资深C++专家、C++11布道师、金山软件资深工程师撰写

深度剖析C++11中最常用新特性，从程序简洁性、性能、代码质量、内存泄露、多线程等多方面给出了代码优化的方法和建议

深入讲解了C++11在线程池开发、流行框架和库的开发、库的封装等各种工程级项目中的应用，包含大量实现源码并开源，可直接使用

华章原创精品

深入应用C++11

代码优化与工程级应用

In-Depth C++ 11

Code Optimization and Engineering Level Application

祁宇 著

机械工业出版社
China Machine Press

华章原创精品

深入应用 C++11：代码优化与工程级应用

祁　宇　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入应用 C++11：代码优化与工程级应用 / 祁宇著 . —北京：机械工业出版社，2015.5
（华章原创精品）

ISBN 978-7-111-50069-8

I. 深… II. 祁… III. C 语言 – 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2015）第 085822 号

深入应用 C++11：代码优化与工程级应用

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：姜 影 责任校对：董纪丽
印 刷： 版 次：2015 年 5 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：26.75
书 号：ISBN 978-7-111-50069-8 定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Preface 前　言

为什么要写这本书

2011 年 C++11 标准刚发布时，广大 C++ 开发者奔走相告，我也在第一时间看了 C++ 之父 Bjarne Stroustrup 的 C++11 FAQ（http://www.stroustrup.com/C++11FAQ.html），虽然只介绍了一部分特性，而且特性的用法介绍也很简短，但给我带来三个震撼：第一个震撼是发现我几乎不认识 C++ 了，这么多新特性，与以前的 C++ 很不同；第二个震撼是很多东西和其他语言类似，比如 C# 或者 Java，感觉很酷；第三个震撼是很潮，比如 lambda 特性，Java 都还没有（那时 Java 8 还没出来），C++11 已经有了。我是一个喜欢研究新技术的人，一下子就被 C++ 那么多新特性吸引住了，连续几天都在看 FAQ，完全着迷了，虽然当时有很多地方没看明白，但仍然很兴奋，因为我知道这就是我想要的 C++。我马上更新编译器尝鲜，学习新特性。经过一段时间的学习，在对一些主要特性有一定的了解之后，我决定在新项目中使用 C++11。用 C++11 的感觉非常好：有了 auto 就不用写冗长的类型定义了，有了 lambda 就不用定义函数对象了，算法也用得更舒服和自然，初始化列表让容器和初始化变得很简便，还有右值引用、智能指针和线程等其他很棒的特性。C++11 确实让项目的开发效率提高了很多。

相比 C++98/03，C++11 做了大幅度的改进，增加了相当多的现代编程语言的特性，使得 C++ 的开发效率有了很大的提高。比如，C++11 增加了右值引用，可以避免无谓的复制，从而提高程序性能；C++11 增加了可变模板参数，使 C++ 的泛型编程能力更加强大，也大幅消除了重复模板定义；C++11 增加了 type_traits，可以使我们很方便地在编译期对类型进行计算、查询、判断、转换和选择；C++11 中增加的智能指针使我们不用担心内存泄露问题了；C++11 中的线程库让我们能很方便地编写可移植的并发程序。除了这些较大的改进之外，C++11 还增加了很多其他实用、便利的特性，提高了开发的便利性。对于一个用过 C# 的开发者来说，

学习 C++11 一定会有一种似曾相识的感觉，比如 C++11 的 auto、for-loop 循环、lambda 表达式、初始化列表、tuple 等分别对应了 C# 中的 var、for-loop 循环、lambda 表达式、初始化列表、tuple，这些小特性使我们编写 C++ 程序更加简洁和顺手。C++11 增加的这些特性使程序编写变得更容易、更简洁、更高效、更安全和更强大，那么我们还有什么理由不去学习这些特性并充分享受这些特性带来的好处呢？

学习和使用 C++11 不要背着 C++ 的历史包袱，要轻装上阵，把它当作一门新的语言来学习，才能发现它的魅力和学习的乐趣。C++11 增加的新特性有一百多项，很多人质疑这会使本已复杂的 C++ 语言变得更加复杂，从而产生一种抗拒心理，其实这是对 C++11 的误解，C++11 并没有变得更复杂，恰恰相反，它在做简化和改进！比如 auto 和 decltype 可以用来避免写冗长的类型，bind 绑定器让我们不用关注到底是用 bind1st 还是 bind2nd 了，lambda 表达式让我们可以不必写大量的不易维护的函数对象等。

语言都是在不断进化之中的，只有跟上时代潮流的语言才是充满活力与魅力的语言。C++ 正是这样一门语言，虽然它已经有三十多年的历史了，但是它还在发展之中。C++14 标准已经制定完成，C++17 也提上了日程，我相信 C++ 的未来会更加美好，C++ 开发者的日子也会越来越美好！

作为比较早使用 C++11 的开发者，我开始在项目中应用 C++11 的时候，可以查阅的资料还很有限，主要是通过 ISO 标准（ISO/IEC 14882:2011）、维基百科、MSDN 和 http://en.cppreference.com/w/ 等来学习 C++11。然而，这些资料对新特性的介绍比较零散，虽然知道这些新特性的基本用法，但有时候不知道为什么需要这个新特性，在实际项目中该如何应用，或者说最佳实践是什么，这些东西网上可没有，也没有人告诉你，因为当时只有很少的人在尝试用 C++11，这些都需要自己不断地去实践、去琢磨，当时多么希望能有一些指导 C++11 实践的资料啊。在不断实践的过程中，我对 C++11 的认识加深了，同时，也把应用 C++11 的一些心得和经验放到我的技术博客（http://www.cnblogs.com/qicosmos/）上分享出来，还开源了不少 C++11 的代码，这些代码大多来自于项目实践。技术分享得到了很多认识的或不认识的朋友的鼓励与支持，曾经不止一个人问过我同一个问题，你坚持写博客分享 C++11 技术是为了什么，有什么好处吗？我想最重要的原因就是 C++11 让我觉得 C++ 语言是非常有意思和有魅力的语言，不断给人带来惊喜，在窥探到 C++11 的妙处之后，我很想和更多的人分享，让更多的人领略 C++11 的魅力。另外一个原因是我的一点梦想，希望 C++ 的世界变得更加美好，C++ 开发者的日子变得更美好。我希望这些经验能帮助学习 C++11 的朋友，让他们少走弯路，快速地将 C++11 应用起来，也希望这些代码能为使用 C++ 的朋友带来便利，解决他们的实际问题。

“独乐乐，与人乐乐，孰乐乎？与少乐乐，与众乐乐，孰乐？”，这是我分享技术和写作此书的初衷。

读者对象

❑ C++ 开发人员。

C++11 新标准发布已经 4 年了，C++11 的使用也越来越普及，这是大势所趋，普通的 C++ 开发者不论是新手还是老手，都有必要学习和应用 C++11，C++11 强大的特性可以大幅提高生产率，让我们开发项目更加得心应手。

❑ C++11 爱好者。

其他语言的开发人员，比如 C# 或者 Java 开发人员，想转到 C++ 开发正是时机，因为新标准的很多特性，C# 和 Java 中也有，学起来也并不陌生，可以乘着新标准的“轻舟”学习 C++11，事半功倍，正当其时。

如何阅读本书

虽然 C++11 的目的是为了提高生产率，让 C++ 变得更好用和更强大，但是，这些新特性毕竟很多，面对这么多特性，初学者可能会茫然无措，找不到头绪。如果对着这些特性一一去查看标准，不仅枯燥乏味，还丧失了学习的乐趣，即使知道了新特性的基本用法，却不知道如何应用到实际开发中。针对这两个问题，本书试图另辟蹊径来解决。本书的前半部分将从另外一个角度去介绍这些新特性，不追求大而全，将重点放在一些常用的 C++11 特性上，有侧重地从另外一个角度将这些特性分门别类，即从利用这些新特性如何去改进我们现有程序的角度介绍。这种方式一来可以让读者掌握这些新特性的用法；二来还可以让读者知道这些特性是如何改进现有程序的，从而能更深刻地领悟 C++11 的新特性。

如果说本书的前半部分贴近实战，那么本书后半部分的工程级应用就是真正的实战。后半部分将通过丰富的开发案例来介绍如何用 C++11 去开发项目，因为只有在实战中才能学到真东西。后半部分实战案例涉及面比较广，是笔者近年来使用 C++11 的经验与心得的总结。这些实践经验是针对实际开发过程中遇到的问题来选取的，它们的价值不仅可以作为 C++11 实践的指导，还可以直接在实际开发中应用（本书开发案例源码遵循 LGPL 开源协议），相信这些实战案例一定能给读者带来更深入的思考。

通过学习本书基础知识与实战案例，相信读者一定能掌握大部分 C++11 新特性，并能应用于自己的实际开发中，充分享受 C++11 带来的好处。

C++ 之父 BjarneStroustrup 曾说过：C++11 看起来像一门新的语言。这个说法是否夸张，读者不妨看完本书之后再来回味这句话。

本书示例代码需要支持 C++11 的编译器：

- Windows：Visual Studio 2013。
- Linux：GCC 4.8+ 或者 Clang 3.4。

由于少数代码用到了 boost 库，还需要编译 boost 1.53 或最新的 boost 库。

勘误和支持

除封面署名外，张轶（木头云）参与了第 1 章大部分内容和 7.4 节的整理，还负责了本书大部分的审稿工作。由于笔者的水平有限，书中错漏之处在所难免，敬请读者批评指正，如有更多宝贵意见请发到我的邮箱 cpp11book@163.com，同时，我们也会把本书的勘误集中公布在我的博客上（http://www.cnblogs.com/qicosmos/）。本书中有少数内容来自 en.cppreference.com、MSDN 和 http://www.ibm.com/developerworks/cn/，以及一些网络博客，虽然大部分都注明了出处，但也可能存在疏漏，如果有些内容引用了但没注明出处，请通过邮箱 cpp11book@163.com 与我联系。

书中的全部源文件除可以从华章网站[㊀]下载外，还可以从 github(https://github.com/qicosmos/cosmos) 上下载，同时我也会将相应的功能更新及时更正出来。

致谢

首先感谢你选择本书，相信本书会成为你学习和应用 C++11 的良师益友。

感谢 C++ 之父 Bjarne Stroustrup 和 C++ 标准委员会，正是他们推动着 C++ 不断改进和完善，才使 C++ 变得更有魅力。

还要感谢一些热心朋友的支持，其中，史建鑫、于洋子、吴楚元、胡宾朔、钟郭福、林曦和翟懿奎审阅了部分章节的内容，并提出了宝贵的意见；还要感谢刘威提供了一些论文资料。

感谢机械工业出版社华章公司的两位编辑杨福川和姜影，在这一年多的时间里始终支持我的写作，他们的帮助与鼓励引导我能顺利完成全部书稿。

接下来我要感谢我的家人：感谢我的父母和妻子，没有他们承担所有的家务和照顾孩子，我不可能完成此书；感谢弟弟和弟妹对我的鼓励与支持。还要对一岁多的女儿说声抱歉，为

㊀ 参见华章网站 www.hzbook.com.——编辑注

了完成本书，已经牺牲了很多陪女儿玩耍的时间，记得女儿经常跑到我写作的书房拉着我的手往外走，边走边说：“爸爸一起玩一下”。在这要对我的家人说声抱歉，在这一年的时间里，由于专注于写作，对他们一直疏于关心和照顾。

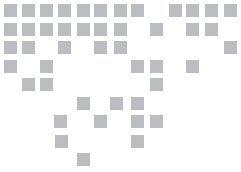
谨以此书献给我最亲爱的家人，以及众多热爱 C++11 的朋友们！

祁宇（qicosmos）

目　录 Contents

第一篇 Part 1

C++11 改进我们的程序



Chapter 1 第 1 章

使用 C++11 让程序更简洁、更现代

本章要讲到的 C++11 特性可以使程序更简洁易读，也更现代。通过这些新特性，可以更方便和高效地撰写代码，并提高开发效率。

用过 C# 的读者可能觉得 C# 中的一些特性非常好用，可以让代码更简洁、易读。比如 var 可以在编译期自动推断出变量的类型；range-base for 循环非常简洁清晰；构造函数初始化列表使创建一个对象变得非常方便；lambda 表达式可以简洁清晰地就定义短小的逻辑，等等。

现在的 C++11 中也增加了类似的特性，不仅实现了上面的这些功能，而且在一些细节的表现上更加灵活。比如 auto 不仅可以自动推断变量类型，还能结合 decltype 来表示函数的返回值。这些新特性可以让我们写出更简洁、更现代的代码。

1.1 类型推导

C++11 引入了 auto 和 decltype 关键字实现类型推导，通过这两个关键字不仅能方便地获取复杂的类型，而且还能简化书写，提高编码效率。

1.1.1 auto 类型推导

1. auto 关键字的新意义

用过 C# 的读者可能知道，从 Visual C# 3.0 开始，在方法范围中声明的变量可以具有隐式类型 var。例如，下面这样的写法（C# 代码）：

```
var i = 10;                    //隐式（implicitly）类型定义
```

```
int i = 10;                          // 显式（explicitly）类型定义
```

其中，隐式的类型定义也是强类型定义，前一行的隐式类型定义写法和后一行的显式写法是等价的。

不同于 Python 等动态类型语言的运行时变量类型推导，隐式类型定义的类型推导发生在编译期。它的作用是让编译器自动推断出这个变量的类型，而不需要显式指定类型。

现在，C++11 中也拥有了类似的功能：auto 类型推导。其写法与上述 C# 代码等价：

```
auto i = 10;
```

是不是和 C# 的隐式类型定义很像呢？

下面看下 auto 的一些基本用法[⊖]：

```
auto x = 5;                          // OK: x 是 int 类型
auto pi = new auto(1);               // OK: pi 被推导为 int*
const auto *v = &x, u = 6;           // OK: v 是 const int* 类型，u 是 const int 类型
static auto y = 0.0;                 // OK: y 是 double 类型
auto int r;                          // error: auto 不再表示存储类型指示符
auto s;                              // error: auto 无法推导出 s 的类型
```

在上面的代码示例中：字面量 5 是一个 const int 类型，变量 x 将被推导为 int 类型（const 被丢弃，后面说明），并被初始化为 5 ；pi 的推导说明 auto 还可以用于 new 操作符。在例子中，new 操作符后面的 auto(1) 被推导为 int(1)，因此 pi 的类型是 int* ；接着，由 &x 的类型为 int*，推导出 const auto* 中的 auto 应该是 int，于是 v 被推导为 const int*，而 u 则被推导为 const int。

v 和 u 的推导需要注意两点：

- 虽然经过前面 const auto*v=&x 的推导，auto 的类型可以确定为 int 了，但是 u 仍然必须要写后面的“=6”，否则编译器不予通过。
- u 的初始化不能使编译器推导产生二义性。例如，把 u 的初始化改成“ u=6.0”，编译器将会报错：

```
const auto *v = &x, u = 6.0;
error: inconsistent deduction for 'const auto': 'int' and then
'double'
```

