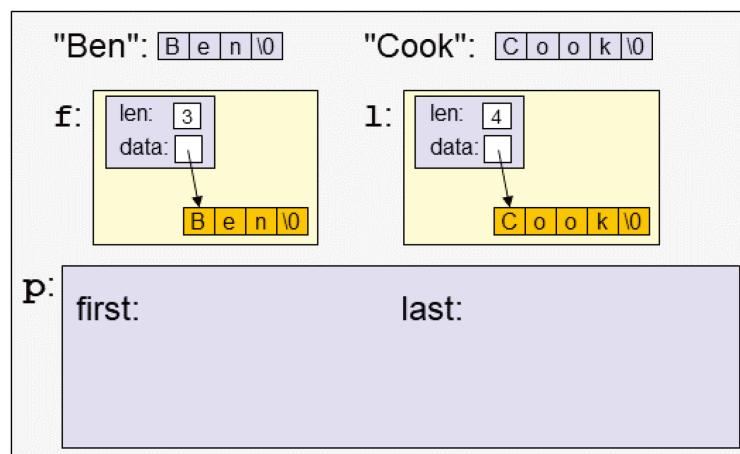
basics/customer.hpp

```
1 #include <string>
2 class Person {
3 private:
4     std::string first; // first name
5     std::string last; // last name
6 public:
7     Person(const std::string& f, const std::string& l)
8         : first{f}, last{l} {
9     }
10    ...
11};
```

当用两个字符串字面值初始化该类的对象时，会发生什么：

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

编译器发现提供的构造函数可以执行初始化，但参数的类型不适合。因此，编译器生成代码首先创建两个临时 std::string，由两个字符串字面量的值初始化，并将形参 *f* 和 *l* 进行绑定：

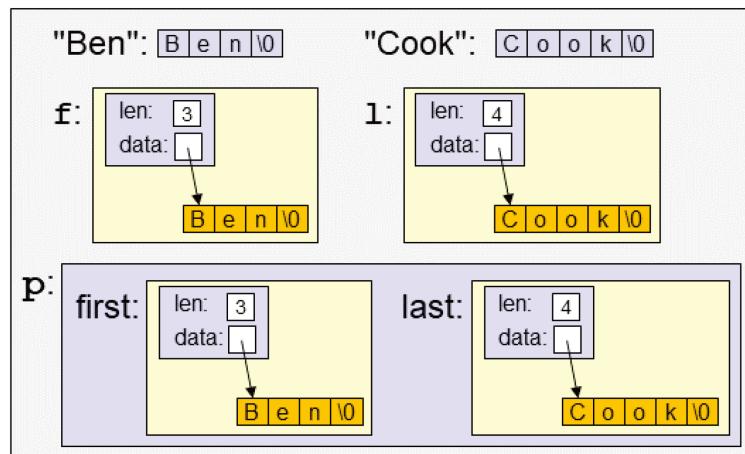


一般来说 (如果小字符串优化 (SSO) 不可用或者字符串太长)，这会为每个 `std::string` 的值分配内存。

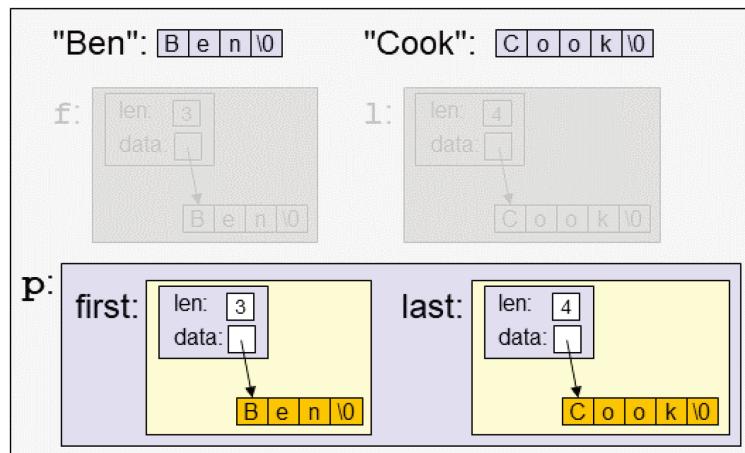
创建的临时字符串不能直接作为 `first` 或 `last` 使用，但现在却用于初始化这些成员。不幸的是，这里没有使用移动语义，原因有二：

- 形参 *f* 和 *l* 是对象，其名称存在的时间长于成员初始化的时间 (仍然可以在构造函数体中使用它们)。
- 形参声明为 *const*，即使使用 `std::move()` 也是禁用移动语义的。

因此，在每个成员初始化时调用 `string` 的复制构造函数，会再次为这些值分配内存：



在构造函数的末尾，销毁临时字符串：



这意味着有四个内存分配，尽管只有两个是必要的。所以，使用移动语义的话可以做得更好。

使用非 `const` 左值引用？

为什么这里不能简单地使用非 `const` 左值引用？

```
1 class Person {  
2     ...  
3     Person(std::string& f, std::string& l)  
4         : first{std::move(f)}, last{std::move(l)} {  
5     }  
6     ...  
7 };
```

但是，传递 `const std::string` 和临时对象（例如，由类型转换创建的）将无法编译：

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

通常，非 `const` 左值引用不会绑定到临时对象。因此，这个构造函数不能将 `f` 和 `l` 绑定到由字面值创建的临时字符串。

4.3.2 通过移动参数的值初始化成员

有了移动语义，构造函数就有了初始化成员的方法：构造函数按值接受每个参数，并将其移动到成员中：

basics/initmove.hpp

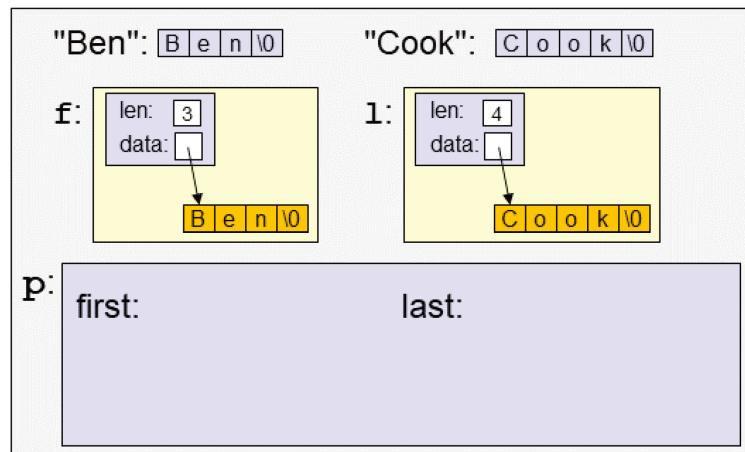
```
1 #include <string>
2 class Person {
3 private:
4     std::string first; // first name
5     std::string last; // last name
6 public:
7     Person(std::string f, std::string l)
8         : first{std::move(f)}, last{std::move(l)} {
9     }
10    ...
11};
```

这个构造函数接受所有可能的参数，并确保每个参数只有一个分配。

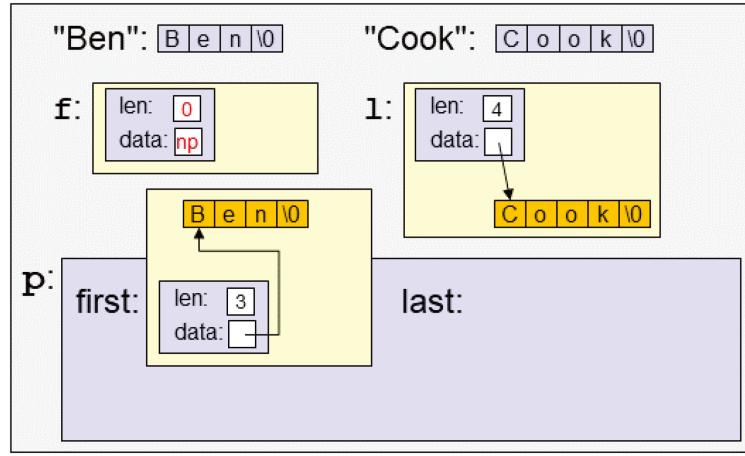
例如，如果传递两个字符串的字面值：

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

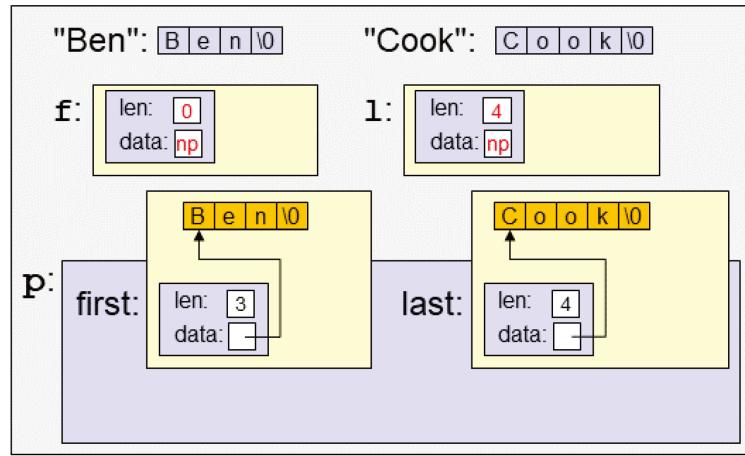
我们首先使用它们来初始化参数 f 和 l ：



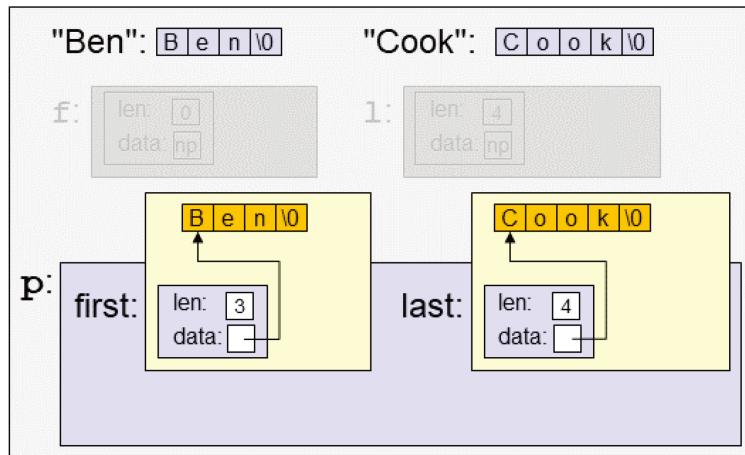
通过使用 $std::move()$ ，可以将形参的值移动到成员中。首先，成员首先从 f 中窃取值：



然后，成员 *last* 从 *l* 中窃取值：



同样，在构造函数的末尾销毁临时字符串。因为字符串的析构函数不再需要释放分配的内存，所以这一次它花费的时间更少：



如果传入 `std::string`，这种初始化成员的方式也能正常工作：

- 如果传递两个现有字符串而不使用 `std::move()` 标记，则将名称复制到形参中，并将它们移动到成员中：

```
1 std::string name1{"Jane"}, name2{"White"};
2 ...
3 Person p{name1, name2}; // OK, copy names into parameters and move them to
4 the members
```

- 如果传递两个不再需要该值的字符串，则根本不需要任何内存分配：

```
1 std::string firstname{"Jane"};
2 ...
3 Person p{std::move(firstname), // OK, move names via parameters to members
4 getLastnameAsString()};
5
```

本例中，移动了传递的字符串两次：一次是初始化参数 *f* 和 *l*，另一次是将 *f* 和 *l* 的值移动到成员中。

如果移动很廉价，那么在只有一个构造函数的实现时，任何初始化都可能是廉价的。

4.3.3 通过右值引用初始化成员

还有更多的方法可以初始化 *Person* 的成员，使用多个构造函数。

使用右值引用

为了支持移动语义，已经了解了可以将形参声明为非 *const* 右值引用。这允许参数从传递的临时对象或标记为 *std::move()* 的对象中窃取值。

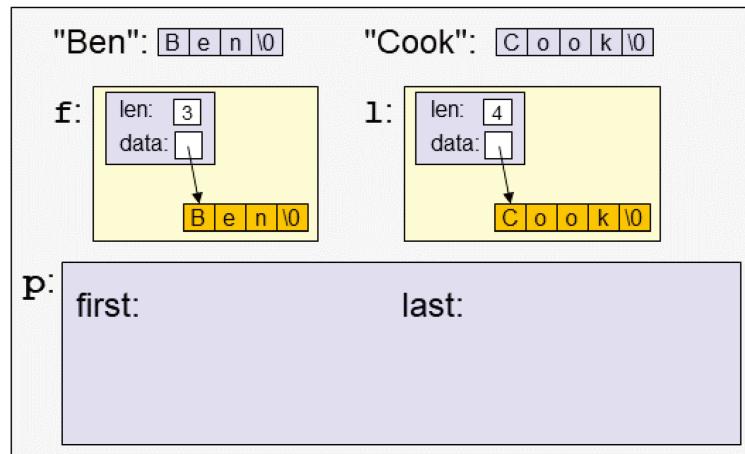
考虑这样声明的构造函数：

```
1 class Person {
2 ...
3 Person(std::string&& f, std::string&& l)
4 : first{std::move(f)}, last{std::move(l)} {
5 }
6 ...
7 };
```

这个初始化也适用于我们传递的字符串：

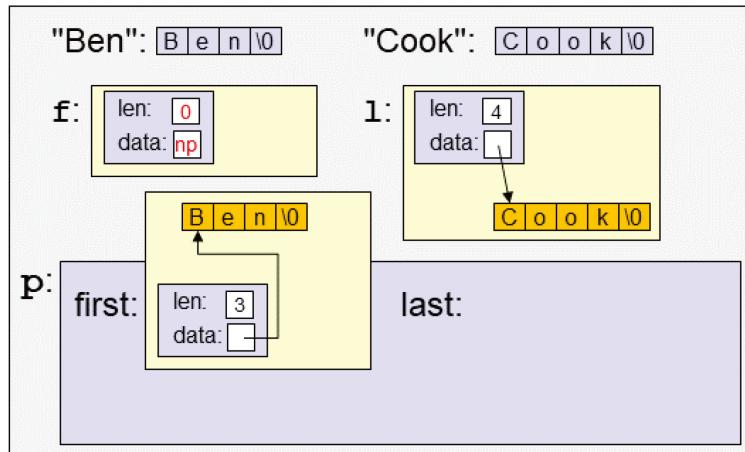
```
1 Person p{"Ben", "Cook"};
```

同样，因为构造函数需要字符串，我们创建 *f* 和 *l* 绑定的两个临时字符串：

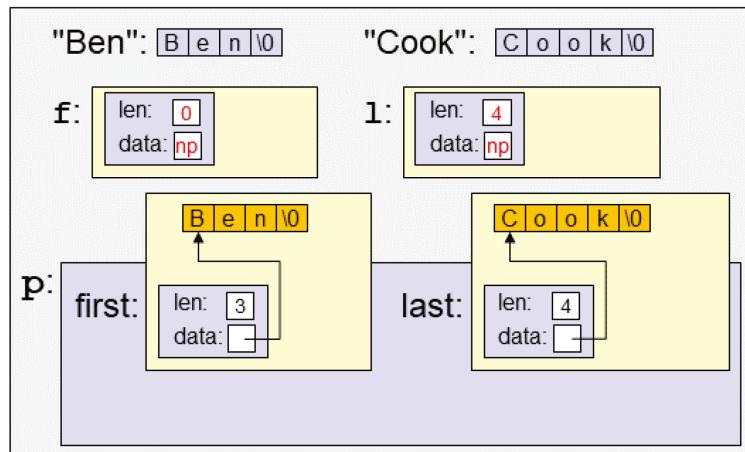


因为有非 *const* 引用，所以可以修改。本例中，用 *std::move()* 标记它们，以便成员的初始化可以窃取值。

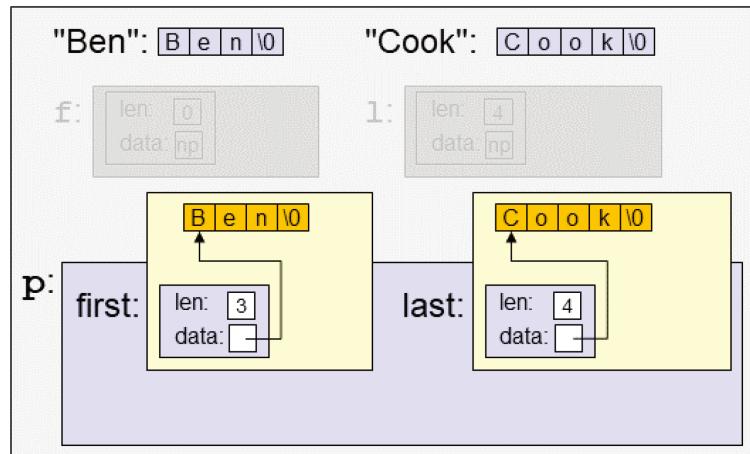
首先，成员首先从 *f* 中窃取值：



然后，成员 *last* 从 *l* 中窃取值：



同样，在构造函数的末尾，销毁临时字符串而不需要释放已分配的内存：



然而，此构造函数并非在所有情况下都有效。

重载右值和左值引用

虽然右值引用的函数在传递临时对象时工作得很好（传递由字符串字面值创建的临时 std::string），但也有限制：不能传递之后还需要值的已命名对象。因此，如果只有一个接受右值引用的构造函数，就不能传递现有的字符串：

```

1 class Person {
2 ...
3 Person(std::string&& f, std::string&& l)
4 : first{std::move(f)}, last{std::move(l)} {
5 }
6 ...
7 };
8
9 Person p1{"Ben", "Cook"}; // OK
10
11 std::string name1{"Jane"}, name2{"White"};
12 ...
13 Person p2{name1, name2}; // ERROR: can't pass a named object to an rvalue reference

```

对于 *p2*，我们需要一个传统的构造函数，使用 *const* 左值引用声明。然而，也可以传递一个字符串字面值和一个现有的字符串。因此，总共需要 4 个构造函数来处理所有可能的组合：

```

1 class Person {
2 ...
3 Person(const std::string& f, const std::string& l)
4 : first{f}, last{l} {
5 }
6 Person(const std::string& f, std::string&& l)
7 : first{f}, last{std::move(l)} {
8 }
9 Person(std::string&& f, const std::string& l)
10 : first{std::move(f)}, last{l} {
11 }
12 Person(std::string&& f, std::string&& l)

```

```
13 : first{std::move(f)}, last{std::move(l)} {  
14 }  
15 ...  
16};
```

这样，我们可以以任意组合传递字符串字面值和现有字符串，而且每个成员只需一个分配。

重载字符串字面值

为了进一步提高性能，甚至可能有特定的实现，将字符串文字作为普通指针。甚至可以避免一些动作。然而，实现所有构造函数有点无趣：

basics/initall.hpp

```
1 #include <string>  
2 class Person {  
3     private:  
4         std::string first; // first name  
5         std::string last; // last name  
6     public:  
7         Person(const std::string& f, const std::string& l)  
8             : first{f}, last{l} {}  
9         Person(const std::string& f, std::string&& l)  
10            : first{f}, last{std::move(l)} {}  
11        Person(std::string&& f, const std::string& l)  
12            : first{std::move(f)}, last{l} {}  
13        Person(std::string&& f, std::string&& l)  
14            : first{std::move(f)}, last{std::move(l)} {}  
15        Person(const char* f, const char* l)  
16            : first{f}, last{l} {}  
17        Person(const char* f, const std::string& l)  
18            : first{f}, last{l} {}  
19        Person(const char* f, std::string&& l)  
20            : first{f}, last{std::move(l)} {}  
21        Person(const std::string& f, const char* l)  
22            : first{f}, last{l} {}  
23        Person(const std::string&& f, const char* l)  
24            : first{f}, last{std::move(l)} {}  
25        Person(std::string&& f, const char* l)  
26            : first{std::move(f)}, last{l} {}  
27        ...  
28    };  
29};
```

这个解决方案的好处是减少了移动的次数。如果传递一个字符串字面值，则直接使用传递的指针初始化成员，而不是创建 std::string 并将其值移动到成员。

4.3.4 比较不同的方法

随着初始化成员的新方法的引入，应该在什么时候使用哪种方式？

通常，应该有一个很好的理由不使用 *const&*，通常是性能表现。因此，测量一下使用三种参数初始化 person 需要多长时间：传递字符串字面值，传递现有字符串，以及传递标有 std::move() 的字符串：

```
1 std :: string fname = "a first name";
2 std :: string lname = "a last name";
3
4 // measure how long this takes:
5 Person p1{"a firstname", "a lastname"};
6 Person p2{fname, lname};
7 Person p3{std :: move(fname), std :: move(lname)};
```

然而，为了避免小字符串优化（这意味着字符串根本不分配任何内存），我们应该使用具有显著长度的字符串。所以，这里有完整的函数来衡量不同的方法：

basics/initmeasure.hpp

```
1 #include <chrono>
2
3 // measure num initializations of whatever is currently defined as Person:
4 std :: chrono :: nanoseconds measure(int num)
5 {
6     std :: chrono :: nanoseconds totalDur{0};
7     for (int i = 0; i < num; ++i) {
8         std :: string fname = "a firstname a bit too long for SSO";
9         std :: string lname = "a lastname a bit too long for SSO";
10
11     // measure how long it takes to create 3 Persons in different ways:
12     auto t0 = std :: chrono :: steady_clock :: now();
13     Person p1{"a firstname too long for SSO", "a lastname too long for SSO"};
14     Person p2{fname, lname};
15     Person p3{std :: move(fname), std :: move(lname)};
16     auto t1 = std :: chrono :: steady_clock :: now();
17     totalDur += t1 - t0;
18 }
19 return totalDur;
20 }
```

函数 measure() 返回执行上述三种初始化的 num 迭代的持续时间，并使用长度够长的字符串。

现在，将上面 Person 的不同定义与 measure 函数结合起来，并使用 main() 函数调用 measure 函数，并打印执行的时间。例如：

basics/initclassicperf.cpp

```
1 #include "initclassic.hpp"
```

```

2 #include "initmeasure.hpp"
3 #include <iostream>
4 #include <cstdlib> // for std::atoi()
5
6 int main(int argc, const char** argv)
7 {
8     int num = 1000; // num iterations to measure
9     if (argc > 1) {
10         num = std::atoi(argv[1]);
11     }
12
13     // a few iterations to avoid measuring initial behavior:
14     measure(5);
15
16     // measure (in integral nano- and floating-point milliseconds):
17     std::chrono::nanoseconds nsDur{measure(num)};
18     std::chrono::duration<double, std::milli> msDur{nsDur};
19
20     // print result:
21     std::cout << num << " iterations take: "
22     << msDur.count() << "ms\n";
23     std::cout << "3 inits take on average: "
24     << nsDur.count() / num << "ns\n";
25 }

```

另外两个程序，[basics/initallperf.cpp](#)和[basics/initmoveperf.cpp](#)，只需对 *Person* 的其他声明使用不同的头文件即可。

使用三个不同的编译器，在三个不同的平台上运行这段代码的效果如下：

- 使用经典左值引用 (*const &*) 的初始化比其他初始化花费的时间要多得多。差不多是其他的 2 倍时间。
- 实现所有 9 个构造函数与只接受参数值和移动的构造函数之间没有很大区别。

如果通过使用非常短的字符串从小字符串优化 (SSO) 中受益，这意味着根本不分配任何内存（并且移动语义应该没有显著帮助），那么耗时是非常接近的。然而，使用接受 *const* 左值引用的传统构造函数仍然有一点缺点。

除了尝试不同的测试程序，还可以使用[basics/initperf.cpp](#)作为一个组合程序，在其他的平台上执行所有测试。

然而，可能有非常大型的成员不能从移动语义中获益（例如一个包含 10000 个双精度值的数据）：

```

1 class Person {
2 private:
3     std::string name;
4     std::array<double, 10000> values; // move can't optimize here
5 public:
6     ...
7 };

```

这种情况下，会遇到一个问题：将初始参数的值乘以 10000 倍，然后移动。我们必须复制参数两次，这几乎需要两倍的时间。请参见[basics/initbigper.cpp](#)，以获得一个完整的程序示例。

4.3.5 成员初始化总结

总的来说，要初始化移动语义对其有显著影响的成员（字符串、容器或具有此类成员的类/数组），应该在以下选项上使用移动语义：

- 从通过左值引用获取形参切换到通过值获取形参并将其移动到成员中
- 重载移动语义的构造函数

第一个选项允许我们只有一个构造函数，这样代码更容易维护。然而，这确实会导致不必要的移动操作。因此，如果移动操作可能花费大量时间，最好使用多个重载。

例如，如果有一个带有字符串和值向量的类，按值和移动通常是正确的方法：

```
1 class Person {
2 private:
3     std::string name;
4     std::vector<std::string> values;
5 public:
6     Person(std::string n, std::vector<std::string> v)
7         : name{std::move(n)}, values{std::move(v)} {
8     }
9     ...
10};
```

但是，如果有 `std::array` 成员，最好重载。因为即使移动 `std::array`，也要花费大量的时间：

```
1 class Person {
2 private:
3     std::string name;
4     std::array<std::string, 1000> values;
5 public:
6     Person(std::string n, const std::array<std::string, 1000>& v)
7         : name{std::move(n)}, values{v} {
8     }
9     Person(std::string n, std::array<std::string, 1000>&& v)
10    : name{std::move(n)}, values{std::move(v)} {
11    }
12    ...
13};
```

4.3.6 总要使用移动进行值传递吗？

上面的讨论引出了这样的问题：现在是否应该总是按值接受参数，并将它们移动来设置内部值或成员。答案是否定的。

这里讨论的特殊情况是创建并初始化一个新值。在这种情况下，这种策略获得了回报。如果已经有一个值，需要对其进行更新或修改，那么使用这种方法将适得其反。

一个简单的例子是 `setter`。`Person` 有以下实现：

```

1 class Person {
2     private:
3         std::string first; // first name
4         std::string last; // last name
5     public:
6         Person(std::string f, std::string l)
7             : first{std::move(f)}, last{std::move(l)} {
8         }
9         ...
10        void setFirstname(std::string s) { // take by value
11            first = std::move(s); // and move
12        }
13        ...
14    };

```

假设这样使用:

```

1 Person p{ "Ben" , "Cook" };
2 std::string name1{ "Ann" };
3 std::string name2{ "Constantin Alexander" };
4
5 p.setFirstname(name1);
6 p.setFirstname(name2);
7 p.setFirstname(name1);
8 p.setFirstname(name2);

```

每次设置新名字时，都创建一个新的临时形参 *s*，并分配内存，然后将其移动到成员的值。因此，有四个分配（假设我们没有 SSO）。

现在考虑以传统的方式实现 setter，采用 *const* 左值引用:

```

1 class Person {
2     private:
3         std::string first; // first name
4         std::string last; // last name
5     public:
6         Person(std::string f, std::string l)
7             : first{std::move(f)}, last{std::move(l)} {
8         }
9         ...
10        void setFirstname(const std::string& s) { // take by reference
11            first = s; // and assign
12        }
13        ...
14    };

```

绑定到传递的参数不会创建新字符串。此外，赋值操作符只在新长度超过当前为该值分配的内存量时才分配新内存。这意味着，因为我们已经有了一个值，通过值移动来获取参数的方法可能会适得其反。

你可能想知道是否重载 setter，以便在新长度超过现有长度时从移动语义中获益:

```

1 class Person {
2     private:
3         std::string first; // first name
4         std::string last; // last name
5     public:
6         Person(std::string f, std::string l)
7             : first{std::move(f)}, last{std::move(l)} {
8         }
9         ...
10        void setFirstname(const std::string& s) { // take by lvalue reference
11            first = s; // and assign
12        }
13        void setFirstname(std::string&& s) { // take by rvalue reference
14            first = std::move(s); // and move assign
15        }
16        ...
17    };

```

然而，即使是这种方法也可能适得其反，因为移动分配可能会收缩容量：

```

1 Person p{ "Ben" , "Cook" };
2
3 p.setFirstname( "Constantin Alexander" ); // would allocate enough memory
4 p.setFirstname( "Ann" ); // would reduce capacity
5 p.setFirstname( "Constantin Alexander" ); // would have to allocate again

```

即使使用移动语义，设置现有值的最佳方法是通过 *const* 左值引用和赋值来获取新值，而不使用 *std::move()*。

按值接受参数并将其移动到需要新值的地方，只有将传递的值存储为新值时才有用（这样我们无论如何都需要新的内存）。当修改现有值时，这个策略可能会适得其反。

然而，初始化成员并不是“取值然后移动”唯一的应用。向容器添加新值的 (*member*) 函数是则是另一种应用：

```

1 class Person {
2     private:
3         std::string name;
4         std::vector<std::string> values;
5     public:
6         Person(std::string n, std::vector<std::string> v)
7             : first{std::move(n)}, values{std::move(v)} {
8         }
9         ...
10        // better pass by value and move to create a new element:
11        void addValue(std::string s) { // take by value
12            values.push_back(std::move(s)); // and move into the collection
13        }
14        ...
15    };

```

4.4 类中使用移动语义

任何复制构造函数、复制赋值或析构函数的声明都禁用了对移动语义的自动支持。这也适用于多态基类。然而，还有一些其他方面需要考虑。

4.4.1 实现多态基类

多态基类通常引入虚成员函数，可以调用派生类的所有对象。例如：

```
1 class GeoObj {
2     public:
3     virtual void draw() const = 0; // pure virtual function (introducing the API)
4     ...
5     virtual ~GeoObj() = default; // let delete call the right destructor
6     ... // other special member functions due to the problem of slicing
7 };
```

这个基类中，禁用移动语义，如果移动 *GeoObj* 对象，基类中声明的成员不会自动支持移动语义。如果有受保护的复制构造函数和一个删除的赋值操作符也适用，通常在多态基类中应该有这样的操作符，避免切片问题。

只要基类不引入成员，不支持移动语义就没有效果。但是，如果这个基类中有一个开销很大的成员，就已经禁用了对移动语义的支持。例如：

```
1 class GeoObj {
2     protected:
3     std::string name; // name of the geometric object
4     public:
5     ...
6     virtual void draw() const = 0; // pure virtual function (introducing the API)
7     ...
8     virtual ~GeoObj() = default; // disables move semantics for name
9     ... // other special member functions due to the problem of slicing
10 };
```

要再次启用移动语义，可以显式声明移动操作为默认值。但正如我们刚刚了解到的，这禁用了复制特殊成员函数。因此，如果想要使用这些函数，就必须显式地提供。

处理切片

但是，有切片的问题。考虑以下代码，使用基类 *GeoObj* 的引用作为派生类 *Circle* 的对象：

```
1 Circle c1{ ... }, c2{ ... };
2
3 GeoObj& geoRef{c1};
4 geoRef = c2; // OOPS: uses GeoObj::operator=() and assigns no Circle members
```

为 *GeoObj* 调用赋值操作符，而且该操作符不是虚操作符，所以编译器调用 *GeoObj::operator=()*，不处理任何派生类的成员。即使用 *virtual* 声明赋值操作符也没有帮助，因为派生类的操作符不会覆盖基类的赋值操作符（第二个操作数的形参类型不同）。

为了避免这个问题，应该禁用在多态类层次结构中使用赋值操作符。此外，如果不是抽象类，还应该避免使用公共复制构造函数来禁用到基类的隐式类型转换。因此，具有移动语义（和成员）的多态基类应该如下声明：

```
1 class GeoObj {
2     protected:
3         std::string name; // name of the geometric object
4         GeoObj(std::string n)
5             : name{std::move(n)} {
6         }
7     public:
8         virtual void draw() const = 0; // pure virtual function (introducing the API)
9         ...
10        virtual ~GeoObj() = default; // would disable move semantics for name
11    protected:
12        // enable copy and move semantics (callable only for derived classes):
13        GeoObj(const GeoObj&) = default;
14        GeoObj(GeoObj&&) = default;
15        // disable assignment operator (due to the problem of slicing):
16        GeoObj& operator=(GeoObj&&) = delete;
17        GeoObj& operator=(const GeoObj&) = delete;
18    };
```

参见[poly/geoobj.hpp](#)获取完整的头文件。

4.4.2 实现派生类的多态

派生类的多态看起来如下所示（参见 `poly/polygon.hpp` 的完整头文件）：

```
1 class Polygon : public GeoObj {
2     protected:
3         std::vector<Coord> points;
4     public:
5         Polygon(std::string s, std::initializer_list<Coord> = {}); // constructor
6         virtual void draw() const override; // implementation of draw()
7     };
```

通常，多态派生类中不需要声明特殊的成员函数。特别是，不需要再次声明虚析构函数（除非必须实现）。再次声明析构函数（无论是否是虚函数）将禁用派生类成员（这里是 `vector`）对移动语义的支持：

```
1 class Polygon : public GeoObj {
2     protected:
3         std::vector<Coord> points;
4     public:
5         Polygon(std::string s, std::initializer_list<Coord> = {}); // constructor
6         ...
7         virtual ~Polygon() = default; // OOPS: don't do that because it disables move
8             semantics
9     };
```

在不声明析构函数的情况下，移动语义适用于 *Polygon* 成员、*name* 和 *points*。

poly/polygon.cpp

```
1 #include "geoobj.hpp"
2 #include "polygon.hpp"
3
4 int main()
5 {
6     Polygon p0{"Poly1", {Coord{1,1}, Coord{1,9}, Coord{9,9}, Coord{9,1}}};
7     Polygon p1{p0}; // copy
8     Polygon p2{std::move(p0)}; // move
9
10    p0.draw();
11    p1.draw();
12    p2.draw();
13 }
```

这个程序有以下输出：

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

对于这两个成员，*name* 和 *points*，值都从 *p0* 移动到 *p2*。

注意，如果必须在 *Polygon* 类中实现移动构造函数，需要特别注意提供正确的条件。

4.5 总结

- 避免有名称的对象。
- 避免不必要的 *std::move()*。尤其不要在返回局部对象时使用。
- 从形参初始化成员的构造函数（对于这种构造函数，移动操作很廉价），应该按值接受实参并将其移动到成员。
- 从形参初始化成员的构造函数，移动操作需要大量时间，应该重载移动语义以获得最佳性能。
- 一般来说，从参数中创建和初始化新值（对于移动操作来说成本很低）应该按值和移动的方式接受参数。但是，不要以值为单位来更新/修改现有的值。
- 不要在派生类中声明虚析构函数（除非必须实现）。

5 引用的重载

本章讨论不同引用限定符的重载成员函数。比如，关于 getter 应该通过值返回，还是通过常量引用返回的问题。

5.1 getter 的返回类型

在 C++11 之前，当为复制开销很大的成员实现 getter 时，有以下几种选择：

- 返回值
- 通过左值引用返回

简单地讨论一下这些方案。

5.1.1 传值返回

按值返回的 getter 看起来是这样的（记住：不要使用 `const` 按值返回，否则禁用移动语义）：

```
1 class Person
2 {
3 private:
4     std::string name;
5 public:
6     ...
7     std::string getName() const {
8         return name;
9     }
10};
```

这段代码是安全的，但每次获取 `name` 时，都可能复制 `name`。

例如，只是检查是否有没名字的人会有很大的开销：

```
1 std::vector<Person> coll;
2 ...
3 for (const auto& person : coll) {
4     if (person.getName().empty()) { // OOPS: copies the name
5         std::cout << "found empty name\n";
6     }
7 }
```

如果将此方法与返回引用的方法进行比较，可以看到按值返回字符串的版本的性能开销是引用的 2 到 100 倍（前提是名称的长度较大，这样 SSO 就没有帮助了）。对图像或数千个元素的集合的成员可能更糟糕。在这种情况下，getter 通常返回 (`const`) 引用以提高性能。

5.1.2 引用返回

通过引用返回的 getter 看起来像这样：

```
1 class Person
2 {
3 private:
```

```

4   std :: string name;
5 public:
6 ...
7 const std :: string& getName() const {
8     return name;
9 }
10 };

```

这样更快，但不安全，因为调用者必须确保返回引用的对象生命周期足够长。实际上，使用返回引用值比原始对象的时间长的话，会有生命周期上的风险。

会踩入这种陷阱的一种方式是使用基于范围的 for 循环，如下所示：

```

1 for (char c : returnPersonByValue().getName()) { // OOPS: undefined behavior
2     if (c == ',') {
3         ...
4     }
5 }

```

注意，循环的右边有一个函数，它返回一个我们用 getter 引用的临时对象。但是，定义了基于范围的 for 循环，使上面的代码等价于以下代码：

```

1 reference range = returnPersonByValue().getName();
2 // OOPS: returned temporary object destroyed here
3 for (auto pos = range.begin(), end = range.end(); pos != end; ++pos) {
4     char c = *pos;
5     if (c == ',') {
6         ...
7     }
8 }

```

开始迭代之前，初始化 *reference1*，必须使用传递的范围两次（一次调用 *begin()*，一次调用 *end()*），并希望避免创建副本（这可能代价很高，甚至不可能）。一般来说，引用会延长所引用内容的生命周期。在本例中，*range* 并不引用由 *returnPersonByValue()* 返回的 *Person*，而 *range* 指向 *getName()* 的返回值，是对返回的 *Person* 的引用。因此，*range* 扩展了引用的生命周期。因此，在第一个语句的末尾，返回的临时对象将销毁，并且在遍历名称的字符时使用对已销毁对象名称的引用。

最好的情况下，在这里会得到一个核心转储（段错误），这样就可以看到出现了明显的错误。最坏的情况是，发布了软件，并出现了未定义行为。

如果 getter 按值返回名称，这样的代码就不会有问题。这样，*range* 将扩展名称副本的生存期，以便我们使用该名称直到 *range* 的生存期结束。

5.1.3 使用移动语义解决困境

通过移动语义，就可以解决这个困境的方法。如果这样做安全，可以通过引用返回。如果遇到生命周期的问题，可以通过值返回。

方法如下：

```

1 class Person

```

```

2 {
3     private:
4         std :: string name;
5     public:
6         ...
7         std :: string getName() && { // when we no longer need the value
8             return std :: move(name); // we steal and return by value
9         }
10        const std :: string& getName() const& { // in all other cases
11            return name; // we give access to the member
12        }
13    };

```

用不同的引用限定符重载 getter，方法与重载带有 `&&` 和 `const&` 形参的函数相同：

- 带有 `&&` 限定符的是有不再需要值的对象时使用（一个即将死亡的对象或用 `std::move()` 标记的对象）。
- 带有 `const&` 限定符的版本用于所有其他情况。总是合适的，但只是无法获得 `&&` 版本时的备选方案。因此，如果有一个不会销毁的对象或没标记为 `std::move()`，就会使用此函数。

现在的性能和安全性都很好：

```

1 Person p{ "Ben" };
2 std :: cout << p.getName(); // 1) fast (returns reference)
3 std :: cout << returnPersonByValue().getName(); // 2) fast (uses move())
4
5 std :: vector<Person> coll;
6 ...
7 for (const auto& person : coll) {
8     if (person.getName().empty()) { // 3) fast (returns reference)
9         std :: cout << "found empty name\n";
10    }
11 }
12
13 for (char c : returnPersonByValue().getName()) { // 4) safe and fast (uses move())
14     if (c == ',') {
15         ...
16     }
17 }

```

语句 1) 和 3) 使用 `const&` 的版本，因为有一个对象没有使用 `std::move()` 标记。语句 2) 和 4) 使用 `&&` 的版本，因为我们为一个临时对象调用 `getName()`。因为临时对象即将销毁，getter 可以将成员名移出作为返回值，这意味着不必为返回值分配新的内存，可以窃取了成员的值。

您可能还记得，`return` 语句不应该使用 `std::move()` 来返回已经销毁的局部对象。本例中，不返回任何局部对象，而是返回一个成员，其生存期不以成员函数的结束为结束。

`std::move()` 用于成员函数中

请注意，这个特性意味着即使在调用成员函数时也可以使用 `std::move()`。例如：

```

1 void foo()
2 {
3     Person p{ ... };
4     ...
5     coll.push_back(p.getName()); // calls getName() const&
6     ...
7     coll.push_back(std::move(p).getName()); // calls getName() && (OK, p no longer
8     used)
}

```

使用 `getName()` 时，使用 `std::move()` 将提高程序的性能。不返回对 `const std::string` 的引用（只能复制），返回值是作为非 `const` 字符串返回的 `p` 的移动名，因此 `push_back()` 可以使用移动语义将其移动到 `coll` 中。通常，在这之后，`p` 处于有效但未定义的状态。

关于在 C++ 标准库中使用此特性的示例，请参见 `std::optional<>`。

5.2 重载

从 C++98 开始，可以重载成员函数来实现 `const` 和非 `const` 版本。例如：

```

1 class C {
2 public:
3     ...
4     void foo(); // foo() for non-const objects
5     void foo() const; // foo() for const objects
6 };

```

圆括号后面的限定符允许限定一个没有传递给形参的对象：可以调用此成员函数的对象。

有了移动语义，就有了用限定符重载函数的新方。考虑以下代码：

[basics/refqual.cpp](#)

```

1 #include <iostream>
2 class C {
3 public:
4     void foo() const& {
5         std::cout << "foo() const&\n";
6     }
7     void foo() && {
8         std::cout << "foo() &&\n";
9     }
10    void foo() & {
11        std::cout << "foo() &\n";
12    }
13    void foo() const&& {
14        std::cout << "foo() const&&\n";
15    }
16 };
17
18 int main()

```

```

19 {
20     C x;
21     x.foo(); // calls foo() &
22     C{}.foo(); // calls foo() &&
23     std::move(x).foo(); // calls foo() &&
24
25     const C cx;
26     cx.foo(); // calls foo() const&
27     std::move(cx).foo(); // calls foo() const&&
28 }

```

这个程序演示了所有可能的引用限定符，以及何时调用。通常，只有两到三个这样的重载，比如对 getter 使用 `&&` 和 `const&(和 &)`。

还要注意，不允许引用和非引用限定符的重载：

```

1 class C {
2 public:
3     void foo() &&;
4     void foo() const; // ERROR: can't overload by both reference and value qualifiers
5 };

```

5.3 何时使用引用

引用限定符允许对特定类别的对象调用函数时，以不同的方式实现函数。目标是在为不再需要其值的对象，调用不同的成员函数。

虽然确实有这个特性，但没有使用。可以（也应该）用它来确保修改对象的操作，不会让即将销毁的临时对象调用。

5.3.1 赋值操作符的引用限定符

更好地使用引用限定符的方式是修改赋值操作符的实现。如 <http://wg21.link/n2819> 中建议的那样，在可能的地方使用引用限定符声明赋值操作符可能会更好。

例如，字符串的赋值操作符声明如下：

```

1 namespace std {
2     template<typename charT, ...>
3     class basic_string {
4     public:
5         ...
6         constexpr basic_string& operator=(const basic_string& str);
7         constexpr basic_string& operator=(basic_string&& str) noexcept( ... );
8         constexpr basic_string& operator=(const charT* s);
9         ...
10    };
11 }

```

这允许将新值赋给临时字符串：

```

1 std::string getString();
2
3 getString() = "hello"; // OK
4 foo(getString() = ""); // passes string instead of bool

```

考虑用引用限定符声明赋值操作符:

```

1 namespace std {
2     template<typename charT, ... >
3     class basic_string {
4     public:
5         ...
6         constexpr basic_string& operator=(const basic_string& str) &;
7         constexpr basic_string& operator=(basic_string&& str) & noexcept( ... );
8         constexpr basic_string& operator=(const charT* s) &;
9         ...
10    };
11 }

```

代码将不再编译:

```

1 std::string getString();
2
3 getString() = "hello"; // ERROR
4 foo(getString() = ""); // ERROR

```

注意，特别是对于可以用作布尔值的类型，将有助于找到如下错误:

```

1 std::optional<int> getValue();
2
3 if (getValue() = 0) { // OOPS: compiles although = is used instead of ==
4     ...
5 }

```

其实，会给临时对象返回基本类型的属性: 右值。

注意，所有这些修改 C++ 标准的建议都被拒绝了，主要是考虑向后兼容性。然而，在实现自己的类时，可以使用以下改进的方式:

```

1 class MyType {
2 public:
3     ...
4     // disable assigning value to temporary objects:
5     MyType& operator=(const MyType& str) & =default;
6     MyType& operator=(MyType&& str) & =default;
7
8     // because this disables the copy/move constructor, also:
9     MyType(const MyType&) =default;
10    MyType(MyType&&) =default;
11    ...
12 }

```

通常，对可能修改对象的成员函数执行此操作。

5.3.2 其他成员函数的引用限定符

如 getter 示例所示，引用限定符也可以而且应该在返回对对象的引用时使用。这样，可以减少访问已销毁临时对象的风险。

同样，标准字符串的当前声明可以作为一个例子：

```
1 namespace std {
2     template<typename charT, ... >
3     class basic_string {
4     public:
5         ...
6         constexpr const charT& operator[](size_type pos) const;
7         constexpr charT& operator[](size_type pos);
8         constexpr const charT& at(size_type n) const;
9         constexpr charT& at(size_type n);
10
11        constexpr const charT& front() const;
12        constexpr charT& front();
13        constexpr const charT& back() const;
14        constexpr charT& back();
15        ...
16    };
17 }
```

相反，下面的重载会更好一些：

```
1 namespace std {
2     template<typename charT, ... >
3     class basic_string {
4     public:
5         ...
6         constexpr const charT& operator[](size_type pos) const&;
7         constexpr charT& operator[](size_type pos) &;
8         constexpr charT operator[](size_type pos) &&;
9         constexpr const charT& at(size_type n) const&;
10        constexpr charT& at(size_type n) &;
11        constexpr charT at(size_type n) &&;
12
13        constexpr const charT& front() const&;
14        constexpr charT& front() &;
15        constexpr charT front() &&;
16        constexpr const charT& back() const&;
17        constexpr charT& back() &;
18        constexpr charT back() &&;
19        ...
20    };
21 }
```

同样，由于向后兼容性，C++ 标准中相应的更改可能会成为一个问题。但可以为自定义的类型提供这些重载。这种情况下，不要忘记右值引用的实现可以移出大型成员。

5.4 总结

- 可以在不同的引用限定符上重载成员函数。
- 重载 getter 用于具有引用限定符的大型成员，使其既安全又快速。
- 使用 `std::move()` 标记对象是有意义的，即使在调用成员函数时也是如此。
- 在赋值操作符中使用引用限定符。

6 已移动状态

虽然生成的或实现的移动语义通常不错，但至少应该了解可能的情况：移动操作会将对象置为 C++ 标准库不支持的状态，或者破坏不变量。

在这章中，我们根据 C++ 标准库，来了解“无效”状态的定义，即已移动的对象所处的状态。

6.1 已移动对象的要求和状态

移动之后，已移动的对象既没有部分销毁，也没有完全销毁。析构函数还未调用，直到已移动对象的生命周期结束时才调用。因此，析构函数至少要没问题。

然而，C++ 标准库为可移动类型提供了更多的保证。已移动对象处于“有效但未定义的状态”，所以可以像使用任何不知道其值的类型的对象一样使用已移动对象。像使用该类型的非 *const* 引用形参，而不知道所传递对象的值一样。例如，可以做的不仅仅是销毁已移动对象，还可以使用移动语义来实现排序和可变序列算法。

为了更详细地理解如何处理已移动对象，最好区分与之相关的信息：

- C++ 标准库安全地使用已移动对象有什么要求？
- 给已移动对象什么样的保证，才能让这些类型的用户知道如何使用？

至少应该满足 C++ 标准库的要求。

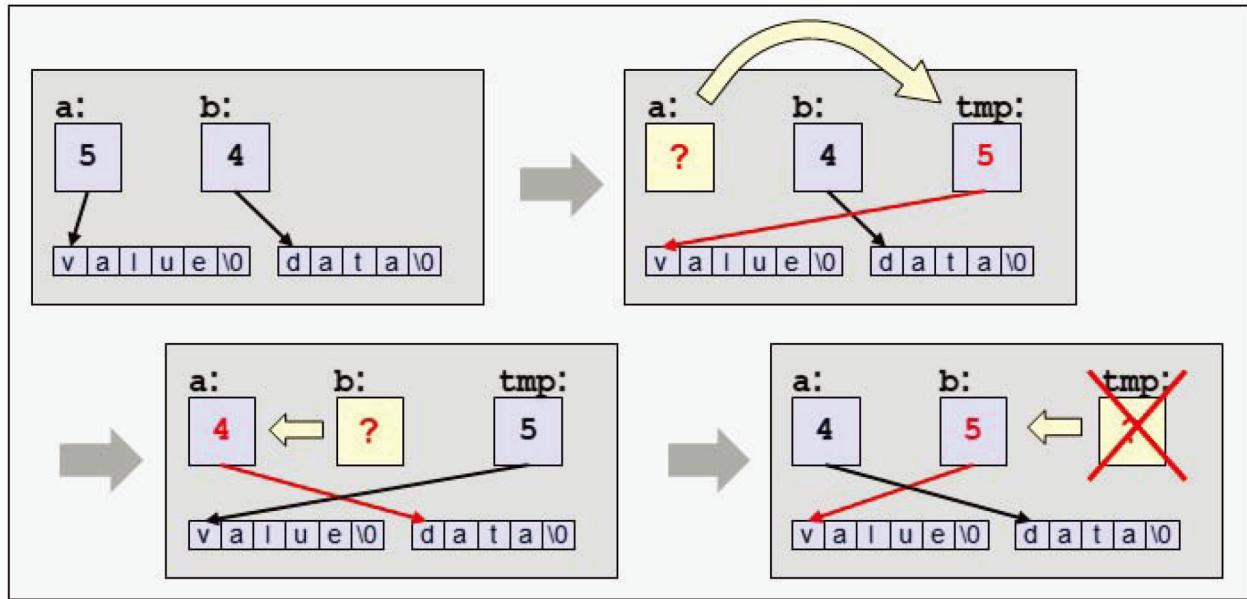
6.1.1 已移动对象的状态要求

C++ 标准库对已移动对象的要求没有什么特别的。对于任何函数，为传递的类型和对象制定的需求，也适用于任何内部传递或是移动来的对象。

已移动对象必须能够销毁。此外，必须能够将新值赋给已移动对象。

例如，考虑如何交换两个对象 *a* 和 *b*(这可能是排序操作的一部分)。交换的实现通常如下(见图 6.1)：

- 将 *a* 移动到一个新的临时对象 *tmp*(这样 *a* 就变成了已移动的对象)。
- 将 *b* 移动赋值给 *a*(这样 *b* 就成为了已移动对象)。
- 将 *tmp* 移动赋值给 *b*(这样 *tmp* 就成了已移动对象)。
- 销毁已移动对象 *tmp*。



注意，可以通过向两个参数传递相同的对象来实现自交换。这种情况下，可以将状态为已移动的对象赋值给它自己。

所以，对于已移动对象，有同样的要求，适用于所有对象：

- 必须能够销毁已移动对象。
- 必须能够为已移动对象赋值。
- 应该能够复制、移动或将已移动对象赋值给另一个对象。

已移动对象还能够处理特殊操作的额外需求。例如排序，所以必须支持 `<` 操作符，或有相应的排序标准，这也适用于已移动的对象。您可能会争辩说，排序算法中应该知道哪个对象移动了，这样就可以避免比较它，但 C++ 标准库并不要求这样做。顺便一说，您可以传递已移动对象，对它们进行排序，只要支持所有必需的操作就可以了。

注意，C++ 标准库的定义需要的不仅仅是 Alexander Stepanov 和 Paul McJones 在《编程元素》一书的内容。

对于 C++ 标准库中使用的所有对象和类型，应该确保已移动对象也支持所调用函数的要求。

6.1.2 保证已移动对象的状态

已移动对象的保证定义在使用哪些代码。通常，类的设计者决定给出哪些保证，并且总在生命周期结束时销毁对象。对于移动状态，必须提定义良好的析构函数。

通常，会有更多要保障的。对于 C++ 标准库的要求，支持基本操作，如复制、赋值相同类型的对象通常就足够了。然而，用户通常希望可以使用处理所有其他方式来为对象赋值。因此，保证可以将新值“赋值”给已移动对象就非常有用。

考虑给标准字符串 `s` 赋新值的不同方法：

```

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

```

例如，下面的循环是将流逐行读入 vector 对象的常用方法：

```
1 std::string row;
2 while (std::getline(myStream, row)) {
3     coll.push_back(std::move(row)); // move the line into the vector
4 }
```

另一个例子中，可以在处理容器的泛型代码中实现以下语句：

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

类似的代码可以用于释放指针所使用的对象的内存：

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

通过声明，已移动对象处于有效但未定义的状态，C++ 标准为其库中的操作进行了保证。

例如，以下是根据 C++ 标准定义的行为：

```
1 std::stack<int> stk;
2 ...
3 foo(std::move(stk)); // stk gets unspecified state
4 stk.push(42);
5 ... // do something else without using stk
6 int i = stk.top();
7 assert(i == 42); // should never fail
```

虽然不知道通过 `std::move()` 将 `stk` 传递给 `foo()` 后的值，但只要使用它来保存 42，直到再次需要，都可以将它当做有效的堆栈。

当然，所提供的保证能达到什么程度取决于您，但要确保您的类型的用户知道这些保证。通常，希望您的类型提供与 C++ 库相同的保证，这样就可以像使用该类型的任何其他对象一样使用已移动对象了。

6.1.3 破坏不变量

C++ 标准库定义了所有处于“有效但未定义状态”的移动对象的含义如下：

对象的值没有指定，除非满足对象的不变量，并且对象上的操作按照指定的类型进行

不变量是适用于所有对象。有了这个保证，可以假定已移动的对象处于某种状态，这意味着它的不变量没有破坏。可以像使用非 `const` 引用参数一样使用对象，而不知道传递的参数的任何信息：可以调用任何没有约束或前置条件的操作，并且效果/结果是为该类型的其他对象指定。

例如，对于已移动的字符串，可以执行没有先决条件的操作：

- 查询大小
- 打印输出
- 遍历字符
- 转换为 C 字符串
- 分配新值

- 添加字符

此外，所有这些函数具有语义：

- 返回的大小可以安全地调用索引操作符。
- 返回的大小与迭代时的字符数相匹配。
- 遍历字符时，打印的字符与字符序列匹配。
- 添加一个字符将把该字符放在值的末尾（不知道是否还有其他字符在前面）。

开发者可以利用这些保证（参见 `std::stack<>` 的例子）。

有时候，必须确保不破坏不变量，否则默认的特殊移动成员函数无法正常工作。在下面的小节中根据示例进行讨论。

受限的不变量

不把 C++ 标准库的全部保证给已移动状态的对象也可以。也就是说，可能有意地限制已移动对象的操作。例如，当对象的有效状态总是需要诸如内存之类的资源时，只有一个部分支持的状态可能会更好地降低移动操作的成本。

理想情况下，不支持所有操作的状态是可查的。对象应该知道这个状态，并提供成员函数来检查这个状态。已移动对象也可能拒绝执行此状态下不支持的操作。然而，相应的检查可能会影响性能。

C++ 标准库中，有些类型提供了 API 来检查对象是否处于移出状态。例如，`std::futures` 有一个成员函数 `valid()`，该函数对已移动对象返回 `false`。所以，不同对象检查状态（是否移动）的接口不同。

通常，已移动状态是默认构造状态，这意味着已移动状态是不变量的一部分。任何情况下，需要让用户知道什么是可用的，什么不可用。

6.2 可销毁和可转移

我们讨论一下如何确保已移动对象满足支持分配和销毁的基本要求。

6.2.1 可分配和可销毁的已移动对象

大多数类中，生成的特殊移动成员函数对已移动的对象可以使用赋值操作符和析构函数。如果每个已移动成员都是可赋值和可销毁的，那么整个已移动对象的赋值和销毁都没问题：

- 通过从相应的源成员中获取状态，赋值将覆盖该成员的未定义状态。
- 析构函数将销毁成员（具有未定义状态）。

通常应该从对象中移动的成员，但对象处于有效但未定义的状态，所以生成的移动操作通常只对创建成员状态正确的对象进行操作。

例如，以下类：

```
1 class Customer {
2 private:
3     std::string name;
4     std::vector<int> values;
```

```
5     ...
6 };
```

当 *customer* 移走时，保证 *name* 和 *values* 都处于有效的状态，以便析构函数（由 *customer* 的析构函数调用）可以正常工作：

```
1 void foo()
2 {
3     Customer c{"Michael Spencer"};
4     ...
5     process(std::move(c));
6     // both name and values have valid but unspecified states
7     ...
8 } // destructor of c will clean up name and values (whatever their state is)
```

另外，给 *c* 赋一个新值也可以，所以就同时赋了 *name* 和 *values*。

6.2.2 不可销毁的已移动对象

这时在极少数情况下可能会出现的问题，通常实现赋值操作符或析构函数来完成对成员赋值或销毁的工作。

考虑以下类，使用固定大小的线程对象数组，其数量可变，为此在析构函数中显式调用 *join()* 来等待线程对象的结束：

basics/tasks.hpp

```
1 #include <array>
2 #include <thread>
3 class Tasks {
4     private:
5         std::array<std::thread, 10> threads; // array of threads for up to 10 tasks
6         int numThreads{0}; // current number of threads/tasks
7     public:
8         Tasks() = default;
9         // pass a new thread:
10        template <typename T>
11        void start(T op) {
12             threads[numThreads] = std::thread{std::move(op)};
13             ++numThreads;
14         }
15         ...
16         // at the end wait for all started threads:
17         ~Tasks() {
18             for (int i = 0; i < numThreads; ++i) {
19                 threads[i].join();
20             }
21         }
22     };
```

目前为止，该类还不支持移动语义，因为用户声明了析构函数。因为禁用复制 *std::thread*，所以也不能复制 *Tasks* 对象。但是，可以启动多个任务并等待结束：

basics/tasks.cpp

```
1 #include "tasks.hpp"
2 #include <iostream>
3 #include <chrono>
4 int main()
5 {
6     Tasks ts;
7     ts.start([]{
8         std::this_thread::sleep_for(std::chrono::seconds{2});
9         std::cout << "\nt1 done" << std::endl;
10    });
11    ts.start([]{
12        std::cout << "\nt2 done" << std::endl;
13    });
14 }
```

现在通过生成默认的移动操作来启用移动语义 (详见[basics/tasksbug.cpp](#)):

```
1 class Tasks {
2 private:
3     std::array<std::thread,10> threads; // array of threads for up to 10 tasks
4     int numThreads{0}; // current number of threads/tasks
5 public:
6     ...
7     // OOPS: enable default move semantics:
8     Tasks(Tasks&&) = default;
9     Tasks& operator=(Tasks&&) = default;
10    // at the end wait for all started threads:
11    ~Tasks() {
12        for (int i = 0; i < numThreads; ++i) {
13            threads[i].join();
14        }
15    }
16};
```

这种情况下会遇到麻烦，因为默认生成的移动操作可能会创建无效的任务。比如:

basics/tasksbug.cpp

```
1 #include "tasksbug.hpp"
2 #include <iostream>
3 #include <chrono>
4 #include <exception>
5 int main()
6 {
7     try {
8         Tasks ts;
9         ts.start([]{
10             std::this_thread::sleep_for(std::chrono::seconds{2});
11             std::cout << "\nt1 done" << std::endl;
12         });
13     }
```

```

13     ts.start([]{
14         std::cout << "\nt2 done" << std::endl;
15     });
16     // OOPS: move tasks:
17     Tasks other{std::move(ts)};
18 }
19 catch (const std::exception& e) {
20     std::cerr << "EXCEPTION: " << e.what() << std::endl;
21 }
22 }
```

开始时，通过将两个任务传递给 *Tasks* 对象 *ts* 来启动它们。因此，在 *ts* 中线程数组有两个元素，*numThreads* 为 2。但是，容器的移动操作会移动元素，因此在 *std::move()* 之后，*ts* 不再包含任何运行的线程对象。因此，*numThreads* 只是复制而已，也就是创建了一个无效的任务。析构函数最终会对前两个任务调用 *join()*，这会抛出一个异常（这将导致析构函数中出现致命错误）。

目前的问题是，需要两个成员一起才能定义一个有效的状态，而默认生成的移动语义会产生不一致的状态，因此会让析构函数失败。但不可能总是避免这个问题，因为对象销毁时，可能需要显式地对对象元素的子集做一些操作。这样，默认生成的移动操作可能不起作用，应该禁用或修复它们。

可能的使用办法是：

- 使用析构函数处理已移动状态（本例中，如果线程对象的 *joinable()* 为 *true*，就只能调用 *join()*）
- 自己实现移动操作
- 禁用移动语义（这是特殊移动成员函数的行为）

根据规则 0，应该将容易出错的资源管理封装在 *helper* 类型中，以便应用程序开发者实现 0 号特殊成员函数。这样，就可以使用一个助手类（模板），其既提供了成员 (*std::array* 和用于实际使用的元素数量的成员)，又提供了移动操作的正确实现。

整个问题也是 *std::thread* 类设计错误的副作用，该类型不遵循 RAII 原则。对于所有运行线程的 *std::thread*，必须在调用析构函数之前调用 *join()*（或 *detach()*）；否则，线程的析构函数将抛出异常。C++20 中，可以使用 *std::jthread* 类，如果对象为一个正在运行的线程，会自动调用 *join()*。

6.3 处理已破坏的不变量

已移动对象可能会破坏“有效但未定义的状态”，这比破坏可销毁要容易得多。我们可以将对象带入一种状态，从而破坏它们的不变量。

幸运的是，只有在显式请求移动语义时才会出现这个问题，因为临时对象无论如何都会立即销毁。然而，明确的移动请求不仅仅是用 *std::move()* 标记对象。可以用以下方法创建已移动对象：

- 对对象使用 *std::move()*
- 移动算法 (*std::move()* 和 *std::move_backward()*)
- 通过将“未删除”的元素移到前面来“删除”元素的算法（例如：*std::remove()*, *std::remove_if()*, *std::unique()*）
- 移动迭代器

如果类的不变量被（生成的）移动操作破坏，有以下解决方法：

- 修复移动操作，使已移动对象进入不变量稳定的状态。
- 禁用移动语义。
- 将定义已移动状态的不变量放宽为有效。特别是，成员函数和使用对象的函数必须以不同的方式实现，以处理新的可能状态。
- 提供一个成员函数来检查“不变量”的状态，这样类型的用户在用 `std::move()` 标记后就不会使用这种类型的对象了（或者只使用一组有限制的操作）。

来看一些不变量破坏的例子，并讨论如何修复它们。

6.3.1 破坏由移动值破坏的不变量

移动操作破坏不变量的第一个原因，与成员的已移动状态就是一个问题，因为该状态是有效的，但不应该发生。

考虑纸牌游戏中的 `Card` 类。假设每个对象都是一张有效的牌，比如红桃 8 或方块 K。还假设由于某种原因，该值是字符串，并且该类的不变式是每个对象都有一个表示有效卡片的状态。这意味着可能没有默认构造函数，而初始化构造函数会断言该值是有效的。例如：

```

1 class Card {
2 private:
3     std::string value; // rank + "–of–" + suit
4 public:
5     Card(const std::string& v)
6         : value{v} {
7         assertValidCard(value); // ensure the value is always valid
8     }
9     std::string getValue() const {
10         return value;
11     }
12 };

```

这个类中，生成的特殊移动成员函数创建了一个无效状态，它破坏了类的不变条件，即花色和数值（比如“红心皇后”）。

只要我们不使用 `std::move()` 或其他移动操作，这就不是问题（该类型的析构函数可以从字符串中移动），但是当调用 `std::move()` 时，就会遇到麻烦。给新值赋值没什么问题：

```

1 std::vector<Card> deck;
2 ... // initialize deck
3 Card c{std::move(deck[0])}; // deck[0] has invalid state
4 deck[0] = Card{"ace-of-hearts"}; // deck[0] is valid again

```

然而，打印已移动卡片的值可能会失败：

```

1 std::vector<Card> deck;
2 ... // initialize deck
3 Card c{std::move(deck[0])}; // deck[0] has invalid state
4 print(deck[0]); // passing an object with broken invariant

```

如果 `print` 函数的不变量没有破坏，我们会得到一个 core dump：

```

1 void print(const Card& c) {
2     std::string val{c.getValue()};
3     auto pos = val.find("-of-"); // find position of substring (no check)
4     std::cout << val.substr(0, pos) << ', '
5     << val.substr(pos+4) << '\n'; // OOPS: possible core dump
6 }

```

此代码可能在运行时失败，因为对于从移动的卡片，不再保证该值包含“-of-”。在这种情况下，`find()`用`std::string::npos`初始化`pos`，当`pos+4`用作`substr()`的第一个参数时，会抛出类型为`std::out_of_range`的异常。

完整示例请参见`basics/card.hpp`和`basics/card.cpp`。

修复方式如下：

- 禁用移动语义：

```

1 class Card {
2     ...
3     Card(const Card&) = default; // disable move semantics
4     Card& operator=(const Card&) = default; // disable move semantics
5 };
6

```

这让移动操作（例如，由`std::sort()`调用）的开销更大。

- 完全禁用复制和移动：

```

1 class Card {
2     ...
3     Card(const Card&) = delete; // disable copy/move semantics
4     Card& operator=(const Card&) = delete; // disable copy/move semantics
5 };
6

```

然而，不能再洗牌或对卡片进行排序。

- 修复损坏的特殊移动成员函数。

然而，什么是有有效的修复（总是分配一个“默认值”，如“梅花 A”）？如何确保具有默认值的对象，在不分配内存的情况下性能良好？

- 内部允许新状态，但不允许调用`getValue()`或其他成员函数。

可以对此进行记录（“对于已移动对象，只允许赋值。所有其他成员函数都以对象不处于已移动状态为前提”），甚至在成员函数内部检查这一点并触发断言或异常。

- 纸牌的方式可能没有新状态来扩展不变量。

这意味着必须实现移动特殊成员函数，必须确保对于一个已移动对象，成员值处于这种状态。通常，已移动状态等同于默认构造状态。因此，这也是提供默认构造函数的机会。理想情况下，还可以提供成员函数来检查这个状态。

这个变化中，这个类的用户必须考虑到字符串的值可能是空，并相应地更新代码。例如：

```

1 void print(const Card& c) {
2     std::string val{c.getValue()};

```

```

3     auto pos = val.find("-of-"); // find position of substring
4     If (pos != std::string::npos) { // check whether it exists
5         std::cout << val.substr(0, pos) << ','
6         << val.substr(pos+4) << '\n';
7     }
8     else {
9         std::cout << "no value\n";
10    }
11}
12

```

另一方面，`getValue()` 可能返回一个 `std::optional`(自 C++17 支持)。似乎没有明显的完美解决方案。必须考虑这些修复对程序更大的不变量意味着什么。比如只有一张牌，或者所有的牌都有效等)，然后决定用哪一张。

这个类在 C++11 之前运行良好，C++11 之前的标准不支持移动语义(这可能意味着第一个选项是最好的)。因此，C++11 可能会为实现类时不可能的类引入状态。这是一种罕见的情况，但移动语义的引入，确实会破坏现有的代码。

请参阅 `Email` 类的另一个例子，在这个类中，内部标记了已移动状态，以便单独处理，并通过“移除”算法使元素处于已移动状态后，使该状态可见。

6.3.2 移动的一致性会破坏成员的不变量

移动操作破坏不变量的第二个原因，与两个成员必须一致但可能被移动破坏的对象有关。正如在线程数组的例子中所看到的，这可能导致破坏析构函数的不一致。通常，析构函数没问题，但自移状态破坏了一个不变量。在类中，有两个不同的值表示形式，一个整数和一个字符串：

`basics/intstring.hpp`

```

1 #include <iostream>
2 #include <string>
3 class IntString
4 {
5     private:
6     int val; // value
7     std::string sval; // cached string representation of the value
8     public:
9     IntString(int i = 0)
10    : val{i}, sval{std::to_string(i)} {
11    }
12     void setValue(int i) {
13        val = i;
14        sval = std::to_string(i);
15    }
16    ...
17     void dump() const {
18        std::cout << "[" << val << "/" << sval << "] \n";
19    }
20};