最后 y、r、s 的推导过程比较简单，就不展开讲解了。读者可自行在支持 C++11 的编译器上实验。

由上面的例子可以看出来，auto 并不能代表一个实际的类型声明（如 s 的编译错误），只是一个类型声明的“占位符”。

使用 auto 声明的变量必须马上初始化，以让编译器推断出它的实际类型，并在编译时将 auto 占位符替换为真正的类型。

⊖ 部分示例来自 ISO/IEC 14882:2011，7.1.6.4 auto specifier，第 3 款。

细心的读者可能会发现，auto 关键字其实并不是一个全新的关键字。在旧标准中，它代表“具有自动存储期的局部变量”，不过其实它在这方面的作用不大，比如：

```
auto int i = 0;          //C++98/03，可以默认写成 int i = 0;
static int j = 0;
```

上述代码中的 auto int 是旧标准中 auto 的使用方法。与之相对的是下面的 static int，它代表了静态类型的定义方法。

实际上，我们很少有机会这样直接使用 auto，因为非 static 的局部变量默认就是“具有自动存储期的”[㊀]。

考虑到 auto 在 C++ 中使用的较少，在 C++11 标准中，auto 关键字不再表示存储类型指示符（storage-class-specifiers，如上文提到的 static，以及 register、mutable 等），而是改成了一个类型指示符（type-specifier），用来提示编译器对此类型的变量做类型的自动推导。

2. auto 的推导规则

从上一节的示例中可以看到 auto 的一些使用方法。它可以同指针、引用结合起来使用，还可以带上 cv 限定符（cv-qualifier，const 和 volatile 限定符的统称）。

再来看一组例子：

```
int x = 0;

auto * a = &x;           //a -> int*，auto 被推导为 int
auto   b = &x;           //b -> int*，auto 被推导为 int*
auto & c = x;            //c -> int&，auto 被推导为 int
auto   d = c;            //d -> int ，auto 被推导为 int

const auto e = x;        //e -> const int
auto f = e;              //f -> int

const auto& g = x;       //e -> const int&
auto& h = g;             //f -> const int&
```

由上面的例子可以看出：

- a 和 c 的推导结果是很显然的，auto 在编译时被替换为 int，因此 a 和 c 分别被推导为 int* 和 int&。
- b 的推导结果说明，其实 auto 不声明为指针，也可以推导出指针类型。
- d 的推导结果说明当表达式是一个引用类型时，auto 会把引用类型抛弃，直接推导成原始类型 int。
- e 的推导结果说明，const auto 会在编译时被替换为 const int。
- f 的推导结果说明，当表达式带有 const（实际上 volatile 也会得到同样的结果）属性时，

㊀ ISO/IEC 14882:2003，7.1.1 Storage class specifiers，第 2 款。

auto 会把 const 属性抛弃掉，推导成 non-const 类型 int。

- g、h 的推导说明，当 auto 和引用（换成指针在这里也将得到同样的结果）结合时，auto 的推导将保留表达式的 const 属性。

通过上面的一系列示例，可以得到下面这两条规则：

1）当不声明为指针或引用时，auto 的推导结果和初始化表达式抛弃引用和 cv 限定符后类型一致。

2）当声明为指针或引用时，auto 的推导结果将保持初始化表达式的 cv 属性。

看到这里，对函数模板自动推导规则比较熟悉的读者可能会发现，auto 的推导和函数模板参数的自动推导有相似之处。比如上面例子中的 auto，和下面的模板参数自动推导出来的类型是一致的：

```
template <typename T> void func(T   x) {}       //T   -> auto
template <typename T> void func(T * x) {}       //T * -> auto *
template <typename T> void func(T & x) {}       //T & -> auto &

template <typename T> void func(const T   x) {}//const T   ->
const auto
template <typename T> void func(const T * x) {}//const T * ->
const auto *
template <typename T> void func(const T & x) {}//const T & ->
const auto &
```

注意：auto 是不能用于函数参数的。上面的示例代码只是单纯比较函数模板参数推导和 auto 推导规则的相似处。

因此，在熟悉 auto 推导规则时，可以借助函数模板的参数自动推导规则来帮助和加强理解。

3. auto 的限制

上一节提到了 auto 是不能用于函数参数的。那么除了这个之外，还有哪些限制呢？

请看下面的示例，如代码清单 1-1 所示。

代码清单 1-1　auto 使用受限的示例

```
void func(auto a = 1) {}                        //error: auto 不能用于函数参数

struct Foo
{
   auto var1_ = 0;                              //error: auto 不能用于非静态成员变量
   static const auto var2_ = 0;                 //OK: var2_ -> static const int
};

template <typename T>
struct Bar {};

int main(void)
{
   int arr[10] = {0};
   auto aa     = arr;                           //OK: aa -> int *
   auto rr[10] = arr;                           //error: auto 无法定义数组
```

```
    Bar<int> bar;
    Bar<auto> bb = bar;                          // error: auto 无法推导出模板参数

    return 0;
}
```

在 Foo 中，auto 仅能用于推导 static const 的整型或者枚举成员（因为其他静态类型在 C++ 标准中无法就地初始化[㊀]），虽然 C++11 中可以接受非静态成员变量的就地初始化，但却不支持 auto 类型非静态成员变量的初始化。

在 main 函数中，auto 定义的数组 rr 和 Bar<auto>bb 都是无法通过编译的。

main 函数中的 aa 不会被推导为 int[10]，而是被推导为 int*。这个结果可以通过上一节中 auto 与函数模板参数自动推导的对比来理解。

4. 什么时候用 auto

前面说了这么多，最重要的是，应该在什么时候使用 auto 呢？

在 C++11 之前，定义了一个 stl 容器以后，在遍历的时候常常会写这样的代码：

```
#include <map>

int main(void)
{
    std::map<double, double> resultMap;

    // ...

    std::map<double,double>::iterator it = resultMap.begin();
    for(; it != resultMap.end(); ++it)
    {
    // do something
    }

    return 0;
}
```

观察上面的迭代器（iterator）变量 it 的定义过程，总感觉有点憋屈。其实通过 resultMap.begin()，已经能够知道 it 的具体类型了，却非要书写上长长的类型定义才能通过编译。

来看看使用了 auto 以后的写法：

```
#include <map>
```

㊀ ISO/IEC 14882:2011，9.4.2 Static data members，第 3 款。

```
int main(void)
{
   std::map<double, double> resultMap;

   // ...

   for(auto it = resultMap.begin(); it != resultMap.end(); ++it)
   {
   // do something
   }

   return 0;
}
```

再次观察 it 的定义过程，是不是感到清爽了很多？

再看一个例子，在一个 unordered_multimap 中查找一个范围，代码如下：

```
#include <map>

int main(void)
{
   std::unordered_multimap<int, int>resultMap;

   // ...

   std::pair<std::unordered_multimap<int,int>::iterator,
std::unordered_multimap<int, int>::iterator>
   range = resultMap.equal_range(key);

   return 0;
}
```

这个 equal_range 返回的类型声明显得烦琐而冗长，而且实际上并不关心这里的具体类型（大概知道是一个 std::pair 就够了）。这时，通过 auto 就能极大的简化书写，省去推导具体类型的过程：

```
#include <map>

int main(void)
{
   std::unordered_multimap<int, int> map;

   // ...

   auto range = map.equal_range(key);

   return 0;
}
```

另外，在很多情况下我们是无法知道变量应该被定义成什么类型的，比如，如代码清单 1-2 所示的例子。

代码清单 1-2 auto 简化函数定义的示例

```
class Foo
{
public:
   static int get(void)
   {
      return 0;
   }
};

class Bar
{
public:
   static const char* get(void)
   {
      return "0";
   }
};

template <class A>
void func(void)
{
   auto val = A::get();

   // ...
}

int main(void)
{
   func<Foo>();
   func<Bar>();
   return 0;
}
```


在这个例子里，我们希望定义一个泛型函数 func，对所有具有静态 get 方法的类型 A，在得到 get 的结果后做统一的后续处理。若不使用 auto，就不得不对 func 再增加一个模板参数，并在外部调用时手动指定 get 的返回值类型。

上面给出的各种示例仅仅只是实际应用中很少的一部分，但也足以说明 auto 关键字的各种常规使用方法。更多的适用场景，希望读者能够在实际的编程中亲身体验。

auto 是一个很强大的工具，但任何工具都有它的两面性。不加选择地随意使用 auto，会带来代码可读性和维护性的严重下降。因此，在使用 auto 的时候，一定要权衡好它带来的“价值”和相应的“损失”。

1.1.2　decltype 关键字

1. 获知表达式的类型

上一节所讲的 auto，用于通过一个表达式在编译时确定待定义的变量类型，auto 所修饰的变量必须被初始化，编译器需要通过初始化来确定 auto 所代表的类型，即必须要定义变量。若仅希望得到类型，而不需要（或不能）定义变量的时候应该怎么办呢？

C++11 新增了 decltype 关键字，用来在编译时推导出一个表达式的类型。它的语法格式如下：

```
decltype(exp)
```

其中，exp 表示一个表达式（expression）。

从格式上来看，decltype 很像 sizeof——用来推导表达式类型大小的操作符。类似于 sizeof，decltype 的推导过程是在编译期完成的，并且不会真正计算表达式的值。

那么怎样使用 decltype 来得到表达式的类型呢？让我们来看一组例子：

```
int x = 0;
decltype(x) y = 1;          //y -> int
decltype(x + y) z = 0;      //z -> int

const int& i = x;
decltype(i) j = y;          //j -> const int &

const decltype(z) * p = &z; //*p  -> const int, p  -> const int *
decltype(z) * pi = &z;      //*pi -> int       , pi -> int *
decltype(pi)* pp = &pi;     //*pp -> int *     , pp -> int * *
```

y 和 z 的结果表明 decltype 可以根据表达式直接推导出它的类型本身。这个功能和上一节的 auto 很像，但又有所不同。auto 只能根据变量的初始化表达式推导出变量应该具有的类型。若想要通过某个表达式得到类型，但不希望新变量和这个表达式具有同样的值，此时 auto 就显得不适用了。

j 的结果表明 decltype 通过表达式得到的类型，可以保留住表达式的引用及 const 限定符。实际上，对于一般的标记符表达式（id-expression），decltype 将精确地推导出表达式定义本身的类型，不会像 auto 那样在某些情况下舍弃掉引用和 cv 限定符。

p、pi 的结果表明 decltype 可以像 auto 一样，加上引用和指针，以及 cv 限定符。

pp 的推导则表明，当表达式是一个指针的时候，decltype 仍然推导出表达式的实际类型（指针类型），之后结合 pp 定义时的指针标记，得到的 pp 是一个二维指针类型。这也是和 auto 推导不同的一点。

对于 decltype 和引用（&）结合的推导结果，与 C++11 中新增的引用折叠规则（Reference Collapsing）有关，因此，留到后面的 2.1 节右值引用（Rvalue Reference）时再详细讲解。

小知识 关于 p、pi、pp 的推导，有个很有意思的地方。像 MicrosoftVisual Studio 这样的 IDE，可以在运行时观察每个变量的类型。我们可以看到 p 的显示是这样的：

*p	0	const int

这其实是 C/C++ 的一个违反常理的地方：指针（*）、引用（&）属于说明符（declarators），在定义的时候，是和变量名，而不是类型标识符（type-specifiers）相结合的。

因此，“const decltype(z)*p”推导出来的其实是 *p 的类型（const int），然后再进一步运算出 p 的类型。

2. decltype 的推导规则

从上面一节内容来看，decltype 的使用是比较简单的。但在简单的使用方法之后，也隐藏了不少细节。

我们先来看看 decltype(exp) 的推导规则㊀：

- 推导规则 1，exp 是标识符、类访问表达式，decltype(exp) 和 exp 的类型一致。
- 推导规则 2，exp 是函数调用，decltype(exp) 和返回值的类型一致。
- 推导规则 3，其他情况，若 exp 是一个左值，则 decltype(exp) 是 exp 类型的左值引用，否则和 exp 类型一致。

只看上面的推导规则，很难理解 decltype(exp) 到底是一个什么类型。为了更好地讲解这些规则的适用场景，下面根据上面的规则分 3 种情况依次讨论：

1）标识符表达式和类访问表达式。

2）函数调用（非标识符表达式，也非类访问表达式）。

3）带括号的表达式和加法运算表达式（其他情况）。

（1）标识符表达式和类访问表达式

先看第一种情况，代码清单 1-3 是一组简单的例子。

代码清单 1-3 decltype 作用于标识符和类访问表达式示例

```
class Foo
{
public:
   static const int Number = 0;
   int x;
};
```

㊀ 关于推导规则，有很多种版本。
-C++ 标准：ISO/IEC 14882:2011，7.1.6.2 Simple type specifiers，第 4 款
-MSDN：decltype Type Specifier，http://msdn.microsoft.com/en-us/library/dd537655.aspx
- 维基百科：decltype，http://en.wikipedia.org/wiki/Decltype
虽然描述不同，但其实是等价的。为了方便理解，这里选取了 MSDN 的版本。

```
int n = 0;
volatile const int & x = n;

decltype(n) a = n;                          //a -> int
decltype(x) b = n;                          //b -> const volatile int &

decltype(Foo::Number) c = 0;                //c -> const int

Foo foo;
decltype(foo.x) d = 0;                      //d -> int，类访问表达式
```

变量 a、b、c 保留了表达式的所有属性（cv、引用）。这里的结果是很简单的，按照推导规则 1，对于标识符表达式而言，decltype 的推导结果就和这个变量的类型定义一致。

d 是一个类访问表达式，因此也符合推导规则 1。

（2）函数调用

接下来，考虑第二种情况：如果表达式是一个函数调用（不符合推导规则 1），结果会如何呢？

请看代码清单 1-4 所示的示例。

代码清单 1-4　decltype 作用于函数调用的示例

```
int& func_int_r(void);                      //左值（lvalue，可简单理解为可寻址值）
int&& func_int_rr(void);                    //x 值（xvalue，右值引用本身是一个 xvalue）
int func_int(void);                         //纯右值（prvalue，将在后面的章节中讲解）

const int& func_cint_r(void);               //左值
const int&& func_cint_rr(void);             //x 值
const int func_cint(void);                  //纯右值

const Foo func_cfoo(void);                  //纯右值

//下面是测试语句
int x = 0;

decltype(func_int_r())   a1 = x;            //a1 -> int &
decltype(func_int_rr())  b1 = 0;            //b1 -> int &&
decltype(func_int())     c1 = 0;            //c1 -> int

decltype(func_cint_r())  a2 = x;            //a2 -> const int &
decltype(func_cint_rr()) b2 = 0;            //b2 -> const int &&
decltype(func_cint())    c2 = 0;            //c2 -> int

decltype(func_cfoo())    ff = Foo();        //ff -> const Foo
```

可以看到，按照推导规则 2，decltype 的结果和函数的返回值类型保持一致。

这里需要注意的是，c2 是 int 而不是 const int。这是因为函数返回的 int 是一个纯右值

（prvalue）。对于纯右值而言，只有类类型可以携带 cv 限定符，此外则一般忽略掉 cv 限定[㊀]。

如果在 gcc 下编译上面的代码，会得到一个警告信息如下：

```
warning: type qualifiers ignored on function return type
[-Wignored-qualifiers]
 cint func_cint(void);
```

因此，decltype 推导出来的 c2 是一个 int。

作为对比，可以看到 decltype 根据 func_cfoo() 推导出来的 ff 的类型是 const Foo。

（3）带括号的表达式和加法运算表达式

最后，来看看第三种情况：

```
struct Foo { int x; };
const Foo foo = Foo();

decltype(foo.x)   a = 0;                //a -> int
decltype((foo.x)) b = a;                //b -> const int &

int n = 0, m = 0;
decltype(n + m) c = 0;                  //c -> int
decltype(n += m) d = c;                 //d -> int &
```

a 和 b 的结果：仅仅多加了一对括号，它们得到的类型却是不同的。

a 的结果是很直接的，根据推导规则 1，a 的类型就是 foo.x 的定义类型。

b 的结果并不适用于推导规则 1 和 2。根据 foo.x 是一个左值，可知括号表达式也是一个左值。因此可以按照推导规则 3，知道 decltype 的结果将是一个左值引用。

foo 的定义是 const Foo，所以 foo.x 是一个 const int 类型左值，因此 decltype 的推导结果是 const int &。

同样，n+m 返回一个右值，按照推导规则 3，decltype 的结果为 int。

最后，n+=m 返回一个左值，按照推导规则 3，decltype 的结果为 int &。

3. decltype 的实际应用

decltype 的应用多出现在泛型编程中。考虑代码清单 1-5 的场景。

代码清单 1-5　泛型类型定义可能存在问题的示例

```
#include <vector>

template <class ContainerT>
class Foo
{
   typename ContainerT::iterator it_; //类型定义可能有问题
```

㊀ ISO/IEC 14882:2011，3.10 Lvalues and rvalues，
第 1 款：“The result of calling a functionwhose return type is not a reference is a prvalue.”
第 4 款：“Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types.”

```
public:
   void func(ContainerT& container)
   {
      it_ = container.begin();
   }

   // ...
};

int main(void)
{
   typedef const std::vector<int> container_t;
   container_t arr;

   Foo<container_t> foo;
   foo.func(arr);

   return 0;
}
```

单独看类 Foo 中的 it_ 成员定义，很难看出会有什么错误，但在使用时，若上下文要求传入一个 const 容器类型，编译器马上会弹出一大堆错误信息。

原因就在于，ContainerT::iterator 并不能包括所有的迭代器类型，当 ContainerT 是一个 const 类型时，应当使用 const_iterator。

要想解决这个问题，在 C++98/03 下只能想办法把 const 类型的容器用模板特化单独处理，比如增加一个像下面这样的模板特化：

```
template <class ContainerT>
class Foo<const ContainerT>
{
   typename ContainerT::const_iterator it_;

public:
   void func(const ContainerT& container)
   {
      it_ = container.begin();
   }

   // ...
};
```

这实在不能说是一个好的解决办法。若 const 类型的特化只是为了配合迭代器的类型限制，Foo 的其他代码也不得不重新写一次。

有了 decltype 以后，就可以直接这样写：

```
template <class ContainerT>
class Foo
```

```
{
   decltype(ContainerT().begin()) it_;

public:
   void func(ContainerT& container)
   {
      it_ = container.begin();
   }

   // ...
};
```

是不是舒服很多了？

decltype 也经常用在通过变量表达式抽取变量类型上，如下面的这种用法：

```
vector<int> v;
// ...
decltype(v)::value_type i = 0;
```

在冗长的代码中，人们往往只会关心变量本身，而并不关心它的具体类型。比如在上例中，只要知道 v 是一个容器就够了（可以提取 value_type），后面的所有算法内容只需要出现 v，而不需要出现像 vector<int> 这种精确的类型名称。这对理解一些变量类型复杂但操作统一的代码片段有很大好处。

实际上，标准库中有些类型都是通过 decltype 来定义的：

```
typedef decltype(nullptr)nullptr_t;// 通过编译器关键字 nullptr 定义类型 nullptr_t
typedef decltype(sizeof(0)) size_t;
```

这种定义方法的好处是，从类型的定义过程上就可以看出来这个类型的含义。

1.1.3 返回类型后置语法——auto 和 decltype 的结合使用

在泛型编程中，可能需要通过参数的运算来得到返回值的类型。

考虑下面这个场景：

```
template <typename R, typename T, typename U>
R add(T t, U u)
{
   return t+u;
}

int a = 1; float b = 2.0;
auto c = add<decltype(a + b)>(a, b);
```

我们并不关心 a+b 的类型是什么，因此，只需要通过 decltype(a+b) 直接得到返回值类型即可。但是像上面这样使用十分不方便，因为外部其实并不知道参数之间应该如何运算，只有 add 函数才知道返回值应当如何推导。那么，在 add 函数的定义上能不能直接通过

decltype 拿到返回值呢?

```
template <typename T, typename U>
decltype(t + u) add(T t, U u) //error: t、u 尚未定义
{
   return t + u;
}
```

当然，直接像上面这样写是编译不过的。因为 t、u 在参数列表中，而 C++ 的返回值是前置语法，在返回值定义的时候参数变量还不存在。

可行的写法如下：

```
template <typename T, typename U>
decltype(T() + U()) add(T t, U u)
{
   return t + u;
}
```

考虑到 T、U 可能是没有无参构造函数的类，正确的写法应该是这样：

```
template <typename T, typename U>
decltype((*(T*)0) + (*(U*)0)) add(T t, U u)
{
   return t + u;
}
```

虽然成功地使用 decltype 完成了返回值的推导，但写法过于晦涩，会大大增加 decltype 在返回值类型推导上的使用难度并降低代码的可读性。

因此，在 C++11 中增加了返回类型后置（trailing-return-type，又称跟踪返回类型）语法，将 decltype 和 auto 结合起来完成返回值类型的推导。

返回类型后置语法是通过 auto 和 decltype 结合起来使用的。上面的 add 函数，使用新的语法可以写成：

```
template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u)
{
   return t + u;
}
```

为了进一步说明这个语法，再看另一个例子：

```
int& foo(int& i);
float foo(float& f);

template <typename T>
auto func(T& val) -> decltype(foo(val))
{
   return foo(val);
}
```

如果说前一个例子中的 add 使用 C++98/03 的返回值写法还勉强可以完成，那么这个例子对于 C++ 而言就是不可能完成的任务了。

在这个例子中，使用 decltype 结合返回值后置语法很容易推导出了 foo(val) 可能出现的返回值类型，并将其用到了 func 上。

返回值类型后置语法，是为了解决函数返回值类型依赖于参数而导致难以确定返回值类型的问题。有了这种语法以后，对返回值类型的推导就可以用清晰的方式（直接通过参数做运算）描述出来，而不需要像 C++98/03 那样使用晦涩难懂的写法。

1.2 模板的细节改进

C++11 改进了编译器的解析规则，尽可能地将多个右尖括号（>）解析成模板参数结束符，方便我们编写模板相关的代码。

1.2.1 模板的右尖括号

在 C++98/03 的泛型编程中，模板实例化有一个很烦琐的地方，那就是连续两个右尖括号（>>）会被编译器解释成右移操作符，而不是模板参数表的结束。

看一下代码清单 1-6 所讲的例子。

代码清单 1-6 C++98/03 中不支持连续两个右尖括号的示例

```
template <typename T>
struct Foo
{
   typedef T type;
};

template <typename T>
class A
{
   // ...
};

int main(void)
{
   Foo<A<int>>::type xx; // 编译出错
   return 0;
}
```

使用 gcc 编译时，会得到如下错误提示：

```
error: '>>' should be '>>' within a nested template argument list
    Foo<A<int>>::type xx;
```

意思就是，“Foo<A<int>>”这种写法是不被支持的，要写成这样：“Foo<A<int> >”（注意两个右尖括号之间的空格）。

这种限制无疑是很没有必要的。在 C++ 的各种成对括号中，目前只有右尖括号连续写两个会出现这种二义性。static_cast、reinterpret_cast 等 C++ 标准转换运算符，都是使用“ <>”来获得待转换类型（type-id）的。若这个 type-id 本身是一个模板，用起来会很不方便。

现在在 C++11 中，这种限制终于被取消了。在 C++11 标准中，要求编译器对模板的右尖括号做单独处理，使编译器能够正确判断出“ >>”是一个右移操作符还是模板参数表的结束标记（delimiter，界定符）[㊀]。

不过这种自动化的处理在某些时候会与老标准不兼容，比如下面这个例子：

```
template <int N>
struct Foo
{
   // ...
};

int main(void)
{
   Foo<100 >> 2> xx;
   return 0;
}
```

在 C++98/03 的编译器中编译是没问题的，但 C++11 的编译器会显示：

```
error: expected unqualified-id before '>' token
    Foo<100 >> 2> xx;
```

解决的方法是这样写：

```
Foo<(100 >> 2)> xx;  // 注意括号
```

这种加括号的写法其实也是一个良好的编程习惯，使得在书写时倾向于写出无二义性的代码。

小知识　各种 C++98/03 编译器除了支持标准（ISO/IEC 14882:2003 及其之前的标准）之外，还自行做了不少的拓展。这些拓展中的一部分，后来经过了 C++ 委员会的斟酌和完善，进入了 C++11。所以有一部分 C++11 的新特征，在一些 C++98/03 的老编译器下也是可以支持的，只是由于没有标准化，无法保证各种平台 / 编译器下的兼容性。比如像 Microsoft Visual C++ 2005 这种不支持 C++11 的编译器，在对模板右尖括号的处理上和现在的 C++11 是一致的。

㊀ ISO/IEC 14882:2011，1.4.2 Names of template specializations，第 3 款。

1.2.2 模板的别名

大家都知道，在 C++ 中可以通过 typedef 重定义一个类型：

```
typedef unsigned int uint_t;
```

被重定义的类型名叫“typedef-name”。它并不是一个新的类型，仅仅只是原有的类型取了一个新的名字。因此，下面这样将不是合法的函数重载：

```
void func(unsigned int);
void func(uint_t);       //error: redefinition
```

使用 typedef 重定义类型是很方便的，但它也有一些限制，比如，无法重定义一个模板。想象下面这个场景：

```
typedef std::map<std::string, int> map_int_t;
// ...
typedef std::map<std::string, std::string> map_str_t;
// ...
```

我们需要的其实是一个固定以 std::string 为 key 的 map，它可以映射到 int 或另一个 std::string。然而这个简单的需求仅通过 typedef 却很难办到。

因此，在 C++98/03 中往往不得不这样写：

```
template <typename Val>
struct str_map
{
   typedef std::map<std::string, Val> type;
};

// ...

str_map<int>::type map1;

// ...
```

一个虽然简单但却略显烦琐的 str_map 外敷类是必要的。这明显让我们在复用某些泛型代码时非常难受。

现在，在 C++11 中终于出现了可以重定义一个模板的语法。请看下面的示例：

```
template <typename Val>
using str_map_t = std::map<std::string, Val>;
// ...
str_map_t<int> map1;
```

这里使用新的 using 别名语法定义了 std::map 的模板别名 str_map_t。比起前面使用外敷模板加 typedef 构建的 str_map，它完全就像是一个新的 map 类模板，因此，简洁了很多。

实际上，using 的别名语法覆盖了 typedef 的全部功能。先来看看对普通类型的重定义示例，将这两种语法对比一下：

```
// 重定义 unsigned int
typedef unsigned int uint_t;
using uint_t = unsigned int;

// 重定义 std::map
typedef std::map<std::string, int> map_int_t;
using map_int_t = std::map<std::string, int>;
```

可以看到，在重定义普通类型上，两种使用方法的效果是等价的，唯一不同的是定义语法。

typedef 的定义方法和变量的声明类似：像声明一个变量一样，声明一个重定义类型，之后在声明之前加上 typedef 即可。这种写法凸显了 C/C++ 中的语法一致性，但有时却会增加代码的阅读难度。比如重定义一个函数指针时：

```
typedef void (*func_t)(int, int);
```

与之相比，using 后面总是立即跟随新标识符（Identifier），之后使用类似赋值的语法，把现有的类型（type-id）赋给新类型：

```
using func_t = void (*)(int, int);
```

从上面的对比中可以发现，C++11 的 using 别名语法比 typedef 更加清晰。因为 typedef 的别名语法本质上类似一种解方程的思路。而 using 语法通过赋值来定义别名，和我们平时的思考方式一致。

下面再通过一个对比示例，看看新的 using 语法是如何定义模板别名的。

```
/* C++98/03 */

template <typename T>
struct func_t
{
   typedef void (*type)(T, T);
};
// 使用 func_t 模板
func_t<int>::type xx_1;

/* C++11 */

template <typename T>
using func_t = void (*)(T, T);
// 使用 func_t 模板
func_t<int> xx_2;
```

从示例中可以看出，通过 using 定义模板别名的语法，只是在普通类型别名语法的基

础上增加 template 的参数列表。使用 using 可以轻松地创建一个新的模板别名，而不需要像 C++98/03 那样使用烦琐的外敷模板。

需要注意的是，using 语法和 typedef 一样，并不会创造新的类型。也就是说，上面示例中 C++11 的 using 写法只是 typedef 的等价物。虽然 using 重定义的 func_t 是一个模板，但 func_t<int> 定义的 xx_2 并不是一个由类模板实例化后的类，而是 void(*)(int, int) 的别名。

因此，下面这样写：

```
void foo(void (*func_call)(int, int));
void foo(func_t<int> func_call);        //error: redefinition
```

同样是无法实现重载的，func_t<int> 只是 void(*)(int, int) 类型的等价物。

细心的读者可以发现，using 重定义的 func_t 是一个模板，但它既不是类模板也不是函数模板（函数模板实例化后是一个函数），而是一种新的模板形式：模板别名（alias template）。

其实，通过 using 可以轻松定义任意类型的模板表达方式。比如下面这样：

```
template <typename T>
using type_t = T;
//...
type_t<int> i;
```

type_t 实例化后的类型和它的模板参数类型等价。这里，type_t<int> 将等价于 int。

1.2.3 函数模板的默认模板参数

在 C++98/03 中，类模板可以有默认的模板参数，如下：

```
template <typename T, typename U = int, U N = 0>
struct Foo
{
   //...
};
```

但是却不支持函数的默认模板参数：

```
template <typename T = int> //error in C++98/03: default template arguments
void func(void)
{
   //...
}
```

现在这一限制在 C++11 中被解除了。上面的 func 函数在 C++11 中可以直接使用，代码如下：

```
int main(void)
{
   func();                                  //如同一个普通的 void(void) 类型函数
```

```
    return 0;
}
```

从上面的例子中可以看出来，当所有模板参数都有默认参数时，函数模板的调用如同一个普通函数。对于类模板而言，哪怕所有参数都有默认参数，在使用时也必须在模板名后跟随“<>”来实例化。

除了上面提到的部分之外，函数模板的默认模板参数在使用规则上和其他的默认参数也有一些不同，它没有必须写在参数表最后的限制。

因此，当默认模板参数和模板参数自动推导结合起来使用时，书写显得非常灵活。我们可以指定函数中的一部分模板参数采用默认参数，而另一部分使用自动推导，比如下面的例子：

```
template <typename R = int, typename U>
R func(U val)
{
    ret val
}

int main(void)
{
    func(123);
    return 0;
}
```

但需要注意的是，在调用函数模板时，若显示指定模板的参数，由于参数填充顺序是从右往左的，因此，像下面这样调用：

```
func<long>(123); // func 的返回值类型是 long
```

函数模板 func 的返回值类型是 long，而不是 int，因为模版参数的填充顺序从右往左，所以指定的模版参数类型 long 会作为 func 的参数类型而不是 func 的返回类型，最终 func 的返回类型为 long。这个细节虽然简单，但在多个默认模板参数和模板参数自动推导穿插使用时很容易被忽略掉，造成使用时的一些意外。