```

这个类中，确保成员 `val` 和成员 `sval` 只是相同值的两种不同表示。这意味着在该类的实现和使用中，通常期望其状态的 int 和 string 表示一致。如果在这里使用移动操作，将保留值 `val`，但是 `sval` 不再保证可用 `val` 的字符串表示。

请看下面程序代码：

`basics/intstring.cpp`

```
1 #include "intstring.hpp"
2 #include <iostream>
3
4 int main()
5 {
6     IntString is1{42};
7     IntString is2;
8     std::cout << "is1 and is2 before move:\n";
9     is1.dump();
10    is2.dump();
11
12    is2 = std::move(is1);
13
14    std::cout << "is1 and is2 after move:\n";
15    is1.dump();
16    is2.dump();
17 }
```

程序有以下输出 (已移动字符串可能会变为空)：

```
is1 and is2 before move:
```

```
[42/'42']
```

```
[0/'0']
```

```
is1 and is2 after move:
```

```
[42/]
```

```
[42/'42']
```

也就是说，自动生成的移动操作打破了两个成员匹配的不变量。

这个问题有多大？这至少是个陷阱。同样，您可能会认为在移动之后不应该再使用该值（直到再次设置）。然而，开发者希望使用针对 C++ 标准库对象的策略，该策略声明对象处于有效但未定义的状态。

事实是，有了相应的 getter，类不再保证 int 和 string 的匹配，这可能是类的不变量（隐式或显式声明）。您可能认为最坏的结果是值（现在是未指定的）看起来不同，这取决于如何使用，但使用它没有问题，因为它仍然是有效的 int 或字符串。

然而，依赖于此不变量的代码可能会破坏。该代码可能假设字符串表示至少有一个数字。例如，如果搜索第一个或最后一个数字，肯定会找到。已移动的字符串（通常是空的）不再是这样了。因此，代码不重复检查字符串值中是否有任何字符，可能会遇到未定义行为。

同样，如何处理这个问题取决于类的设计者。但是，如果遵循 C++ 标准库的规则，应该使已

移动对象处于有效状态，这可能要表示为“没有任何值”的状态。

通常，当一个对象的状态具有以某种方式相互依赖时，必须显式地确保已移动状态属于有效状态。但有些情况会出现例外：

- 对相同值有不同的表示，但是其中一些已移走。
- 像 *counter* 这样的成员对应于成员中的元素数量。
- 用布尔值声明字符串的值已验证，但是该验证值已移走。
- 所有元素的平均值的缓存值仍然存在，但是值（在容器成员中）已移走。

注意，这个类在 C++11 之前没问题，因为不支持移动语义。当切换到 C++11 或更高版本，并使用已移动对象时，不变量就破坏了。

6.3.3 移动的类指针成员破坏不变量

移动操作破坏不变量的第三个原因，与具有类似指针语义（如（智能）指针）成员的对象有关。

考虑以下类的例子，其中对象使用 `std::shared_ptr<>` 共享整型值：

```
1 class SharedInt {
2 private:
3     std::shared_ptr<int> sp;
4 public:
5     explicit SharedInt(int val)
6         : sp{std::make_shared<int>(val)} {
7     }
8     std::string asString() const {
9         return std::to_string(*sp); // OOPS: assume there is always an int value
10    }
11};
```

这个类的对象与副本共享的初始整数值。只要新对象是复制的，一切都好：

```
1 SharedInt si1{42};
2 SharedInt si2{si1}; // si1 and si2 share the value 42
3
4 std::cout << si1.asString() << '\n'; // OK
```

由于只是复制，*SharedInt* 成员 *sp* 总是为它的值分配内存（从 `std::make_shared<>()` 或从已分配内存的现有共享指针复制）。

然而，当使用移动语义时，如果使用已移动对象，就会遇到未定义行为：

```
1 SharedInt si1{42};
2 SharedInt si3{std::move(si1)}; // OOPS: moves away the allocated memory in si1
3
4 std::cout << si1.asString() << '\n'; // undefined behavior (probably core dump)
```

问题是在类内部，没有正确处理值可能已移走，因为默认生成的移动操作调用了共享指针的移动操作，该操作将所有权从原始对象移走。这意味着从 *SharedInt* 的状态移动到成员 *sp* 不再拥有对象的情况，其成员函数 *asString()* 无法正确处理。

您可能会争辩说，为具有已移动状态的对象调用 `asString()` 没有意义，因为使用的是未定义的值，但至少标准库保证已移动类型的对象处于有效状态，因此可以调用没有限制的所有操作。如果在用户定义的类型中不提供相同的保证，该类型的用户可能会感到非常惊讶。

从鲁棒编程（避免意外、陷阱和未定义行为）的角度来看，我建议您遵循 C++ 标准库的规则。也就是说：移动操作不应该将对象带入破坏不变量的状态。

本例中，必须执行以下操作之一：

- 通过正确处理所有可能的已移动状态来修复类的所有错误操作
- 禁用移动语义，在复制对象时没有优化
- 显式实现移动操作
- 调整并记录类或特定操作的不变量（约束/先决条件）（例如“为已移动对象调用 `asString()` 是未定义的行为”）。

分配内存非常昂贵，最好的解决办法是正确处理所有权移走的情况。这将使用默认构造函数创建对象应具有的状态，这样就可以引入这个状态了。

下面的小节将演示这一点和其他修缮的代码。

修复损坏的成员函数

第一个选项，修复所有损坏的操作，扩展类的不变量（所有对象的可能状态），以便所有操作都可以处理已移动状态。我们仍然需要做出设计决策，例如：为已移动对象（或更一般地说，共享指针不拥有整型值的对象）调用 `asString()` 时，可以这样：

- 返回备选值：

```
1 class SharedInt {  
2     ...  
3     std::string asString() const {  
4         return sp ? std::to_string(*sp) : "";  
5     }  
6     ...  
7 };  
8
```

- 抛出异常：

```
1 class SharedInt {  
2     ...  
3     std::string asString() const {  
4         if (!sp) throw ...;  
5         return std::to_string(*sp);  
6     }  
7     ...  
8 };  
9
```

- 在调试模式下强制出现运行时错误：

```
1 class SharedInt {  
2     ...  
3 }
```

```

1      ...
2      std::string asString() const {
3          assert(sp);
4          return std::to_string(*sp);
5      }
6      ...
7  };
8
9

```

禁用移动语义

第二个选项是禁用移动语义，只使用复制语义。我们前面描述了如何禁用移动语义，必须声明另一个特殊成员函数。通常，对复制特殊成员函数使用 `=default`:

```

1 class SharedInt {
2     ...
3     SharedInt(const SharedInt&) = default; // disable move semantics
4     SharedInt& operator=(const SharedInt&) = default; // disable move semantics
5     ...
6 };

```

实现移动语义

第三种选择是实现移动操作，这样就不会破坏类的不变量。

为此，必须确定已移动对象的状态应该是什么。为了支持 `asString()` 可以调用 `operator*`(解引用) 而不检查值是否存在，所以必须提供一个值。例如，可以有一个静态的已移动的值，把它赋给那些已移动的对象:

[basics/sharedint.hpp](#)

```

1 #include <memory>
2 #include <string>
3 class SharedInt {
4     private:
5         std::shared_ptr<int> sp;
6         // special "value" for moved-from objects:
7         inline static std::shared_ptr<int> movedFromValue{std::make_shared<int>(0)};
8     public:
9         explicit SharedInt(int val)
10            : sp{std::make_shared<int>(val)} {
11        }
12         std::string asString() const {
13             return std::to_string(*sp); // OOPS: unconditional deref
14         }
15         // fix moving special member functions:
16         SharedInt(SharedInt&& si)
17            : sp{std::move(si.sp)} {
18             si.sp = movedFromValue;
19         }

```

```

20 SharedInt& operator= (SharedInt&& si) noexcept {
21     if (this != &si) {
22         sp = std::move(si.sp);
23         si.sp = movedFromValue;
24     }
25     return *this;
26 }
27 // enable copying (deleted with user-declared move operations):
28 SharedInt (const SharedInt&) = default;
29 SharedInt& operator= (const SharedInt&) = default;
30 };

```

注意，我们必须对复制特殊成员的函数使用 `=default`，因为当我们有声明特殊成员移动函数时，这些函数的默认版本会删除。

6.4 总结

- 每个类需要说明对象的移动状态，必须确保是可销毁的（通常情况下没有实现特殊成员函数）。然而，类的用户可能期望/要求更多。
- C++ 标准库函数的要求也适用于已移动对象。
- 生成的特殊移动成员函数可能会将已移动对象带入某种状态，从而破坏类的不变量。这种情况可能会发生，特别是当：
 - 类没有具有确定值的默认构造函数（因此没有自然的移出状态）
 - 成员的值有限制（比如断言）
 - 成员的值相互依赖
 - 使用了具有类似指针语义的成员（指针、智能指针等）。
- 如果已移动状态中断不变量或使操作无效，应该使用以下选项来修复这个问题：
 - 禁用移动语义
 - 修缮了移动语义的实现
 - 类内部处理已破坏的不变量，并将对外隐藏
 - 通过记录已移动对象的约束和前置条件来消除对类不变量的影响

7 移动语义和 noexcept

移动语义在 C++11 中非常完整时，我们发现了一个问题：vector 重新分配时，不能使用移动语义。因此，引入了新的关键字 noexcept。

本章就来解释这个问题，以及使用 noexcept 意味着什么。

7.1 移动构造函数有和没有 noexcept 的区别

通过使用 noexcept 关键字的示例来开始这个主题。

7.1.1 无 noexcept 的移动构造函数

下面的类，引入了带有 string 成员的类型，并实现了复制和移动构造函数：

basics/person.hpp

```
1 #include <string>
2 #include <iostream>
3
4 class Person {
5     private:
6         std::string name;
7     public:
8         Person(const char* n)
9             : name{n} {
10     }
11
12     std::string getName() const {
13         return name;
14     }
15
16 // print out when we copy or move:
17     Person(const Person& p)
18         : name{p.name} {
19         std::cout << "COPY " << name << '\n';
20     }
21     Person(Person&& p)
22         : name{std::move(p.name)} {
23         std::cout << "MOVE " << name << '\n';
24     }
25     ...
26 };
```

现在创建并初始化一个 Person 的 vector，并插入一个 Person 对象：

basics/person.cpp

```
1 #include "person.hpp"
2 #include <iostream>
3 #include <vector>
4
5 int main()
```

```

6 {
7     std::vector<Person> coll{ "Wolfgang Amadeus Mozart",
8         "Johann Sebastian Bach",
9         "Ludwig van Beethoven"};
10    std::cout << "capacity: " << coll.capacity() << '\n';
11    coll.push_back("Pjotr Iljitsch Tschaikowski");
12 }

```

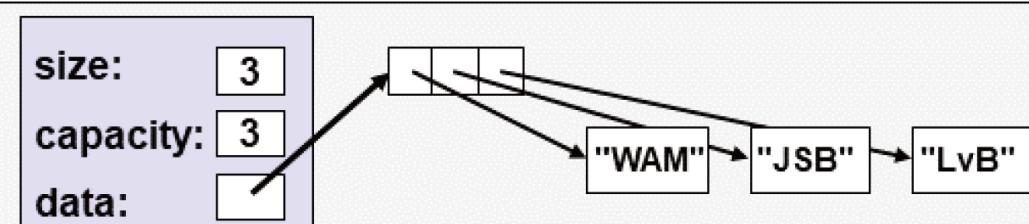
程序输出如下:

```

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

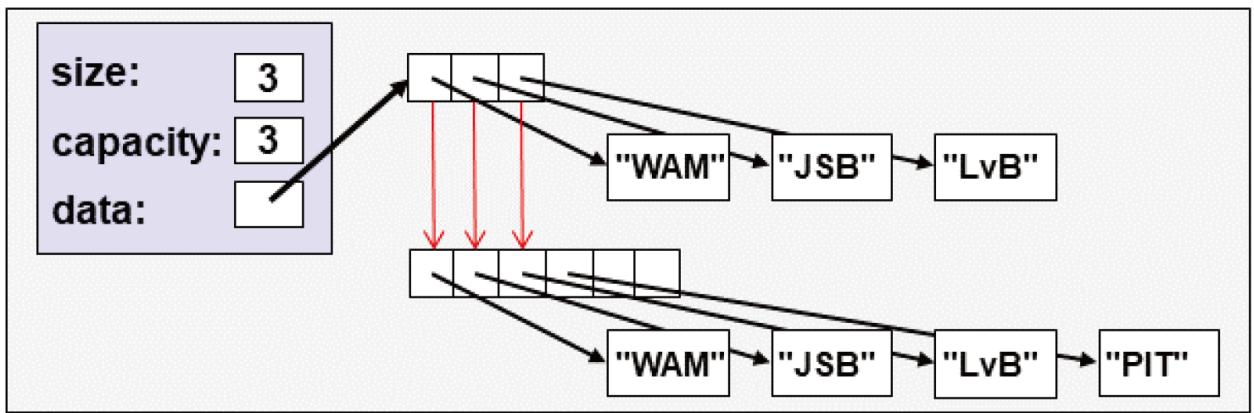
```

首先，将初始值复制到 vector 中 (容器的 `std::initializer_list<>` 构造函数按值接受传递的参数)。因此，vector 通常为三个元素分配内存:

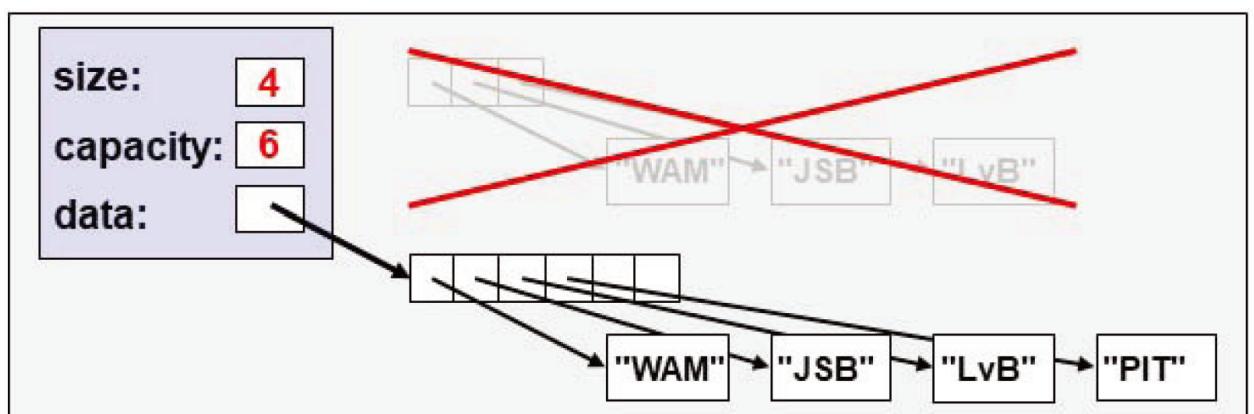


接下来会发生什么: 用 `push_back()` 插入第四个元素，结果如下:

- vector 在内部需要更多内存，所以会分配新的内存 (比如，6 个元素)，移动第 4 个字符串 (创建一个临时的 `Person` 并使用 `push_back()` 方法将其移动到 vector 中)，但也会将现有的元素复制到新内存中:



- 这个操作结束时，vector 销毁旧的元素，释放这些元素的旧内存，并更新成员：



问题是，为什么 vector 不使用移动构造函数，将元素从旧内存移动到新内存呢？

强异常安全保证

vector 的重新分配不使用移动语义的原因是，`push_back()` 提供了强异常处理保证：在 vector 的重新分配过程中抛出异常时，C++ 标准库保证将 vector 回退到之前的状态。也就是说，`push_back()` 提供了一种保证：要么成功，要么无效。

C++ 标准能够在 C++98 和 C++03 中提供这种保证，因为 C++ 只能复制元素。如果在复制元素时出现错误，源对象仍然可用。处理异常的内部代码只需销毁创建的副本，并释放新的内存，使 vector 返回到之前的状态（C++ 标准库要求析构函数不抛出异常；否则，无法回退）。

重新分配是使用移动语义的最优位置，因为只是元素从一个位置移动到另一个位置。因此，C++11 希望在这里使用移动语义。但是，如果在重新分配期间抛出异常，就无法回退了。新内存中的元素已经窃取了旧内存中元素的值。因此，只是销毁新的元素还不够，还得把他们移回去。但怎么知道把移回去的时候就不会失败呢？

你可能会说一个移动构造函数永远不抛出异常。这可能是正确的字符串（因为我们只是移动值和指针），而是需要对象在一个有效的状态，这种状态可能需要额外的内存，所以当额外内存有问题时会丢弃状态信息（例如，基于节点的容器 Visual C++ 实现方式）。

我们也不能放弃，因为程序可能已经使用这个特性来避免创建向量的备份，而丢失数据可能是（安全）关键的。不再支持 `push_back()` 对于使用 C++11 来说将是个噩梦。

最后，只有当元素类型的移动构造函数保证不抛出异常时，才在重新分配时使用移动语义。

7.1.2 使用 noexcept 的移动构造函数

因此，保证 *Person* 类的移动构造函数永远不会抛出异常时，示例程序就改变了行为：

basics/personmove.hpp

```
1 #include <string>
2 #include <iostream>
3
4 class Person {
5     private:
6         std::string name;
7     public:
8         Person(const char* n)
9             : name{n} {}
10
11         std::string getName() const {
12             return name;
13         }
14
15 // print out when we copy or move:
16         Person(const Person& p)
17             : name{p.name} {
18                 std::cout << "COPY " << name << '\n';
19             }
20         Person(Person&& p) noexcept // guarantee not to throw
21             : name{std::move(p.name)} {
22                 std::cout << "MOVE " << name << '\n';
23             }
24         ...
25     };
```

如果像以前一样使用 *Persons*:

basics/personmove.cpp

```
1 #include "personmove.hpp"
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<Person> coll{"Wolfgang Amadeus Mozart",
8         "Johann Sebastian Bach",
9         "Ludwig van Beethoven"};
10    std::cout << "capacity: " << coll.capacity() << '\n';
11    coll.push_back("Pjotr Iljitsch Tschaikowski");
12 }
```

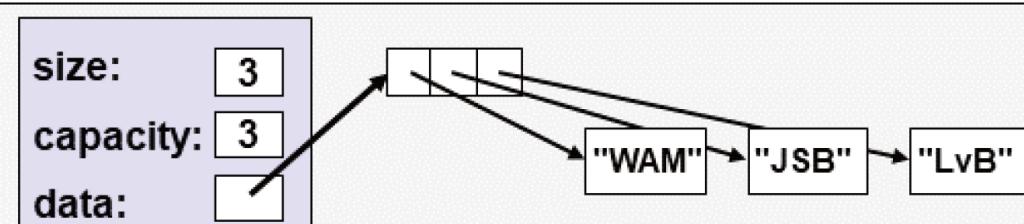
可得到以下输出：

```

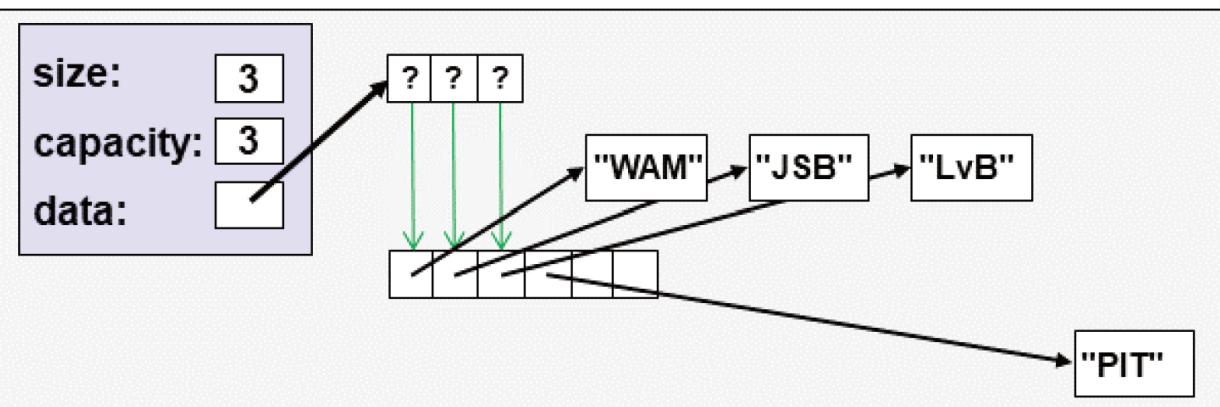
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
MOVE Pjotr Iljitsch Tschaikowski
MOVE Wolfgang Amadeus Mozart
MOVE Johann Sebastian Bach
MOVE Ludwig van Beethoven

```

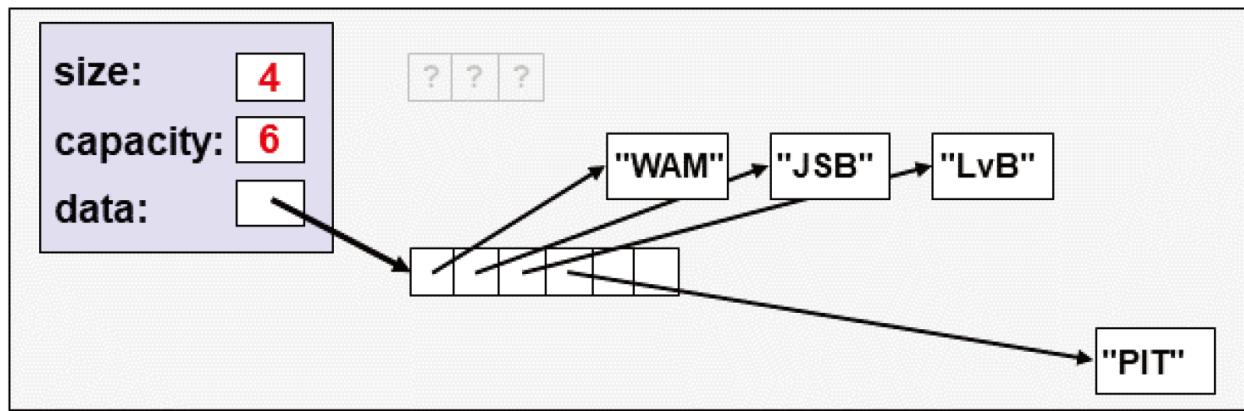
同样，首先将初始值复制到 vector 中



然而，随着输出的最后三行可视化，vector 现在使用移动构造函数将元素移动到新的重新分配的内存中：



最后，vector 只需要释放旧内存并更新其成员：



具有条件的 noexcept 声明

然而，是否可以将移动构造函数标记为 noexcept? 移动 `name`(一个 `std::string`)，并向标准输出流写入一些内容。如果在那里抛出异常，就违反了无异常的保证。这种情况下，程序在运行时将调用 `std::terminate()`，后者通常调用 `std::abort()` 来发出程序异常结束的信号(通常会出现 core dump)。

因此，如果 `string` 成员和输出操作不抛出异常，则应该保证不抛出异常。引入 noexcept 关键字是用来指定不抛出的条件保证。[\(basics/personcond.hpp是一个完整的例子\)](#):

```

1 class Person {
2     private:
3         std :: string name;
4     public:
5         ...
6         Person (Person&& p)
7             noexcept (std :: is_nothrow_move_constructible_v<std :: string>
8             && noexcept (std :: cout << name))
9             : name{std :: move(p.name)} {
10                 std :: cout << "MOVE " << name << '\n';
11             }
12         ...

```

使用 `noexcept(...)`，保证括号内的编译时表达式为真时不会抛出异常。这种情况下，需要两样东西来保证：

- 对于 `std::is_nothrow_move_constructible_v<std::string>`(在 C++20 之前，必须使用 `std::is_nothrow_m` 使用标准类型特征 (type) 来告诉 `std::string` 的移动构造函数是否保证不抛出异常。
- 对于 `noexcept(std::cout << name)`，对名称的输出表达式是否保证不抛出异常。这里，使用 `noexcept` 作为操作符，说明是否所有执行传递的表达式都保证不会抛出异常。

通过此声明，重新分配将再次使用复制构造函数。字符串的移动构造函数不保证抛出异常，但输出操作符不保证抛出异常。然而，移动构造函数通常不输出任何内容。因此，通常当成员不抛出异常时，可以为整个移动构造函数提供不抛出异常的保证。

好消息是，如果没有自己实现移动构造函数，编译器将为你提供 noexcept 保证。对于所有成员都保证不抛出移动构造函数的类，生成的或默认的移动构造函数将作为整体提供保证。

考虑下面的声明：

basics/persondefault.hpp

```
1 #include <string>
2 #include <iostream>
3
4 class Person {
5     private:
6         std::string name;
7     public:
8         Person(const char* n)
9             : name{n} {
10
11
12         std::string getName() const {
13             return name;
14
15
16         // print out when we copy:
17         Person(const Person& p)
18             : name{p.name} {
19                 std::cout << "COPY " << name << '\n';
20             }
21         // force default generated move constructor:
22         Person(Person&& p) = default;
23
24     ...  
};
```

这种情况下，应该生成默认的移动构造函数：

```
1 class Person {
2     ...
3     // force default generated move constructor:
4     Person(Person&& p) = default;
5     ...
6 };
```

这意味着只有在复制时才输出。当使用生成的移动构造函数时，只是没有执行复制。

现在用常用的程序在末尾打印一些 *Persons*:

basics/persondefault.cpp

```
1 #include "persondefault.hpp"
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<Person> coll{"Wolfgang Amadeus Mozart",
8         "Johann Sebastian Bach",
9         "Ludwig van Beethoven"};
10    std::cout << "capacity: " << coll.capacity() << '\n';
```

```
11 coll.push_back("Pjotr Iljitsch Tschaikowski");
12
13 std::cout << "name of coll[0]: " << coll[0].getName() << '\n';
14 }
```

会得到以下输出:

```
COPY Wolfgang Amadeus Mozart
COPY Johann Sebastian Bach
COPY Ludwig van Beethoven
capacity: 3
name of coll[0]: Wolfgang Amadeus Mozart
```

只看到初始化器列表中元素的副本。对于其他任何的操作，包括重新分配，都使用默认的移动构造函数。在重新分配的内存中，第一个人名是正确的。

如果不指定任何成员函数，则具有相同的行为:

- 如果实现了移动构造函数，应该声明它是否，以及何时保证不会抛出异常。
- 如果不需要实现移动构造函数，根本不需要指定任何东西。

如果类的性能或该类的重新分配对象很重要，开发者可能还想在编译时再次检查类的移动构造函数，是否保证不会抛出异常:

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

或升级到 C++17:

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

7.1.3 值得使用 noexcept 吗？

您可能想知道用或多或少的 noexcept 表达式声明移动构造函数是否值得使用。Howard Hinnant 用一个简单的程序演示了这种效果（本书中进行了简单的改编）：

[basics/movenoexcept.cpp](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <chrono>
5
6 // string wrapper with move constructor:
7 struct Str
8 {
9     std::string val;
10    // ensure each string has 100 characters:
11    Str()
12        : val(100, 'a') { // don't use braces here
13    }
```

```

14
15 // enable copying:
16 Str(const Str&) = default;
17
18 // enable moving (with and without noexcept):
19 Str (Str&& s) NOEXCEPT
20 : val{std::move(s.val)} {
21 }
22 };
23
24 int main()
25 {
26 // create vector of 1 Million wrapped strings:
27 std::vector<Str> coll;
28 coll.resize(1000000);
29
30 // measure time to reallocate memory for all elements:
31 auto t0 = std::chrono::steady_clock::now();
32 coll.reserve(coll.capacity() + 1);
33 auto t1 = std::chrono::steady_clock::now();
34
35 std::chrono::duration<double, std::milli> d{t1 - t0};
36 std::cout << d.count() << "ms\n";
37 }
```

提供了一个类来包装长度有效的字符串 (以避免小字符串优化)。注意，必须用圆括号初始化，因为大括号会将 100 解释为值为 100 的初始字符。

类中用 NOEXCEPT 标记移动构造函数，预处理程序可以用空或 noexcept (例如，用-DNOEXCEPT=NOEXCEPT 编译) 进行替换。然后，再看看重新分配 vector 中的 100 万个对象需要多长时间。

几乎所有平台上，声明 noexcept 的移动构造函数会使重新分配速度提高 10 倍 (确保使用了相应的编译优化级别)。也就是说，重新分配 (通常通过插入新元素强制执行) 可能需要 20 毫秒，而不是 200 毫秒。这意味着减少了 180 毫秒，不能使用任何其他向量，这对于性能来说是巨大的收益。

7.2 noexcept 声明的细节

正如所见，noexcept 的引入是为了允许条件保证不抛出异常。通常，在编译时知道函数不能抛出异常可以改进代码和优化，因为不必处理由于抛出异常而进行的清理。注意，如果违反了 noexcept 保证，程序会调用 std::terminate()，而后者通常会调用 std::abort() 来导致“异常的程序终止”(例如，core dump)。

7.2.1 声明 noexcept 的函数的规则

当声明 noexcept 条件时，有几个适用的规则：

- noexcept 条件必须是编译时表达式，该表达式的值可转换为 bool 类型。
- 不能重载不同条件的函数。

- 在类层次结构中，noexcept 条件是接口的一部分。用不是 noexcept 的函数覆盖不是 noexcept 的基类函数是错误的。

例如：

```

1 class Base {
2     public:
3     ...
4     virtual void foo(int) noexcept;
5     virtual void foo(int); // ERROR: overload on different noexcept clause only
6     virtual void bar(int);
7 };
8
9 class Derived : public Base {
10    public:
11    ...
12    virtual void foo(int) override; // ERROR: override giving up the noexcept
13    guarantee
14    virtual void bar(int) noexcept; // OK (here we also guarantee not to throw)
15 };

```

但是，对于非虚函数，派生类成员可以使用不同的 noexcept 声明隐藏基类成员：

```

1 class Base {
2     public:
3     ...
4     void foo(int) noexcept;
5 };
6
7 class Derived : public Base {
8     public:
9     ...
10    void foo(int); // OK, hiding instead of overriding
11 };

```

条件在编译时计算后遵循相同的规则。例如，考虑以下类的层次结构：

```

1 class Base {
2     public:
3     virtual void func() noexcept(sizeof(int) < 8); // might throw if sizeof(int) >= 8
4 };
5
6 class Derived : public Base {
7     public:
8     void func() noexcept(sizeof(int) < 4) override; // might throw if sizeof(int) >= 4
9 };

```

这种情况下，如果 int 的大小为 4，将出现编译时错误，因为基类保证在调用 `func()` 时不抛出，而派生类不再为 `func()` 提供这种保证。当 int 的大小小于 4 时，两者都是 noexcept，这挺不错。当 int 的大小至少为 8 时，两者都不是 noexcept，这也没问题。因此，派生类只需要进一步限制异常保证。

7.2.2 noexcept 的特殊成员函数

不能为特殊成员函数自动生成条件。

noexcept 的复制和移动特殊成员函数

按照规则，当生成但未实现特殊成员函数时，将生成 noexcept 条件。这种情况下，如果对所有基类和非静态成员使用相应操作应该保证不抛出异常。

例如：

basics/specialnoexcept.cpp

```
1 #include <iostream>
2 #include <type_traits>
3
4 class B
5 {
6     std::string s;
7 };
8
9 int main()
10{
11    std::cout << std::boolalpha;
12    std::cout << std::is_nothrow_default_constructible<B>::value << '\n';
13    std::cout << std::is_nothrow_copy_constructible<B>::value << '\n';
14    std::cout << std::is_nothrow_move_constructible<B>::value << '\n';
15    std::cout << std::is_nothrow_copy_assignable<B>::value << '\n';
16    std::cout << std::is_nothrow_move_assignable<B>::value << '\n';
17}
```

程序输出如下：

```
true
false
true
false
true
```

生成的复制构造函数和复制赋值操作符可能会抛出异常，因为复制 std::string 可能会抛出异常。但生成的默认构造函数、移动构造函数和移动赋值操作符保证不会抛出异常，因为类 std::string 的默认构造函数、移动构造函数和移动赋值操作符保证不会抛出异常。

请注意，noexcept 条件甚至在这些特殊成员函数使用 =default 进行声明时也会生成。因此，如果像下面这样声明 B 类，效果是一样的：

```
1 class B
2 {
3     std::string s;
4 public:
```

```
5 B(const B&) = default; // noexcept condition automatically generated
6 B(B&&) = default; // noexcept condition automatically generated
7 B& operator=(const B&) = default; // noexcept condition automatically generated
8 B& operator=(B&&) = default; // noexcept condition automatically generated
9 };
```

当有一个默认的特殊成员函数时，可以显式地指定不同于生成的 noexcept 保证。例如：

```
1 class C
2 {
3     ...
4 public:
5     C(const C&) noexcept = default; // guarantees not to throw (OK since C++20)
6     C(C&&) noexcept(false) = default; // specifies that it might throw (OK since C
7         ++20)
8     ...
9 };
```

C++20 之前，如果生成的 noexcept 条件与指定的 noexcept 条件相矛盾，则删除已定义的函数。

noexcept 的析构函数

按照规则，析构函数总是保证在默认情况下不会抛出异常。这既适用于生成的析构函数，也适用于实现的析构函数。

例如：

```
1 class B
2 {
3     std::string s;
4 public:
5     ...
6     ~B() { // automatically always declared as ~B() noexcept
7         ...
8     }
9 };
```

对于 noexcept(false)，可以在没有这个保证的情况下进行声明，但这通常没有意义，因为 C++ 标准库保证析构函数从不抛出异常。

7.3 noexcept 在类中的声明

可以看到，特别是必须实现一个移动构造函数时，应该用 noexcept 保证来声明。通常，遵循 C++ 标准的规则，当所有基类和所有成员类型在移动赋值时都没有抛出异常时，应该声明为 noexcept。

通常的模式如下：

```
1 class Base {
2     ...
```

```

3 } ;
4
5 class Drv : public Base {
6     MemType member ;
7     ...
8     // move constructor:
9     Drv(Drv&&) noexcept( std::is_nothrow_move_constructible_v<Base> &&
10        std::is_nothrow_move_constructible_v<MemType> );
11 }

```

这里, *Drv* 类的移动构造函数保证, 如果基类 *Base* 和成员类型 *MemType* 提供了这个保证, 则不会抛出异常。

移动赋值操作符可能使用相同的模式。但请注意, 多态类型中应该删除移动赋值操作符, 所以通常不需要在派生类中实现它们。

7.3.1 检查抽象基类中的移动构造函数

请注意, 类型特征 `std::is_nothrow_move_constructible<>` 并不会总如预期。对于抽象基类, 总是产生 *false*, 因为还会检查是否可以用移动构造函数创建该类型的对象, 而这对于抽象类型是不可能的。

因此, “如果抽象基类保证不抛出异常, 则继承类也应该保证不抛出异常”的声明不能使用标准类型特征来表述。通常, 只需(必须)知道基类的移动构造函数是否可能抛出。

为了能够检查一个类的移动构造函数是否保证不会抛出, 可以实现以下 helper 类型特征(使用 C++20 实现)。但请注意, 必须为每个纯虚函数提供实现:

[poly/isnothrowmovable.hpp](#)

```

1 // type trait to check whether a base class guarantees not to throw
2 // in the move constructor (even if the constructor is not callable)
3 #ifndef IS_NOTHROW_MOVABLE_HPP
4 #define IS_NOTHROW_MOVABLE_HPP
5
6 #include <type_traits>
7
8 template<typename Base>
9 struct Wrapper : Base {
10     using Base::Base;
11     // implement all possibly wrapped pure virtual functions:
12     void print() const {}
13     ...
14 };
15
16 template<typename T>
17 static constexpr inline bool is_nothrow_movable_v
18 = std::is_nothrow_move_constructible_v<Wrapper<T>>;
19
20 #endif // IS_NOTHROW_MOVABLE_HPP

```

现在甚至可以检查抽象基类的移动构造函数是否为 noexcept。下面的程序演示了标准和用户定义的类型特征的不同行为：

poly/isnothrowmovable.cpp

```
1 #include "isnothrowmovable.hpp"
2 #include <iostream>
3
4 class Base {
5     std::string id;
6     ...
7 public:
8     virtual void print() const = 0; // pure virtual function (forces abstract base
9         class)
10    ...
11 protected:
12     // protected copy and move semantics (also forces abstract base class):
13     Base(const Base&) = default;
14     Base(Base&&) = default;
15     // disable assignment operator (due to the problem of slicing):
16     Base& operator=(Base&&) = delete;
17     Base& operator=(const Base&) = delete;
18 };
19
20 int main()
21 {
22     std::cout << std::boolalpha;
23     std::cout << "std::is_nothrow_move_constructible_v<Base>: "
24     << std::is_nothrow_move_constructible_v<Base> << '\n';
25     std::cout << "is_nothrow_movable_v<Base>: "
26     << is_nothrow_movable_v<Base> << '\n';
27 }
```

该程序有以下输出：

```
std::is_nothrow_move_constructible_v<Base>: false
is_nothrow_movable_v<Base>: true
```

因此，如果必须在抽象基类派生的类中实现移动构造函数，可以使用 helper 类型特征来声明移动构造函数。如下所示：

```
1 class Drv : public Base {
2     MemType member;
3     ...
4     // move constructor:
5     Drv(Drv&&) noexcept(is_nothrow_movable_v<Base> &&
6     is_nothrow_movable_v<MemType>);
7 };
```

由于不需要实现所有纯虚函数，C++ 标准中会缺少编译器支持的类型特征。

7.4 何时何地使用 noexcept

noexcept 在 C++11 中引入，是为了解决在 vector 中重新分配元素时不能使用移动语义的问题。

原则上，可以用 noexcept(条件式) 标记每个函数。这不仅有助于像 vector 重新分配这样的情况（注意，移动构造函数可能会调用其他需要 noexcept 保证的函数），还有助于编译器优化代码，因为不需要生成代码来处理异常。所以问题是，应该在哪使用 noexcept？

对于 C++11 标准库，会急于确定将 noexcept 放在哪里（重新分配问题很晚才发现，noexcept 的最终语义在发布 C++11 的那一周才解释清楚）。因此，遵循 <http://wg21.link/n3279> 中提出方案是一种相当保守的方法，但对代码编写是一个有用的指导方针。大致来说，指导方针为：

- 每个库的工作组同意不能抛出异常，且具有“广泛契约”的库函数（即，由于前提条件而没有指定未定义的行为）都应该标记为“无条件的 noexcept”。
- 如果库中的交换函数、移动构造函数或移动赋值操作符是“条件宽松”的（即，可以通过应用 noexcept 操作符保证不抛出异常），则应将其标记为有条件的 noexcept。任何其他函数都不应该使用条件 noexcept。
- 任何标准库析构函数都不应该抛出异常，应该使用隐式提供的（非抛出的）异常规范。
- 为兼容 C 代码而设计的库函数，可以标记为无条件的 noexcept。

下面的例子阐明了第一项的意思：

- 对于容器和字符串，成员函数 empty() 和 clear() 标记为 noexcept，因为没有办法来实现它们并抛出异常。
- 即使在正确使用时保证不会抛出异常，vector 和 string 的索引操作符不会标记为 noexcept。然而，当传递无效索引时，会出现有未定义的行为。不使用 noexcept 来声明时，这些实现会在这种情况下抛出异常。

第二项是在实现移动语义时应该遵循的指导原则。建议只在实现移动构造函数、移动赋值操作符或 swap() 函数时，使用条件 noexcept。对于所有其他函数，考虑态详细的条件，可能会透露太多的实现细节。

对于 <http://wg21.link/p0884>，包装类型也应该有条件 noexcept 声明：

- 如果标准库类型具有包装语义，以提供与底层类型相同的行为，则默认构造函数、复制构造函数和复制赋值操作符应有条件地标记为 noexcept，而底层异常规范仍然有效。

最后，请注意，根据规则，任何析构函数都总是隐式声明为 noexcept。

代码的应该这么写：

- 应该实现移动构造函数、移动赋值操作符和一个带有 noexcept(条件) 的 swap() 函数。
- 对于所有其他函数，在知道不能抛出异常时，就使用无条件的 noexcept。
- 析构函数不需要 noexcept 规范。

7.5 总结

- 使用 noexcept，可以 (有条件的) 保证不抛出异常。
- 如果实现了移动构造函数、移动赋值操作符或 swap()，可以用 (有条件的)noexcept 表达式进行声明。
- 对于其他函数，如果从不抛出异常，可以用无条件的 noexcept 标记。
- 析构函数总是使用 noexcept 来声明 (即使是在实现的时候)。

8 值的种类

本章会介绍移动语义的形式术语和规则。并且会正式的介绍值的类别，如 lvalue、rvalue、prvalue 和 xvalue，并讨论了在绑定对象引用时的作用。也会讨论移动语义不会自动传递的细节，以及 decltype 在表达式调用时的微妙行为。

这一章是这本书中最复杂的一章。可能会看到一些事实和特征，可能很难相信或理解。当您再次阅读值类别、对象的绑定引用和 decltype 时，可以再回到这里看看。

8.1 值的种类

要编译表达式或语句，所涉及的类型是否合适并不重要。例如，如果在赋值符的左边使用了 int 型字面值，则不能将 int 型赋值给 int 型字面值：

```
1 int i = 42;
2
3 i = 77; // OK
4 77 = i; // ERROR
```

因此，C++ 程序中的每个表达式都有值类别。除了类型之外，值类别对于决定表达式可以做什么也很重要。

然而，值类别在 C++ 中随着时间的推移而改变。

8.1.1 值类型的历史

从历史上看（引用 Kernighan&Ritchie C, K&R C），最初只有 lvalue 和 rvalue：

- lvalue 可以出现在赋值的左边
- rvalue 只能出现在赋值的右侧

根据这个定义，当使用 int 对象/变量时，使用的是 lvalue，但当使用 int 字面值时，使用的是 rvalue：

```
1 int x; // x is an lvalue when used in an expression
2
3 x = 42; // OK, because x is an lvalue and the type matches
4 42 = x; // ERROR: 42 is an rvalue and can be only on the right-hand side of an
           assignment
```

然而，这些类别不仅重要，并且通常用于指定表达式是否以及在何处可以使用。例如：

```
1 int x; // x is an lvalue when used in an expression
2
3 int* p1 = &x; // OK: & is fine for lvalues (object has a specified location)
4 int* p2 = &42; // ERROR: & is not allowed for rvalues (object has no specified
                 location)
```

然而，在 ANSI-C 中，事情变得更加复杂。因为声明为 *const int* 的 x 不能放在赋值函数的左边，但仍然可以在其他只能使用左值的地方使用：

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

```
2  
3 c = 42; // now an ERROR (so that c should no longer be an lvalue)  
4 const int* p1 = &c; // still OK (so that c should still be an lvalue)
```

C 语言中，声明为 *const int* 的 *c* 仍然是 lvalue，因为对于特定类型的 *const* 对象，仍然可以调用大多数 lvalue 操作。唯一不能做的就是在赋值函数的左边有一个 *const* 对象。

因此，在 ANSI-C 中，l 的含义变成了定位。lvalue 现在是程序中具有指定位置的对象（例如，以便您可以获取地址）。以同样的方式，rvalue 现在只是一个可读的值。

C++98 采用了这些值类别的定义。然而，随着移动语义的引入，问题出现了：用 *std::move()* 标记的对象应该有哪些值类别，用 *std::move()* 标记的类的对象应该遵循以下规则：

```
1 std :: string s;  
2 ...  
3 std :: move(s) = "hello"; // OK (behaves like an lvalue)  
4 auto ps = &std :: move(s); // ERROR (behaves like an rvalue)
```

但是，请注意基本数据类型 (FDT) 的行为：

```
1 int i;  
2 ...  
3 std :: move(i) = 42; // ERROR  
4 auto pi = &std :: move(i); // ERROR
```

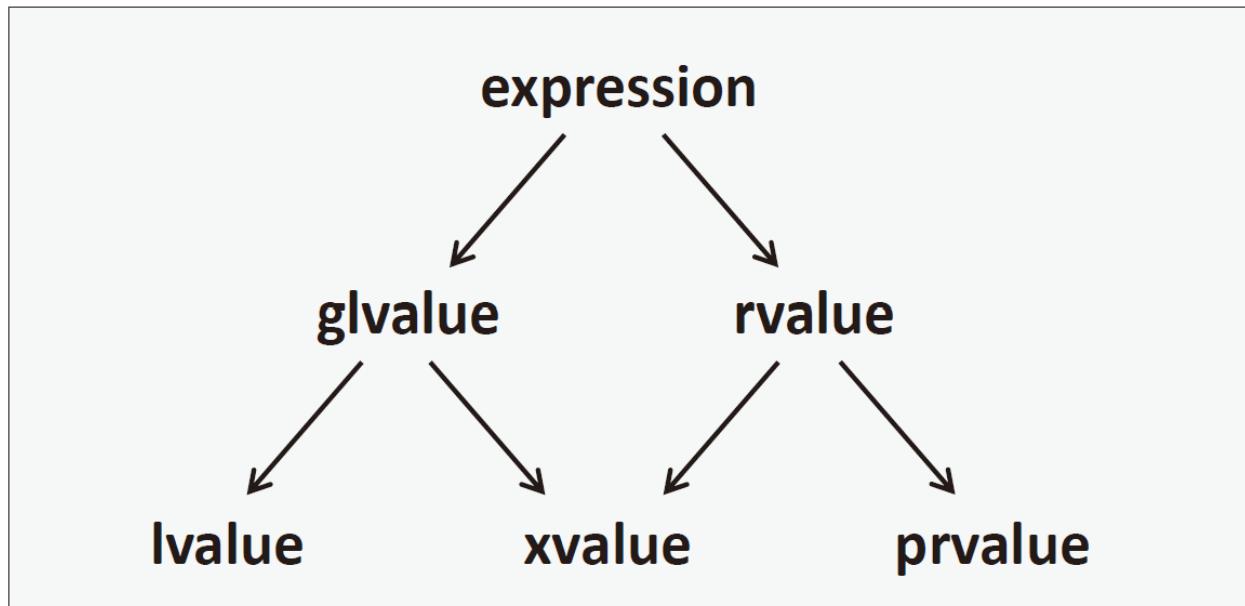
除了基本数据类型之外，标记为 *std::move()* 的对象仍应该像 lvalue 一样，允许修改它的值。另一方面，也存在一些限制，比如不能获取该地址。

因此，引入了一个新的类别 xvalue(“eXpire value”) 来为显式标记的对象指定规则，因为这里不再需要这个值（主要是用 *std::move()* 标记的对象）。大多数 C++11 前的 rvalue 的规则也适用于 xvalue。因此，以前的 rvalue 变成了一个复合值类别，现在表示新的主值类别 prvalue(对于以前的所有 rvalue) 和 xvalue。关于提出这些改变的论文，请参阅 <http://wg21.link/n3055>。

8.1.2 C++11 的值类别

C++11 的值类别如图 8.1 所示。

图 8.1 C++11 的值类别



有以下主要类别:

- lvalue (“定位值”)
- prvalue (“纯可读值”)
- xvalue (“过期值”)

综合类别为:

- glvalue (“广义 lvalue”) 作为 “左 lvalue 或 xvalue”的常用术语
- rvalue 作为 “xvalue 或 prvalue”的常用术语

基本表达式的值分类

lvalue 的例子有:

- 仅为变量、函数或数据成员 (除右值的普通值成员外) 的名称的表达式
- 只是字符串字面量的表达式 (例如, "hello")
- 如果函数声明返回左值引用, 则返回函数的值 (返回类型 type &)
- 任何对函数的引用, 即使标记为 `std::move()`(参见下面)
- 内置的一元操作符 * 的结果 (即, 对原始指针进行解引用所产生的结果)

prvalue 的例子有:

- 由非字符串字面量的内置字面量组成的表达式 (例如, 42、true 或 nullptr)
- 如果函数声明为按值返回, 则按类型返回值 (返回类型为 Type)。
- 内置的一元操作符 & 的结果 (即, 获取表达式的地址所产生的结果)
- Lambda 表达式

xvalues 的示例如下:

- 用 `std::move()` 标记对象的结果
- 对对象类型 (不是函数类型) 的 rvalue 引用的强制转换

- 函数声明返回 rvalue 引用 (返回类型 type $\&&$)
- 右值的非静态值成员 (见下面)

例如:

```

1 class X {
2 };
3
4 X v;
5 const X c;
6
7 f(v); // passes a modifiable lvalue
8 f(c); // passes a non-modifiable lvalue
9 f(X()); // passes a prvalue (old syntax of creating a temporary)
10 f(X{}); // passes a prvalue (new syntax of creating a temporary)
11 f(std::move(v)); // passes an xvalue

```

说些经验法则:

- 所有用作表达式的名称都是 lvalue。
- 所有用作表达式的字符串字面值都是 lvalue。
- 所有非字符串字面值 (4.2、true 或 nullptr) 都是 prvalue。
- 所有没有名称的临时对象 (特别是回的对象) 都是 prvalues。
- 所有标记为 `std::move()` 的对象，及其值成员都是 xvalues。

严格来说，glvalues、prvalues 和 xvalues 是表达式的术语，而不是值的术语 (这意味着这些术语用词不当)。例如，变量本身不是 lvalue，只有表示该变量为 lvalue 的表达式:

```

1 int x = 3; // here, x is a variable, not an lvalue
2 int y = x; // here, x is an lvalue

```

第一个语句中，3 是一个初始化变量 x 的 prvalue(不是 lvalue)。第二个语句中，x 是一个 lvalue(它的计算值指定了一个包含值 3 的对象)。lvalue x 用作 rvalue，它初始化变量 y。

8.1.3 C++17 新加的值类别

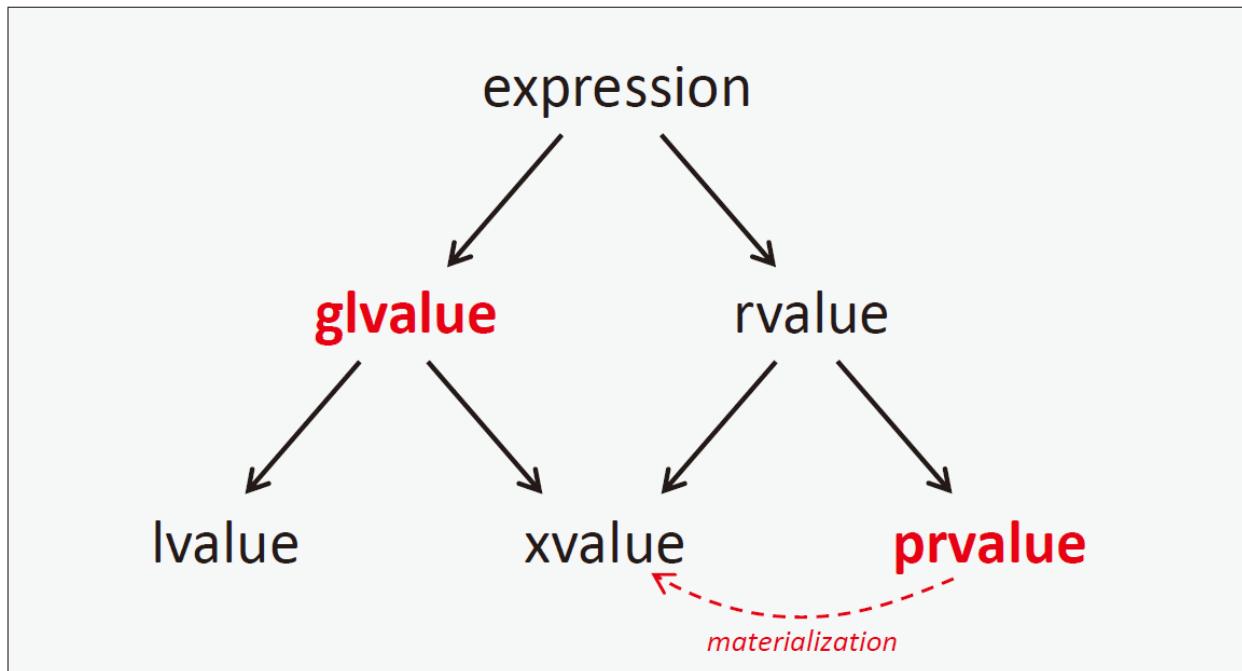
C++17 具有相同的值类别，图 8.2 中描述了值类别的语义。

现在解释值类别有两种主要的表达方式:

- glvalues: 用于长生命周期对象或函数位置的表达式
- prvalues: 用于短生命周期对象的初始化表达式

然后，xvalue 是表示不再需要其资源/值的 (长期存在的) 对象。

图 8.2 C++17 新加的值类别



通过值传递 prvalues

有了这个更改，即使没有定义有效的副本和有效的移动构造函数，现在也可以将 prvalue 作为未命名的初始值按值传递：

```

1 class C {
2     public:
3     C( ... );
4     C(const C&) = delete; // this class is neither copyable ...
5     C(C&&) = delete; // ... nor movable
6 };
7
8 C createC() {
9     return C{ ... }; // Always creates a conceptual temporary prior to C++17.
10} // In C++17, no temporary object is created at this point.
11
12 void takeC(C val) {
13     ...
14 }
15
16 auto n = createC(); // OK since C++17 (error prior to C++17)
17
18 takeC(createC()); // OK since C++17 (error prior to C++17)
  
```

C++17 之前，如果没有复制或移动支持，传递 prvalue(例如：`createC()` 的创建和初始化返回值) 是不可能的。但是，从 C++17 开始，只要是不需要有地址的对象，就可以按值传递 prvalues。

具象化

C++17 随后引入了一个新术语，称为具象化（未命名的临时对象），此时 prvalue 变成了临时对象。因此，临时物化转换是 prvalue 到 xvalue 转换（通常是隐式的）。

需要 glvalue（左值或 xvalue）的地方使用 prvalue，就会创建临时对象，并使用该 prvalue 初始化（记住，prvalue 主要是“初始化值”），并且该 prvalue 会指定临时对象的 xvalue。因此，在上面的例子中：

```
1 void f(const X& p); // accepts an expression of any value category but expects a
2                                     glvalue
3 f(X{}); // creates a temporary prvalue and passes it materialized as an xvalue
```

因为本例中的 `f()` 有一个引用形参，所以需要一个 glvalue 参数。然而，表达式 `X{}` 是 prvalue。因此，“临时具象化”的规则开始起作用，表达式 `X{}` 被“转换”为 xvalue，该 xvalue 指定使用默认构造函数初始化的临时对象。

请注意，具象化并不意味着我们创建新的/不同的对象。左值引用 `p` 仍然绑定到 xvalue 和 prvalue，尽管后者现在会涉及到向 xvalue 的转换。

8.2 值类别的特殊规则

对于影响移动语义的函数和成员的值类型，有特殊的规则。

8.2.1 函数的值类型

C++ 标准中的特殊规则是，所有引用函数的表达式都是 lvalue。

例如：

```
1 void f(int) {
2 }
3
4 void(&fref1)(int) = f; // fref1 is an lvalue
5 void(&&fref2)(int) = f; // fref2 is also an lvalue
6
7 auto& ar = std::move(f); // OK: ar is lvalue of type void(&)(int)
```

与对象类型相反，可以将非 `const` lvalue 引用绑定到标记为 `std::move()` 的函数指针，因为标记为 `std::move()` 的函数指针仍然是 lvalue。

8.2.2 数据成员的值类型

如果使用对象的数据成员（例如，使用 `std::pair<>` 的第一个和第二个成员时），将使用特殊规则。

通常，数据成员的值类型如下：

- lvalue 的数据成员是 lvalue。
- rvalue 的引用和静态数据成员是 lvalue。
- rvalue 的普通数据成员是 xvalue。

这些规则反映了引用或静态成员实际上不是对象的一部分。如果不再需要对象的值，这也适用于对象的普通数据成员。但是，引用或静态的成员的值可能被其他对象所使用。

例如：

```
1 std::pair<std::string, std::string&> foo(); // note: member second is reference
2
3 std::vector<std::string> coll;
4 ...
5 coll.push_back(foo().first); // moves because first is an xvalue here
6 coll.push_back(foo().second); // copies because second is an lvalue here
```

需要使用 `std::move()` 来移动第二个成员：

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

如果有 lvalue(一个有名字的对象)，就有两种使用 `std::move` 的方式来移动成员：

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

`std::move()` 的意思是“不再需要这个值”，所以不再需要对象的值，应该标记 `obj`。如果不再需要成员的值，应该标记 `member`。然而，实际情况会比较复杂。

`std::move()` 用于普通数据成员

如果成员既不是静态也不是引用，`std::move()` 能将成员转换为 xvalue，以便能够使用移动语义。

考虑声明了以下内容：

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

以下代码先将成员移动，然后再将成员移动到 `coll` 中：

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

但是，下面的代码具有相同的效果：

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

看起来有点奇怪，`std::move()` 标记对象之后仍然使用 `obj`。在本例中，知道对象的哪个部分可以移动，所以可以使用未移动的部分。因此，当必须实现移动构造函数时，我更喜欢用 `std::move()` 标记成员。

`std::move()` 用于引用或静态成员

如果成员是引用或静态的，则使用不同的规则:rvalue 的引用或静态成员是 lvalue。同样，这条规则反映了，成员的值并不是对象的一部分。“不再需要对象的值”并不意味着“不再需要不属于对象的值(成员的值)”。

因此，如果有引用或静态成员，那么如何使用 `std::move()` 是有区别的：

- 对对象使用 `std::move()` 不起作用：

```
1 struct S {  
2     static std::string statString; // static member  
3     std::string& refString; // reference member  
4 };  
5 S obj;  
6 ...  
7 coll.push_back(std::move(obj).statString); // copies statString  
8 coll.push_back(std::move(obj).refString); // copies refString  
9
```

- 对成员使用 `std::move()` 具有的效果：

```
1 struct S {  
2     static std::string statString;  
3     std::string& refString;  
4 };  
5 S obj;  
6 ...  
7 coll.push_back(std::move(obj.statString)); // moves statString  
8 coll.push_back(std::move(obj.refString)); // moves refString  
9
```

这样的举措是否有用是另一个问题。窃取静态成员或引用成员的值意味着修改所使用对象外部的值，这还能说得通，但也可能是意外和危险的。通常，类型应该更好地保护对这些成员的访问。

泛型代码中，可能不知道成员是静态的还是引用的。因此，使用 `std::move()` 来标记对象是不那么危险的，就是看起来奇怪：

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

稍后将介绍的 `std::forward<>()` 可以用来完美地转发对象的成员。参见 `basics/members.cpp` 获取完整的示例。

8.3 绑定引用时值类型的影响

将引用绑定到对象时，值类型起着重要的作用。例如，在 C++98/C++03 中，定义了可以将 rvalue(没有名称的临时对象或标有 `std::move()` 的对象) 赋值或传递给 `const` lvalue 引用，但不能传递给非 `const` lvalue 引用：

```
1 std::string createString(); // forward declaration  
2
```

```

3 const std::string& r1{createString(); // OK
4
5 std::string& r2{createString(); // ERROR

```

这里编译器打印的错误消息是“不能将非 *const* lvalue 引用绑定到 rvalue”。

调用 *foo2()* 时也会得到这个错误消息:

```

1 void foo1(const std::string&); // forward declaration
2 void foo2(std::string&); // forward declaration
3
4 foo1(std::string{"hello"}); // OK
5 foo2(std::string{"hello"}); // ERROR

```

8.3.1 解析 rvalue 引用的重载

让我们看看传递对象给引用时的规则。

假设类 *X* 中有一个非 *const* 变量 *v* 和一个 *const* 变量 *c*:

```

1 class X {
2 ...
3 };
4
5 X v{ ... };
6 const X c{ ... };

```

如果提供了函数 *f()* 的所有引用重载，则绑定引用的规则表会列出了传递参数的绑定引用的规则:

```

1 void f(const X&); // read-only access
2 void f(X&); // OUT parameter (usually long-living object)
3 void f(X&&); // can steal value (object usually about to die)
4 void f(const X&&); // no clear semantic meaning

```

数字列出了重载解析的优先级，以便了解在提供多个重载时调用了哪个函数。数字越小，优先级越高 (优先级 1 表示最优先)。

注意，只能将 rvalue(prvalues, 如没有名称的临时对象) 或 xvalues(用 *std::move()* 标记的对象) 传递给 rvalue 引用。

通常可以忽略表的最后一列，因为 *const* rvalue 引用在语义上没有多大意义，这意味着我们有以下规则:

表 8.1 绑定引用规则表

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

- 非 *const lvalue* 引用只接受非 *const lvalue*。
- rvalue 引用只接受非 *const rvalue*。
- *const lvalue* 引用可以接受所有内容，并在没有提供其他重载的情况下充当备选机制。

下面是从表中提取的移动语义的备选机制规则：

Call	<code>f(const X&)</code>	<code>f(X&&)</code>
<code>f(X{})</code>	3	1
<code>f(move(v))</code>	3	1

如果向函数传递 rvalue(临时对象或标记为 `std::move()` 的对象)，而移动语义没有特定的实现(通过接受 rvalue 引用声明)，则使用通常的复制语义，`const&` 接受实参。

请注意，在介绍通用引用/转发引用时会扩展此表。

有时可以将 lvalue 传递给 rvalue 引用(当使用模板形参时)。请注意，并非每个带有 `&&` 的声明都遵循相同的规则。这里的规则适用于使用 `&&` 声明类型(或类型别名)的情况。

8.3.2 通过引用和值进行重载

可以通过引用和值来声明函数：

```

1 void f(X); // call-by-value
2 void f(const X&); // call-by-reference
3 void f(X&);
4 void f(X&&);
5 void f(const X&&);

```

原则上，这些重载的声明没问题。但是，按值调用和按引用调用之间没有特定的优先级。如果函数声明以值作为参数(它可以接受任何值类别的任何参数)，那么任何匹配声明以引用作为参数都会造成歧义。

因此，只能通过值或引用(使用认为有用的尽可能多的引用重载)接受参数，但永远不要两者都接受。

8.4 当 lvalue 变成 rvalue

当使用具体类型的 rvalue 引用形参声明函数时，只能将这些形参绑定到 rvalue。例如：

```

1 void rvFunc(std::string&&); // forward declaration
2
3 std::string s{ ... };
4 rvFunc(s); // ERROR: passing an lvalue to an rvalue reference
5 rvFunc(std::move(s)); // OK, passing an xvalue

```

但请注意，有时传递 lvalue 是可行的。例如：

```

1 void rvFunc(std::string&&); // forward declaration
2
3 rvFunc("hello"); // OK, although "hello" is an lvalue

```

记住，字符串文字作为表达式使用时是 lvalue。因此，不能传递给 rvalue 引用。但是，这里涉及到一个隐藏的操作，因为实参的类型（6 个常量字符的数组）与形参的类型不匹配。隐式类型转换由 string 构造函数执行，创建了一个没有名称的临时对象。

因此，真正的使用方式如下：

```
1 void rvFunc(std::string&&); // forward declaration
2
3 rvFunc(std::string{"hello"}); // OK, "hello" converted to a string is a prvalue
```

8.5 当 rvalue 变成 lvalue

现在让了解一下将形参声明为 rvalue 引用的函数的实现：

```
1 void rvFunc(std::string&& str) {
2     ...
3 }
```

只能传递 rvalue：

```
1 std::string s{ ... };
2 rvFunc(s); // ERROR: passing an lvalue to an rvalue reference
3 rvFunc(std::move(s)); // OK, passing an xvalue
4 rvFunc(std::string{"hello"}); // OK, passing a prvalue
```

然而，当在函数内部使用 str 形参时，处理的是有名称的对象。这意味着使用 str 作为 lvalue。

不能直接递归地调用自己的函数：

```
1 void rvFunc(std::string&& str) {
2     rvFunc(str); // ERROR: passing an lvalue to an rvalue reference
3 }
```

必须再次用 `std::move()` 标记 str：

```
1 void rvFunc(std::string&& str) {
2     rvFunc(std::move(str)); // OK, passing an xvalue
3 }
```

这是没有传递移动语义规则的规范。这是特性，而不是 bug。如果传递了移动语义，就不能使用两次传递了移动语义的对象，因为第一次使用后，就会失去它的值。或者，需要临时禁用移动语义的特性。

如果将 rvalue 引用参数绑定到 rvalue(prvalue 或 xvalue)，该对象将作为 lvalue，必须再次将其转换为 rvalue，以便传递给 rvalue 引用。

现在，请记住 `std::move()` 只不过是将 rvalue 引用的 `static_cast`。也就是说，可以在递归调用中编写如下程序：

```
1 void rvFunc(std::string&& str) {
2     rvFunc(static_cast<std::string&&>(str)); // OK, passing an xvalue
3 }
```

将对象 *str* 转换为 *string* 类型。通过强制转换，改变值的类型。根据规则，通过对 rvalue 引用的强制转换，lvalue 变成了 xvalue，因此允许将对象传递给 rvalue 引用。

这并不是什么新鲜事：即使在 C++11 之前，声明为 lvalue 引用的形参在使用时也遵循 lvalue 规则。关键是声明中的引用指定了可以传递给函数的内容。对于函数内部的行为，与引用无关。

困惑吗？这就是在 C++ 标准中定义移动语义和值类型的规则。是否有足够的了解，其实并不重要，编译器明白这些规则其实就足够了。

这里需要了解的是移动语义没有传递。如果传递一个带有移动语义的对象，必须再次用 *std::move()* 标记，将其语义转发给另一个函数。

8.6 使用 decltype 检查值类别

与移动语义一起，C++11 引入了一个新的关键字 *decltype*。这个关键字的主要目标是获得声明对象的确切类型，也可以用于确定表达式的值类型。

8.6.1 使用 decltype 检查名称的类型

在接受 rvalue 引用形参的函数中，可以使用 *decltype* 查询并使用形参的确切类型。只需将参数的名称传递给 *decltype*。例如：

```
1 void rvFunc(std::string&& str)
2 {
3     std::cout << std::is_same<decltype(str), std::string>::value; // false
4     std::cout << std::is_same<decltype(str), std::string&>::value; // false
5     std::cout << std::is_same<decltype(str), std::string&&>::value; // true
6     std::cout << std::is_reference<decltype(str)>::value; // true
7     std::cout << std::is_lvalue_reference<decltype(str)>::value; // false
8     std::cout << std::is_rvalue_reference<decltype(str)>::value; // true
9 }
```

decltype(str) 表达式总是表示 *str* 的类型，即 *std::string&&*。在表达式中任何需要该类型的地方都可以使用该类型。类型特征（类型函数如 *std::is_same<>*）会帮助我们处理这些类型。

例如，要声明传递的形参类型不是引用的新对象，可以声明：

```
1 void rvFunc(std::string&& str)
2 {
3     std::remove_reference<decltype(str)>::type tmp;
4     ...
5 }
```

tmp 在这个函数中是 *std::string* 类型（也可以显式地声明，如果使它成为 T 类型对象的泛型函数，代码仍可以工作）。

8.6.2 使用 decltype 检查值类型

目前为止，只向 *decltype* 传递了名称来查询类型。但是，也可以将表达式（不仅仅是名称）传递给 *decltype*，会根据以下约定生成值类型：

- 对于 prvalue，产生值类型:type

- 对于 lvalue，将其类型作为 lvalue 引用:type&
- 对于 xvalue，将其类型作为 rvalue 引用:type&&

例如:

```

1 void rvFunc(std::string&& str)
2 {
3     decltype(str + str) // yields std::string because s+s is a prvalue
4     decltype(str[0]) // yields char& because the index operator yields an lvalue
5     ...
6 }
```

这意味着，如果只是传递一个放在圆括号内的名称（这是一个表达式，而不再只是名称），decltype 将生成其类型。行为如下：