另外，当默认模板参数和模板参数自动推导同时使用时，若函数模板无法自动推导出参数类型，则编译器将使用默认模板参数；否则将使用自动推导出的参数类型。请看下面这个例子：

```
template <typename T>
struct identity
{
    typedef T type;
};

template <typename T = int>
void func(typename identity<T>::type val, T = 0)
```

```
{
   // ...
}

int main(void)
{
   func(123);                    // T -> int
   func(123, 123.0);             // T -> double
   return 0;
}
```

在例子中，通过 identity 外敷模板禁用了形参 val 的类型自动推导。但由于 func 指定了模板参数 T 的默认类型为 int，因此，在 func(123) 中，func 的 val 参数将为 int 类型。而在 func(123, 123.0) 中，由于 func 的第二个参数 123.0 为 double 类型，模板参数 T 将优先被自动推导为 double，因此，此时 val 参数将为 double 类型。

这里要注意的是，不能将默认模板参数当作模板参数自动推导的“建议”，因为模板参数自动推导总是根据实参推导来的，当自动推导生效时，默认模板参数会被直接忽略。

1.3 列表初始化

我们知道，在 C++98/03 中的对象初始化方法有很多种，如代码清单 1-7 所示。

代码清单 1-7 对象初始化示例

```
// initializer list
int i_arr[3] = { 1, 2, 3 };          // 普通数组

struct A
{
   int x;
   struct B
   {
      int i;
      int j;
   } b;
} a = { 1, { 2, 3 } };               // POD 类型

// 拷贝初始化 (copy-initialization)

int i = 0;

class Foo
{
public:
   Foo(int) {}
} foo = 123;                         // 需要拷贝构造函数

// 直接初始化 (direct-initialization)
```

```
int j(0);
Foo bar(123);
```

这些不同的初始化方法，都有各自的适用范围和作用。最关键的是，这些种类繁多的初始化方法，没有一种可以通用所有情况。

为了统一初始化方式，并且让初始化行为具有确定的效果，C++11 中提出了列表初始化（List-initialization）的概念。

1.3.1　统一的初始化

在上面我们已经看到了，对于普通数组和 POD 类型[㊀]，C++98/03 可以使用初始化列表（initializer list）进行初始化：

```
int i_arr[3] = { 1, 2, 3 };
long l_arr[] = { 1, 3, 2, 4 };

struct A
{
   int x;
   int y;
} a = { 1, 2 };
```

但是这种初始化方式的适用性非常狭窄，只有上面提到的这两种数据类型[㊁]可以使用初始化列表。

在 C++11 中，初始化列表的适用性被大大增加了。它现在可以用于任何类型对象的初始化，如代码清单 1-8 所示。

代码清单 1-8　通过初始化列表初始化对象

```
class Foo
{
public:
       Foo(int) {}

private:
       Foo(const Foo &);
};

int main(void)
{
       Foo a1(123);
       Foo a2 = 123; //error: 'Foo::Foo(const Foo &)' is private
```

㊀ 即 plain old data 类型，简单来说，是可以直接使用 memcpy 复制的对象。参考：ISO/IEC 14882:2011，9 Classes，第 10 款。

㊁ 实际上在 C++98/03 标准中，对于可以使用这种初始化方式的类型有明确的定义，将在后面展开讲解。

```
        Foo a3 = { 123 };
        Foo a4 { 123 };

        int a5 = { 3 };
        int a6 { 3 };

        return 0;
}
```

在上例中，a3、a4 使用了新的初始化方式来初始化对象，效果如同 a1 的直接初始化。

a5、a6 则是基本数据类型的列表初始化方式。可以看到，它们的形式都是统一的。

这里需要注意的是，a3 虽然使用了等于号，但它仍然是列表初始化，因此，私有的拷贝构造并不会影响到它。

a4 和 a6 的写法，是 C++98/03 所不具备的。在 C++11 中，可以直接在变量名后面跟上初始化列表，来进行对象的初始化。

这种变量名后面跟上初始化列表方法同样适用于普通数组和 POD 类型的初始化：

```
int i_arr[3] { 1, 2, 3 };// 普通数组

struct A
{
   int x;
   struct B
   {
      int i;
      int j;
   } b;
} a { 1, { 2, 3 } };      // POD 类型
```

在初始化时，{ } 前面的等于号是否书写对初始化行为没有影响。

另外，如同读者所想的那样，new 操作符等可以用圆括号进行初始化的地方，也可以使用初始化列表：

```
int* a = new int { 123 };
double b = double { 12.12 };
int* arr = new int[3] { 1, 2, 3 };
```

指针 a 指向了一个 new 操作符返回的内存，通过初始化列表方式在内存初始化时指定了值为 123。

b 则是对匿名对象使用列表初始化后，再进行拷贝初始化。

这里让人眼前一亮的是 arr 的初始化方式。堆上动态分配的数组终于也可以使用初始化列表进行初始化了。

除了上面所述的内容之外，列表初始化还可以直接使用在函数的返回值上：

```
struct Foo
```

```
{
    Foo(int, double) {}
};

Foo func(void)
{
    return { 123, 321.0 };
}
```

这里的 return 语句就如同返回了一个 Foo(123, 321.0)。

由上面的这些例子可以看到，在 C++11 中使用初始化列表是非常便利的。它不仅统一了各种对象的初始化方式，而且还使代码的书写更加简单清晰。

1.3.2　列表初始化的使用细节

在 C++11 中，初始化列表的使用范围被大大增强了。一些模糊的概念也随之而来。

上一节，读者已经看到了初始化列表可以被用于一个自定义类型的初始化。但是对于一个自定义类型，初始化列表现在可能有两种执行结果：

```
struct A
{
    int x;
    int y;
} a = { 123, 321 };    //a.x = 123, a.y = 321

struct B
{
    int x;
    int y;
    B(int, int) : x(0), y(0) {}
} b = { 123, 321 };    //b.x = 0, b.y = 0
```

其实，上述变量 a 的初始化过程是 C++98/03 中就有的聚合类型（Aggregates）的初始化。它将以拷贝的形式，用初始化列表中的值来初始化 struct A 中的成员。

struct B 由于定义了一个自定义的构造函数，因此，实际上初始化是以构造函数进行的。

看到这里，读者可能会希望能够有一个确定的判断方法，能够清晰地知道初始化列表的赋值方式。

具体来说，在使用初始化列表时，对于什么样的类型 C++ 会认为它是一个聚合体？

下面来看看聚合类型的定义[㊀]：

（1）类型是一个普通数组（如 int[10]、char[]、long[2][3]）。

（2）类型是一个类（class、struct、union），且

❑ 无用户自定义的构造函数。

㊀ ISO/IEC 14882:2011，8.5.1 Aggregates，第 1 款。

❑ 无私有（Private）或保护（Protected）的非静态数据成员。

❑ 无基类。

❑ 无虚函数。

❑ 不能有 { } 和 = 直接初始化（brace-or-equal-initializer）的非静态数据成员。

对于数组而言，情况是很清晰的。只要该类型是一个普通数组，哪怕数组的元素并非一个聚合类型，这个数组本身也是一个聚合类型：

```
int x[] = { 1, 3, 5 };
float y[4][3] =
{
   { 1, 3, 5 },
   { 2, 4, 6 },
   { 3, 5, 7 },
};
char cv[4] = { 'a', 's', 'd', 'f' };

std::string sa[3] = { "123", "321", "312" };
```

下面重点介绍当类型是一个类时的情况。首先是存在用户自定义构造函数时的例子，代码如下：

```
struct Foo
{
   int x;
   double y;
   int z;
   Foo(int, int) {}
};
Foo foo { 1, 2.5, 1 };    //error
```

这时无法将 Foo 看做一个聚合类型，因此，必须以自定义的构造函数来构造对象。

私有（Private）或保护（Protected）的非静态数据成员的情况如下：

```
struct ST
{
   int x;
   double y;
protected:
   int z;
};
ST s { 1, 2.5, 1 };       //error

struct Foo
{
   int x;
   double y;
protected:
   static int z;
};
```

```
Foo foo { 1, 2.5 }; //ok
```

在上面的示例中，ST 的初始化是非法的。因为 ST 的成员 z 是一个受保护的非静态数据成员。

而 Foo 的初始化则是成功的，因为它的受保护成员是一个静态数据成员。

这里需要注意，Foo 中的静态成员是不能通过实例 foo 的初始化列表进行初始化的，它的初始化遵循静态成员的初始化方式。

对于有基类和虚函数的情况：

```
struct ST
{
   int x;
   double y;
   virtual void F(){}
};
ST s { 1, 2.5 };    //error

struct Base {};
struct Foo : public Base
{
   int x;
   double y;
};
Foo foo { 1, 2.5 }; //error
```

ST 和 Foo 的初始化都会编译失败。因为 ST 中存在一个虚函数 F，而 Foo 则有一个基类 Base。

最后，介绍“不能有 { } 和 = 直接初始化（brace-or-equal-initializer）的非静态数据成员”这条规则，代码如下：

```
struct ST
{
   int x;
   double y = 0.0;
};
ST s { 1, 2.5 };    //error
```

在 ST 中，y 在声明时即被 = 直接初始化为 0.0，因此，ST 并不是一个聚合类型，不能直接使用初始化列表。

在 C++98/03 中，对于 y 这种非静态数据成员，本身就不能在声明时进行这种初始化工作。但是在 C++11 中放宽了这方面的限制。可以看到，在 C++11 中，非静态数据成员也可以在声明的同时进行初始化工作（即使用 { } 或 = 进行初始化）。

对于一个类来说，如果它的非静态数据成员在声明的同时进行了初始化，那么它就不再是一个聚合类型，因此，也不能直接使用初始化列表。

对于上述非聚合类型的情形，想要使用初始化列表的方法就是自定义一个构造函数，比如：

```
struct ST
{
   int x;
   double y;
   virtual void F(){}
private:
   int z;
public:
   ST(int i, double j, int k) : x(i), y(j), z(k) {}
};
ST s { 1, 2.5, 2 };
```

需要注意的是，聚合类型的定义并非递归的。简单来说，当一个类的非静态成员是非聚合类型时，这个类也有可能是聚合类型。比如下面这个例子：

```
struct ST
{
   int x;
   double y;
private:
   int z;
};
ST s { 1, 2.5, 1 };      //error

struct Foo
{
   ST st;
   int x;
   double y;
};
Foo foo { {}, 1, 2.5 };  //OK
```

可以看到，ST 并非一个聚合类型，因为它有一个 Private 的非静态成员。

但是尽管 Foo 含有这个非聚合类型的非静态成员 st，它仍然是一个聚合类型，可以直接使用初始化列表。

注意到 foo 的初始化过程，对非聚合类型成员 st 做初始化的时候，可以直接写一对空的大括号“{}”，相当于调用 ST 的无参构造函数。

现在，对于使用初始化列表时的一些细节有了更深刻的了解。对于一个聚合类型，使用初始化列表相当于对其中的每个元素分别赋值；而对于非集合类型，则需要先自定义一个合适的构造函数，此时使用初始化列表将调用它对应的构造函数。

1.3.3　初始化列表

1. 任意长度的初始化列表

读者可能注意到了，C++11 中的 stl 容器拥有和未显示指定长度的数组一样的初始化能力，代码如下：

```
int arr[] { 1, 2, 3 };

std::map<std::string, int> mm =
{
   { "1", 1 }, { "2", 2 }, { "3", 3 }
};

std::set<int> ss = { 1, 2, 3 };

std::vector<int> arr = { 1, 2, 3, 4, 5 };
```

这里 arr 没有显式指定长度，因此，它的初始化列表可以是任意长度。

同样，std::map、std::set、std::vector 也可以在初始化时任意书写需要初始化的内容。

前面自定义的 Foo 却不具备这种能力，只能按部就班地按照构造函数指定的参数列表进行赋值。

实际上，stl 中的容器是通过使用 std::initializer_list 这个轻量级的类模板来完成上述功能支持的。我们只需要为 Foo 添加一个 std::initializer_list 构造函数，它也将拥有这种任意长度初始化的能力，代码如下：

```
class Foo
{
public:
   Foo(std::initializer_list<int>) {}
};

Foo foo = { 1, 2, 3, 4, 5 }; // OK!
```

那么，知道了使用 std::initializer_list 来接收 {...}，如何通过它来给自定义容器赋值呢？来看代码清单 1-9 中的例子。

代码清单 1-9　通过 std::initializer_list 给自定义容器赋值示例

```
class FooVector
{
   std::vector<int> content_;

public:
   FooVector(std::initializer_list<int> list)
   {
      for (auto it = list.begin(); it != list.end(); ++it)
```

```
        {
            content_.push_back(*it);
        }
    }
};

class FooMap
{
    std::map<int, int> content_;
    using pair_t = std::map<int, int>::value_type;

public:
    FooMap(std::initializer_list<pair_t> list)
    {
        for (auto it = list.begin(); it != list.end(); ++it)
        {
            content_.insert(*it);
        }
    }
};

FooVector foo_1 = { 1, 2, 3, 4, 5 };
FooMap    foo_2 = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

这里定义了两个自定义容器，一个是 FooVector，采用 std::vector<int> 作为内部存储；另一个是 FooMap，采用 std::map<int, int> 作为内部存储。

可以看到，FooVector、FooMap 的初始化过程，就和它们使用的内部存储结构一样。

这两个自定义容器的构造函数中，std::initializer_list 负责接收初始化列表。并通过我们熟知的 for 循环过程，把列表中的每个元素取出来，并放入内部的存储空间中。

std::initializer_list 不仅可以用来对自定义类型做初始化，还可以用来传递同类型的数据集合，代码如下：

```
void func(std::initializer_list<int> l)
{
    for (auto it = l.begin(); it != l.end(); ++it)
    {
        std::cout << *it << std::endl;
    }
}

int main(void)
{
    func({});            // 一个空集合
    func({ 1, 2, 3 });   // 传递 { 1, 2, 3 }
    return 0;
}
```

如上述所示，在任何需要的时候，std::initializer_list 都可以当作参数来一次性传递同类型的多个数据。

2. std::initializer_list 的一些细节

了解了 std::initializer_list 之后，再来看看它的一些特点，如下：

- 它是一个轻量级的容器类型，内部定义了 iterator 等容器必需的概念。
- 对于 std::initializer_list<T> 而言，它可以接收任意长度的初始化列表，但要求元素必须是同种类型 T（或可转换为 T）。
- 它有 3 个成员接口：size()、begin()、end()。
- 它只能被整体初始化或赋值。

通过前面的例子，已经知道了 std::initializer_list 的前几个特点。其中没有涉及的接口 size() 是用来获得 std::initializer_list 的长度的，比如：

```
std::initializer_list<int> list = { 1, 2, 3 };
size_t n = list.size(); //n == 3
```

最后，对 std::initializer_list 的访问只能通过 begin() 和 end() 进行循环遍历，遍历时取得的迭代器是只读的。因此，无法修改 std::initializer_list 中某一个元素的值，但是可以通过初始化列表的赋值对 std::initializer_list 做整体修改，代码如下：