```

1 void rvFunc(std::string&& str)
2 {
3     std::cout << std::is_same<decltype((str)), std::string>::value; // false
4     std::cout << std::is_same<decltype((str)), std::string&>::value; // true
5     std::cout << std::is_same<decltype((str)), std::string&&>::value; // false
6     std::cout << std::is_reference<decltype((str))>::value; // true
7     std::cout << std::is_lvalue_reference<decltype((str))>::value; // true
8     std::cout << std::is_rvalue_reference<decltype((str))>::value; // false
9 }
```

将此函数与不使用括号的前一个函数实现进行比较。这里，decltype(str) 的结果是 std::string&，因为 str 是 lvalue 的 std::string 类型。

对于 decltype，当传递的名称周围加上圆括号时，会产生不同的结果，这在稍后讨论 decltype(auto) 时会很重要。

检查值类型内部代码

现在可以在代码中检查特定的值类别，如下所示：

- !std::is_reference_v<decltype((expr))> 检查 expr 是否为 prvalue。
- std::is_lvalue_reference_v<decltype((expr))> 检查 expr 是否为 lvalue。
- std::is_rvalue_reference_v<decltype((expr))> 检查 expr 是否为 xvalue。
- !std::is_lvalue_reference_v<decltype((expr))> 检查 expr 是否为 rvalue。

请再次注意这里使用的括号，以确保即使只传递名称 *expr*，也使用 decltype 的值-类别检查形式。

C++20 之前，必须使用::value 来替代后缀 _v。

8.7 总结

- C++ 程序中的任何表达式都只属于以下主要值类别中的一种：
 - lvalue (用于命名对象或字符串字面量)
 - prvalue (用于未命名的临时对象)

- xvalue (对于标记为 `std::move()` 的对象)
 - C++ 中的调用或操作是否有效取决于类型和值类别。
 - 类型的 rvalue 引用只能绑定到 rvalue(prvalues 或 xvalues)。
 - 隐式操作可能更改传递参数的值类别。
 - 将 rvalue 传递给 rvalue 引用，可以将其绑定到 lvalue。
 - 移动语义不可传递。
 - 函数和对函数的引用总是 lvalue。
- 对于 rvalue(临时对象或标记为 `std::move()` 的对象)，普通值成员具有移动语义，而引用或静态成员没有。
- decltype 既可以检查所传递名称的声明类型，也可以检查所传递表达式的类型和值类别。

2 Part II: 泛型代码中的移动语义

这一部分介绍了 C++ 为泛型编程 (模板) 提供的移动语义。

9 完美转发

本章介绍泛型代码中的移动语义。特别讨论了通用引用(转发引用)和完美转发。

接下来的内容将讨论引用和完美转发的细节，以及如何处理泛型代码中的移动返回值。

9.1 完美转发的动机

知道了移动语义不能自动传递，所以对泛型代码会有影响。

9.1.1 为什么需要完美转发

要将带有移动语义的对象转发给函数，不仅需要绑定到 rvalue 引用，还需要再次使用 `std::move()` 将其移动语义转发给另一个函数。

例如，引用重载函数的规则：

```
1 class X {
2     ...
3 };
4
5 // forward declarations:
6 void foo(const X&); // for constant values (read-only access)
7 void foo(X&); // for variable values (out parameters)
8 void foo(X&&); // for values that are no longer used (move semantics)
```

调用这些函数时，有以下规则：

```
1 X v;
2 const X c;
3
4 foo(v); // calls foo(X&)
5 foo(c); // calls foo(const X&)
6 foo(X{}); // calls foo(X&&)
7 foo(std::move(v)); // calls foo(X&&)
8 foo(std::move(c)); // calls foo(const X&)
```

假设通过协助函数 `callFoo()` 间接地调用相同的参数 `foo()`。函数还需要三个重载：

```
1 void callFoo(const X& arg) { // arg binds to all const objects
2     foo(arg); // calls foo(const X&)
3 }
4 void callFoo(X& arg) { // arg binds to lvalues
5     foo(arg); // calls foo(X&)
6 }
7 void callFoo(X&& arg) { // arg binds to rvalues
8     foo(std::move(arg)); // needs std::move() to call foo(X&&)
9 }
```

这里，`arg` 都用作 lvalue(具有名称的对象)。第一个版本将其作为 `const` 对象转发，但其他两种情况实现了转发非 `const` 参数的两种不同方式：

- 声明为 lvalue 引用(绑定到没有移动语义的对象)的参数按原样传递。

- 声明为 rvalue 引用 (绑定到具有移动语义的对象) 的参数通过 `std::move()` 传递。

这可以完美地转发移动语义: 对于任何通过移动语义传递的参数, 保持移动语义。当遇到没有移动语义的参数时, 不添加移动语义。

看下 `callFoo()` 如何调用不同的 `foo()`:

```

1 X v;
2 const X c;
3 callFoo(v); // calls foo(X&)
4 callFoo(c); // calls foo(const X&)
5 callFoo(X{}); // calls foo(X&&)
6 callFoo(std::move(v)); // calls foo(X&&)
7 callFoo(std::move(c)); // calls foo(const X&)

```

请记住, 传递给 rvalue 引用的 rvalue 在使用时成为 lvalue, 需要 `std::move()` 再次将其作为 rvalue 传递。但是, 有些地方不能使用 `std::move()`。对于其他重载, 当传递 rvalue 时, 使用 `std::move()` 将调用 `foo()` 的重载实现来获取 rvalue 引用。

为了在泛型代码中实现完美转发, 需要为每个参数进行重载。为了支持所有组合, 对 2 个泛型参数有 9 个重载, 对 3 个泛型参数有 27 个重载。

因此, C++11 引入了一种方式来完美地转发给定的参数, 不需要任何重载, 仍然保持类型和具体值。

完美转发 `const rvalue` 引用

虽然 `const rvalue` 引用没有语义上的含义, 但想要用 `std::move()` 标记的常量对象的类型和值, 还需要第四个重载:

```

1 void callFoo(const X&& arg) { // arg binds to const rvalues
2   foo(std::move(arg)); // needs std::move() to call foo(const X&&)
3 }

```

否则, 将调用 `foo(const X&)`。这通常没问题的, 但在某些情况下, 可能希望保留传递 `const rvalue` 引用的信息 (例如, 出于某种原因, 提供了 `foo(const X&&)` 的重载)。

有了完美转发的特性, 泛型代码就没必要为了对两个或三个参数, 进行 16 和 64 次重载了。

9.2 实现完美转发

为了避免重载具有不同值类型的参数的函数, C++ 引入了一种机制来实现完美转发。需要三样东西:

1. 将调用形参作为纯 rvalue 引用 (使用 `&&` 声明, 但不使用 `const` 或 `volatile`)。
2. 形参的类型必须是函数模板的形参。
3. 将形参转发给另一个函数时, 可以使用 `std::forward<>()` 的辅助函数, 该函数在 `<utility>` 头文件中声明。

函数的完美转发参数, 如下所示:

```

1 template<typename T>
2 void callFoo(T&& arg) {

```

```
3     foo(std::forward<T>(arg)); // equivalent to foo(std::move(arg)) for passed rvalues
4 }
```

`std::forward<>()` 实际上是一个条件性 `std::move()`, 这样就得到了与上面 `callFoo()` 的三个 (或四个) 重载等价的行为:

- 如果传递 rvalue 给 `arg`, 就会产生与调用 `foo(std::move(arg))` 相同的效果。
- 如果我们传递一个 lvalue 给 `arg`, 就会产生与调用 `foo(arg)` 相同的效果。

同样, 可以完美传递两个参数:

```
1 template<typename T1, typename T2>
2 void callFoo(T1&& arg1, T2&& arg2) {
3     foo(std::forward<T1>(arg1), std::forward<T2>(arg2));
4 }
```

也可以将 `std::forward<>()` 应用于可变参的实参, 完美地将进行转发:

```
1 template<typename ... Ts>
2 void callFoo(Ts&&... args) {
3     foo(std::forward<Ts>(args)...);
4 }
```

注意, 不会对所有参数一次性使用 `forward<>()`, 而是会分别为每个参数使用。因此, 必须将省略号 ("...") 放在 `forward()` 表达式的末尾, 而不是直接放在 `args` 的后面。

然而, 这里到底发生了什么, 需要详细的解释。

9.2.1 通用 (或转发) 引用

首先, 将 `arg` 声明为 rvalue 引用形参:

```
1 template<typename T>
2 void callFoo(T&& arg); // arg is universal/forwarding reference
```

这可能会给人一种应该适用 rvalue 引用的一般规则的感觉。然而, 事实并非如此。函数模板形参的 rvalue 引用 (未限定为 `const` 或 `volatile`), 不遵循普通 rvalue 引用的规则。所以, 不是一回事

两个术语: 通用引用和转发引用

这样的引用称为通用引用。但 C++ 标准中还使用了另一个术语: 转发引用。这两个术语没有区别, 只是一个历史问题, 两个术语的含义是相同的。

这两个术语都描述了通用引用/转发引用的基本面:

- 可以统一绑定到所有类型的对象 (`const` 和非 `const`) 和值类别。
- 通常用来转发参数, 但这并不是唯一的用法 (这也是我更喜欢“通用引用”的原因之一)。

通用引用绑定到所有值类别

通用引用的重要特性是, 可以绑定到任何值类别的对象和表达式:

```

1 template<typename T>
2 void callFoo(T&& arg); // arg is a universal/forwarding reference
3
4 X v;
5 const X c;
6 callFoo(v); // OK
7 callFoo(c); // OK
8 callFoo(X{}); // OK
9 callFoo(std::move(v)); // OK
10 callFoo(std::move(c)); // OK

```

此外，保持所绑定对象的常量和值类别 (无论我们有 rvalue 还是 lvalue):

```

1 template<typename T>
2 void callFoo(T&& arg); // arg is a universal/forwarding reference
3
4 X v;
5 const X c;
6 callFoo(v); // OK, arg is X&
7 callFoo(c); // OK, arg is const X&
8 callFoo(X{}); // OK, arg is X&&
9 callFoo(std::move(v)); // OK, arg is X&&
10 callFoo(std::move(c)); // OK, arg is const X&&

```

按照规则，类型 $T\&\&$ 是 arg 的类型

- 如果引用 lvalue，则为 lvalue 引用
- 如果引用 rvalue，则为 rvalue 引用

注意，用 *const*(或 *volatile*) 限定的泛型 rvalue 引用不是通用引用。只能传递 rvalue:

```

1 template<typename T>
2 void callFoo(const T&& arg); // arg is not a universal/forwarding reference
3
4 const X c;
5 callFoo(c); // ERROR: c is not an rvalue
6 callFoo(std::move(c)); // OK, arg is const X&&

```

这里，还没有谈到 T 的类型。稍后将解释什么样的类型可以导出为通用引用的 T 。

稍后，将讨论使用 Lambda 的相应示例。

9.2.2 $std::forward<>()$

callFoo() 内部，使用如下所示的通用引用:

```

1 template<typename T>
2 void callFoo(T&& arg) {
3     foo(std::forward<T>(arg)); // becomes foo(std::move(arg)) for passed rvalues
4 }

```

和 *std::move()* 一样，*std::forward<>()* 也定义在头文件 *<utility>* 中。

`std::forward<t>(arg)` 其实是这样实现的:

- 如果传递给函数的是 T 类型的 rvalue，则表达式等价于 `std::move(arg)`。
- 如果传递给函数的是 T 类型的 lvalue，则表达式等价于 `arg`。

也就是说，`std::forward<>()` 是 `std::move()`，仅用于传递 rvalue。

就像 `std::move()` 一样，`std::forward<>()` 的语义是在这里不再需要这个值，另外保留了要传递的通用引用绑定的对象类型（包括常量）和值类别。你可以争辩说，需要达成条件才不再需要这个值，但是因为不知道 `std::forward<>()` 是否变成了 `std::move()`，所以假设对象之后有值就是错误的。因此，使用 `std::forward<>()` 之后，对象通常有效，但可能不知道具体值。

`std::forward<>()` 用于成员函数

注意，可以在调用成员函数时使用 `std::forward<>()` 作为通用引用。记住，成员函数可能使用引用限定符对移动语义有特定的重载。如果不再需要该对象的值，可以使用 `std::forward<>()` 来调用成员函数。

例如，假设重载了 getter 来提高返回临时人员名称的性能:

```
1 class Person
2 {
3     private:
4         std::string name;
5     public:
6         ...
7         void print() const {
8             std::cout << "print()\n";
9         }
10
11        std::string getName() && { // when we no longer need the value
12            return std::move(name); // we steal and return by value
13        }
14        const std::string& getName() const& { // in all other cases
15            return name; // we give access to the member
16        }
17    };
```

采用通用引用的函数中，可以使用 `std::forward<>()`，如下所示:

```
1 template<typename T>
2 void foo(T&& x)
3 {
4     x.print(); // OK, no need to forward the passed value category
5
6     x.getName(); // calls getName() const&
7     std::forward<T>(x).getName(); // calls getName() && for rvalues (OK, no longer
8     need x)
9 }
```

使用 `std::forward<>()` 之后, `x` 处于有效但未指定的状态。无论何时使用 `std::forward<>()`, 请确保不再使用该对象。

9.2.3 完美转发的效果

结合声明通用引用的行为和 `std::forward<>()` 的使用, 得到了以下行为:

```
1 void foo(const X&); // for constant values (read-only access)
2 void foo(X&); // for variable values (out parameters)
3 void foo(X&&); // for values that are no longer used (move semantics)
4 template<typename T>
5
6 void callFoo(T&& arg) { // arg is a universal/forwarding reference
7     foo(std::forward<T>(arg)); // becomes foo(std::move(arg)) for passed rvalues
8 }
9
10 X v;
11 const X c;
12
13 callFoo(v); // OK, expands to foo(arg), so it calls foo(X&)
14 callFoo(c); // OK, expands to foo(arg), so it calls foo(const X&)
15 callFoo(X{}); // OK, expands to foo(std::move(arg)), so it calls foo(X&&)
16 callFoo(std::move(v)); // OK, expands to foo(std::move(arg)), so it calls foo(X&&)
17 callFoo(std::move(c)); // OK, expands to foo(std::move(arg)), so it calls foo(const
18 // X&)
```

传递给 `callFoo()` 的任何参数都会变成 lvalue(因为参数 `arg` 是有名称的对象)。然而, `arg` 的类型取决于传递的内容:

- 如果传递 lvalue, `arg` 就是一个 lvalue 引用 (传递非 `const X` 时是 `X&`, 传递 `const X` 时是 `X&&`)。
- 如果传递 rvalue(未命名的临时对象或用 `std::move()` 标记的对象), 则 `arg` 是 rvalue 引用 (`X&&` 或 `const X&&`)。

当有 rvalue 引用时 (即, `arg` 绑定到右值), 通过 `std::forward<>()`, 就可以用 `std::move()` 转发形参。

9.3 右值引用与通用引用

不幸的是, 通用/转发引用使用与普通 rvalue 引用相同的语法 (形式上, 是特殊的 rvalue 引用)。这是混乱的根源, 如果看到有两个 & 符号的声明, 必须再次检查使用的是真实类型名, 还是函数模板形参名。

换句话说, 两者之间有很大的区别

```
1 void foo(Coll&& arg) // arg is an ordinary rvalue reference of type Coll
```

和

```
1 template<typename Coll>
2 void foo(Coll&& arg) // arg is a universal/forwarding reference of any type
```

详细讨论一下两者的区别。

9.3.1 实际类型的 rvalue 引用

普通的 rvalue 引用就不是使用函数的模板形参名 (或者该引用是用 *const* 或 *volatile* 声明的), 所以只能将这些引用绑定到 rvalue。此外, 传递的实参还不是 *const*:

```
1 using Coll = std::vector<std::string>;
2
3 void foo(Coll&& arg) // arg is an ordinary rvalue reference
{
4     Coll coll; // coll can't be const
5     ...
6     bar(std::move(arg)); // perfectly forward to bar() (no need to use std::forward
7         <>() here)
8 }
9
10 Coll v;
11 const Coll c;
12
13 foo(v); // ERROR: can't bind rvalue reference to lvalue
14 foo(c); // ERROR: can't bind rvalue reference to lvalue
15 foo(Coll{});
16 foo(std::move(v)); // OK, arg binds to a non-const prvalue
17 foo(std::move(c)); // ERROR: can't bind non-const rvalue reference to const xvalue
```

foo() 中:

- *arg* 的类型 *Coll* 绝不是 *const*。
- 使用 *std::forward<>()* 没有意义。只有不再需要该值, 并想将其转发给另一个函数时, 使用 *std::move()* 才有意义 (这里可以使用 *std::forward<>()*, 因为当使用 rvalue 引用时, 它等同于 *std::move()*)。

9.3.2 函数模板形参的 rvalue 引用

如果函数模板形参有非 *const/volatile* 的 rvalue 引用, 则可以传递所有值类别的对象。传递的实参可以是 *const*, 也可以不是:

```
1 template<typename Coll>
2 void foo(Coll&& arg) // arg is a universal/forwarding reference
3 {
4     Coll coll; // coll may be const
5     ...
6     bar(std::forward<Coll>(arg)); // perfectly forward to bar() (don't use std::move
7         () here)
8 }
9
10 std::vector<std::string> v;
11 const std::vector<std::string> c;
```

```

12 foo(v); // OK, arg binds to a non-const lvalue
13 foo(c); // OK, arg binds to a const lvalue
14 foo(Coll{}); // OK, arg binds to a non-const prvalue
15 foo(std::move(v)); // OK, arg binds to a non-const xvalue
16 foo(std::move(c)); // OK, arg binds to a const xvalue

```

foo() 中:

- *arg* 的类型现在可以是 *const*, 也可以不是。
- 这种情况下, 使用 *std::move()* 没有意义。只有不再需要该值, 并想要将其转发给另一个函数时, 才有必要使用 *std::forward<>()*(也可以使用 *std::move()*, 但会将所有参数传递给带有移动语义的 *foo()*, 将非 *const* lvalue *v* 作为 xvalue 传递)。

9.4 使用通用引用的重载

已经讨论了将对象绑定到普通引用时的规则。现在, 在引入通用引用之后, 必须扩展这些规则。

再次假设有一个类 X, 有一个非 *const* 变量 *v*, 有一个 *const* 变量 *c*:

```

1 class X {
2     ...
3 };
4
5 X v;
6 const X c{ ... };

```

如果提供函数 *f()* 的所有重载, 根据正式规则, 就是下面列出所有重载:

```

1 void f(const X&); // read-only access
2 void f(X&); // OUT parameter (usually long-living object)
3 void f(X&&); // can steal value (object usually about to die)
4 void f(const X&&); // contradicting semantic meaning
5 template<typename T>
6 void f(T&&); // to use perfect forwarding

```

表 9.1. 绑定所有引用规则

Call	<i>f(X&)</i>	<i>f(const X&)</i>	<i>f(X&&)</i>	<i>f(const X&&)</i>	<i>template<typename T> f(T&&)</i>
<i>f(v)</i>	1	3	no	no	2
<i>f(c)</i>	no	1	no	no	2
<i>f(X{})</i>	no	4	1	3	2
<i>f(move(v))</i>	no	4	1	3	2
<i>f(move(c))</i>	no	3	no	1	2

同样, 这些数字列出了重载解析的优先级(最小的数字具有最高的优先级), 以便确定重载时调用了哪个函数。

请注意, 通用引用总是次优选择。完美匹配总是优先, 但是需要转换类型(例如使其为 *const* 或将 rvalue 转换为 lvalue)是比为精确类型实例化函数模板更糟糕的匹配方式。

9.4.1 用通用引用修正重载解析

重载解析中，通用引用绑定比类型转换更好，这有一个非常糟糕的副作用：如果有一个接受单个通用引用的构造函数，那么这个匹配就要比以下方式更优

- 如果传递非 *const* 对象，则使用复制构造函数
- 如果传递 *const* 对象，则使用移动构造函数

因此，实现只有一个通用引用参数的构造函数时，必须谨慎。

思考下面的代码：

generic/universalconstructor.cpp

```
1 #include <iostream>
2
3 class X {
4 public:
5     X() = default;
6     X(const X&) {
7         std::cout << "copy constructor\n";
8     }
9     X(X&&) {
10        std::cout << "move constructor\n";
11    }
12
13     template<typename T>
14     X(T&&) {
15         std::cout << "universal constructor\n";
16     }
17 };
18 int main()
19 {
20     X xv;
21     const X xc;
22
23     X xcc{xc}; // OK: calls copy constructor
24     X xvc{xv}; // OOPS: calls universal constructor
25     X xvm{std::move(xv)}; // OK: calls move constructor
26     X xcm{std::move(xc)}; // OOPS: calls universal constructor
27 }
```

如注释中所示，该程序有以下输出：

```
copy constructor
universal constructor
move constructor
universal constructor
```

因此，最好避免实现将第一个形参声明为通用引用，并为任意类型的实参调用的泛型构造函数。

另一种选择是，传递的是类的类型（或可转换为），则以禁用构造函数的方式约束构造函数。其必须使用在复制或移动构造函数上才有效果。从 C++20 起，可以写成这样：

```
1 class X {
2     public:
3     ...
4     template<typename T>
5     requires (!std::is_same_v<std::remove_cvref_t<T>, X>)
6     X(T&&) {
7         std::cout << "universal constructor\n";
8     }
9 };
```

C++20 前，需要这样写：

```
1 class X {
2     public:
3     ...
4     template<typename T,
5              typename
6             = typename std::enable_if<!std::is_same<typename std::decay<T>::type,
7             X>::value
8             >::type>
9     X(T&&) {
10        std::cout << "universal constructor\n";
11    }
12 };
```

9.5 Lambda 中的完美转发

如果想要完美地转发 Lambda 的参数，必须使用通用引用和 `std::forward<>()`。但是，通常用 `auto&&` 声明通用引用。

C++20 中，也可以在 Lambda 中使用模板形参：

```
1 auto callFoo = []<typename T>(T&& arg) { // OK, universal reference since C++20
2     foo(std::forward<T>(arg)); // perfectly forward arg
3 };
```

9.6 总结

- 带有两个 & 符 (&&) 的声明可以是两个不同的类型：
 - 如果不是函数模板形参，它是一个普通的 rvalue 引用，只绑定到 rvalue。
 - 如果是函数模板参数，它是一个通用引用，可以绑定到所有值类别。
- 通用引用（在 C++ 标准中称为转发引用）是一种可以通用地引用任何类型和值类别对象的引用。类型是：
 - lvalue 引用（类型 &）：绑定到 lvalue

- rvalue 引用 (类型 `&&`): 绑定到 rvalue

- 要完美地传递传递的实参, 请使用 `std::forward<>()`, 并将该形参声明为函数模板形参的通用引用。
 - `std::forward<>()` 是一个条件 `std::move()`。如果参数是 rvalue, 则扩展为 `std::move()`。
 - 使用 `std::forward<>()` 标记对象可能是有意义的, 即使在调用成员函数时也是如此。
 - 通用引用是所有重载解析的次优选择。
 - 不要为通用引用实现泛型构造函数 (或为特定类型进行约束)。

10 完美转发的细节

前一章介绍了(完美)转发和通用/转发引用之后，本章会来介绍完美转发和通用/转发引用的细节，例如：

- 非转发的通用引用
- 通用引用
- 通用引用的类型推导

接下来的章节将讨论如何处理泛型代码中，可能有或没有移动语义的返回值。

10.1 通用引用作为非转发引用

有一些通用引用(也称为转发引用)的应用与转发无关，因为可以绑定所有对象，同时仍然知道值的类别和/或对象是否为 *const*。

本章中，将讨论一些例子。介绍了第二种通用引用 *auto&&* 之后，将讨论使用通用引用作为非转发引用的实际示例。

10.1.1 通用引用和 *const*

根据绑定引用的正式规则，通用引用是将引用绑定到任何值类别的对象，并需要保持其为 *const* 的唯一方法。另一个绑定到所有对象的引用是 *const&*，但丢失了传递参数是否为 *const* 的信息。

转发常量

如果想避免重载，但又希望对 *const* 和非 *const* 参数有不同的行为，并支持所有值类别，就必须使用通用引用。

考虑以下代码：

generic/universalconst.cpp

```
1 #include <iostream>
2 #include <string>
3
4 void iterate(std::string::iterator beg, std::string::iterator end)
5 {
6     std::cout << "do some non-const stuff with the passed range\n";
7 }
8
9 void iterate(std::string::const_iterator beg, std::string::const_iterator end)
10 {
11     std::cout << "do some const stuff with the passed range\n";
12 }
13
14 template<typename T>
15 void process(T&& coll)
16 {
17     iterate(coll.begin(), coll.end());
18 }
```

```

20 int main()
21 {
22     std::string v{"v"};
23     const std::string c{"c"};
24
25     process(v); // coll binds to a non-const lvalue, iterators passed
26     process(c); // coll binds to a const lvalue, const_iterators passed
27     process(std::string{"t"}); // coll binds to a non-const prvalue, iterators passed
28     process(std::move(v)); // coll binds to a non-const xvalue, iterators passed
29     process(std::move(c)); // coll binds to a const xvalue , const_iterators passed
30 }

```

该程序有以下输出:

```

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

```

代码中, 用一个形参声明 `process()` 作为对集合(容器、字符串等)的通用引用。与通用引用一样, 可以传递所有可能值类别的字符串。

`process()` 内部, 可以根据实参是否为 `const`, 有不同的行为。本例中, 调用 `begin()` 和 `end()` 来传递传递的集合, 将它们作为一个半开范围传递给遍历所有元素的函数:

```

1 template<typename T>
2 void process(T&& coll)
3 {
4     iterate(coll.begin(), coll.end());
5 }

```

但这里调用了 `iterate()` 函数的不同实现: 一个用于 `iterator`(能够修改元素), 另一个用于 `const_iterator`(不能修改元素)。通用引用保留了传递的 `const` 对象, 所以调用与传递集合匹配的 `iterate()`。

注意, `process()` 内部没有使用(完美)转发。只想通用引用 `const` 和非 `const` 对象, 甚至可以遍历所有元素后使用 `coll`。

您可能会认为, 应该知道在迭代元素时是否修改了它们。假设这个泛型函数允许为每个元素调用进行操作, 则可以读取或修改元素, 因此 `const` 的正确性非常重要。

注意, 这里使用 `std::forward<>()` 是有问题的:

```

1 template<typename T>
2 void process(T&& coll)
3 {
4     iterate(std::forward<T>(coll).begin(), std::forward<T>(coll).end()); // ???
5 }

```

在两个不同的位置，声明不再需要同一对象是一切麻烦的起源，因为两个位置都可能将此解释为从对象中窃取值的原因。因此，窃取 *last* 的位置可能没有获得正确的值。对于同一个对象，永远不要使用 *std::move()* 或 *std::forward<>()* 两次 (除非该对象在第二次使用之前重新初始化)。

这里只使用 *std::forward<>()* 一次也是麻烦，因为不保证函数调用参数的求值顺序：

```
1 template<typename T>
2 void process(T&& coll)
3 {
4     iterate(coll.begin(), std::forward<T>(coll).end()); // ???
5 }
```

这个特定的例子中，使用 *std::forward<>()* 一次或两次可能有效，因为 *begin()* 和 *end()* 不会从传递的对象中窃取/修改值。但是，除非确切地知道如何使用这些信息，否则就是错误。

10.1.2 通用引用的细节

前面的示例演示了将参数声明为通用引用比将其完全转发给另一个函数更有用。进一步分析一下。

考虑以下声明：

```
1 template<typename T>
2 void foo(T&& arg) {
3     ...
4 }
```

当声明 *arg* 时，有一个引用，可以统一地绑定到所有类型和值类别。对于非 *const* 对象 *v* 和 *const* 对象 *c*，类型 *T* 和参数类型推导如下：

	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	cosnt Type&&

关于传递的参数的信息拆分如下：

- *arg* 知道值及其类型，包括是否为 *const*。如果传递了 lvalue，则为 lvalue 引用；如果传递了 rvalue，则为 rvalue 引用。
- *T* 类型拥有关于传递参数的值类别的一些信息 (是传递 lvalue 还是 rvalue)。根据特定模板类型的推导规则，如果传递了一个 lvalue，*T* 就是一个 lvalue 引用；否则，*T* 不是一个引用。

调用 *std::forward<t>(arg)* 将所有信息再次集合在一起，以恢复常量和值类别，以完美地转发传递的参数及其当前值。但是，如果不需要传递参数的值类别来实现完美转发，则不需要 *std::forward<>()*。

依赖常量的代码

如果只需要知道传递的参数是否为 *const*，可以同时使用 *arg* 和 *t*：

```

1 template<typename T>
2 void foo(T&& arg)
{
3
4     if constexpr( std ::is_const_v<std ::remove_reference_t<T>>) {
5         ... // passed argument is const
6     }
7     else {
8         ... // passed argument is not const
9     }
10 }
```

这里，我们使用 `if constexpr` (C++17 引入的编译时 `if`) 根据传递的实参是否为 `const` 来执行不同的操作。

注意，在检查 `T` 的一致性之前，使用 `std::remove_reference<>` 来删除对 `T` 的引用是很重要的。`const` 类型的引用不是整体的 `const`:

```

1 std ::is_const_v<int> // false
2 std ::is_const_v<const int> // true
3 std ::is_const_v<const int&> // false
4 std ::is_const_v<std ::remove_reference_t<const int&>> // true
```

说些大家不知道的:

- `std::remove_reference_t<T>` (自 C++14 可用) 是 `std::remove_reference<T>::type` 的缩写。
- `std::is_const_v<T>` (自 C++17 可用) 是 `std::is_const<T>::value` 的缩写。

依赖值类别的代码

通过使用 `T`，可以获得/检查关于传递的值类别的信息 (至少是传递了 lvalue 还是 rvalue)。例如:

```

1 template<typename T>
2 void foo(T&& arg)
{
3
4     if constexpr( std ::is_lvalue_reference_t<T>) {
5         ... // passed argument is lvalue (has no move semantics)
6     }
7     else {
8         ... // passed argument is rvalue (has move semantics)
9     }
10 }
```

有时这样的检查是必要的 (例如，根据传递的是 lvalue 还是 rvalue 来不同地处理子对象)。

10.1.3 特定类型的通用引用

普通 rvalue 引用和通用引用共享相同的语法，这会导致了多个问题。不仅不能确定有什么，还不能声明特定类型的通用引用。

例如，声明为使用通用引用的函数:

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

希望将该函数限制为只接受字符串 (*const* 和非 *const* 都可以, 且不丢失信息), 但并非轻易就能做到。

因为 C++20, 用关键字 *require* 来约束对特定类型的通用引用成为可能。但是, 必须确定是否支持, 以及支持哪种类型的类型转换:

- 当类型必须匹配时 (不允许隐式转换):

```
1 template<typename T>
2 requires std::is_same_v<std::remove_cvref_t<T>, std::string>
3 void processString(T&&) {
4     ...
5 }
```

- 允许隐式转换:

```
1 template<typename T>
2 requires std::is_convertible_v<T, std::string>
3 void processString(T&&) {
4     ...
5 }
```

- 允许显式转换:

```
1 template<typename T>
2 requires std::is_constructible_v<std::string, T>
3 void processString(T&&) {
4     ...
5 }
```

通常, `std::is_convertible` 是期望的, 因为符合函数调用的通常规则。注意, `std::is_convertible` 和 `std::is_constructible` 的转换顺序与源类型和目标类型相反。

请参考 `generic/universaltypes.cpp` 以获得所有情况的完整示例。

C++20 之前, 需要 `enable_if<>` 类型特征, 而不是 `require`, 并且不支持类型特征带有 `_v` 和 `_t` 后缀的缩写形式。例如, 以下代码支持 (自 C++11 起) 隐式转换为 `std::string` 的所有类型:

```
1 template<typename T,
2 typename = typename std::enable_if<
3 std::is_convertible<T, std::string>::value
4 >::type>
5
6 void processString(T&& args);
```

要在 C++11 中限制为 `std::string` 类型, 我们需要:

```

1 template<typename T,
2 typename = typename std::enable_if<
3 std::is_same<typename std::decay<T>::type,
4             std::string
5             >::value
6             >::type>
7
8 void processString(T&& arg);

```

这里,类型 trait std::decay<> 用于从类型 T 中移除引用和一致性 (类型 trait std::remove_cvref<> 在 C++20 后可用)。

以类似的方式, 可以约束泛型代码, 让通用引用只绑定到 rvalue。但是, 如果有通用引用的特定语法, 那么这些都不需要。例如:

```

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

```

但是, 现在没有这种语法, 现在两个程序都使用的是 &&。

10.2 通用或普通 rvalue 引用?

对普通 rvalue 引用和通用/转发引用使用相同的语法, 这为对象是普通 rvalue 引用还是通用引用带来了一些有趣的情况。

10.2.1 泛型类型成员的 rvalue 引用

对模板形参的成员类型的 rvalue 引用不是通用引用。

例如:

```

1 template<typename T>
2 void foo(typename T::value_type&& arg); // not a universal reference

```

这里有一个完整的例子:

[generic/universalmem.cpp](#)

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 template<typename T>
6 void insert(T& coll, typename T::value_type&& arg)
7 {
8     coll.push_back(arg);
9 }
10
11 int main()
12 {
13     std::vector<std::string> coll;
14     ...
15     insert(coll, std::string{"prvalue"}); // OK

```

```

16 ...
17 std::string str{ "lvalue" };
18 insert(coll, str); // ERROR: T::value_type&& is not a universal reference
19 insert(coll, std::move(str)); // OK
20 ...
21 }

```

10.2.2 类模板中参数的 rvalue 引用

对类模板的模板形参的 rvalue 引用不是通用引用。

例如:

```

1 template<typename T>
2 class C {
3     T&& member; // member is not a universal reference
4     ...
5     void foo(T&& arg); // arg is not a universal reference
6 };

```

完整的例子:

[generic/universalclass.cpp](#)

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 template<typename T>
6 class Coll {
7 private:
8     std::vector<T> values;
9 public:
10    Coll() = default;
11
12    // function in class template:
13    void insert(T&& val) {
14        values.push_back(val);
15    }
16 };
17
18 int main()
19 {
20     Coll<std::string> coll;
21     ...
22     coll.insert(std::string{ "prvalue" }); // OK
23     std::string str{ "lvalue" };
24     coll.insert(str); // ERROR: && of Coll<T> is not a universal reference
25     coll.insert(std::move(str)); // OK
26     ...
27 }

```

通常，类模板中的函数不遵循函数模板规则。就是 *temploid*，泛型代码在实例化类时遵循普通函数的规则。

10.2.3 完全特化中参数的 rvalue 引用

对函数模板的完全特化的形参的 rvalue 引用不是通用引用。

例如：

```
1 template<typename T> // primary template
2 void foo(T&& arg); // - arg is a universal reference
3 ...
4 template<> // full specialization (for rvalues only)
5 void foo(std::string&& arg); // - arg is not a universal reference
```

对于 std::string 类型的 lvalue，仍然会调用主模板。要特化 std::string 的主模板的所有情况，必须提供第二个全特化版本：

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

完整的例子：

generic/universalspec.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 // primary template taking a universal reference:
6 template<typename Coll, typename T>
7 void insert(Coll& coll, T&& arg)
8 {
9     std::cout << "primary template for type T called\n";
10    coll.push_back(arg);
11 }
12
13 // full specialization for rvalues of type std::string:
14 template<>
15 void insert(std::vector<std::string>& coll, std::string&& arg)
16 {
17     std::cout << "full specialization for type std::string&& called\n";
18     coll.push_back(arg);
19 }
20
21 // full specialization for lvalues of type std::string:
22 template<>
23 void insert(std::vector<std::string>& coll, const std::string& arg)
24 {
25     std::cout << "full specialization for type const std::string& called\n";
26     coll.push_back(arg);
27 }
```

```

28
29 int main()
30 {
31     std::vector<std::string> coll;
32     ...
33     insert(coll, std::string{"prvalue"}); // calls full specialization for rvalues
34     std::string str{"lvalue"};
35     insert(coll, str); // calls full specialization for lvalues
36     insert(coll, std::move(str)); // calls full specialization for rvalues
37     ...
38 }
```

注意，必须在类定义之外声明/定义成员函数模板的全特化：

```

1 template<typename T>
2 class Cont {
3     ...
4     // primary template:
5     template<typename U>
6     void insert(U&& v) { // universal reference
7         coll.push_back(std::forward<U>(v));
8     }
9     ...
10 };
11
12 // full specializations for Cont<T>::insert<>():
13 // - have to be outside the class
14 // - need specializations for rvalues and lvalues
15
16 template<>
17     template<>
18 void Cont<std::string>::insert<>(std::string&& v)
19 {
20     coll.push_back(std::move(v));
21 }
22
23 template<>
24     template<>
25 void Cont<std::string>::insert<>(const std::string& v)
26 {
27     coll.push_back(v);
28 }
```

10.3 C++ 标准如何指定完美转发

为了理解完美转发的所有规则，来看看这些规则是如何在 C++ 标准中指定的。

同样，有以下声明：

```

1 template<typename T>
```

```

2 void f(T&& arg) // arg is universal/forwarding reference
3 {
4     g(std::forward<T>(arg)); // perfectly forward (move() only for passed rvalues)
5 }

```

通常, T 只有传递参数的类型:

```

1 MyType v;
2
3 f(MyType{}); // T is deduced as MyType, so arg is declared as MyType&&
4 f(std::move(v)); // T is deduced as MyType, so arg is declared as MyType&&

```

然而, 对于通用引用传递 lvalue, 有一个特殊的规则 (参见 C++ 标准的 [temp.debit.call] 节):

如果形参类型是对 cv-非限定模板形参的右值引用, 实参是 lvalue, 类型“对 T 的 lvalue 引用”将用来代替 T 进行类型推断。

这意味着在这种情况下:

- 如果形参的类型是用 `&&` 声明的, 而不是用 `const` 或 `volatile` 声明的
- 并且传递的是一个 lvalue
- 那么将 T 推导为 `T&`。

例子:

```

1 template<typename T>
2 void f(T&& arg); // arg is a universal/forwarding reference
3
4 MyType v;
5 const MyType c;
6
7 f(v); // T is deduced as MyType&
8 f(c); // T is deduced as const MyType&

```

但已经将 `arg` 声明为 `T&&`。如果 T 是 `T&`, 那这里 C++ 的引用折叠规则 (参见 [dcl.] 一节。参考] 的 C++ 标准) 给出了一个答案:

- `Type& &` 成为 `Type&`
- `Type& &&` 成为 `Type&`
- `Type&& &` 成为 `Type&`
- `Type&& &&` 成为 `Type&&`

这意味着:

```

1 MyType v;
2 const MyType c;
3
4 f(v); // T is deduced as MyType& and arg has this type
5 f(c); // T is deduced as const MyType& and arg has this type

```

现在考虑一下 `std::forward<>()` 是如何定义的, 与 `std::move()` 相反:

- `std::move()` 总是将类型转换为 rvalue 引用:

```
1 static_cast<remove_reference_t<T>&&>(t)  
2
```

它删除引用并转换为相应的 rvalue 引用类型 (删除任何 & 并添加 &&)。

- `std::forward<>()` 只向传递的类型参数添加 rvalue 引用:

```
1 static_cast<T&&>(t)  
2
```

引用折叠规则再次适用:

- 如果类型 T 是 lvalue 引用, `T&&` 仍然是 lvalue 引用 (`&&` 无效)。因此, 将 `arg` 强制转换为 lvalue 引用, 这意味着 `arg` 没有移动语义。
- 但是, 如果 T 是 rvalue 引用 (或者根本不是引用), `T&&(仍然)` 是 rvalue 引用。因此, 将 `arg` 强制转换为 rvalue 引用, 这样就将值类别更改为 xvalue, 这是 `std::move()` 的效果。

因此:

```
1 template<typename T>  
2 void f(T&& arg) // arg is a universal/forwarding reference  
3 {  
4     g(std::forward<T>(arg)); // perfectly forward (move() only for passed rvalues)  
5 }  
6  
7 MyType v;  
8 const MyType c;  
9  
10 f(v); // T and arg are MyType&, forward() has no effect in this case  
11 f(c); // T and arg are const MyType&, forward() has no effect in this case  
12 f(MyType{}); // T is MyType, arg is MyType&&, forward() is equivalent to move()  
13 f(std::move(v)); // T is MyType, arg is MyType&&, forward() is equivalent to move()
```

注意, 字符串字面值是 rvalue, 因此我们可以推导出的 T 和 `arg`:

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

还请记住, 对函数的引用总是 lvalue, 因此, 如果对函数的引用传递给通用引用, 那么 T 总是推断为 lvalue 引用:

```
1 void func(int) {  
2 }  
3 f(func); // lvalue passed to f(), so T and arg have type void(&)(int)  
4 f(std::move(func)); // lvalue passed to f(), so T and arg have type void(&)(int)
```

10.3.1 通用引用类型的说明

声明通用/转发引用时，还可以显式指定模板形参的类型，而不是推导。但是，请记住参数声明为 `T&&`。因此，有以下行为：

```
1 template<typename T>
2 void f(T&& arg) // arg is universal/forwarding reference
3 {
4     g(std::forward<T>(arg)); // perfectly forward (move() only for passed rvalues)
5 }
6
7 f<std::string>(...); // arg is a raw rvalue reference binding to rvalues only
8 f<std::string&>(...); // arg is an lvalue reference binding to non-const lvalues
9     only
10 f<const std::string&>(...); // arg is a const lvalue reference binding to
11     everything
12 f<std::string&&>(...); // arg is a raw rvalue reference binding to rvalues only
```

因此，有了明确的规范，通用引用不再作为通用引用。作为调用者，可以指定获得的引用的具体类型。

因此，要传递 lvalue(这里仍然需要值)，请确保将模板参数指定为 lvalue 引用。否则，代码将无法编译：

```
1 template<typename T>
2 void f(T&& arg) // arg is universal/forwarding reference
3 {
4     g(std::forward<T>(arg)); // perfectly forward (move() only for passed rvalues)
5 }
6
7 std::string s;
8 ...
9 f<std::string>(s); // ERROR: cannot bind rvalue reference to lvalue
10 f<std::string&>(s); // OK, does not move and forward s
11 f<std::string>(std::move(s)); // OK, does move and forward s
12 f<std::string&&>(std::move(s)); // OK, does move and forward s
```

最后两个调用是等价的。

这些规则同样适用于使用 C++20 特性，在声明普通函数时使用 `auto&&`：

```
1 void f(auto&& arg) {
2     g(std::forward<decltype(arg)>(arg));
3 }
```

10.3.2 与通用引用冲突的模板参数推断

推导通用引用模板参数的特殊规则（当传递 lvalue 时，将类型作为 lvalue 引用进行推导）可能会导致看似正确的代码出现意外错误。

以下代码无法编译时，开发者通常会感到惊讶：

```
1 template<typename T>
2 void insert(std::vector<T>& vec, T&& elem)
```

```

3 {
4     vec.push_back(std::forward<T>(elem));
5 }
6
7 std::vector<std::string> coll;
8 std::string s;
9 ...
10 insert(coll, s); // ERROR: no matching function call

```

问题是两个参数都可以推导出参数 T，但推导出的类型不一样：

- 使用参数 *coll*, T 将其推导为 std::string。
- 但是，根据通用引用的特殊规则，参数 *elem* 会强制将 T 推导为 std::string&。

因此，编译器会产生歧义错误。

有两种方法可以解决这个问题：

- 可以使用 std::remove_reference<>:

```

1 template<typename T>
2 void insert(std::vector<std::remove_reference_t<T>&> &vec, T&& elem)
3 {
4     vec.push_back(std::forward<T>(elem));
5 }
6 std::vector<std::string> coll;
7 std::string s;
8 ...
9 insert(coll, s); // OK, with T deduced as std::string& vec now binds to coll
10

```

- 可以使用两个模板参数:

```

1 template<typename T1, typename T2>
2 void insert(std::vector<T1>& vec, T2&& elem)
3 {
4     vec.push_back(std::forward<T2>(elem));
5 }
6

```

或者只是：

```

1 template<typename Coll, typename T>
2 void insert(Coll& coll, T&& elem)
3 {
4     coll.push_back(std::forward<T>(elem));
5 }
6

```

10.3.3 泛型类型的纯 rvalue 引用

通过导出通用引用的模板形参的特殊规则(当传递左值时,将类型作为 lvalue 引用),可以约束泛型引用形参仅绑定到 rvalue:

```
1 template<typename T>
2 requires (!std::is_lvalue_reference_v<T>) // bind to rvalues only
3 void callFoo(T&& arg) {
4     foo(std::forward<T>(arg));
5 }
```

C++20 之前,必须对类型特征再次使用 std::enable_if<>:

```
1 template<typename T,
2         typename
3             = typename std::enable_if<!std::is_lvalue_reference<T>::value
4             >::type>
5 void callFoo(T&& arg) {
6     foo(std::forward<T>(arg));
7 }
```

10.4 完美转发(不美丽的)细节

每当像移动语义这样新的复杂的东西出现,人们就会犯错。犯错是人的天性,在很长一段时间内,我们无法避免犯错,因为不知道未来所有的移动语义的应用和扩展。C++ 标准中,也对一些关于完美转发和通用引用的错误用法进行了讨论。

10.4.1 “通用引用”与“转发引用”

如前所述,对于可以引用任何值类别的引用,有两个不同的术语:通用引用和转发引用。在 C++11 中,标准没有引入任何通用引用的特殊术语,只是简单地介绍了函数模板形参的 rvalue 引用的特殊规则。

2012 年,当描述这些引用的行为时,Scott Meyers(C++ 社区的主要作者之一)引入了通用引用。为了与描述引用绑定到什么的术语右值引用和 rvalue 引用保持一致,目的是将通用引用作为一种描述,这种引用可以绑定所有类别。

不幸的是,几年后,C++ 标准委员会决定在 C++17 中引入一个不同的术语:转发引用。官方的理由是,这些引用不是“通用的”,因为它们不能绑定到所有类别,它们的主要目的是完美地转发参数(见 <http://wg21.link/n4164>)。

虽然转发是这些引用的用例,但转发不是唯一的用例。另一个非常重要的用例是,将引用普遍地绑定到任何对象,以保存关于其值类别和/或是否为 *const* 的信息。例如:

- 可能必须使用同一个对象(不管是什对象)两次,才能不丢失该对象是否为 *const* 的信息。典型的例子是为传递的集合调用 *begin()* 和(正如我们之前看到的那样,稍后还将在基于范围的 for 循环的实现中使用)。
- 可能需要引用任何值类别的对象,而不将其声明为 *const*(稍后给出一个使用通用引用调用基于范围的 for 循环的例子)。

问题是,为什么 C++ 标准委员会不采纳已经在 C++ 社区建立的术语?争论焦点是这个术语

会令人困惑，但 C++ 标准中有更多令人困惑的术语，现在有两个术语表示同一件事，这种困惑肯定更糟糕。

无论如何，这是 C++ 标准委员会的慎重决定，提出了一个不同的术语。毫无疑问，通用引用已经足够 (C++ 标准中有更糟糕的术语)；然而，对于 C++ 标准委员会的主流群体来说，不采用这个术语而使用另一个或更好的术语就更加重要。

我仍然倾向于使用通用引用 (本书至少是一致的，本书常使用自己的术语，无论如何那只对专家有效)。然而，为了澄清，我不得不使用通用引用/转发引用来处理这种混乱的情况。

对于读者来说，这意味着无论何时听到“转发引用”这个术语，都必须将其转换为通用引用 (反之亦然)。如果有疑问，使用通用引用/转发引用。

10.4.2 为什么普通 rvalue 和通用引用都用 `&&` ?

通用/转发引用使用与普通 rvalue 引用相同的语法，这是麻烦的来源。那么，为什么不为通用引用引入一种特定的语法呢？

例如，另一种建议可能是 (有时讨论为一种修正)：

- 对普通的 rvalue 引用使用两个 & 号：

```
1 void foo( Coll&& arg ) // arg is an ordinary rvalue reference  
2
```

- 使用三个 & 符号为通用引用：

```
1 template<typename Coll>  
2 void foo( Coll&&& arg ) // arg is universal/forwarding reference  
3
```

然而，这三个 & 符号可能看起来太可笑了 (每当我显示这个选项时，人们都会笑)。不幸的是，使用三个 & 号会更好，因为这会使代码更直观。

如前所述，当约束对具体类型的通用引用时，只需声明即可

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

而不是

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

甚至：

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

这里有一个重要的教训：有一个可笑但清晰语法比有一个很酷但令人困惑的混乱术语要好。

10.5 总结

- 可以使用通用引用来绑定到所有 *const* 和非 *const* 对象，而不会丢失该对象的 *const* 信息。
- 即使不使用 `std::forward<>()`，也可以使用通用引用来实现对传递的参数的特殊处理。
- 要拥有特定类型的通用引用，需要使用 *concepts/requirements*(自 C++20 起) 或一些模板技巧(到 C++17 为止)。
- 只有函数模板形参的 rvalue 引用是通用引用。类模板形参的 rvalue 引用、模板形参的成员以及全特化都是普通的 rvalue 引用，只能绑定到 rvalue。
- 当显式指定通用引用的类型时，不再作为通用引用。而是使用类型 `&` 来传递 lvalue。
- C++ 标准委员会将转发引用作为通用引用的“更好”术语。不幸的是，术语转发引用限制了通用引用对特定用例的用途，并造成了对同一事物使用两个术语的不必要混淆。因此，使用通用引用/转发引用可以避免更多的混淆。

11 用 auto&& 完美传递

讨论了泛型代码中移动语义、完美转发参数后，现在讨论一下如何完美处理返回值。本章中，将讨论返回值的完美传递。为此，引入 auto&& 作为通用引用（同样也称为转发引用）的另一种方式。但也会讨论 auto&& 与转发值无关的部分。

下一章将讨论如何完美地返回值。

11.1 默认的完美的传递

我们经常需要将返回值传递给另一个函数：

```
1 // pass return value of compute() to process():
2 process(compute(t)); // OK, uses perfect forwarding of returned value
```

非泛型代码中，需要知道所涉及的类型。然而，泛型代码中，也希望 *compute()* 的返回值完全传递给 *process()*。

好消息是：如果直接将返回值传递给另一个函数，该值会完美传递，保持其类型和值类别。不必担心移动语义（如果支持将自动使用）。

11.1.1 默认完美传递的细节

完整的例子：

generic/perfectpassing.cpp

```
1 #include <iostream>
2 #include <string>
3
4 void process(const std::string&) {
5     std::cout << "process(const std::string&)\n";
6 }
7 void process(std::string&) {
8     std::cout << "process(std::string&)\n";
9 }
10 void process(std::string&&) {
11     std::cout << "process(std::string&&)\n";
12 }
13
14 const std::string& computeConstLRef(const std::string& str) {
15     return str;
16 }
17 std::string& computeLRef(std::string& str) {
18     return str;
19 }
20 std::string&& computeRRef(std::string&& str) {
21     return std::move(str);
22 }
23 std::string computeValue(const std::string& str) {
24     return str;
25 }
```

```

26
27 int main()
28 {
29     process(computeConstLRef("tmp")); // calls process(const std::string&)
30
31     std::string str{"lvalue"};
32     process(computeLRef(str)); // calls process(std::string&)
33
34     process(computeRRef("tmp")); // calls process(std::string&&)
35     process(computeRRef(std::move(str))); // calls process(std::string&&)
36
37     process(computeValue("tmp")); // calls process(std::string&&)
38 }
```

- 如果 *compute()* 返回一个 const lvalue 引用:

```

1 const std::string& computeConstLRef(const std::string& str) {
2     return str;
3 }
4
```

返回值的值类别是 lvalue, 这意味着返回值会完美转发, 并与 *const lvalue* 引用匹配:

```

1 process(computeConstLRef("tmp")); // calls process(const std::string&)
2
```

- 如果 *compute()* 返回一个非 *const lvalue* 引用:

```

1 std::string& computeLRef(std::string& str) {
2     return str;
3 }
4
```

返回值的值类别是 lvalue, 这意味着返回值会完全转发, 并与非 *const lvalue* 引用的最佳匹配:

```

1 std::string str{"lvalue"};
2 process(computeLRef(str)); // calls process(std::string&)
3
```

- 如果 *compute()* 返回 rvalue 引用:

```

1 std::string&& computeRRef(std::string&& str) {
2     return std::move(str);
3 }
4
```

返回值的值类别是 xvalue, 这意味着返回值会完全转发为 rvalue 引用, 允许 *process()* 窃取值:

```

1 process(computeRRef("tmp")); // calls process(std::string&&)
2 process(computeRRef(std::move(str))); // calls process(std::string&&)
3
```

- 如果 `compute()` 按值返回临时对象:

```

1 std::string computeValue(const std::string& str) {
2     return str;
3 }
4

```

返回值的值类别是 prvalue，返回值完全转发为 rvalue 引用，也允许 `process()` 窃取值:

```

1 process(computeValue("tmp")); // calls process(std::string&&)
2

```

注意，通过返回 `const` 值:

```

1 const std::string computeConstValue(const std::string& str) {
2     return str;
3 }

```

或 `const` rvalue 引用:

```

1 const std::string&& computeConstRRef(std::string&& str) {
2     return std::move(str);
3 }

```

再次禁用移动语义:

```

1 process(computeConstValue("tmp")); // calls process(const std::string&)
2 process(computeConstRRef("tmp")); // calls process(const std::string&)

```

如果有 `const&&` 的声明，可以接受这样的重载。

因此: 不要将 value 返回的值标记为 `const`，也不要将返回的非 `const` rvalue 引用标记为 `const`。

11.2 使用 `auto&&` 的通用引用

但在泛型代码中，如何在传递返回值的同时，仍然保持其类型和值类别？

答案是通用/转发引用，但没有声明为参数。为此，需要 `auto&&`。

调用时:

```

1 // pass return value of compute() to process():
2 process(compute(t)); // OK, uses perfect forwarding of returned value

```

还可以实现以下功能:

```

1 // pass return value of compute() to process() with some delay:
2 auto&& ret = compute(t); // initialize a universal reference with the return value
3 ...
4 process(std::forward<decltype(ret)>(ret)); // OK, uses perfect forwarding of
                                                 returned value

```

或者，使用大括号初始化时:

```

1 // pass return value of compute() to process() with some delay:
2 auto&& ret{compute(t)}; // initialize a universal reference with the return value
3 ...
4 process(std::forward<decltype(ret)>(ret)); // OK, uses perfect forwarding of
      returned value

```

请参阅 generic/perfectautorefref.cpp 以获得所有情况下的完整示例。

11.2.1 auto&& 的类型定义

使用 auto&& 声明时，也声明了一个通用引用。定义一个绑定到所有值类别的引用，该引用的类型保留其初始值的类型和值类别。

如果声明

```

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

```

类型 ref 是根据函数模板参数的通用引用类型推导出来的:

```

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

```

根据规则，ref(即 auto&& 或 T&& 类型) 的类型是

- 如果引用 lvalue，则为 lvalue 引用
- 如果引用 rvalue，则为 rvalue 引用

例如:

```

1 // forward declarations:
2 std::string retByValue();
3 std::string& retByRef();
4 std::string&& retByRefRef();
5 const std::string& retByConstRef();
6 const std::string&& retByConstRefRef();
7
8 // deduced auto&& types:
9 std::string s;
10 auto&& r1{s}; // std::string&
11 auto&& r2{std::move(s)}; // std::string&&
12
13 auto&& r3{retByValue()}; // std::string&&
14 auto&& r4{retByRef()}; // std::string&
15 auto&& r5{retByRefRef()}; // std::string&&
16 auto&& r6{retByConstRef()}; // const std::string&
17 auto&& r7{retByConstRefRef()}; // const std::string&&

```

当使用 rvalue(prvalue 或 xvalue) 初始化用 auto&& 声明的引用时，就声明了一个 rvalue 引用。例如，将引用绑定到标记为 `std::move()` 的对象或返回的普通值或 rvalue 引用时，就会出现这种情况。

但是，当使用 lvalue 初始化用 auto&& 声明的引用时，就声明了一个 lvalue 引用。例如，将引用绑定到已命名的对象或返回 lvalue 引用的函数的返回值时，就会出现这种情况。

因为字符串字面值是 lvalue(以字符数组为类型)，所以将通用引用绑定到字符串字面值时，也会得到 lvalue 引用：

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

因为对函数的引用总是 lvalue，所以将通用引用绑定到函数时，也会得到 lvalue 引用：

```
1 std :: string foo( int ); // forward declaration
2
3 auto&& r9{ foo }; // lvalue of type std :: string(&)( int )
```

11.2.2 完美转发 auto&& 引用

同样，也可以完美地将传递给通用引用的值作为函数模板形参：

```
1 template<typename T>
2 void callFoo( T&& ref ) {
3     foo( std :: forward<T>( ref )); // becomes foo( std :: move( ref )) for passed rvalues
4 }
```

完全可以转发 auto&& 的通用引用：

```
1 auto&& ref{ ... };
2
3 foo( std :: forward<decltype( ref )>( ref )); // becomes foo( std :: move( ref )) for rvalues
```

表达式 `std::forward<decltype(ref)>(ref)` 是这样实现的：

- 如果传递的 `decltype(ref)` 类型是一个 lvalue 引用，如果 `ref` 是用返回的 lvalue 引用初始化的，则表达式将 `ref` 强制转换为 lvalue 引用，这意味着 `ref` 传递时没有移动语义。
- 如果传递的 `decltype(ref)` 类型是 rvalue 引用，如果 `ref` 是用返回的普通值或 rvalue 引用初始化的，则表达式将 `ref` 转换为 rvalue 引用，这是 `std::move(ref)` 的效果。

因此，如果用函数的返回值初始化通用引用：

```
1 auto&& ret{ compute( t ) }; // initialize a universal reference with the return value
```

表达式为

```
1 process( std :: forward<decltype( ret )>( ret )); // perfectly forward the return value
```

当 `compute()` 返回 rvalue(如临时对象或 rvalue 引用) 时，扩展为

```
1 process( std :: move( ret ));
```

11.3 auto&& 的非转发引用

再次注意，通用引用是将引用绑定到任何类型和值类别的任何对象，并且仍然保留其值类别和是否为 `const` 的唯一方法。这也适用于用 auto&& 声明的通用引用。

11.3.1 通用引用和基于范围的 for 循环

使用基于范围的 for 循环时，使用 auto&& 声明的非转发通用引用起着重要的作用。

基于范围的循环规范

在 C++ 标准中，基于范围的 for 循环可以指定使用普通 for 循环遍历范围内的元素。

类似这样的调用：

```
1 std::vector<std::string> coll;
2 ...
3 for (const auto& s : coll) {
4     ...
5 }
```

相当于：

```
1 std::vector<std::string> coll;
2 ...
3 auto&& range = coll; // initialize a universal reference
4 auto pos = range.begin(); // to use the given range coll here
5 auto end = range.end(); // and here
6 for ( ; pos != end; ++pos ) {
7     const auto& s = *pos;
8     ...
9 }
```

将 range 声明为一个通用引用，希望能够将其绑定到每个 range，所以可以使用它两次 (一次是开始，一次是结束) 没有创建副本或丢失信息。

循环应该为：

- 非 *const* lvalue:

```
1 std::vector<int> coll;
2 ...
3 for (int& i : coll) {
4     i *= 2;
5 }
```

- *const* lvalue:

```
1 const std::vector<int> coll{0, 8, 15};
2 ...
3 for (int i : coll) {
4     ...
5 }
```

- prvalue:

```
1  for (int i : std::vector<int>{0, 8, 15}) {  
2      ...  
3  }  
4
```

注意，对于这些情况，没有其他方法声明 range:

- 使用 auto，将创建 range 的副本 (这需要花费时间并禁用对元素的修改)。
- 使用 auto&，可以用临时的 prvalue 禁用 range 的初始化。
- 使用 const auto&，将失去所遍历 range 的非常量性。

请注意，现在指定的基于范围的 for 循环有一个问题。代码如以下:

```
1  std::vector<std::string> createStrings();  
2  ...  
3  for (char c : createStrings().at(0)) { // fatal runtime error  
4      ...  
5 }
```

成为:

```
1  std::vector<std::string> createStrings();  
2  ...  
3  auto&& range = createStrings().at(0); // OOPS: universal reference to reference  
4  auto pos = range.begin(); // return value of createStrings() destroyed here  
5  auto end = range.end();  
6  for ( ; pos != end; ++pos ) {  
7      char c = *pos;  
8      ...  
9 }
```

所有有效引用都延长了所绑定值的生命周期，这也适用于 rvalue 引用。但是，没有绑定到 *createString()* 的返回值 (这样可以正常工作)；而是绑定到引用，该引用指向由 *at()* 返回的 *createStrings()* 的返回类型，扩展了引用的生命周期。因此，该循环将遍历已经销毁的字符串。

使用基于范围的 for 循环

即使在调用基于范围的 for 循环时，通用引用也有意义。

要在迭代时修改元素，必须使用非 *const* 引用。考虑函数模板，将传递值赋给传递集合中的所有元素:

```
1 template<typename Coll, typename T>  
2 void assign(Coll& coll, const T& value) {  
3     for (auto& elem : coll) {  
4         elem = value;  
5     }  
6 }
```

看起来适用于所有容器类型和元素类型 (其中支持赋值):

```

1 std::vector<int> coll1{0, 8, 15};
2 ...
3 assign(coll1, 42); // OK
4
5 std::vector<std::string> coll2{"hello", "world"};
6 ...
7 assign(coll2, "ok"); // OK

```

然而，有种情况行不通：

```

1 std::vector<bool> collB{false, true, false};
2 ...
3 assign(collB, true); // ERROR: cannot bind non-const lvalue reference to an rvalue

```

发生了什么事？看一下基于范围的 for 循环展开的代码：

```

1 std::vector<bool> coll{false, true, false};
2 ...
3 {
4     auto&& range = coll; // OK: universal reference to reference
5     auto pos = range.begin(); // OK
6     auto end = range.end(); // OK
7     for ( ; pos != end; ++pos ) { // OK
8         auto& elem = *pos; // ERROR: cannot bind non-const lvalue reference to an rvalue
9         elem = elem + elem;
10    }
11 }

```

问题是 `std::vector<bool>` 中的元素不是 `bool` 类型的对象，而是单个比特位。实现方法是：对于 `std::vector<bool>`，元素引用的类型不是元素类型的引用。`std::vector<bool>` 的实现是主模板 `std::vector<T>` 实现的偏特化，其中元素的引用是代理类的对象，可以像引用一样使用：

```

1 namespace std {
2     template< ... >
3     class vector<bool, ... > {
4         public:
5             ...
6         class reference {
7             ...
8         };
9         ...
10    };
11 }

```

当对迭代器解引用时，返回 `std::vector<bool>::reference` 的值。因此，在基于范围的 for 循环的扩展代码中的语句为

```

1 auto& elem = *pos;

```

尝试将非 `const` lvalue 引用绑定到临时对象 (prvalue)，这是不允许的。

然而，这个问题有一个解决方案：调用基于范围的 for 循环时使用通用引用：

```

1 template<typename Coll, typename T>
2 void assign(Coll&& coll, const T& value) {
3     for (auto&& elem : coll) { // note: universal reference support proxy element
4         elem = value;
5     }
6 }
```

因为通用引用可以绑定到任何对象 (甚至是 prvalue)，所以 `vector<bool>` 的代码现在可以编译：

```

1 std::vector<bool> collB{false, true, false};
2 ...
3 assign(collB, true); // OK (universal reference used to bind to an element)
```

因此，找到了使用非转发通用引用的另一个原因：绑定到没有作为引用实现的引用类型。或者说：允许绑定作为代理类型提供的非 `const` 对象来进行操作。

11.4 完美的 Lambda 转发

如果想要完美地转发通用 Lambda 的参数，必须使用通用引用和 `std::forward<>()`。但现在只需使用 `auto&&` 来声明通用引用。

已经有了一个函数模板：

```

1 auto callFoo = [] (auto&& arg) { // arg is a universal/forwarding reference
2     foo(std::forward<decltype(arg)>(arg)); // perfectly forward arg
3 };
4
5 std::string s{"BLM"};
6 callFoo(s); // OK, arg is std::string&
7 callFoo(std::move(s)); // OK, arg is std::string&&
```

在 C++20 中，可以通过用模板形参声明 Lambda 来避免使用 `decltype(arg)`。

下面的泛型 Lambda 使用此函数，完美地转发了可变数量的参数：

```

1 [] (auto&&... args) {
2     ...
3     foo(std::forward<decltype(args)>(args)...);
4 };
```

请记住 Lambda 只是定义函数对象的一种简单方法 (定义了 `operator()` 的对象允许其作为函数使用)。上面的定义扩展为编译器定义的类 (闭包类型)，并将通用引用定义为模板形参：

```

1 class NameDefinedByCompiler {
2     ...
3     public:
4         template<typename ... Args>
5         auto operator() (Args&&... args) const {
6             ...
```

```
7     foo(std::forward<decltype(args)>(args)...);  
8 }  
9 };
```

再次注意，用 *const*(或 *volatile*) 限定的泛型 rvalue 引用不是通用引用，这也适用于 Lambda。如果形参用 *const auto&&* 声明，只能传递 rvalue:

```
1 auto callFoo = [](const auto&& arg) { // arg is not a universal reference  
2 ...  
3 };  
4  
5 const std::string cs{"BLM"};  
6 callFoo(cs); // ERROR: s is not an rvalue  
7 callFoo(std::move(cs)); // OK, arg is const std::string&&
```

Lambda 内部，使用 *std::move()* 完美传递传递参数就可以了。

11.5 C++20 中函数声明使用 *auto&&*

因为 C++20，也可以用 *auto&&* 来声明普通函数，处理方式一样：用通用引用来声明函数模板。

如下定义：

```
1 void callFoo(auto&& val) {  
2     foo(std::forward<decltype(arg)>(arg));  
3 }
```

相当于：

```
1 template<typename T>  
2 void callFoo(T&& val) {  
3     foo(std::forward<decltype(arg)>(arg));  
4 }
```

还可以在这里显式地指定参数的类型。然而，与普通函数模板一样，必须用适合所传递参数的值类别的类型来限定模板形参：

```
1 std::string s;  
2 ...  
3 callFoo<std::string>(s); // ERROR: cannot bind rvalue reference to lvalue  
4 callFoo<std::string&>(s); // OK, does not move and forward s  
5 callFoo<std::string>(std::move(s)); // OK, does move and forward s  
6 callFoo<std::string&&>(std::move(s)); // OK, does move and forward s
```

11.6 总结

- 不要使用 *const* 返回值 (否则将禁用返回值的移动语义)。
- 不要将返回的非 *const* rvalue 引用标记为 *const*。
- auto&&* 可用于声明不是参数的通用引用。就像任何通用引用一样，它可以引用任何类型和值类别的所有对象，其类型是

- 如果绑定到 lvalue，则为 lvalue 引用 (类型 &)
- 如果绑定到 rvalue，则为 rvalue 引用 (类型 &&)
- 使用 `std::forward<decltype(ref)>(ref)` 完美地转发用 `auto&&` 声明的通用引用 `ref`。
- 可以使用通用引用同时引用 `const` 和非 `const` 对象，并多次使用而不会丢失 `const` 信息。
- 可以使用通用引用来绑定到代理类型的引用。
- 考虑在迭代集合元素以修改元素的泛型代码中使用 `auto&&`。这样，代码就可以用于代理类型的引用。
- 声明 Lambda(或函数，因为 C++20) 的形参时使用 `auto&&` 是在函数模板中声明作为通用引用形参的方式。

12 完美返回 decltype(auto)

讨论了参数的完美转发和返回值的完美传递之后，现在来讨论完美返回。本章介绍了新的占位符类型 decltype(auto)。还会看到一些令人惊讶的结果，比如：建议不必要的括号不要出现在 return 语句中。