```
std::initializer_list<int> list;
size_t n = list.size(); //n == 0
list = { 1, 2, 3, 4, 5 };
n = list.size();        //n == 5
list = { 3, 1, 2, 4 };
n = list.size();        //n == 4
```

std::initializer_list 拥有一个无参数的构造函数，因此，它可以直接定义实例，此时将得到一个空的 std::initializer_list。

之后，我们对 std::initializer_list 进行赋值操作（注意，它只能通过初始化列表赋值），可以发现 std::initializer_list 被改写成了 {1, 2, 3, 4, 5}。

然后，还可以对它再次赋值，std::initializer_list 被修改成了 {3, 1, 2, 4}。

看到这里，可能有读者会关心 std::initializer_list 的传递或赋值效率。

假如 std::initializer_list 在传递或赋值的时候如同 vector 之类的容器一样，把每个元素都复制了一遍，那么使用它传递类对象的时候就要斟酌一下了。

实际上，std::initializer_list 是非常高效的。它的内部并不负责保存初始化列表中元素的拷贝，仅仅存储了列表中元素的引用而已。

因此，我们不应该像这样使用：

```
std::initializer_list<int> func(void)
{
   int a = 1, b = 2;
```

```
    return { a, b }; // a、b 在返回时并没有被拷贝
}
```

虽然这能够正常通过编译，但却无法传递出我们希望的结果（a、b 在函数结束时，生存期也结束了，因此，返回的将是不确定的内容）。

这种情况下最好的做法应该是这样：

```
std::vector<int> func(void)
{
    int a = 1, b = 2;
    return { a, b };
}
```

使用真正的容器，或具有转移 / 拷贝语义的物件来替代 std::initializer_list 返回需要的结果。

我们应当总是把 std::initializer_list 看做保存对象的引用，并在它持有对象的生存期结束之前完成传递。

1.3.4 防止类型收窄

类型收窄指的是导致数据内容发生变化或者精度丢失的隐式类型转换。考虑下面这种情况：

```
struct Foo
{
    Foo(int i) { std::cout << i << std::endl; }
};

Foo foo(1.2);
```

以上代码在 C++ 中能够正常通过编译，但是传递之后的 i 却不能完整地保存一个浮点型的数据。

上面的示例让我们对类型收窄有了一个大概的了解。具体来说，类型收窄包括以下几种情况[㊀]：

1）从一个浮点数隐式转换为一个整型数，如 int i=2.2。

2）从高精度浮点数隐式转换为低精度浮点数，如从 long double 隐式转换为 double 或 float。

3）从一个整型数隐式转换为一个浮点数，并且超出了浮点数的表示范围，如 float x=(unsigned long long)−1。

4）从一个整型数隐式转换为一个长度较短的整型数，并且超出了长度较短的整型数的表示范围，如 char x=65536。

在 C++98/03 中，像上面所示类型收窄的情况，编译器并不会报错（或报一个警告，如

㊀ ISO/IEC 14882:2011，8.5.4 List-initialization，第 7 款。

Microsoft Visual C++)。这往往会导致一些隐藏的错误。在 C++11 中，可以通过列表初始化来检查及防止类型收窄。

请看代码清单 1-10 的示例。

代码清单 1-10　列表初始化防止类型收窄示例

```
int a = 1.1;                              //OK
int b = { 1.1 };                          //error

float fa = 1e40;                          //OK
float fb = { 1e40 };                      //error

float fc = (unsigned long long)-1;        //OK
float fd = { (unsigned long long)-1 };    //error
float fe = (unsigned long long)1;         //OK
float ff = { (unsigned long long)1 };     //OK

const int x = 1024, y = 1;
char c = x;                               //OK
char d = { x };                           //error
char e = y;                               //OK
char f = { y };                           //OK
```

在上面的各种隐式类型转换中，只要遇到了类型收窄的情况，初始化列表就不会允许这种转换发生。

其中需要注意的是 x、y 被定义成了 const int。如果去掉 const 限定符，那么最后一个变量 f 也会因为类型收窄而报错。

对于类型收窄的编译错误，不同的编译器表现并不相同。

在 gcc 4.8 中，会得到如下警告信息：

```
warning: narrowing conversion of '1.0e+40' from 'double' to
'float' inside { } [-Wnarrowing]
   float fb = { 1e40 };
                ^
```

在 Microsoft Visual C++2013 中，同样的语句则直接给出了一个错误：

```
error C2397: conversion from 'double' to 'float' requires a
narrowing conversion
```

另外，对于精度不同的浮点数的隐式转换，如下面这种：

```
float ff = 1.2;                           //OK
float fd = { 1.2 };                       //OK (gcc)
```

fd 的初始化并没有引起类型收窄，因此，在 gcc 4.8 下没有任何错误或警告。但在 Microsoft Visual C++2013 中，fd 的初始化语句将会得到一个 error C2397 的编译错误。

C++11 中新增的这种初始化方式，为程序的编写带来了很多便利。在后面的章节中也会经常使用这种新的初始化方式，读者可以在后面章节的示例代码中不断体会到新初始化方法带来的便捷。

1.4 基于范围的 for 循环

在 C++03/98 中，不同的容器和数组，遍历的方法不尽相同，写法不统一，也不够简洁，而 C++11 基于范围的 for 循环以统一、简洁的方式来遍历容器和数组，用起来更方便了。

1.4.1 for 循环的新用法

我们知道，在 C++ 中遍历一个容器的方法一般是这样的：

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

int main(void)
{
   std::vector<int> arr;

   // ...

   for(auto it = arr.begin(); it != arr.end(); ++it)
   {
      std::cout << *it << std::endl;
   }

   return 0;
}
```

上面借助前面介绍过的 auto 关键字，省略了迭代器的声明。

当然，熟悉 stl 的读者肯定还知道在 <algorithm> 中有一个 for_each 算法可以用来完成和上述同样的功能：

```
#include <algorithm>
#include <iostream>
#include <vector>

void do_cout(int n)
{
   std::cout << n << std::endl;
}

int main(void)
{
   std::vector<int> arr;
```

```
    // ...

    std::for_each(arr.begin(), arr.end(), do_cout);

    return 0;
}
```

std::for_each 比起前面的 for 循环，最大的好处是不再需要关注迭代器（Iterator）的概念，只需要关心容器中的元素类型即可。

但不管是上述哪一种遍历方法，都必须显式给出容器的开头（Begin）和结尾（End）。这是因为上面的两种方法都不是基于“范围（Range）”来设计的。

我们先来看一段简单的 C# 代码[㊀]：

```
int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibarray)
{
    System.Console.WriteLine(element);
}
```

上面这段代码通过“foreach”关键字使用了基于范围的 for 循环。可以看到，在这种 for 循环中，不再需要传递容器的两端，循环会自动以容器为范围展开，并且循环中也屏蔽掉了迭代器的遍历细节，直接抽取出容器中的元素进行运算。

与普通的 for 循环相比，基于范围的循环方式是“自明”的。这种语法构成的循环不需要额外的注释或语言基础，很容易就可以看清楚它想表达的意义。在实际项目中经常会遇到需要针对容器做遍历的情况，使用这种循环方式无疑会让编码和维护变得更加简便。

现在，在 C++11 中终于有了基于范围的 for 循环（The range-based for statement）。再来看一开始的 vector 遍历使用基于范围的 for 循环应该如何书写：

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

int main(void)
{
    std::vector<int> arr = { 1, 2, 3 };

    // ...

    for(auto n : arr) // 使用基于范围的 for 循环
    {
        std::cout << n << std::endl;
    }

    return 0;
}
```

㊀ 示例来自 MSDN：foreach, in(C# Reference)，http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx

在上面的基于范围的 for 循环中，n 表示 arr 中的一个元素，auto 则是让编译器自动推导出 n 的类型。在这里，n 的类型将被自动推导为 vector 中的元素类型 int。

在 n 的定义之后，紧跟一个冒号（:），之后直接写上需要遍历的表达式，for 循环将自动以表达式返回的容器为范围进行迭代。

在上面的例子中，我们使用 auto 自动推导了 n 的类型。当然在使用时也可以直接写上我们需要的类型：

```
std::vector<int> arr;
for(int n : arr) ;
```

基于范围的 for 循环，对于冒号前面的局部变量声明（for-range-declaration）只要求能够支持容器类型的隐式转换。因此，在使用时需要注意，像下面这样写也是可以通过编译的：

```
std::vector<int> arr;
for(char n : arr) ; // int 会被隐式转换为 char
```

在上面的例子中，我们都是在使用只读方式遍历容器。如果需要在遍历时修改容器中的值，则需要使用引用，代码如下：

```
for(auto& n : arr)
{
   std::cout << n++ << std::endl;
}
```

在完成上面的遍历后，arr 中的每个元素都会被自加 1。

当然，若只是希望遍历，而不希望修改，可以使用 const auto& 来定义 n 的类型。这样对于复制负担比较大的容器元素（比如一个 std::vector<std::string> 数组）也可以无损耗地进行遍历。

1.4.2 基于范围的 for 循环的使用细节

从上一节的示例中可以看出，range-based for 的使用是比较简单的。但是再简单的使用方法也有一些需要注意的细节。

首先，看一下使用 range-based for 对 map 的遍历方法：

```
#include <iostream>
#include <map>

int main(void)
{
   std::map<std::string, int> mm =
   {
      { "1", 1 }, { "2", 2 }, { "3", 3 }
   };
```

```
    for(auto& val : mm)
    {
        std::cout << val.first << " -> " << val.second << std::endl;
    }

    return 0;
}
```

这里需要注意两点：

1）for 循环中 val 的类型是 std::pair。因此，对于 map 这种关联性容器而言，需要使用 val.first 或 val.second 来提取键值。

2）auto 自动推导出的类型是容器中的 value_type，而不是迭代器。

关于上述第二点，我们再来看一个对比的例子：

```
std::map<std::string, int> mm =
{
    { "1", 1 }, { "2", 2 }, { "3", 3 }
};

for(auto ite = mm.begin(); ite != mm.end(); ++ite)
{
    std::cout << ite->first << " -> " << ite->second << std::endl;
}

for(auto& val : mm) //使用基于范围的 for 循环
{
    std::cout << val.first << " -> " << val.second << std::endl;
}
```

从这里就可以很清晰地看出，在基于范围的 for 循环中每次迭代时使用的类型和普通 for 循环有何不同。

在使用基于范围的 for 循环时，还需要注意容器本身的一些约束。比如下面这个例子：

```
#include <iostream>
#include <set>

int main(void)
{
    std::set<int> ss = { 1, 2, 3 };

    for(auto& val : ss)
    {
        //error: increment of read-only reference 'val'
        std::cout << val++ << std::endl;
    }

    return 0;
}
```

例子中使用 auto& 定义了 std::set<int> 中元素的引用，希望能够在循环中对 set 的值进行修改，但 std::set 的内部元素是只读的——这是由 std::set 的特征决定的，因此，for 循环中的 auto& 会被推导为 const int &。

同样的细节也会出现在 std::map 的遍历中。基于范围的 for 循环中的 std::pair 引用，是不能够修改 first 的。

接下来，看看基于范围的 for 循环对容器的访问频率。看下面这段代码：

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

std::vector<int> arr = { 1, 2, 3, 4, 5 };

std::vector<int>& get_range(void)
{
   std::cout << "get_range ->: " << std::endl;
   return arr;
}

int main(void)
{
   for(auto val : get_range())
   {
      std::cout << val << std::endl;
   }
   return 0;
}
```

输出结果：

```
get_range ->:
1
2
3
4
5
```

从上面的结果中可以看到，不论基于范围的 for 循环迭代了多少次，get_range() 只在第一次迭代之前被调用。

因此，对于基于范围的 for 循环而言，冒号后面的表达式只会被执行一次。

最后，让我们看看在基于范围的 for 循环迭代时修改容器会出现什么情况。比如，下面这段代码：

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

int main(void)
```

```
{
        std::vector<int>arr = { 1, 2, 3, 4, 5 };
        for(auto val : arr)
        {
                std::cout << val << std::endl;
                arr.push_back(0); //扩大容器
        }
        return 0;
}
```

执行结果（32 位 mingw4.8）：

```
1
5189584
-17891602
-17891602
-17891602
```

若把上面的 vector 换成 list，结果又将发生变化。

这是因为基于范围的 for 循环其实是普通 for 循环的语法糖，因此，同普通的 for 循环一样，在迭代时修改容器很可能会引起迭代器失效，导致一些意料之外的结果。由于在这里我们是看不到迭代器的，因此，直接分析对基于范围的 for 循环中的容器修改会造成什么样的影响是比较困难的。

其实对于上面的基于范围的 for 循环而言，等价的普通 for 循环如下[㊀]：

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

int main(void)
{
   std::vector<int> arr = { 1, 2, 3, 4, 5 };

   auto && __range = (arr);
   for (auto __begin = __range.begin(), __end = __range.end();
        __begin != __end; ++__begin)
   {
      auto val = *__begin;
      std::cout << val << std::endl;
      arr.push_back(0); //扩大容器
   }

   return 0;
}
```

从这里可以很清晰地看到，和我们平时写的容器遍历不同，基于范围的 for 循环倾向于在循环开始之前确定好迭代的范围，而不是在每次迭代之前都去调用一次 arr.end()。

㊀ ISO/IEC 14882:2011，6.5.4 The range-based for statement，第 1 款。

当然，良好的编程习惯是尽量不要在迭代过程中修改迭代的容器。但是实际情况要求我们不得不这样做的时候，通过理解基于范围的 for 循环的这个特点，就可以方便地分析每次迭代的结果，提前避免算法的错误。

1.4.3 让基于范围的 for 循环支持自定义类型

假如有一个自己自定义的容器类，如何让它能够支持 range-based for 呢？其实上面已经提到了，基于范围的 for 循环只是普通 for 循环的语法糖。它需要查找到容器提供的 begin 和 end 迭代器。

具体来说，基于范围的 for 循环将以下面的方式查找容器的 begin 和 end：

1）若容器是一个普通 array 对象（如 int arr[10]），那么 begin 将为 array 的首地址，end 将为首地址加容器长度。

2）若容器是一个类对象，那么 range-based for 将试图通过查找类的 begin() 和 end() 方法来定位 begin、end 迭代器。

3）否则，range-based for 将试图使用全局的 begin 和 end 函数来定位 begin、end 迭代器。

由上述可知，对自定义类类型来说，分别实现 begin()、end() 方法即可。下面通过自定义一个 range 对象来看看具体的实现方法。

我们知道，标准库中有很多容器，如 vector、list、queue、map，等等。这些对象在概念上属于 Containers。但是如果我们并不需要对容器进行迭代，而是对某个“区间”进行迭代，此时标准库中并没有对应的概念。

在这种情况下，只能使用普通的 for 循环，代码如下：

```
for (int n = 2; n < 8; n += 2) // [2, 8)
{
   std::cout << " " << n;
}
```

上面的 for 循环其实是在区间 [2, 8) 中做迭代，迭代步长为 2。如果我们可以实现一个 range 方法，使用 range(2, 8, 2) 来代表区间 [2, 8)，步长为 2，就可以通过基于范围的 for 循环直接对这个区间做迭代了[㊀]。

我们来看一下 range 的实现。首先，需要一个迭代器来负责对一个范围取值。

考虑到 for 循环中迭代器的使用方法（结束条件：ite !=end），迭代器的对外接口可以像下面这样：

```
template <typename T>
class iterator
{
public:
using value_type = T;
```

㊀ 这里提到的 range 概念，和 Python 中的 range 类似。参考：https://docs.python.org/release/1.5.1p1/tut/range.html

```
using size_type  = size_t;

   iterator(size_type cur_start, value_type begin_val,
value_type step_val);

   value_type operator*() const;

   bool operator!=(const iterator& rhs) const;

   iterator& operator++(void); //prefix ++ operator only
};
```

构造函数传递 3 个参数来进行初始化，分别是开始的迭代次数、初始值，以及迭代的步长。如果容器的 begin() 方法将设置初始值 cur_start 为 0，end() 方法将限定最多能够迭代的次数，那么在对 ite 和 end 做比较的时候，只需要比较迭代的次数就可以了。由于迭代次数不可能是负数，所以类型使用 size_t。

这里为何不直接使用 iterator 当中的值做比较呢？这是因为迭代的步长 step 并不一定是 begin-end 的约数。如果我们直接采用 iterator 中的值做比较，就不能使用 != 了。

operator*() 用于取得迭代器中的值。

operator!= 用于和另一个迭代器做比较。因为在基于范围的 for 循环的迭代中，我们只需要用到 !=，因此，只需要实现这一个比较方法。

operator++ 用于对迭代器做正向迭代。当 step 为负数时，实际上会减少 iterator 的值。同样，在基于范围的 for 循环的迭代中也只需要实现前置 ++。

整理出这些必需的接口之后，它的功能实现就不困难了，如代码清单 1-11 所示。

代码清单 1-11　迭代器类的实现

```
namespace detail_range {

//////////////////////////////////////////////////////////////
/// The range iterator
//////////////////////////////////////////////////////////////

template <typename T>
class iterator
{
public:
   using value_type = T;
   using size_type = size_t;

private:
   size_type        cursor_;
   const value_type step_;
   value_type       value_;

public:
   iterator(size_type cur_start, value_type begin_val, value_type
step_val)
```

```
        : cursor_(cur_start)
        , step_  (step_val)
        , value_ (begin_val)
    {
        value_ += (step_ * cursor_);
    }

    value_type operator*() const { return value_; }

    bool operator!=(const iterator& rhs) const
    {
        return (cursor_ != rhs.cursor_);
    }

    iterator& operator++(void) //prefix ++ operator only
    {
        value_ += step_;
        ++ cursor_;
        return (*this);
    }
};

} //namespace detail_range
```

上述分别定义了 cursor_、step_ 和 value_ 这 3 个成员变量，来保存构造函数中传递进来的参数。

在 iterator 的构造中，我们通过一个简单的计算得到 value_ 当前正确的值。

在之后的 operator++（前置 ++）运算符中，我们通过 += 操作计算出 value_ 的下一个值。

在上面的代码中，step_ 被定义为一个常量。这是因为 step_ 作为步长的存储，在迭代器中是不能被修改的。

接下来思考 range 类型的具体实现。

基于范围的 for 循环只需要对象具有 begin 和 end 方法，因此，impl 类可以如代码清单 1-12 所示。

代码清单 1-12 impl 类的声明

```
template <typename T>
class impl
{
public:
    using value_type      = T;
    using reference       = const value_type&;
    using const_reference = const value_type&;
    using iterator        = const
detail_range::iterator<value_type>;
    using const_iterator  = const
detail_range::iterator<value_type>;
    using size_type       = typename iterator::size_type;

    impl(value_type begin_val, value_type end_val, value_type
step_val);
```

```
size_type size(void) const;

const_iterator begin(void)const;
const_iterator end(void)const;
};
```

impl 是 range 函数最终返回的一个类似容器的对象，因此，我们给它定义了容器所拥有的基本的概念抽象，比如 value_type、reference 及 iterator。

很明显，我们不希望外部能够修改 range，这个 range 应当是一个仅能通过构造函数作一次性赋值的对象，因此 reference 及 iterator 都应该是 const 类型，并且 begin、end 接口也应当返回 const_iterator。

下面，让我们来实现 impl 类，如代码清单 1-13 所示。

代码清单 1-13　impl 类的实现

```
namespace detail_range {

////////////////////////////////////////////////////////
/// The impl class
////////////////////////////////////////////////////////

template <typename T>
class impl
{
public:
   using value_type      = T;
   using reference       = const value_type&;
   using const_reference = const value_type&;
   using iterator        = const
detail_range::iterator<value_type>;
   using const_iterator  = const
detail_range::iterator<value_type>;
   using size_type       = typename iterator::size_type;

private:
   const value_type begin_;
   const value_type end_;
   const value_type step_;
   const size_type  max_count_;

   size_type get_adjusted_count(void) const
   {
      if (step_ > 0 && begin_ >= end_)
         throw std::logic_error("End value must be greater than begin value.");
      else
      if (step_ < 0 && begin_ <= end_)
         throw std::logic_error("End value must be less than begin value.");

      size_type x = static_cast<size_type>((end_ - begin_) /
step_);
      if (begin_ + (step_ * x) != end_) ++x;
```

```
        return x;
    }

public:
    impl(value_type begin_val, value_type end_val, value_type
step_val)
        : begin_(begin_val)
        , end_  (end_val)
        , step_ (step_val)
        , max_count_(get_adjusted_count())
    {}

    size_type size(void) const
    {
        return max_count_;
    }

    const_iterator begin(void) const
    {
        return { 0, begin_, step_ };
    }

    const_iterator end(void) const
    {
        return { max_count_, begin_, step_ };
    }
};

} // namespace detail_range
```

在上面的实现中，只允许 impl 做初始化时的赋值，而不允许中途修改，因此，它的成员变量都是 const 类型的。

通过 private 的 get_adjusted_count 方法，我们在获取 max_count_ 也就是最大迭代次数时会先判断 begin_、end_ 和 step_ 的合法性。很显然，合法的组合将只可能计算出大于等于 0 的最大迭代次数。

之所以在 get_adjusted_count 中对计算的值做自增的调整，是为了让计算的结果向上取整。

最后，我们实现出 range 的外覆函数模板，如代码清单 1-14 所示。

代码清单 1-14　range 类的外覆函数模板

```
////////////////////////////////////////////////////////////
/// Make a range of [begin, end) with a custom step
////////////////////////////////////////////////////////////

template <typename T>
detail_range::impl<T> range(T end)
{
    return{ {}, end, 1 };
}
```

```
template <typename T>
detail_range::impl<T> range(T begin, T end)
{
   return{ begin, end, 1 };
}

template <typename T, typename U>
auto range(T begin, T end, U step)
   -> detail_range::impl<decltype(begin + step)>
{
   //may be narrowing
   using r_t = detail_range::impl<decltype(begin + step)>;
   return r_t(begin, end, step);
}
```

可以看到，通过初始化列表，可以让 return 的书写非常便捷。

这里需要注意的是第三个 range 重载。我们当然不希望限制用户使用 range(1, 2, 0.1) 这种方式来创建区间 [1, 2)，因此，step 可以是不同的数据类型。

那么相对的，impl 的模板参数类型应当是 begin+step 的返回类型。因此，我们需要通过返回值后置语法，并使用 decltype(begin+step) 来推断出 range 的返回值类型。

这样，在运行 return 的时候，可能会导致类型收窄（整型和浮点型的加法运算将总是返回浮点型），不过此处的类型收窄一般不会影响计算结果，因此，最后一个重载函数需要使用直接初始化方式返回 impl 对象。

下面运行一组示例来看看效果，如代码清单 1-15 所示。

代码清单 1-15　测试代码

```
void test_range(void)
{
   std::cout << "range(15):";
   for (int i : range(15))
   {
      std::cout << " " << i;
   }
   std::cout << std::endl;

   std::cout << "range(2, 6):";
   for (auto i : range(2, 6))
   {
      std::cout << " " << i;
   }
   std::cout << std::endl;

   const int x = 2, y = 6, z = 3;
   std::cout << "range(2, 6, 3):";
   for (auto i : range(x, y, z))
```

```
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range(-2, -6, -3):";
    for (auto i : range(-2, -6, -3))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range(10.5, 15.5):";
    for (auto i : range(10.5, 15.5))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range(35, 27, -1):";
    for (int i : range(35, 27, -1))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range(2, 8, 0.5):";
    for (auto i : range(2, 8, 0.5))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range(8, 7, -0.1):";
    for (auto i : range(8, 7, -0.1))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;

    std::cout << "range('a', 'z'):";
    for (auto i : range('a', 'z'))
    {
        std::cout << " " << i;
    }
    std::cout << std::endl;
}

int main(void)
{
    test_range();
```

```
return 0;
}
```

示例的运行结果如下：

```
range(15): 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
range(2, 6): 2 3 4 5
range(2, 6, 3): 2 5
range(-2, -6, -3): -2 -5
range(10.5, 15.5): 10.5 11.5 12.5 13.5 14.5
range(35, 27, -1): 35 34 33 32 31 30 29 28
range(2, 8, 0.5): 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5
range(8, 7, -0.1): 8 7.9 7.8 7.7 7.6 7.5 7.4 7.3 7.2 7.1
range('a', 'z'): a b c d e f g h i j k l m n o p q r s t u v w x y
```

通过上面的 range 实现，我们不仅完成了自定义类型的 range-based for 支持，同时也使用了我们前面所学的自动类型推导、模板别名以及初始化列表的相关知识。

我们完成的这个 range，目前还有一些限制（比如参数传递采用值语义，不支持 range 返回值的存储后修改等）。如果读者感兴趣，可以自己进一步完善 range 的实现。

1.5　std::function 和 bind 绑定器

C++11 增加了 std::function 和 std::bind，不仅让我们使用标准库函数时变得更加方便，而且还能方便地实现延迟求值。

1.5.1　可调用对象

在 C++ 中，存在“可调用对象（Callable Objects）”这么一个概念。准确来说，可调用对象有如下几种定义[㊀]：

1）是一个函数指针。

2）是一个具有 operator() 成员函数的类对象（仿函数）。

3）是一个可被转换为函数指针的类对象。

4）是一个类成员（函数）指针。

它们在程序中的应用如代码清单 1-16 所示。

代码清单 1-16　可调用对象的使用示例

```
void func(void)
{
   // ...
}
```

㊀ ISO/IEC 14882:2011，20.8.1 Definitions，第 3 款、第 4 款；20.8 Function objects，第 1 款。

```
struct Foo
{
   void operator()(void)
   {
      //...
   }
};

struct Bar
{
   using fr_t = void(*)(void);

   static void func(void)
   {
      //...
   }

   operator fr_t(void)
   {
      return func;
   }
};

struct A
{
   int a_;

   void mem_func(void)
   {
      //...
   }
};

int main(void)
{
   void(* func_ptr)(void) = &func; //1. 函数指针
   func_ptr();

   Foo foo;                        //2. 仿函数
   foo();

   Bar bar;                        //3. 可被转换为函数指针的类对象
   bar();

   void (A::*mem_func_ptr)(void)   //4. 类成员函数指针
         = &A::mem_func;
   int A::*mem_obj_ptr             //    或者是类成员指针
         = &A::a_;
   A aa;
   (aa.*mem_func_ptr)();
   aa.*mem_obj_ptr = 123;
```

```
    return 0;
}
```

从上述可以看到，除了类成员指针之外，上面定义涉及的对象均可以像一个函数那样做调用操作。

在 C++11 中，像上面例子中的这些对象（func_ptr、foo、bar、mem_func_ptr、mem_obj_ptr）都被称做可调用对象。相对应的，这些对象的类型被统称为“可调用类型”。

细心的读者可能会发现，上面对可调用类型的定义里并没有包括函数类型或者函数引用（只有函数指针）。这是因为函数类型并不能直接用来定义对象；而函数引用从某种意义上来说，可以看做一个 const 的函数指针。

C++ 中的可调用对象虽然具有比较统一的操作形式（除了类成员指针之外，都是后面加括号进行调用），但定义方法五花八门。这样在我们试图使用统一的方式保存，或传递一个可调用对象时，会十分烦琐。

现在，C++11 通过提供 std::function 和 std::bind 统一了可调用对象的各种操作。

1.5.2　可调用对象包装器——std::function

std::function 是可调用对象的包装器。它是一个类模板，可以容纳除了类成员（函数）指针之外的所有可调用对象。通过指定它的模板参数，它可以用统一的方式处理函数、函数对象、函数指针，并允许保存和延迟执行它们。

下面看一个示例，如代码清单 1-17 所示。

代码清单 1-17　std::function 的基本用法

```
#include <iostream>   // std::cout
#include <functional> // std::function

void func(void)
{
    std::cout << __FUNCTION__ << std::endl;
}

class Foo
{
public:
    static int foo_func(int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ") ->: ";
        return a;
    }
};

class Bar
{
public:
```

```
    int operator()(int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ") ->: ";
        return a;
    }
};

int main(void)
{
    std::function<void(void)> fr1 = func;        //绑定一个普通函数
    fr1();

    //绑定一个类的静态成员函数
    std::function<int(int)> fr2 = Foo::foo_func;
    std::cout << fr2(123) << std::endl;

    Bar bar;
    fr2 = bar;                                   //绑定一个仿函数
    std::cout << fr2(123) << std::endl;

    return 0;
}
```

运行结果如下：

```
func
foo_func(123) ->: 123
operator()(123) ->: 123
```

从上面我们可以看到 std::function 的使用方法，当我们给 std::function 填入合适的函数签名（即一个函数类型，只需要包括返回值和参数表）之后，它就变成了一个可以容纳所有这一类调用方式的“函数包装器”。

再来看如代码清单 1-18 所示的例子。

代码清单 1-18 std::function 作为回调函数的示例

```
#include <iostream>                              //std::cout
#include <functional>                            //std::function

class A
{
    std::function<void()> callback_;

public:
    A(const std::function<void()>& f)
        : callback_(f)
    {}

    void notify(void)
```

```
    {
        callback_();    // 回调到上层
    }
};

class Foo
{
public:
    void operator()(void)
    {
        std::cout << __FUNCTION__ << std::endl;
    }
};

int main(void)
{
    Foo foo;
    A aa(foo);
    aa.notify();

    return 0;
}
```

从上面的例子中可以看到，std::function 可以取代函数指针的作用。因为它可以保存函数延迟执行，所以比较适合作为回调函数，也可以把它看做类似于 C# 中特殊的委托（只有一个成员的委托）。

同样，std::function 还可以作为函数入参，如代码清单 1-19 所示。

代码清单 1-19　std::function 作为函数入参的示例

```
#include <iostream>    // std::cout
#include <functional> // std::function

void call_when_even(int x, const std::function<void(int)>& f)
{
    if (!(x & 1)) // x % 2 == 0
    {
        f(x);
    }
}

void output(int x)
{
    std::cout << x << " ";
}

int main(void)
{
    for(int i = 0; i < 10; ++i)
```

```
    {
        call_when_even(i, output);
    }
    std::cout << std::endl;
    return 0;
}
```

输出结果如下：

```
0 2 4 6 8
```

从上面的例子中可以看到，std::function 比普通函数指针更灵活和便利。

在下一节我们可以看到当 std::function 和 std::bind 配合起来使用时，所有的可调用对象（包括类成员函数指针和类成员指针）都将具有统一的调用方式。