12.1 完美返回

泛型代码中，经常会计算一个值，然后返回给调用者。问题是，如何完美地返回值，但仍然保留类型和值类别？换句话说：应该如何声明以下函数的返回类型：

```
1 template<typename T>
2 ??? callFoo (T&& arg)
3 {
4     return foo (std :: forward<T>(arg));
5 }
```

这个函数中，调用了名为 *foo()* 的函数，形参是完美转发的 *arg*。不知道这种类型的 *foo()* 返回什么；可能是临时值 (prvalue)、lvalue 引用或 rvalue 引用。返回类型可以是 *const* 或非 *const*。

那么，如何完美地将 *foo()* 的返回值返回给 *callFoo()* 的调用者呢？先说有几个种不起作用的方式：

- 返回类型 *auto* 将删除 *foo()* 返回类型的引用。例如，如果提供对容器元素的访问权限（将 *foo()* 视为 *at()* 成员函数或 *vector* 的索引操作符），*callFoo()* 将不再提供对该元素的访问权限。此外，可能会创建不必要的副本（如果没有优化掉的话）。
- 任何作为引用的返回类型 (*auto&*, *const auto&*, 和 *auto&&*) 将返回对局部对象的引用，如果 *foo()* 按值返回一个临时对象。幸运的是，编译器在检测到此类 bug 时会发出警告。

也就是说，需要一种表示方式：

- 如果有一个值，则按值返回
- 如果得到/有一个引用，则按引用返回

但仍然保留返回的类型和值类别。

C++14 为此引入了一个新的占位符类型:decltype(auto)。

```
1 template<typename T>
2 decltype(auto) callFoo (T&& arg) // since C++14
3 {
4     return foo (std :: forward<T>(arg));
5 }
```

有了这个声明，如果 *foo()* 按值返回，*callFoo()* 就能按值返回；如果 *foo()* 按引用返回，*callFoo()* 按引用返回，类型和值类别都可以保留。

12.2 decltype(auto)

就像其他占位符类型 *auto* 一样，*decltype(auto)* 是一个占位符类型，编译器在初始化时会推断出类型。然而，该类型是根据 *decltype* 的规则推导出来的：

- 如果用普通名称初始化或返回普通名称，则返回类型是具有该名称的对象的类型。
- 如果使用表达式初始化或返回表达式，则返回类型为求值表达式的类型和值类别：
 - 对于 **prvalue**, 只产生值类型:type
 - 对于 **lvalue**, 将其类型作为 lvalue 引用:type&
 - 对于 **xvalue**, 将其类型作为 rvalue 引用:type&&

例如:

```

1 std :: string s = "hello";
2 std :: string& r = s;
3
4 // initialized with name:
5 decltype(auto) da1 = s; // std::string
6 decltype(auto) da2(s); // same
7 decltype(auto) da3{s}; // same
8 decltype(auto) da4 = r; // std::string&
9
10 // initialized with expression:
11 decltype(auto) da5 = std :: move(s); // std::string&&
12 decltype(auto) da6 = s+s; // std::string
13 decltype(auto) da7 = s[0]; // char&
14 decltype(auto) da8 = (s); // std::string&

```

对于表达式，根据规则，类型推导如下：

- 因为 `std::move(s)` 是一个 xvalue，所以 `da5` 是 rvalue 引用。
- 因为字符串的加法操作符按值返回新的临时字符串（因此它是 prvalue），所以 `da6` 是普通值类型。
- 因为 `s[0]` 返回对第一个字符的 lvalue 引用，所以它是 lvalue，并强制 `da7` 也是 lvalue 引用。
- 因为 `(s)` 是 lvalue，所以 `da8` 是 lvalue 引用。是的，这里会因为括号有所不同。

与总是引用的 `auto&&` 相反，`decltype(auto)` 有时只是一个值（如果用值类型对象的名称或用 `prvalue` 表达式初始化）。

注意 `decltype(auto)` 不能有其他限定符：

```

1 decltype(auto) da{s}; // OK
2 const decltype(auto)& da1{s}; // ERROR
3 decltype(auto)* da2{&s}; // ERROR

```

12.2.1 返回类型的 `decltype(auto)`

当使用 `decltype(auto)` 作为返回类型时，使用 `decltype` 的规则如下：

- 如果表达式返回/产生普通值，那么值类别是 prvalue，`decltype(auto)` 推导出值类型。
- 如果表达式返回/产生 lvalue 引用，那么值类别是 lvalue，`decltype(auto)` 推导出 lvalue 引用。
- 如果表达式返回/产生 rvalue 引用，那么值类别是 xvalue，`decltype(auto)` 推导出 rvalue 引用。

这正是完美返回所需要的：对于普通值，推导一个值；对于引用，推导一个相同类型的引用。

更通用的例子，考虑 helper 函数（在初始化之后）透明地调用函数，就像直接调用函数一样：

generic/call.hpp

```
1 #include <utility> // for forward<>()
2 template <typename Func, typename... Args>
3 decltype(auto) call (Func f, Args&&... args)
4 {
5     ...
6     return f(std::forward<Args>(args)...);
7 }
```

该函数将 *args* 声明为一个可变数量的通用引用（也称为转发引用）。通过 *std::forward<>()*，它完美地将这些给定的参数转发给 *f* 作为第一个参数传递的函数。因为我们使用 *decltype(auto)* 作为返回类型，所以完美地将 *f()* 的返回值返回给 *call()* 的调用者。因此，可以同时调用按值返回和按引用返回的函数。例如：

generic/call.cpp

```
1 #include "call.hpp"
2 #include <iostream>
3 #include <string>
4
5 std::string nextString()
6 {
7     return "Let's dance";
8 }
9
10 std::ostream& print(std::ostream& strm, const std::string& val)
11 {
12     strm << "value: " << val;
13 }
14
15 std::string&& returnArg(std::string&& arg)
16 {
17     return std::move(arg);
18 }
19
20 int main()
21 {
22     auto s = call(nextString); // call() returns temporary object
23
24     auto&& ref = call(returnArg, std::move(s)); // call() returns rvalue reference to
25         s
26     std::cout << "s: " << s << '\n';
27     std::cout << "ref: " << ref << '\n';
28
29     auto str = std::move(ref); // move value from s and ref to str
30     std::cout << "s: " << s << '\n';
31     std::cout << "ref: " << ref << '\n';
32     std::cout << "str: " << str << '\n';
```

```
32     call(print, std::cout, str) << '\n'; // call() returns reference to std::cout
33 }
34 }
```

当调用

```
1 auto s = call(nextString);
```

函数 *call()* 调用函数 *nextString()*, 不带任何参数, 并返回它的返回值来初始化 *s*。

当调用

```
1 auto&& ref = call(returnArg, std::move(s));
```

函数调用带有 *std::move()* 标记的函数 *returnArg()*。*returnArg()* 将传递的参数作为右值引用返回, 然后 *call()* 完美地返回给调用者来初始化 *ref*。*str* 仍然有它的值, 并且 *ref* 会对其进行引用:

```
s: Let's dance
ref: Let's dance
```

使用

```
1 auto str = std::move(ref);
```

将 *s* 和 *ref* 的值移动到 *str*, 得到以下状态:

```
s:
ref:
str: Let's dance
```

当调用

```
1 call(print, std::cout, ref) << '\n';
```

函数使用 *std::cout* 和 *ref* 作为完全转发的参数调用 *print()* 函数。*print()* 将传递的流作为 lvalue 引用返回, 然后完美地返回给 *call()* 的调用者。

12.2.2 延迟完美返回

为了完美地返回之前计算的值, 必须使用 *decltype(auto)* 声明一个局部对象, 当它是 rvalue 引用时, 使用 *std::move()* 返回它。例如:

```
1 template<typename Func, typename... Args>
2 decltype(auto) call(Func f, Args&&... args)
3 {
4     decltype(auto) ret{f(std::forward<Args>(args)...)};
5     ...
6     if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
7         return std::move(ret); // move xvalue returned by f() to the caller
8     }
9     else {
```

```
10     return ret; // return the plain value or the lvalue reference
11 }
12 }
```

函数中, *ret* 的类型就是 *f()* 的完美推导类型。通过使用 `if constexpr`(从 C++17 起), 可以使用 `decltype(auto)` 和 `decltype` 两种方式来推导类型, 如下所示:

- 如果 *ret* 声明为 rvalue 引用, `decltype(auto)` 使用表达式 `std::move(ret)`, 这是 xvalue, 来推导 rvalue 引用。因此, 将 *f()* 返回的值移动到这个函数的调用者。
- 如果 *ret* 声明为普通值或 lvalue 引用, `decltype(auto)` 使用名为 *ret* 的类型, 这也是某个值类型或 lvalue 引用类型。

其他的解决方案并不总是有效:

- 即使在 C++20 之前, 以下内容也会做正确的事情, 但有性能问题:

```
1 decltype(auto) call( ... )
2 {
3     decltype(auto) ret{f( ... )};
4     ...
5     return static_cast<decltype(ret)>(ret); // perfect return but unnecessary
6     copy
7 }
```

事实上, 总是使用 `static_cast<>` 可能会禁用移动语义和复制备选。对于普通值, 这就像在 `return` 语句中有不必要的 `std::move()`。

- 简单地返回 *ret* 并不总有效:

```
1 decltype(auto) call( ... )
2 {
3     decltype(auto) ret{f( ... )};
4     ...
5     return ret; // may be an ERROR
6 }
7 }
```

call() 的返回类型正确。但是, 如果 *f()* 返回 rvalue 引用, 则不能返回左值 *ret*, 因为非 `const` 引用没有绑定到 lvalue。

- 使用 `auto&&` 来声明 *ret* 不起作用, 因为将通过引用返回:

```
1 decltype(auto) call( ... )
2 {
3     auto&& ret{f( ... )};
4     ...
5     return ret; // fatal runtime error: returns a reference to a local object
6 }
7 }
```

使用 `decltype(auto)` 时, 不要在返回的名字后面加括号:

```

1 decltype(auto) call( ... )
2 {
3     decltype(auto) ret{f( ... )};
4     ...
5     if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
6         return std::move(ret); // move value returned by f() to the caller
7     }
8     else {
9         return (ret); // FATAL RUNTIME ERROR: always returns an lvalue reference
10    }
11 }

```

这样，返回类型 decltype(auto) 会切换到表达式规则，并推断出 lvalue 引用，因为 ret 是 lvalue(有名称的对象)。

如果习惯在 return 语句中把名字和表达式用括号括起来，那就不要再这样做了。若继续括起来，使用 decltype(auto) 时可能会出现错误。

12.2.3 完美的 Lambda 转发和返回

如果 Lambda 完美返回，则必须更改返回类型。声明如下：

```

1 [] (auto f, auto&&... args) {
2     ...
3 }

```

代表：

```

1 [] (auto f, auto&&... args) -> auto {
2     ...
3 }

```

这意味着在默认情况下，Lambda 总是按值返回。

通过用返回类型 decltype(auto) 显式声明 Lambda，可以实现完美返回：

```

1 [] (auto f, auto&&... args) -> decltype(auto) {
2     ...
3     return f(std::forward<decltype(args)>(args)...);
4 }

```

对于延迟完美返回，需要和前面介绍的一样的技巧：如果返回 rvalue 引用，必须使用 `std::move()`：

```

1 [] (auto f, auto&&... args) -> decltype(auto) {
2     decltype(auto) ret = f(std::forward<decltype(args)>(args)...);
3     ...
4     if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
5         return std::move(ret); // move value returned by f() to the caller
6     }
7     else {
8         return ret; // return the value or the lvalue reference
9     }

```

```
10 };
```

同样，不要在返回名称周围加上额外的括号，因为 decltype(auto) 会将其推断为 lvalue 引用：

```
1 [] (auto f, auto&&... args) -> decltype(auto) {
2 ...
3     decltype(auto) ret = f(std::forward<decltype(args)>(args)...);
4 ...
5     if constexpr (std::is_rvalue_reference_v<decltype(ret)>) {
6         return std::move(ret); // move value returned by f() to the caller
7     }
8     else {
9         return (ret); // FATAL RUNTIME ERROR: always returns an lvalue reference
10    }
11};
```

12.3 总结

- decltype(auto) 是占位符类型，用于从值推导值类型，从引用推导引用类型。
- 使用 decltype(auto) 在泛型函数中完美地返回值。
- return 语句中，不要把返回值/表达式作为整体用括号括起来。

3 Part III: C++ 标准库中的移动语义

介绍了移动语义的所有特性之后，这一部分描述了这些特性在 C++ 标准库中的应用。

13 只移动类型

移动语义的主要应用是只移动类型。这些类型中，对象代表某个值或“拥有”某个资源，对这些资源的复制没有任何意义。但是，仍然可以传递值或所有权（例如，将其作为参数传递、返回或将其存储在容器中）。

C++ 标准库中的只移动类型如下：

- 输入输出流
- 线程
- unique 智能指针

这些例子中，对象表示资源（打开的流、运行的线程或分配的对象）。对象“拥有”资源的意义在于对象的析构函数将释放资源（关闭流、结束或等待线程结束、释放分配的对象）。

可以传递所有权，但不支持资源的复制。出于语义原因（什么是已打开文件的副本，什么是正在运行的线程的副本？）或技术原因（如果拥有相同资源的两个所有者，必须同步访问或处理潜在的后果），复制可能没有意义。

因此，只移动类型简化了对资源的管理。类型系统禁用复制，但仍然可以传递这些资源。每次存储/管理资源的位置总是只有一个，但仍然可以使用指针或引用来引用这些资源。

13.1 声明和使用只移动类型

只移动类型和对象在声明或使用时有一些共同点。

13.1.1 声明只移动类型

只移动类型禁用了复制。通常，复制构造函数和复制赋值操作符会删除，而移动构造函数和移动赋值操作符会默认实现：

例如：

```
1 class MoveOnly {
2 public:
3     // constructors:
4     MoveOnly();
5     ...
6     // copying disabled:
7     MoveOnly(const MoveOnly&) = delete;
8     MoveOnly& operator=(const MoveOnly&) = delete;
9     // moving enabled:
10    MoveOnly(MoveOnly&&) noexcept;
11    MoveOnly& operator=(MoveOnly&&) noexcept;
12};
```

按照规则，声明移动特殊成员函数就足够了（因为声明特殊移动成员将复制成员标记为已删除）。但是，显式地将复制特殊成员函数标记为 `=delete` 会使意图更加明确。

13.1.2 使用只移动类型

通过上面的声明，可以创建和移动对象，但不能复制。例如：

```

1 std::vector<MoveOnly> coll;
2 ...
3 coll.push_back(MoveOnly{}); // OK, creates a temporary object, which is moved into
   coll
4 ...
5 MoveOnly mo;
6 coll.push_back(mo); // ERROR: can't copy mo into coll
7 coll.push_back(std::move(mo)); // OK, moves mo into coll

```

要将只移动元素的值移出容器，需要使用 `std::move()` 作为元素的引用。例如：

```

1 mo = std::move(coll[0]); // move assign first element (still there with moved-from
   state)

```

但请记住，调用之后元素仍然在容器中，状态为已移动。

循环中，也可以移除所有元素：

```

1 for (auto& elem : coll) { // note: non-const reference
2     coll2.push_back(std::move(elem)); // move element to coll2
3 }

```

同样的，元素仍然在容器中，状态为已移动。

对于只移动类型，有两个不可能操作：

- 不能使用 `std::initializer_lists`，因为其是按值传递的，这需要复制元素：

```

1 std::vector<MoveOnly> coll{ MoveOnly{}, ... }; // ERROR
2

```

- 通过引用迭代，可以访问容器中所有只允许移动的元素：

```

1 std::vector<MoveOnly> coll;
2 ...
3 for (const auto& elem : coll) { // OK
4     ...
5 }
6 ...
7 for (auto elem : coll) { // ERROR: can't copy move-only elements
8     ...
9 }
10

```

请参阅 `lib/moveonly.cpp`，其中包含本节中的所有示例语句。

13.1.3 将只移动对象作为参数传递

如果使用了移动语义，可以通过值传递和返回只移动的对象：

```

1 void sink(MoveOnly arg); // sink() takes ownership of the passed argument
2
3 sink(MoveOnly{}); // OK, moves temporary objects to arg

```

```

4 MoveOnly mo;
5 sink(mo); // ERROR: can't copy mo to arg
6 sink(std::move(mo)); // OK, moves mo to arg because passed by value

```

从语义上讲，这里将相关资源的所有权传递给函数。但请注意，只有当参数按值接受时才会出现这种情况。

sink() 函数也可以通过 (rvalue 或通用) 引用声明为只接受移动对象，仍然需要通过 *std::move()* 传递 lvalue。但是，不知道传递的资源的所有权是否由 *sink()* 获取。

```

1 void sink(MoveOnly&& arg); // sink() might take ownership of the passed argument
2
3 MoveOnly mo;
4 sink(mo); // ERROR: can't pass lvalue mo to arg
5 sink(std::move(mo)); // OK, might move mo to something inside sink()

```

Scott Meyers 和 Herb Sutter(C++ 社区的两位主要作者) 讨论了应该如何声明只移动类型的接收器函数。Herb 的立场是通过值来取参数，Scott 的立场是通过 rvalue 引用来取参数。

据我所知，他们后来一致认为最好采用 rvalue 引用的方法。然而，真正的答案不重要。*std::move()* 的规则也应该适用于这里：如果用 *std::move()* 传递只移动的对象，可能会丢失值，也可能不会。如果放弃所有权很重要（希望确保文件已关闭、线程已停止或相关的资源已释放），请在调用后使用相应语句直接明示。例如：

```

1 MoveOnly mo;
2
3 foo(std::move(mo)); // might move ownership
4 // ensure mo's resource is longer acquired/owned/open/running:
5 mo.close(); // or mo.reset() or mo.release() or so

```

只移动对象通常有这样的函数，但名称不同（例如，在 C++ 标准库中，称为流的 *close()*，线程的 *join()*，或 unique 指针的 *reset()*）。这些函数通常将对象带入默认构造状态。

13.1.4 按值返回只移动的对象

还可以实现按值返回（新的）只移动对象的源函数，这意味着将所有权传递给函数的调用者。

如果以这种方式返回一个局部对象，就会自动使用移动语义：

```

1 MoveOnly source()
2 {
3     MoveOnly mo;
4     ...
5     return mo; // moves mo to the caller
6 }
7
8 MoveOnly m{source()}; // takes ownership of the associated value/resource

```

有非本地数据时，才可能需要在 *return* 语句中使用 *std::move()*（例如，成员函数中移出成员的值）。

13.1.5 只移动对象的状态

只移动对象处于已移动状态时，通常不再拥有自己的资源。这是一个已定义的状态，该类型的用户应该能够对其进行双重检查。有时，移动操作只是交换内部数据，以便移动赋值将另一个资源分配给已移动对象（例如，流类这样做）。

C++ 标准库使用不同的方式和名称来检查不再拥有资源的“已移动”状态。例如，正校验（是否仍拥有资源？）可能如下所示：

- if(s.is_open()) 是文件流
- if(up) 对于 unique 指针
- if(t.joinable()) 对于 thread 对象
- if(f.valid()) 对于 std::future 对象

13.2 总结

- 只移动类型允许我们移动“拥有的”资源，而不能够复制。复制特殊成员函数会删除。
- 不能在 std::initializer_lists 中使用只移动类型。
- 不能在只移动类型的集合上按值迭代。
- 如果将只移动对象传递给接收器函数，并且确保已经失去了所有权（文件关闭，内存释放等），那么在此之后可以直接显式地释放资源。

14 移动算法和迭代器

C++ 标准库对迭代元素时移动元素提供了特殊支持。为此，C++11 中引入了特殊算法和特殊迭代器（移动迭代器）。本章讨论如何使用它们。

14.1 移动算法

C++ 标准库提供了一些移动元素的算法。这些算法是：

- **std::move()**, 将元素移动到另一个范围或在同一范围内向后移动 (不要将此算法与 *std::move()* 标记不再需要其值的对象混淆)
指定目标范围的开头，元素将从源范围的开头移动到末尾。
- **std::move_backward()**, 将元素移动到另一个范围或在同一范围内向前移动
指定目标范围的结束，元素将从源范围的结束移动到开始。

这些算法使用的是 *std::copy()* 和 *std::copy_backward()* 算法的对等操作。是的，*std::move()* 还有另一个重载，接受多个形参（三个迭代器（从 C++17 起），还有一个可选的并行执行策略）。

这些算法的效果是在迭代每个元素时调用 *std::move(elem)* 对目标范围进行移动赋值。

考虑以下例子：

lib/movealgo.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <list>
5 #include <algorithm>
6
7 template<typename T>
8 void print(const std::string& name, const T& coll)
9 {
10    std::cout << name << " (" << coll.size() << " elems): ";
11    for (const auto& elem : coll) {
12        std::cout << " " << elem << " ";
13    }
14    std::cout << "\n";
15 }
16
17 int main(int argc, char** argv)
18 {
19     std::list<std::string> coll1 { "love", "is", "all", "you", "need" };
20     std::vector<std::string> coll2;
21
22     // ensure coll2 has enough elements to overwrite their values:
23     coll2.resize(coll1.size());
24
25     // print out size and values:
26     print("coll1", coll1);
27     print("coll2", coll2);
```

```

28 // move assign the values from coll1 to coll2
29 // — not changing any size
30 std::move(coll1.begin(), coll1.end(), // source range
31 coll2.begin()); // destination range
32
33 // print out size and values:
34 print("coll1", coll1);
35 print("coll2", coll2);
36
37 // move assign the first three values inside coll2 to the end
38 // — not changing any size
39 std::move_backward(coll2.begin(), coll2.begin() + 3, // source range
40 coll2.end()); // destination range
41
42 // print out size and values:
43 print("coll1", coll1);
44 print("coll2", coll2);
45 }

```

当调用 `std::move()` 时，将源容器的所有值赋给目标容器：

```

1 // move assign the values from coll1 to coll2
2 // — not changing any size
3 std::move(coll1.begin(), coll1.end(), // source range
4         coll2.begin()); // destination range

```

与覆盖算法一样，目标容器必须有足够的元素（否则就会有未定义的行为）。元素的数量不会改变（无论是在源范围还是在目标范围）。但是，源范围的元素处于“已移动”状态。因此，这个调用之后，就不知道源范围内字符串的值（除非为已移动对象指定了行为，比如只移动类型）。

当调用 `std::move_backward()` 算法时，将前三个元素赋值到同一个集合的末尾：

```

1 // move assign the first three values inside coll2 to the end
2 // — not changing any size
3 std::move_backward(coll2.begin(), coll2.begin() + 3, // source range
4                     coll2.end()); // destination range

```

同样，如果没有另一个值移走，元素的状态将会是已移动。因此，不再知道前两个元素的值（第三个元素的值被第一个元素的移动赋值覆盖）。在这通调用之后，我们只知道 `coll2` 的最后三个元素是 `love`, `is` 和 `all`。

因此，整个程序的输出是这样的（? 表示不确定的值）：

```

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'

```

字符串移动后通常是空的，但这不能保证。实践中，我甚至在某个平台上发现了这样的输出：

```
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'
```

所以，与往常一样：使用已移动元素时要谨慎。

14.2 移除性算法

根据设计，C++ 算法使用迭代器来处理容器和范围的元素。然而，就像指针操作数组一样，迭代器只能读取和写入值，不能插入或删除元素。因此，“删除”算法并不是真正删除元素，只是将所有未移除元素的值移动到已处理范围的前面。

例如，给定以下整数序列：

```
1 2 3 4 5 4 3 2 1
```

调用 `std::remove()` 算法来删除值为 2 的所有元素，修改后的序列：

```
1 3 4 5 4 3 1 2 1
```

所有不是 2 的元素都移到前面，作为新的结束（“last”元素后面的位置），返回 2 的位置。

可能的话，这些算法会移动，会以已移状态保留元素。本例中，如果元素是字符串，那么最后 2 个元素将保持不变，但最后 1 个元素将向前移动到 2 之前，以便最后一个元素处于已移动状态。

因此，这些算法也可以将元素保留为已移动状态。事实上，以下算法可以创建已移动状态：

- `std::remove()` 和 `std::remove_if()`
- `std::unique()`

看一个完整的例子，可以看到元素是否为已移动状态：

[lib/email.hpp](#)

```
1 #include <iostream>  
2 #include <cassert>  
3 #include <string>  
4  
5 // class for email addresses  
6 // — asserts that each email address has a @  
7 // — except when in a moved-from state  
8 class Email {  
9     private:  
10         std::string value; // email address  
11         bool movedFrom{false}; // special moved-from state
```

```

12 public:
13 Email(const std::string& val)
14 : value{val} {
15     assert(value.find('@') != std::string::npos);
16 }
17 Email(const char* val) // enable implicit conversions for string literals
18 : Email{std::string(val)} {
19 }
20 std::string getValue() const {
21     assert(!movedFrom); // or throw
22     return value;
23 }
24 ...
25 // implement move operations to signal a moved-from state:
26 Email(Email&& e) noexcept
27 : value{std::move(e.value)}, movedFrom{e.movedFrom} {
28     e.movedFrom = true;
29 }
30 Email& operator=(Email&& e) noexcept {
31     value = std::move(e.value);
32     movedFrom = e.movedFrom;
33     e.movedFrom = true;
34     return *this;
35 }
36 // enable copying:
37 Email(const Email&) = default;
38 Email& operator=(const Email&) = default;
39
40 // print out the current state (even if it is a moved-from state):
41 friend std::ostream& operator<< (std::ostream& strm, const Email& e) {
42     return strm << (e.movedFrom ? "MOVED-FROM" : e.value);
43 }
44 };

```

通过实现移动构造函数和移动赋值操作符，设置了一个成员 *movedFrom*，并在输出操作符中求值。

现在让我们将该类中的一些元素放入 vector 中，并使用算法 *std::remove_if()* 删除一些元素。本例中，我们删除所有以".de" 结尾的电子邮件地址：

lib/removeif.cpp

```

1 #include "email.hpp"
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <algorithm>
6 int main()
{
7     std::vector<Email> coll{ "tomdomain.de" , "jillcompany.com" ,
8     "sarahdomain.de" , "hanacompany.com" };
9

```

```

10
11 // remove all email addresses ending with ".de":
12 auto newEnd = std::remove_if(coll.begin(), coll.end(),
13 [] (const Email& e) {
14     auto&& val = e.getValue();
15     return val.size() > 2 &&
16     val.substr(val.size() - 3) == ".de";
17 });
18
19 // print elements up to the new end:
20 std::cout << "remaining elements:\n";
21 for (auto pos = coll.begin(); pos != newEnd; ++pos) {
22     std::cout << " " << *pos << "\n";
23 }
24
25 // print all elements in the container:
26 std::cout << "all elements:\n";
27 for (const auto& elem : coll) {
28     std::cout << " " << elem << "\n";
29 }
30 }
```

程序输出如下:

```

remaining elements:
"jill@company.com"
"hana@company.com"
all elements:
"jill@company.com"
"hana@company.com"
"sarah@domain.de"
"MOVED-FROM"
```

“删除”以“.de”结尾的元素之后，新的结束位置是第二个元素后面的位置。但在容器中，第三个元素没有移动，第四个元素处于已移动状态，因为它移动赋值给了第二个元素（之前移动到第一个元素）。

使用这些已移动对象时要小心，原因在关于已移动状态的章节中已经讨论过了。

14.3.1 移动算法中的迭代器

算法中使用移动迭代器，通常只有在算法保证每个元素只使用一次时才有意义。因此，该算法应该：

- 要求源的输入迭代器类别和目标的输出迭代器类别
- 或者保证每个元素只使用一次（例如，为 `std::for_each()` 算法指定）

对于具有可调用对象的算法，允许规范提供详细的功能，元素通过 `std::move()` 传递给可调用

对象。在可调用对象内部，可以决定如何处理：

- 按值接受参数，总是移动/窃取值或资源
- 通过右值/通用引用参数来决定移动/窃取哪个值/资源

例如：

lib/foreachmove.cpp|

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <algorithm>
5 template<typename T>
6 void print(const std::string& name, const T& coll)
7 {
8     std::cout << name << " (" << coll.size() << " elems): ";
9     for (const auto& elem : coll) {
10         std::cout << " \\" << elem << "\\" ;
11     }
12     std::cout << "\n";
13 }
14 void process(std::string s) // gets moved value from rvalues
15 {
16     std::cout << "- process(" << s << " )\n";
17     ...
18 }
19 int main()
20 {
21     std::vector<std::string> coll{"don't", "vote", "for", "liars"};
22     print("coll", coll);
23     // move away only the elements processed:
24     std::for_each(std::make_move_iterator(coll.begin()), 
25                 std::make_move_iterator(coll.end()),
26                 [] (auto&& elem) {
27                     if (elem.size() != 4) {
28                         process(std::move(elem));
29                     }
30                 });
31     print("coll", coll);
32 }
```

代码中，将所有大小为 4 的元素移动到 helper 函数 `process()` 中：

```
1 // move away only the elements processed:
2 std::for_each(std::make_move_iterator(coll.begin()), 
3               std::make_move_iterator(coll.end()),
4               [] (auto&& elem) {
5                   if (elem.size() != 4) {
6                       process(std::move(elem));
7                   }
8               });
9 }
```

为此，将标记的元素（移动迭代器标记为 `std::move()`）通过通用引用传递给 `process()`，并将其与 `std::move()` 一起传递给 `process()`。因为 `process()` 按值接受参数，所以值实际上已经移走。

因此，所有大小不为 4 的元素都从容器 `coll` 中的对象中移出。该程序的输出如下（? 表示未知值）：

```
coll (4 elems): "don't" "vote" "for" "liars"
- process(don't)
- process(for)
- process(liars)
coll (4 elems): "?" "vote" "?" "?"
```

最后一行是这样的：

```
coll (4 elems): "" "vote" "" ""
```

这里使用了辅助函数 `std::make_move_iterator()`，这样在声明迭代器时就不必指定元素类型了。从 C++17 开始，类模板实参推导 (CTAD) 允许直接声明类型 `std::move_iterator`，而不需要指定元素类型：

```
1 std :: for_each ( std :: move_iterator { coll . begin () } ,
2 std :: move_iterator { coll . end () } ,
3 [] ( auto&& elem ) {
4     if ( elem . size () != 4 ) {
5         process ( std :: move ( elem )) ;
6     }
7});
```

14.3.2 在构造函数和成员函数中移动迭代器

也可以在只读取一次元素的算法中使用移动迭代器，场景可能是将源容器的元素移动到另一个容器（相同或不同类型的）。例如：

lib/moveitor.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <list>
4 #include <vector>
5
6 template<typename T>
7 void print( const std :: string& name , const T& coll )
8 {
9     std :: cout << name << " (" << coll . size () << " elems ) : " ;
10    for ( const auto& elem : coll ) {
11        std :: cout << " \ " << elem << "\ " ;
```

```

12     }
13     std::cout << "\n";
14 }
15
16 int main()
{
17 {
18     std::list<std::string> src{ "don't" , "vote" , "for" , "liars" };
19     // move all elements from the list to the vector:
20     std::vector<std::string> vec{ std::make_move_iterator(src.begin()) ,
21         std::make_move_iterator(src.end()) };
22     print("src" , src);
23     print("vec" , vec);
24 }
```

该程序有以下输出:(? 表示未知值):

```

src (4 elems): "?" "?" "?" "?"
vec (4 elems): "don't" "vote" "for" "liars"
```

请再次注意，源容器中的元素数量没有改变。所有元素会移动到初始化的新容器中。因此，源范围中的元素随后处于已移动状态，所以不知道它们的值。

14.3 移动迭代器

通过使用移动迭代器 (在 C++11 中引入)，甚至可以在其他算法中使用移动语义，也可以在任何输入范围内使用移动语义 (例如，在构造函数中)。

但是，使用这些迭代器时要小心。在迭代容器或范围的元素时，每次对元素的访问都使用 `std::move()`。这可能会更快，会使元素处于有效但未定义的状态，所以一个元素不应该使用两次 `std::move()`。

14.4 总结

- C++ 标准库提供了移动多个元素的特殊算法，称为 `std::move()`(是 `std::move()` 的重载，用于将单个对象标记为可移动) 和 `std::move_backward()`。它们让元素处于离开状态。
- 移除算法可以将元素保留在已移动状态。
- 移动迭代器允许在迭代元素时使用移动语义。可以在算法或构造函数中使用这些迭代器，构造函数中，范围用于初始化/设置值。但是，要确保迭代器只对每个元素使用一次。

15 C++ 标准库类型中的移动语义

本章讨论了 C++ 标准库中移动语义最重要的应用。

可以帮助您更好地理解基本类型，如字符串和容器，并了解一些移动语义的巧妙使用。

15.1 字符串的移动语义

字符串是可以分配内存来保存其值的对象。因此，可以对其使用移动语义。

书中已经有了几个关于字符串如何支持移动语义的例子：

- 将字符串移动到某个 vector 中
- 为字符串实现移动构造函数
- 成员初始化字符串
- 循环中读取字符串，并在 `move()` 之后使用它们

本节中，将更详细地介绍移动语义对字符串的影响。

15.1.1 字符串分配和容量

字符串的容量（当前可用于该值的内存）通常不会减少。只有移动操作、`swap()` 或 `shrink_to_fit()` 可以减小容量。

考虑下面例子：

[lib/stringmoveassign.cpp](#)

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string s0;
7     std::string s1{"short"};
8     std::string s2{"a string with an extraordinarily long value"};
9     std::cout << "— s0 capa: " << s0.capacity() << " (" << s0 << ")" \n";
10    std::cout << " s1 capa: " << s1.capacity() << " (" << s1 << ")" \n";
11    std::cout << " s2 capa: " << s2.capacity() << " (" << s2 << ")" \n";
12
13    std::string s3{std::move(s1)};
14    std::string s4{std::move(s2)};
15    std::cout << "— s1 capa: " << s1.capacity() << " (" << s1 << ")" \n";
16    std::cout << " s2 capa: " << s2.capacity() << " (" << s2 << ")" \n";
17    std::cout << " s3 capa: " << s3.capacity() << " (" << s3 << ")" \n";
18    std::cout << " s4 capa: " << s4.capacity() << " (" << s4 << ")" \n";
19
20    std::string s5{"quite a reasonable value"};
21    std::cout << "— s4 capa: " << s4.capacity() << " (" << s4 << ")" \n";
22    std::cout << " s5 capa: " << s5.capacity() << " (" << s5 << ")" \n";
23
24    s4 = std::move(s5);
25    std::cout << "— s4 capa: " << s4.capacity() << " (" << s4 << ")" \n";
```

```
26     std::cout << " s5 capa: " << s5.capacity() << " (" << s5 << ")" \n";  
27 }
```