1.5.3 std::bind 绑定器

std::bind 用来将可调用对象与其参数一起进行绑定。绑定后的结果可以使用 std::function 进行保存，并延迟调用到任何我们需要的时候。

通俗来讲，它主要有两大作用：

1）将可调用对象与其参数一起绑定成一个仿函数。

2）将多元（参数个数为 n，n>1）可调用对象转成一元或者（n–1）元可调用对象，即只绑定部分参数。

下面来看看它的实际使用，如代码清单 1-20 所示。

代码清单 1-20 std::bind 的基本用法

```
#include <iostream>       // std::cout
#include <functional>     // std::bind

void call_when_even(int x, const std::function<void(int)>& f)
{
    if (!(x & 1))         // x % 2 == 0
    {
        f(x);
    }
}

void output(int x)
{
    std::cout << x << " ";
}

void output_add_2(int x)
{
    std::cout << x + 2 << " ";
}
```

```
int main(void)
{
   {
      auto fr = std::bind(output, std::placeholders::_1);
      for(int i = 0; i < 10; ++i)
      {
         call_when_even(i, fr);
      }
      std::cout << std::endl;
   }
   {
      auto fr = std::bind(output_add_2, std::placeholders::_1);
      for(int i = 0; i < 10; ++i)
      {
         call_when_even(i, fr);
      }
      std::cout << std::endl;
   }
   return 0;
}
```

输出结果如下：

```
0 2 4 6 8
2 4 6 8 10
```

同样还是上面 std::function 中最后的一个例子，只是在这里，我们使用了 std::bind，在函数外部通过绑定不同的函数，控制了最后的执行结果。

我们使用 auto fr 保存 std::bind 的返回结果，是因为我们并不关心 std::bind 真正的返回类型（实际上 std::bind 的返回类型是一个 stl 内部定义的仿函数类型），只需要知道它是一个仿函数，可以直接赋值给一个 std::function。当然，这里直接使用 std::function 类型来保存 std::bind 的返回值也是可以的。

std::placeholders::_1 是一个占位符，代表这个位置将在函数调用时，被传入的第一个参数所替代。

因为有了占位符的概念，std::bind 的使用非常灵活，如代码清单 1-21 所示。

代码清单 1-21　std::bind 的占位符

```
#include <iostream>      // std::cout
#include <functional>    // std::bind

void output(int x, int y)
{
   std::cout << x << " " << y << std::endl;
}
```

```
int main(void)
{
   std::bind(output, 1, 2)();                              //输出：1 2
   std::bind(output, std::placeholders::_1, 2)(1);         //输出：1 2
   std::bind(output, 2, std::placeholders::_1)(1);         //输出：2 1
   //error: 调用时没有第二个参数
   std::bind(output, 2, std::placeholders::_2)(1);
   std::bind(output, 2, std::placeholders::_2)(1, 2);      //输出：2 2
   //调用时的第一个参数被吞掉了
   std::bind(output, std::placeholders::_1,
   std::placeholders::_2)(1, 2);                           //输出：1 2

   std::bind(output, std::placeholders::_2,
   std::placeholders::_1)(1, 2);                           //输出：2 1
   return 0;
}
```

上面对 std::bind 的返回结果直接施以调用。可以看到，std::bind 可以直接绑定函数的所有参数，也可以仅绑定部分参数。

在绑定部分参数的时候，通过使用 std::placeholders，来决定空位参数将会属于调用发生时的第几个参数。

下面再来看一个例子，如代码清单 1-22 所示。

代码清单 1-22 std::bind 和 std::function 配合使用

```
#include <iostream>
#include <functional>

class A
{
public:
   int i_ = 0;

   void output(int x, int y)
   {
      std::cout << x << " " << y << std::endl;
   }
};

int main(void)
{
   A a;
   std::function<void(int, int)> fr =
         std::bind(&A::output, &a, std::placeholders::_1
, std::placeholders::_2);
   fr(1, 2);                                               //输出：1 2

   std::function<int&(void)> fr_i =std::bind(&A::i_, &a);
   fr_i() = 123;
```

```
    std::cout << a.i_ << std::endl; // 输出 : 123

    return 0;
}
```

fr 的类型是 std::function<void(int, int)>。我们通过使用 std::bind，将 A 的成员函数 output 的指针和 a 绑定，并转换为一个仿函数放入 fr 中存储。

之后，std::bind 将 A 的成员 i_ 的指针和 a 绑定，返回的结果被放入 std::function<int&(void)> 中存储，并可以在需要时修改访问这个成员。

现在，通过 std::function 和 std::bind 的配合，所有的可调用对象均有了统一的操作方法。

下面再来看几个 std::bind 的使用例子。

1. 使用 bind 简化和增强 bind1st 和 bind2nd

其实 bind 简化和增强了之前标准库中 bind1st 和 bind2nd，它完全可以替代 bind1s 和 bind2st，并且能组合函数。我们知道，bind1st 和 bind2nd 的作用是将一个二元算子转换成一个一元算子，代码如下：

```
// 查找元素值大于 10 的元素的个数
int count = std::count_if(coll.begin(), coll.end(),
std::bind1st(less<int>(), 10));
// 查找元素之小于 10 的元素
int count = std::count_if(coll.begin(), coll.end(),
std::bind2nd(less<int>(), 10));
```

本质上是对一个二元函数 less<int> 的调用，但是它却要分别用 bind1st 和 bind2nd，并且还要想想到底是用 bind1st 还是 bind2nd，用起来十分不便。

现在我们有了 bind，就可以以统一的方式去实现了，代码如下：

```
using std::placeholders::_1;
// 查找元素值大于 10 的元素的个数
int count = std::count_if(coll.begin(),
coll.end(),std::bind(less<int>(), 10, _1));
// 查找元素之小于 10 的元素
int count = std::count_if(coll.begin(),
coll.end(),std::bind(less<int>(), _1, 10));
```

这样就不用关心到底是用 bind1st 还是 bind2nd，只需要使用 bind 即可。

2. 使用组合 bind 函数

bind 还有一个强大之处就是可以组合多个函数。假设要找出集合中大于 5 小于 10 的元素个数应该怎么做呢？

首先，需要一个用来判断是否大于 5 的功能闭包，代码如下：

```
std::bind(std::greater<int>(), std::placeholders::_1, 5);
```

这里 std::bind 返回的仿函数只有一个 int 参数。当输入了这个 int 参数后，输入的 int 值

将直接和 5 进行大小比较，并在大于 5 时返回 true。

然后，我们需要一个判断是否小于 10 的功能闭包：

```
std::bind(std::less_equal<int>(),std::placeholders::_1,10);
```

有了这两个闭包之后，只需要用逻辑与把它们连起来：

```
using std::placeholders::_1;
std::bind(std::logical_and<bool>(),
std::bind(std::greater<int>(), _1, 5),
std::bind(std::less_equal<int>(), _1, 10));
```

然后就可以复合多个函数（或者说闭包）的功能：

```
using std::placeholders::_1;
// 查找集合中大于 5 小于 10 的元素个数
auto f = std::bind(std::logical_and<bool>(),
std::bind(std::greater<int>(), _1, 5),
std::bind(std::less_equal<int>(), _1, 10));
int count = std::count_if(coll.begin(), coll.end(), f);
```

1.6 lambda 表达式

lambda 表达式是 C++11 最重要也最常用的一个特性之一。其实在 C#3.5 中就引入了 lambda，Java 则至今还没引入，要等到 Java 8 中才有 lambda 表达式。

lambda 来源于函数式编程的概念，也是现代编程语言的一个特点。C++11 这次终于把 lambda 加进来了。

lambda 表达式有如下优点：

- 声明式编程风格：就地匿名定义目标函数或函数对象，不需要额外写一个命名函数或者函数对象。以更直接的方式去写程序，好的可读性和可维护性。
- 简洁：不需要额外再写一个函数或者函数对象，避免了代码膨胀和功能分散，让开发者更加集中精力在手边的问题，同时也获取了更高的生产率。
- 在需要的时间和地点实现功能闭包，使程序更灵活。

下面，先从 lambda 表达式的基本功能开始介绍它。

1.6.1 lambda 表达式的概念和基本用法

lambda 表达式定义了一个匿名函数，并且可以捕获一定范围内的变量。lambda 表达式的语法形式可简单归纳如下：

```
[ capture ] ( params ) opt -> ret { body; };
```

其中：

capture 是捕获列表；params 是参数表；opt 是函数选项；ret 是返回值类型；body 是函数体。

因此，一个完整的 lambda 表达式看起来像这样：

```
auto f = [](int a) -> int { return a + 1; };
std::cout << f(1) << std::endl;          //输出：2
```

可以看到，上面通过一行代码定义了一个小小的功能闭包，用来将输入加 1 并返回。

在 C++11 中，lambda 表达式的返回值是通过前面介绍的返回值后置语法来定义的。其实很多时候，lambda 表达式的返回值是非常明显的，比如上例。因此，C++11 中允许省略 lambda 表达式的返回值定义：

```
auto f = [](int a){ return a + 1; };
```

这样编译器就会根据 return 语句自动推导出返回值类型。

需要注意的是，初始化列表不能用于返回值的自动推导：

```
auto x1 = [](int i){ return i; };        //OK: return type is int
auto x2 = [](){ return { 1, 2 }; };      //error: 无法推导出返回值类型
```

这时我们需要显式给出具体的返回值类型。

另外，lambda 表达式在没有参数列表时，参数列表是可以省略的。因此像下面的写法都是正确的：

```
auto f1 = [](){ return 1; };
auto f2 = []{ return 1; };               //省略空参数表
```

lambda 表达式可以通过捕获列表捕获一定范围内的变量：

- [] 不捕获任何变量。
- [&] 捕获外部作用域中所有变量，并作为引用在函数体中使用（按引用捕获）。
- [=] 捕获外部作用域中所有变量，并作为副本在函数体中使用（按值捕获）。
- [=, &foo] 按值捕获外部作用域中所有变量，并按引用捕获 foo 变量。
- [bar] 按值捕获 bar 变量，同时不捕获其他变量。
- [this] 捕获当前类中的 this 指针，让 lambda 表达式拥有和当前类成员函数同样的访问权限。如果已经使用了 & 或者 =，就默认添加此选项。捕获 this 的目的是可以在 lamda 中使用当前类的成员函数和成员变量。

下面看一下它的具体用法，如代码清单 1-23 所示。

代码清单 1-23　lambda 表达式的基本用法

```
class A
{
public:
```

```
    int i_ = 0;

    void func(int x, int y)
    {
        auto x1 = []{ return i_; };               //error, 没有捕获外部变量
        auto x2 = [=]{ return i_ + x + y; };      //OK, 捕获所有外部变量
        auto x3 = [&]{ return i_ + x + y; };      //OK, 捕获所有外部变量
        auto x4 = [this]{ return i_; };           //OK, 捕获 this 指针
        auto x5 = [this]{ return i_ + x + y; };   //error, 没有捕获 x、y
        auto x6 = [this, x, y]{ return i_ + x + y; }; //OK, 捕获 this 指针、x、y
        auto x7 = [this]{ return i_++; };         //OK, 捕获 this 指针, 并修改成员的值
    }
};

int a = 0, b = 1;
auto f1 = []{ return a; };              //error, 没有捕获外部变量
auto f2 = [&]{ return a++; };           //OK, 捕获所有外部变量, 并对 a 执行自加运算
auto f3 = [=]{ return a; };             //OK, 捕获所有外部变量, 并返回 a
auto f4 = [=]{ return a++; };           //error, a 是以复制方式捕获的, 无法修改
auto f5 = [a]{ return a + b; };         //error, 没有捕获变量 b
auto f6 = [a, &b]{ return a + (b++); };//OK, 捕获 a 和 b 的引用, 并对 b 做自加运算
auto f7 = [=, &b]{ return a + (b++); }; //OK, 捕获所有外部变量和 b 的引用, 并对 b 做自加运算
```

从上例中可以看到，lambda 表达式的捕获列表精细地控制了 lambda 表达式能够访问的外部变量，以及如何访问这些变量。

需要注意的是，默认状态下 lambda 表达式无法修改通过复制方式捕获的外部变量。如果希望修改这些变量的话，我们需要使用引用方式进行捕获。

一个容易出错的细节是关于 lambda 表达式的延迟调用的：

```
int a = 0;
auto f = [=]{ return a; };              //按值捕获外部变量

a += 1;                                 //a 被修改了

std::cout << f() << std::endl;          //输出?
```

在这个例子中，lambda 表达式按值捕获了所有外部变量。在捕获的一瞬间，a 的值就已经被复制到 f 中了。之后 a 被修改，但此时 f 中存储的 a 仍然还是捕获时的值，因此，最终输出结果是 0。

如果希望 lambda 表达式在调用时能够即时访问外部变量，我们应当使用引用方式捕获。

从上面的例子中我们知道，按值捕获得到的外部变量值是在 lambda 表达式定义时的值。此时所有外部变量均被复制了一份存储在 lambda 表达式变量中。此时虽然修改 lambda 表达式中的这些外部变量并不会真正影响到外部，我们却仍然无法修改它们。

那么如果希望去修改按值捕获的外部变量应当怎么办呢？这时，需要显式指明 lambda 表达式为 mutable：

```
int a = 0;
auto f1 = [=]{ return a++; };              //error, 修改按值捕获的外部变量
auto f2 = [=]() mutable { return a++; }; //OK, mutable
```

需要注意的一点是，被 mutable 修饰的 lambda 表达式就算没有参数也要写明参数列表。

最后，介绍一下 lambda 表达式的类型。

lambda 表达式的类型在 C++11 中被称为“闭包类型（Closure Type）”。它是一个特殊的，匿名的非 nunion 的类类型[㈠]。

因此，我们可以认为它是一个带有 operator() 的类，即仿函数。因此，我们可以使用 std::function 和 std::bind 来存储和操作 lambda 表达式：

```
std::function<int(int)>  f1 = [](int a){ return a; };
std::function<int(void)> f2 = std::bind([](int a){ return a; },
123);
```

另外，对于没有捕获任何变量的 lambda 表达式，还可以被转换成一个普通的函数指针：

```
using func_t = int(*)(int);
func_t f = [](int a){ return a; };
f(123);
```

lambda 表达式可以说是就地定义仿函数闭包的“语法糖”。它的捕获列表捕获住的任何外部变量，最终均会变为闭包类型的成员变量。而一个使用了成员变量的类的 operator()，如果能直接被转换为普通的函数指针，那么 lambda 表达式本身的 this 指针就丢失掉了。而没有捕获任何外部变量的 lambda 表达式则不存在这个问题。

这里也可以很自然地解释为何按值捕获无法修改捕获的外部变量。因为按照 C++ 标准，lambda 表达式的 operator() 默认是 const 的[㈡]。一个 const 成员函数是无法修改成员变量的值的。而 mutable 的作用，就在于取消 operator() 的 const。

需要注意的是，没有捕获变量的 lambda 表达式可以直接转换为函数指针，而捕获变量的 lambda 表达式则不能转换为函数指针[㈢]。看看下面的代码：

```
typedef void(*Ptr)(int*);
// 正确，没有状态的 lambda（没有捕获）的 lambda 表达式可以直接转换为函数指针
Ptr p = [](int* p){delete p;};
Ptr p1 = [&](int* p){delete p;};          //错误，有状态的 lambda 不能直接转换为函数指针
```

上面第二行代码能编译通过，而第三行代码不能编译通过，因为第三行的代码捕获了变量，不能直接转换为函数指针。

1.6.2　声明式的编程风格，简洁的代码

就地定义匿名函数，不再需要定义函数对象，大大简化了标准库算法的调用。比如，在

㈠ ISO/IEC 14882:2011，5.1.2 Lambda expressions，第 3 款。

㈡ ISO/IEC 14882:2011，5.1.2 Lambda expressions，第 5 款。

㈢ ISO/IEC 14882:2011 93 页第六条。

C++11 之前，我们要调用 for_each 函数将 vector 中的偶数打印出来，如代码清单 1-24 所示。

代码清单 1-24 lambda 表达式代替函数对象的示例

```
class CountEven
{
   int& count_;

public:
   CountEven(int& count)
      : count_(count)
   {}

   void operator()(int val)
   {
      if (!(val & 1))     //val % 2 == 0
      {
         ++ count_;
      }
   }
};

std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), CountEven(even_count));
std::cout << "The number of even is " << even_count << std::endl;
```