代码中，由于小字符串优化 (SSO)，即使是空字符串也有容纳某些字符的能力，这通常为字符串本身的值保留 15 或 22 个字节。除了 SSO 的大小之外，字符串在堆上分配内存，堆至少有存储值所需的大小。因此：

```
1 std::string s0;  
2 std::string s1{ "short" };  
3 std::string s2{ "a string with an extraordinarily long value" };
```

可能会得到如下输出：

- 平台 A:

```
- s0 capa: 15 ("")  
s1 capa: 15 ('short')  
s2 capa: 43 ('a string with an extraordinarily long value')
```

- 平台 B:

```
- s0 capa: 22 ("")  
s1 capa: 22 ('short')  
s2 capa: 47 ('a string with an extraordinarily long value')
```

这里，两个平台都支持 SSO(最多 15 或 22 个字符)，当只需要 43 个字符的内存时，第二个平台为 47 个字符分配内存。

所有平台上，已移动字符串通常为空，即使值没有存储在外部分配的内存中 (因此我们必须复制所有字符)。

```
1 std::string s3{ std::move(s1) };  
2 std::string s4{ std::move(s2) };
```

会得到这样的结果：

- 平台 A:

```
- s1 capa: 15 ("")  
s2 capa: 15 ("")  
s3 capa: 15 ('short')  
s4 capa: 43 ('a string with an extraordinarily long value')
```

- 平台 B:

```
- s1 capa: 22 ("")
s2 capa: 22 ("")
s3 capa: 22 ('short')
s4 capa: 47 ('a string with an extraordinarily long value')
```

注意，这里不能保证 *s1* 变成空的。C++ 标准库只保证从字符串中移动的字符串处于有效但未指定的状态，这意味着 *s1* 的值仍然可以是“short”，或是其他值。

移动时分配不同的字符串值可能会缩小容量。示例程序的最后两个步骤基本上执行：

```
1 std :: string s4{ "a string with an extraordinarily long value" };
2 std :: string s5{ "quite a reasonable value" };
3 s4 = std :: move(s5);
```

在实践中会有以下输出：

- 内存交换：

```
- 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 ("")
```

- 移动内存 (释放旧内存后)：

```
- 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 ("")
```

s4 的容量通常会缩小，有时缩小到目标的容量，有时缩小到空字符串的最小容量。然而，两者都不能完全保证。

15.2 容器的移动语义

容器通常是必须分配内存来保存其元素的对象。因此，可以从使用移动语义中获益。但有例外：`std::array` 不在堆上分配内存，这意味着对 `std::array` 应该使用特殊规则。

书中已经有了几个关于容器如何支持移动语义的例子。最初的例子中，了解了支持的移动语义：

- 通过重载 `push_back()`，C++ 标准支持移动语义来插入新元素。
- 通过提供移动构造函数和移动赋值操作符，降低复制临时对象（比如返回值）的成本。

执行以下操作时，所有容器都支持移动语义：

- 复制容器
- 赋值容器

- 插入容器

然而，还有更多。

15.2.1 基本移动支持容器整体

所有容器都定义了一个移动构造函数和移动赋值操作符，以支持未命名临时对象和 `std::move()` 标记对象的移动语义。

例如，`std::list<>` 的声明如下：

```

1 template<typename T, typename Allocator = allocator<T>>
2 class list {
3     public:
4     ...
5     list(const list&); // copy constructor
6     list(list&&); // move constructor
7     list& operator=(const list&); // copy assignment
8     list& operator=(list&&) noexcept( ... ); // move assignment
9     ...
10 };

```

这使得按值返回/传递容器并赋值的成本变低。例如：

```

1 std::list<std::string> createAndInsert()
2 {
3     std::list<std::string> coll;
4     ...
5     return coll; // move constructor if not optimized away
6 }
7 std::list<std::string> v;
8 ...
9 v = createAndInsert(); // move assignment

```

但请注意，这里有附加的要求和保证，这适用于除 `std::array<>` 之外的所有容器的移动构造函数和移动赋值操作符。这些要求和保证意味着已移动的容器移动是空的。

容器移动构造函数的保证

对于移动构造函数：

```

1 ContainerType cont1{ ... };
2 ContainerType cont2{ std::move(cont1) }; // move the container

```

C++ 标准规定了常量复杂度，移动的持续时间不取决于元素的数量。

有了这种保证，实现者没有其他选择，只能从源对象 `cont1` 整体窃取元素的内存到目标对象 `cont2`，使源对象 `cont1` 处于初始/空状态。

可能会认为移动构造函数也可以在源对象中创建新值，但这没有多大意义，因为这只会使操作变慢。

对于 `vector`，甚至间接禁止从已移动对象中获取值，因为 `std::vector<>` 的移动构造函数不会抛出异常：

```

1 template<typename T, typename Allocator = allocator<T>>
2 class vector {
3     public:
4     ...
5     vector(const vector&); // copy constructor
6     vector(vector&& noexcept; // move constructor
7     ...
8 };

```

总之，在移动构造函数中使用容器作为源时，有以下保证：

- 对于 `vector`，本质上要求已移动的容器为空。
- 对于其他容器（除了 `std::array<>`），不是严格要求为空，但实现为其他也没什么意义。

容器移动赋值操作符的保证

对于移动赋值操作符：

```

1 ContainerType cont1{ ... }, cont2{ ... };
2 cont2 = std::move(cont1); // move assign the container

```

C++ 标准保证此操作会覆盖或销毁目标对象 `cont2` 的每个元素。这保证了目标容器 `dest2` 的元素在条目时拥有的所有资源都会进行释放。因此，只有两种方法来实现移动赋值：

- 销毁旧的元素，并将源的全部内容移动到目标（即，将指向内存的指针从源移动到目标）。
- 一个元素一个元素地从源 `cont1` 移动到目标 `cont2`，并销毁目标中未覆盖的所有剩余元素。

这两种方法的复制度都线性的，在这个定义中，不允许仅交换源和目标的内容。

然而，自 C++17 以来，所有容器都保证在内存可互换时不会抛出异常。例如：

```

1 template<typename T, typename Allocator = allocator<T>>
2 class list {
3     public:
4     ...
5     list& operator=(list&&)
6     noexcept(allocator_traits<Allocator>::is_always_equal::value);
7     ...
8 };

```

`noexcept` 对赋值操作符的保证，排除了将移动赋值实现为逐个元素移动的第二种方法。移动操作可能会抛出异常，只有销毁旧元素的实现才不会抛出。因此，当内存可以互换时，必须使用第一种方法来实现移动赋值操作符。

总之，移动赋值中使用容器作为源，实际上有以下保证：

- 如果内存是可互换的（使用默认标准分配器时尤其如此），则基本上要求从移动的容器为空。这适用于除 `std::array<>` 之外的所有容器。
- 否则，移动的容器处于有效但未定义的状态。

但请注意，在给自己移动赋值之后，容器总处于未定义但有效的状态。

15.2.2 Insert 和 Emplace 函数

所有容器都支持将新元素移动到容器中。

Insert 函数

例如，vector 通过 `push_back()` 的两种不同实现来支持移动语义：

```
1 template<typename T, typename Allocator = allocator<T>>
2 class vector {
3 public:
4 ...
5
6 // insert a copy of elem:
7 void push_back (const T& elem);
8
9 // insert elem when the value of elem is no longer needed:
10 void push_back (T&& elem);
11 ...
12 };
```

rvalue 的 `push_back()` 函数用 `std::move()` 传递传递的元素，这样就调用了元素类型的移动构造函数，而不是复制构造函数。

同样，所有容器都有相应的重载。例如：

```
1 template<typename Key, typename T, typename Compare = less<Key>,
2 typename Allocator = allocator<pair<const Key, T>>>
3 class map {
4 public:
5 ...
6 pair<iterator, bool> insert(const value_type& x);
7 pair<iterator, bool> insert(value_type&& x);
8 ...
9 };
```

Emplace 函数

从 C++11 开始，容器也提供了 Emplace 函数（比如为向量提供了 `emplace_back()`）。可以传递多个参数来直接在容器中初始化新元素，而不是传递单个元素类型参数（或可转换为元素类型）。这样就可以保存副本或移动。

注意，即使这样，容器也可以通过为构造函数的初始参数支持移动语义，并从移动语义中获益。

像 `emplace_back()` 这样的函数可以使用完美转发来避免创建所传递参数的副本。例如，对于 `std::vector<>`，`emplace_back()` 成员函数的定义如下：

```
1 template<typename T, typename Allocator = allocator<T>>
2 class vector {
3 public:
4 ...
5 // insert a new element with perfectly forwarded arguments:
```

```

6  template<typename ... Args>
7  constexpr T& emplace_back(Args&&... args) {
8
9      ...
10     // call the constructor with the perfectly forwarded arguments:
11     place_element_in_memory(T(std::forward<Args>(args)...));
12
13     ...
14 };

```

在 vector 内部使用完全转发的参数初始化新元素。

15.2.3 std::array<> 的移动语义

array<> 是唯一没有在堆上分配内存的容器。实际上，是通过带有数组成员的模板化 C 数据结构实现的：

```

1 template<typename T, size_t N>
2 struct array {
3     T elems[N];
4     ...
5 };

```

因此，不能以移动指针到内部内存的方式来实现移动操作。

因此，std::array<> 有两个保证：

- 移动构造函数具有线性复杂度，必须逐个元素地移动。
- 移动赋值操作符可能总是抛出异常，因为它必须逐个元素地移动赋值。

因此，复制或移动一个数值数组没有区别：

```

1 std::array<double, 1000> arr;
2 ...
3 auto arr2{arr}; // copies all double elements/values
4 auto arr3{std::move(arr)}; // still copies all double elements/values

```

对于所有其他容器，后者将只移动指向新对象的内部指针，这是一个成本非常低的操作。

但是，如果移动元素比复制元素成本还要低，那么移动仍然比复制要好。例如：

```

1 std::array<std::string, 1000> arr;
2 ...
3 auto arr2{arr}; // copies string by string
4 auto arr3{std::move(arr)}; // moves string by string

```

如果字符串分配堆内存（即，如果使用小字符串优化（SSO），则有一个显著的大小），那么移动字符串数组通常会更快。

可以通过 lib/contmove.cpp 看到这一点，它检查复制和移动不同元素类型（双精度、小字符串和大字符串）的数组和 vector 之间的区别。请注意，在不同平台上，因为生成的代码具有不同的优化，复制和移动双精度数组或小字符串之间可能存在微小的性能差异。

15.3 词汇类型的移动语义

C++ 标准库提供了两种词汇表类型，用于使用值语义处理一个或多个值（对象自动地和整体地保存和复制它们的值）。

原则上，都提供了移动语义。然而，其中有一些需要专门讨论和说明。

15.3.1 pair 的移动语义

`std::pair<>` 是很好的例子，展示了移动语义的好处和复杂性。原则上，只有一个具有两个成员的通用数据结构（在命名空间 `std` 中定义）：

```
1 template<typename T1, typename T2>
2 struct pair {
3     T1 first;
4     T2 second;
5     ...
6 };
```

为了支持移动语义（以及其他一些棘手的案例，如引用成员），有以下声明（这里，使用了 C++14 版本，并做了一些利于可读性的工作）：

```
1 template<typename T1, typename T2>
2 struct pair {
3     // types of each member:
4     using first_type = T1; // same as: typedef T1 first_type
5     using second_type = T2;
6     // the members:
7     T1 first;
8     T2 second;
9
10    // constructors:
11    constexpr pair();
12    constexpr pair(const T1& x, const T2& y);
13    template<typename U, typename V> constexpr pair(U&& x, V&& y);
14    pair(const pair&) = default;
15    pair(pair&&) = default;
16    template<typename U, typename V> constexpr pair(const pair<U, V>& p);
17    template<typename U, typename V> constexpr pair(pair<U, V>&& p);
18    template<typename ... Args1, typename ... Args2>
19    pair(piecewise_construct_t, tuple<Args1...> first_args,
20          tuple<Args2...> second_args);
21
22    // assignments:
23    pair& operator=(const pair& p);
24    pair& operator=(pair&& p) noexcept( ... );
25    template<typename U, typename V> pair& operator=(const pair<U, V>& p);
26    template<typename U, typename V> pair& operator=(pair<U, V>&& p);
27
28    // other:
29    void swap(pair& p) noexcept( ... );
```

30 };

该类支持移动语义。有一个默认的移动构造函数和一个已实现的移动赋值操作符 (如果两种成员类型都保证不抛出异常，则对应的 noexcept 条件是不抛出异常):

```
1 template<typename T1, typename T2>
2 struct pair {
3     ...
4     pair(pair&&) = default;
5     ...
6     pair& operator=(pair&& p) noexcept( ... );
7     ...
8 };
```

因此，代码如下:

```
1 std::pair<std::string, std::string> p1{"some value", "some other value"};
2 auto p2{p1};
3 auto p3{std::move(p1)};
4 std::cout << "p1: " << p1.first << '/' << p1.second << '\n';
5 std::cout << "p2: " << p2.first << '/' << p2.second << '\n';
6 std::cout << "p3: " << p3.first << '/' << p3.second << '\n';
```

有如下输出:

```
p1: /
p2: some value/some other value
p3: some value/some other value
```

但是，`std::pair<>` 也可以通过处理通用/转发引用支持完美转发:

```
1 template<typename T1, typename T2>
2 struct pair {
3     ...
4     template<typename U, typename V> constexpr pair(U&& x, V&& y);
5     ...
6 };
```

因此，可以在初始化 `pair` 时使用移动语义。例如:

```
1 int val = 42;
2 std::string s1{"value of s1"};
3 std::pair<std::string, std::string> p4{std::to_string(val), std::move(s1)};
4
5 std::cout << "s1: " << s1 << '\n';
6 std::cout << "p4: " << p4.first << '/' << p4.second << '\n';
```

有如下输出:

```
s1:  
p4: 42/value of s1
```

相应的成员模板实现为期望的通用/转发引用:

```
1 template<typename U, typename V>  
2 constexpr pair::pair(U&& x, V&& y)  
3 : first(std::forward<U>(x)), second(std::forward<V>(y)) {  
4 }
```

注意，通用引用/转发引用还意味着，只要定义了相应的类型转换，就可以创建和赋值不同的 pair。例如：

```
1 std::pair<const char*, std::string> p5{ "answer", "is 42" };  
2 auto p6{std::move(p5)};  
3  
4 std::cout << "p5: " << p5.first << '/' << p5.second << '\n';  
5 std::cout << "p6: " << p6.first << '/' << p6.second << '\n';
```

有如下输出：

```
p5: answer/  
p6: answer/is 42
```

初始化 *p6* 时，将 *p5* 的第一个成员（声明为 `const char*`）转换为 `std::string`，而在使用 *p5* 的第二个成员初始化 *p6* 的第二个成员时使用移动语义。

最后，注意 `std::pair<>` 支持具有引用类型的成员。在这种情况下，当为这些成员使用 `std::move()` 时将应用特殊规则。参见 `basics/members.cpp` 获取完整的示例。

`std::make_pair()`

`std::pair<>` 附带了一个方便的函数模板 `std::make_pair<>()`，用于创建 `pair` 而不必指定成员的类型：

```
1 auto p{std::make_pair(42, "hello")}; // creates std::pair<int, const char*>
```

`std::make_pair<>()` 是一个很好的例子，它演示了在 rvalue 和通用/转发引用中使用移动语义时必须考虑的另一件事。它的声明在不同的 C++ 标准中有所不同：

- 第一个 C++ 标准是 C++98 中，`make_pair<>()` 是在命名空间 `std` 中使用引用来声明的，以避免不必要的复制：

```
1 template<typename T1, typename T2>  
2 pair<T1, T2> make_pair (const T1& a, const T2& b)  
3 {  
4     return pair<T1, T2>(a, b);  
5 }
```

然而，当使用成对的字符串字面值或原始数组时，会导致了严重的问题。例如，当"hello"作为第二个实参传递时，对应形参 *b* 的类型成为对 const char 数组 (const char(&)[6]) 的引用。因此，char 类型 [6] 推导为 T2 类型，并用作第二个成员的类型。但是，不能使用数组初始化数组成员，因为不能复制数组。

这种情况下，应该使用衰变的类型作为成员类型，这是按值传递参数时获得的类型 (const char* 表示字符串)。

- 因此，C++03 中，函数定义改为使用按值调用：

```
1 template<typename T1, typename T2>
2     pair<T1,T2> make_pair (T1 a, T2 b)
3 {
4     return pair<T1,T2>(a,b);
5 }
6
```

正如在问题解决方案的基本原理中看到的那样，“这似乎是对标准的建议小得多的更改，并且效率方面都被解决方案的优势抵消了。”

- C++11 中，make_pair() 必须支持移动语义，这意味着参数必须成为通用/转发引用。同样，对于引用，参数的类型不会衰减。因此，定义变更如下：

```
1 template<typename T1, typename T2>
2     constexpr pair<typename decay<T1>::type, typename decay<T2>::type>
3         make_pair (T1&& a, T2&& b)
4     {
5         return pair<typename decay<T1>::type,
6             typename decay<T2>::type>(forward<T1>(a),
7             forward<T2>(b));
8     }
9
```

C++14 中可以写成这样：

```
1 template<typename T1, typename T2>
2     constexpr pair<decay_t<T1>, decay_t<T2>>
3         make_pair (T1&& a, T2&& b)
4     {
5         return pair<decay_t<T1>, decay_t<T2>>(forward<T1>(a), forward<T2>(b));
6     }
7
```

真正的实现更加复杂，因为 C++11：为了支持 *std::ref()* 和 *std:: cref()*，还使用了 *std::reference_wrapper<>*。C++ 标准库以类似的方式在许多地方完美地转发参数，通常还会结合使用 *std::decay<>*。

15.3.2 std::optional<> 的移动语义

std::optional<> 是 C++17 中可用的值类型，通过“没有任何值”扩展包含所有可能的类型值。这避免了为具有此语义而标记该类型的特定值（例如，指针值 0）。

optional 对象也支持移动语义。如果将对象作为一个整体移动，状态将复制，所包含的对象（如果有的话）将移动。因此，一个已移动的对象仍然具有相同的状态，但任何值都变成未定义的。

但也可以将值移进或移出所包含的对象。例如：

```
1 std::optional<std::string> os;
2 std::string s = "a very very very long string";
3 os = std::move(s); // OK, moves
4 std::string s2 = *os; // OK, copies
5 std::string s3 = std::move(*os); // OK, moves
```

注意，在最后一次调用之后，*os* 仍然有一个字符串值，但与通常的已移动对象一样，这个值未定义。因此，只要不对其值做任何假设，就可以使用它。甚至可以在那里赋一个新的字符串值。

还请注意，有些重载确保临时 optional 可移动。考虑一个返回可选字符串的函数：

```
1 std::optional<std::string> func();
```

这种情况下，定义了移动值：

```
1 std::string s4 = func().value(); // OK, moves
2 std::string s5 = *func(); // OK, moves
```

这种行为可以通过使用引用限定符，为相应的成员函数提供 rvalue 重载来实现：

```
1 namespace std {
2     template<typename T>
3     class optional {
4         ...
5         constexpr T& operator*() &;
6         constexpr const T& operator*() const&;
7         constexpr T&& operator*() &&;
8         constexpr const T&& operator*() const&&;
9
10        constexpr T& value() &;
11        constexpr const T& value() const&;
12        constexpr T&& value() &&;
13        constexpr const T&& value() const&&;
14    };
15 }
```

通过使用引用限定符，类可以在对 rvalue(临时对象或标记为 *std::move()* 的对象) 调用操作时返回移动值。：

```
1 std::vector<std::string> coll;
2 std::optional<std::string> optStr;
3 ...
4 coll.push_back(std::move(optStr).value()); // OK, moves from member into coll
```

注意，*std::optional<>* 是 C++ 标准库中少数使用 *const rvalue* 引用的类型。原因是 *std::optional<>* 是一个包装器类型，它希望确保操作正确，即使 *const* 对象标记为 *std::move()*，并且所包含的类型为 *const rvalue* 引用提供了特殊行为。

15.4 智能指针的移动语义

原始指针不能从移动语义中获益 (它们的地址值总是副本)，而智能指针可以从移动语义中获益。

- 共享指针 (`std::shared_ptr<T>`) 支持移动语义，因为移动共享指针比复制共享指针廉价得多。
- 唯一指针 (`std::unique_ptr<T>`) 甚至只支持移动语义，因为不可能复制 unique 指针。

15.4.1 `std::shared_ptr<T>` 的移动语义

共享指针有共享所有权的概念。多个共享指针可以“拥有”同一个对象，当最后一个所有者销毁 (或获得一个新值) 时，将调用所拥有对象的“删除器”。

例如：

```
1 {
2     std::shared_ptr<int> sp1; // init shared pointer that does not own anything
3     {
4         auto sp2{std::make_shared<int>(42)}; // init shared pointer that owns new int
5         ...
6         sp1 = sp2; // sp1 and sp2 now share ownership
7         ...
8         *sp2 = 77; // modify value via sp2
9         ...
10    } // sp2 destroyed, sp1 is the only owner
11    std::cout << *sp1 << '\n'; // use modified value via sp1
12 } // last owner destroyed so that delete is called
```

这个例子中，赋值操作符复制了 `int` 对象的所有。之后，两个共享指针都拥有该对象。但请注意，复制所有权是非常昂贵的操作。这是因为计数器必须跟踪所有者的数量：

- 每次复制一个共享指针时，所有者计数器就递增
- 每次销毁共享指针或赋值时，所有者计数器就递减

此外，修改计数器的值代价很高，因为修改是原子操作，可以避免多线程处理拥有相同对象的共享指针时出现的问题。

因此，通过引用遍历共享指针集合要廉价得多：

```
1 std::vector<std::shared_ptr<...>> coll;
2 ...
3 for (auto sp : coll) { // expensive
4     ...
5 }
6 ...
7 for (const auto& sp : coll) { // cheap
8     ...
9 }
```

对于移动语义，最好是移动共享指针而不是复制它们。例如，不要这样实现：

```
1 std::shared_ptr<int> lastPtr; // init shared pointer that does not own anything
```

```

2 while ( ... ) {
3     auto ptr{std::make_shared<int>(getValue())}; // init shared pointer that owns new
4     int
5     ...
6     lastPtr = ptr; // expensive (note: ptr no longer used)
} // ptr destroyed, lastPtr is the only owner

```

最好这样做:

```

1 std::shared_ptr<int> lastPtr; // init shared pointer that does not own anything
2 while ( ... ) {
3     auto ptr{std::make_shared<int>(getValue())}; // init shared pointer that owns new
4     int
5     ...
6     lastPtr = std::move(ptr); // cheap
} // ptr destroyed, lastPtr is the only owner

```

因此,当对象移动时,具有共享指针成员的对象将失去这些成员所指向的资源的所有权。这对性能有好处,但会创建无效的从状态移动。因此,应该对拥有 `std::shared_ptr<>` 类型成员的已移动对象的状态进行双重检查。

15.4.2 `std::unique_ptr<>` 的移动语义

类模板 `std::unique_ptr<>` 实现了独占所有权的概念。类型系统确保一个对象在任何时候只能有一个所有者。技巧是使用类型系统来禁用复制唯一指针。因为这个检查是在编译时完成的,所以这种方法不会在运行时引入任何显著的性能开销。

可以在函数中创建 unique 指针,并将其返回给调用者:

`lib/uniqueptr1.cpp`

```

1 #include <iostream>
2 #include <string>
3 #include <memory>
4
5 std::unique_ptr<std::string> source()
6 {
7     static long id{0};
8     // create string with new and let ptr own it:
9     auto ptr = std::make_unique<std::string>("obj" + std::to_string(++id));
10    ...
11    return ptr; // transfer ownership to caller
12 }
13
14 int main()
15 {
16     std::unique_ptr<std::string> p;
17     for (int i = 0; i < 10; ++i) {
18         p = source(); // p gets ownership of the returned object
19                     //((previously returned object of source() is deleted))
20         std::cout << *p << '\n';

```

```
21     ...
22 }
23 } // last-owned object of p is deleted
```

和已移动类型一样，要将所有权传递给接收器函数，只能通过 `std::move()` 传递：

```
1 std::vector<std::unique_ptr<std::string>> coll;
2 std::unique_ptr<std::string> up;
3 ...
4 coll.push_back(up); // ERROR: copying disabled
5 coll.push_back(std::move(up)); // OK, moves ownership into new element of coll
```

如果传递 unique 指针给一个通过引用接受参数的潜在接收器函数，则不知道所有权是否已经移动。所有权是否转移取决于功能的实现。这种情况下，可以用 `bool()` 操作符再次检查状态：

```
1 std::unique_ptr<std::string> up;
2 ...
3 sink(std::move(up)); // might move ownership to sink()
4 if (up) { // does it still have the ownership?
5     ...
6 }
```

或者，可以确保所有权消失（资源释放）：

```
1 std::unique_ptr<std::string> up;
2 ...
3 sink(std::move(up)); // might move ownership to sink()
4 up.reset(); // ensure ownership is gone (resource deleted)
```

15.5 输入输出流的移动语义

IOStreams 在 C++98 中引入，作为一种可以读写的抽象（标准 I/O，文件，甚至字符串）。这是早期的设定，不可能复制这些对象（复制打开的文件的对象意味着什么，对相同的文件有两个句柄或复制文件，以及如何同步访问？）

自从 C++11 以来，移动语义允许我们移动 IOStream 对象，并使用临时流。

15.5.1 移动 IOStream 对象

考虑如下示例：

lib/outfile.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <stream>
4
5 std::ofstream openToWrite(const std::string& name)
6 {
7     std::ofstream file(name); // open a file to write to
8     if (!file) {
9         std::cerr << "can't open file '" << name << "'\n";
```

```

10     std::exit(EXIT_FAILURE);
11 }
12 return file; // return ownership (open file)
13 }
14
15 void storeData(std::ofstream fstrm) // takes ownership of file (but this might
16   change)
17 {
18     fstrm << 42 << '\n';
19 } // closes the file
20
21 int main()
22 {
23     auto outFile{openToWrite("iostream.tmp")}; // open file
24     storeData(std::move(outFile)); // store data
25
26     // better ensure that the file is closed:
27     if (outFile.is_open()) {
28         outFile.close();
29     }

```

这里，函数 `openToWrite()` 打开并返回输出文件流:

```

1 std::ofstream openToWrite(const std::string& name)
2 {
3     std::ofstream file(name); // open a file to write to
4     ...
5     return file; // return ownership (open file)
6 }

```

使用返回值初始化 `outFile`，并将其传递给 `storeData()`:

```

1 auto outFile{openToWrite("iostream.tmp")}; // open file
2 storeData(std::move(outFile)); // store data

```

因为 `storeData()` 按值接受参数，所以它有打开文件的所有权。因此，在 `storeData()` 的末尾，文件关闭:

```

1 void storeData(std::ofstream fstrm) // takes ownership of file (but this might
2   change)
3 {
4     ...
5 } // closes the file

```

但是，`storeData()` 也可以通过引用来获取参数，这意味着它不一定需要获取所有权。这种情况下，需要再次检查传递参数 `outFile` 的状态:

```

1 // better ensure that the file is closed:
2 if (!outFile.is_closed()) {
3     outFile.close();

```

```
4 }
```

调用 `outFile.close()` 通常就足够了，但如果文件流已经关闭，则会设置该文件流的 `failbit`。

15.5.2 使用临时 IOStreams

自 C++11 开始，IOStreams 库也提供了函数重载来接受 rvalue 引用，允许接受临时对象。例如：

```
1 std::string s = "hello, world";
2 std::ofstream("fstream1.tmp") << s << '\n'; // OK since C++11
```

甚至可以这样向流中写一个字符串字面值 (使用 C++11 标准之前编译，其使用 operator«(const void*) 输出字符串字面值的地址)：

```
1 std::ofstream("fstream1.tmp") << "hello, world\n"; // correct since C++11
2 // (wrote address before)
```

同样，可以用临时字符串流来解析给定的字符串：

```
1 std::string name, firstname, lastname;
2 ...
3 name = "Tina Turner";
4 std::istringstream{name} >> firstname >> lastname; // OK since C++11
```

最后，可以使用 `std::getline()` 来解析临时流的第一行：

```
1 std::string multiLineString, firstLine;
2 ...
3 std::getline(std::istringstream{multiLineString}, // read from temporary string
             firstLine);
4 firstLine);
```

15.6 多线程的移动语义

C++ 标准库有两个特殊的类型，代表正在运行的线程或用于同步。其中一些既不能复制也不能移动 (例如，原子类型或条件变量)。然而，其中一些是只移动类型。让我们从移动语义的角度来看一些细节。

15.6.1 `std::thread<>` 和 `std::jthread<>`

表示运行线程的对象 (`std::thread` 或 C++20 `std::jthread`) 可以传递，但不能复制。可以把它放在容器里。

考虑以下例子：

`lib/thread.cpp`

```
1 #include <iostream>
2 #include <thread>
3 #include <vector>
4
```

```

5 void doThis(const std::string& arg) {
6     std::cout << "doThis(): " << arg << '\n';
7 }
8 void doThat(const std::string& arg) {
9     std::cout << "doThat(): " << arg << '\n';
10}
11
12 int main()
13{
14    std::vector<std::thread> threads; // better std::jthread since C++20
15
16    // start a couple of threads:
17    std::string someArg{ "Black Lives Matter" };
18    threads.push_back(std::thread{doThis, someArg});
19    threads.push_back(std::thread{doThat, std::move(someArg)} );
20    ...
21
22    // wait for all threads to end:
23    for (auto& t : threads) {
24        t.join();
25    }
26}

```

启动两个线程，并将它们放入所有运行线程的集合中：

```

1 std::vector<std::thread> threads; // better std::jthread since C++20
2
3 std::string someArg{ "black lives matter" };
4 threads.push_back(std::thread{doThis, someArg});
5 threads.push_back(std::thread{doThat, std::move(someArg)} );

```

通过声明 `std::thread` 类型的对象 (从 C++20 开始，最好使用 `std::jthread`) 来启动一个线程，并将临时对象移动到线程中。在可调用对象 (函数, Lambda, 函数对象) 的参数后面，可以传递额外的参数，这些参数在线程启动时传递给可调用对象。默认情况下，线程类的构造函数会复制这些参数。使用 `std::move()` 切换到移动语义。因此，`doThis()` 获得传递给 `someArg` 字符串的副本，而 `doThat()` 获得该字符串的移动值。

最后，等待所有线程的结束 (使用 `std::thread` 时，这是避免段错误的必要条件)：

```

1 // wait for all threads to end:
2 for (auto& t : threads) {
3     t.join();
4 }

```

15.6.2 Future, Promise 和打包任务

对于一些用于同步两个线程之间 (返回) 值交换的 helper 类型，还使用了只移动类型 (`future`、`promise` 和打包任务都是只移动类型)。

考虑以下例子：

lib/future.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <thread>
5 #include <future>
6
7 void getValue(std::promise<std::string> p)
8 {
9     try {
10         std::string ret{"vote"};
11         ...
12         // store result:
13         p.set_value_at_thread_exit(std::move(ret));
14     }
15     catch (...) {
16         // store exception:
17         p.set_exception_at_thread_exit(std::current_exception());
18     }
19 }
20 int main()
21 {
22     std::vector<std::future<std::string>> results;
23
24     // create promise and future to deal with outcome of the thread started:
25     std::promise<std::string> p;
26     std::future<std::string> f{p.get_future()};
27     results.push_back(std::move(f));
28
29     // start thread and move the promise to it:
30     std::thread t{getValue, move(p)};
31     t.detach(); // would not be necessary for std::jthread
32     ...
33
34     // wait for all threads to end:
35     for (auto& fut : results) {
36         std::cout << fut.get() << '\n';
37     }
38 }
```

main() 中，首先创建了 promise，即能够将结果（值或异常）发送给能够读取它的关联 future：

```
1 std::promise<std::string> p;
2 std::future<std::string> f{p.get_future()};
```

future 移动到 future 的集合中：

```
1 results.push_back(std::move(f));
```

也可以将 `p.get_future()` 的结果直接传递给 `push_back()`：

```
1 results.push_back(p.get_future());
```

promise 移动到已启动的线程:

```
1 std::thread t{getValue, move(p)};
```

线程将作为 `getValue()` 的参数, `getValue()` 按值接受传递的 promise:

```
1 void getValue(std::promise<std::string> p)
2 {
3     ...
4 }
```

通过这种方式, `getValue()` 接收 promise 的所有权, 并在线程结束时销毁通信机制的源。

`getValue()` 也可以通过引用的方式获取 promise, 因为启动的线程保存着已移动的 promise 值, 直到结束为止。

15.7 总结

从本章讨论的类型中可以学到很多东西, 在这里总结一下在 C++ 标准库的类型中使用移动语义的几个通用的方面。

- 已移动的标准字符串通常为空, 但不能保证。
- 如果没有使用特殊的分配器, 则标准容器 (`std::array<>` 除外) 通常为空。对于向量, 这是可(间接)保证的; 对于其他容器, 可以间接地保证所有元素都移除或销毁, 插入新成员没有意义。
- 移动赋值可以改变字符串和 `vector` 的容量。
- 为了在向通用/转发引用传递值时支持衰退(对于不同长度的字符串, 通常需要推导出相同的类型), 使用类型特征 `std::decay<>`。
- 泛型包装器类型应该使用引用限定符重载访问包装对象的成员函数。这甚至可能意味着对 `const rvalue` 引用使用重载。
- 避免复制共享指针(例如, 通过值传递它们)。
- 用 `std::jthread(C++20 之后可用)` 代替 `std::thread`。