这样写既烦琐又容易出错。有了 lambda 表达式以后，我们可以使用真正的闭包概念来替换掉这里的仿函数，代码如下：

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), [&even_count](int val)
{
   if (!(val & 1))        //val % 2 == 0
   {
      ++ even_count;
   }
});
std::cout << "The number of even is " << even_count << std::endl;
```

lambda 表达式的价值在于，就地封装短小的功能闭包，可以极其方便地表达出我们希望执行的具体操作，并让上下文结合得更加紧密。

1.6.3 在需要的时间和地点实现闭包，使程序更灵活

在 1.5.3 节中使用了 std::bind 组合了多个函数，实现了计算集合中大于 5 小于 10 的元素个数的功能：

```
using std::placeholders::_1;
//查找集合中大于 5 小于 10 的元素个数
auto f = std::bind(std::logical_and<bool>(),
                   std::bind(std::greater<int>(), _1, 5),
                   std::bind(std::less_equal<int>(), _1, 10));
int count = std::count_if(coll.begin(), coll.end(), f);
```

通过使用 lambda 表达式，可以轻松地实现类似的功能：

```
//查找大于 5 小于 10 的元素的个数
int count = std::count_if(coll.begin(), coll.end(), [](int
x){return x > 5 && x < 10;});
```

可以看到，lambda 表达式比 std::bind 的灵活性更好，也更为简洁。当然，这都得益于 lambda 表达式的特征，它可以任意封装出功能闭包，使得短小的逻辑可以以最简洁清晰的方式表达出来。

比如说，我们这时候的需求变更了，只希望查找大于 10，或小于 10 的个数：

```
//查找大于 10 的元素的个数
int count = std::count_if(coll.begin(), coll.end(), [](int
x){return x > 10;});
//查找小于 10 的元素的个数
int count = std::count_if(coll.begin(), coll.end(), [](int
x){return x < 10;});
```

不论如何去写，使用 lambda 表达式的修改量是非常小的，清晰度也很好。

lambda 和 std::function 的效果是一样的，代码还更简洁了。一般情况下可以直接用 lambda 来代替 function，但还不能完全替代，因为还有些老的库，比如 boost 的一些库就不支持 lambda，还需要 function。

C++11 引入函数式编程的概念中的 lambda，让代码更简洁，更灵活，也更强大，并提高了开发效率，提高了可维护性。

1.7　tupe 元组

tuple 元组是一个固定大小的不同类型值的集合，是泛化的 std::pair。和 C# 中的 tuple 类似，但是比 C# 中的 tuple 强大得多。我们也可以把它当作一个通用的结构体来用，不需要创建结构体又获取结构体的特征，在某些情况下可以取代结构体，使程序更简洁、直观。

tuple 看似简单，其实它是简约而不简单，可以说它是 C++11 中一个既简单又复杂的类型，简单的一面是它很容易使用，复杂的一面是它内部隐藏了太多细节，往往要和模板元的一些技巧结合起来使用。下面看看 tuple 的基本用法。

先构造一个 tuple：

```
tuple<constchar*, int>tp = make_tuple(sendPack,nSendSize); //构造一个 tuple
```

这个 tuple 等价于一个结构体：

```
struct A
{
   char* p;
   int len;
};
```

用 tuple<const char*, int>tp 就可以不用创建这个结构体了，而作用是一样的，是不是更简洁直观了？还有一种方法也可以创建元组，用 std::tie，它会创建一个元组的左值引用。

```
auto tp = return std::tie(1, "aa", 2);
// tp 的类型实际是: std::tuple<int&,string&, int&>
```

再看看如何获取元组的值：

```
constchar* data = tp.get<0>();                    // 获取第一个值
int len = tp.get<1>();                            // 获取第二个值
```

还有一种方法也可以获取元组的值，通过 std::tie 解包 tuple。

```
int x,y;
string a;
std::tie(x,a,y) = tp;
```

通过 tie 解包后，tp 中 3 个值会自动赋值给 3 个变量。解包时，如果只想解某个位置的值时，可以用 std::ignore 占位符来表示不解某个位置的值。比如我们只想解第 3 个值：

```
std::tie(std::ignore,std::ignore,y) = tp;         // 只解第 3 个值了
```

还有一个创建右值的引用元组方法：forward_as_tuple。

```
std::map<int, std::string> m;
m.emplace(std::forward_as_tuple(10, std::string(20, 'a')));
```

它实际上创建了一个类似于 std::tuple<int&&, std::string&&> 类型的 tuple。

我们还可以通过 tuple_cat 连接多个 tupe，代码如下：

```
int main()
{
   std::tuple<int, std::string, float> t1(10, "Test", 3.14);
   int n = 7;
   auto t2 = std::tuple_cat(t1, std::make_pair("Foo", "bar"), t1,
std::tie(n));
   n = 10;
   print(t2);
}
```

输出结果如下：

```
(10, Test, 3.14, Foo, bar, 10, Test, 3.14, 10)
```

可以看到 tuple 的基本用法还是比较简单的，也很容易使用。然而 tuple 是简约而不简单，它有很多高级的用法，具体内容将在第 2 章中介绍。

tuple 虽然可以用来代替简单的结构体，但不要滥用，如果用 tuple 来替代 3 个以上字段的结构体时就不太合适了，因为使用 tuple 可能会导致代码的易读性降低，如果到处都是 std::get<N>(tuple)，反而不直观，建议对于多个字段的结构体时，不要使用 tuple。

1.8　总结

本章主要介绍了通过一些 C++11 的特性简化代码，使代码更方便、简洁和优雅。首先讨论了自动类型推断的两个关键字 auto 和 decltype，通过这两个关键字可以化繁为简，使我们不仅能方便地声明变量，还能获取复杂表达式的类型，将二者和返回值后置组合起来能解决函数的返回值难以推断的难题。

模板别名和模板默认参数可以使我们更方便地定义模板，将复杂的模板定义用一个简短更可读的名称来表示，既减少了烦琐的编码又提高了代码的可读性。

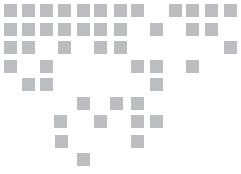
range-based for 循环可以用更简洁的方式去遍历数组和容器，它还可以支持自定义的类型，只要自定义类型满足 3 个条件即可。

初始化列表和统一的初始化使得初始化对象的方式变得更加简单、直接和统一。

std::function 不仅是一个函数语义的包装器，还能绑定任意参数，可以更灵活地实现函数的延迟执行。

lambda 表达式能更方便地使用 STL 算法，就地定义的匿名函数免除了维护一大堆函数对象的烦琐，也提高了程序的可读性。

tuple 元组可以作为一个灵活的轻量级的小结构体，可以用来替代简单的结构体，它有一个很好的特点就是能容纳任意类型和任意数量的元素，比普通的容器更灵活，功能也更强大。但是它也有复杂的一面，tuple 的解析和应用往往需要模板元的一些技巧，对使用者有一定的要求。



Chapter 2 第 2 章

使用 C++11 改进程序性能

C++11 中引入了右值引用和移动语义，可以避免无谓的复制，提高了程序性能，相应的，C++11 的容器还增加了一些右值版本的插入函数。C++11 还增加了一些无序容器，如 unordered_map、unordered_multimap，这些容器不同于标准库的 map，map 在插入元素时，会自动排序，在一些场景下这种自动排序会影响性能，而且在不需要排序的场景下，它还是会自动排序，造成了额外的性能损耗。为了改进这些缺点，C++11 引入了无序容器，这些无序容器在插入元素时不会自动排序，在不需要排序时，不会带来额外的性能损耗，提高了程序性能。本章将分别介绍右值引用相关的新特性。

2.1 右值引用

C++11 增加了一个新的类型，称为右值引用（R-value reference），标记为 T &&。在介绍右值引用类型之前先要了解什么是左值和右值。左值是指表达式结束后依然存在的持久对象，右值是指表达式结束时就不再存在的临时对象。一个区分左值与右值的便捷方法是：看能不能对表达式取地址，如果能，则为左值，否则为右值[㊀]。所有的具名变量或对象都是左值，而右值不具名。

在 C++11 中，右值由两个概念构成，一个是将亡值（xvalue, expiring value），另一个则是纯右值（prvalue, PureRvalue），比如，非引用返回的临时变量、运算表达式产生的临时变量、原始字面量和 lambda 表达式等都是纯右值。而将亡值是 C++11 新增的、与右值引用相

㊀ http://www.cnblogs.com/hujian/archive/2012/02/13/2348621.html

关的表达式，比如，将要被移动的对象、T&& 函数返回值、std::move 返回值和转换为 T&& 的类型的转换函数的返回值。

C++11 中所有的值必属于左值、将亡值、纯右值三者之一[㈠]，将亡值和纯右值都属于右值。区分表达式的左右值属性有一个简便方法：若可对表达式用 & 符取址，则为左值，否则为右值。

比如，简单的赋值语句：

```
int i = 0;
```

在这条语句中，i 是左值，0 是字面量，就是右值。在上面的代码中，i 可以被引用，0 就不可以了。字面量都是右值。[㈡]

2.1.1　&& 的特性

右值引用就是对一个右值进行引用的类型。因为右值不具名，所以我们只能通过引用的方式找到它。

无论声明左值引用还是右值引用都必须立即进行初始化，因为引用类型本身并不拥有所绑定对象的内存，只是该对象的一个别名。通过右值引用的声明，该右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样，只要该变量还活着，该右值临时量将会一直存活下去。[㈢]

看一下下面的代码：

```
#include <iostream>
using namespace std;

int g_constructCount=0;
int g_copyConstructCount=0;
int g_destructCount=0;
struct A
{
        A(){
                cout<<"construct: "<<++g_constructCount<<endl;
        }

        A(const A& a)
        {
                cout<<"copy construct: "<<++g_copyConstructCount <<endl;
        }
        ~A()
        {
                cout<<"destruct: "<<++g_destructCount<<endl;
        }
};
```

㈠ 《深入理解 C++11》(机械工业出版社 2013 年版）3.3.3 节。

㈡ http://www.ibm.com/developerworks/cn/aix/library/1307_lisl_c11/

㈢ 《深入理解 C++11》3.3.3 节。

```
A GetA()
{
        return A();
}

int main() {
        A a = GetA();
        return 0;
}
```

为了清楚地观察临时值，在 GCC 下编译时设置编译选项 -fno-elide-constructors 来关闭返回值优化效果。

输出结果：

```
construct: 1
copy construct: 1
destruct: 1
copy construct: 2
destruct: 2
destruct: 3
```

从上面的例子中可以看到，在没有返回值优化的情况下，拷贝构造函数调用了两次，一次是 GetA() 函数内部创建的对象返回后构造一个临时对象产生的，另一次是在 main 函数中构造 a 对象产生的。第二次的 destruct 是因为临时对象在构造 a 对象之后就销毁了。如果开启返回值优化，输出结果将是：

```
construct: 1
destruct: 1
```

可以看到返回值优化将会把临时对象优化掉，但这不是 C++ 标准，是各编译器的优化规则。我们在回到之前提到的可以通过右值引用来延长临时右值的生命周期，如果在上面的代码中我们通过右值引用来绑定函数返回值，结果又会是什么样的呢？在编译时设置编译选项 -fno-elide-constructors。

```
int main() {
        A&& a = GetA();
        return 0;
}
```

输出结果：

```
construct: 1
copy construct: 1
destruct: 1
destruct: 2
```

通过右值引用，比之前少了一次拷贝构造和一次析构，原因在于右值引用绑定了右值，让临时右值的生命周期延长了。我们可以利用这个特点做一些性能优化，即避免临时对象的

拷贝构造和析构，事实上，在 C++98/03 中，通过常量左值引用也经常用来做性能优化。将上面的代码改成：

```
const A& a = GetA();
```

输出的结果和右值引用一样，因为常量左值引用是一个“万能”的引用类型，可以接受左值、右值、常量左值和常量右值。需要注意的是普通的左值引用不能接受右值，比如这样的写法是不对的：

```
A& a = GetA();
```

上面的代码会报一个编译错误，因为非常量左值引用只能接受左值。

实际上 T&& 并不是一定表示右值，它绑定的类型是未定的，既可能是左值又可能是右值。看看这个例子：

```
template<typename T>
void f(T&& param);

f(10);                    // 10 是右值
int x = 10;
f(x);                     // x 是左值
```

从这个例子可以看出，param 有时是左值，有时是右值，因为在上面的例子中有 &&，这表示 param 实际上是一个未定的引用类型。这个未定的引用类型称为 universal references [⊖]（可以认为它是一种未定的引用类型），它必须被初始化，它是左值还是右值引用取决于它的初始化，如果 && 被一个左值初始化，它就是一个左值；如果它被一个右值初始化，它就是一个右值。

需要注意的是，只有当发生自动类型推断时（如函数模板的类型自动推导，或 auto 关键字），&& 才是一个 universal references。

```
template<typename T>
void f(T&& param);        // 这里 T 的类型需要推导，所以 && 是一个 universal references

template<typename T>
class Test {
   ...
   Test(Test&& rhs);      // 已经定义了一个特定的类型，没有类型推断
   ... // && 是一个右值引用
};

void f(Test&& param);     // 已经定义了一个确定的类型，没有类型推断，&& 是一个右值引用
```

再看一个复杂一点的例子：

```
template<typename T>
```

⊖ http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

```
void f(std::vector<T>&& param);
```

这里既有推断类型 T 又有确定类型 vector，那么这个 param 到底是什么类型呢？

答案是它是右值引用类型，因为在调用这个函数之前，这个 vector<T> 中的推断类型已经确定了，所以到调用 f 时没有类型推断了。再看下面这个例子：

```
template<typename T>
void f(const T&& param);
```

这个 param 是 universal references 吗？其实它也是一个右值引用类型。读者也许会不明白，T 不是推断类型吗，怎么会是右值引用类型？其实还有一条很关键的规则：universal references 仅仅在 T&& 下发生，任何一点附加条件都会使之失效，而变成一个普通的右值引用。

因此，上面的 T&& 在被 const 修饰之后就成为右值引用了。

由于存在 T&& 这种未定的引用类型，当它作为参数时，有可能被一个左值引用或者右值引用的参数初始化，这时经过类型推导的 T&& 类型，相比右值引用（&&）会发生类型的变化，这种变化被称为引用折叠。C++11 中的引用折叠规则如下：

1）所有的右值引用叠加到右值引用上仍然还是一个右值引用。

2）所有的其他引用类型之间的叠加都将变成左值引用。

左值或者右值是独立于它的类型的，右值引用可能是左值也可能是右值。比如下面的例子：

```
int&& var1 = x; // var1 的类型是 int&&

// var2 存在类型推导，因此是一个 universal references。这里 auto&& 最终会被推导为 int&
auto&& var2 = var1;
```

其中，var1 的类型是一个左值类型，但 var1 本身是一个左值；var1 是一个左值，根据引用折叠规则，var2 是一个 int&。下面再来看一个例子：

```
int w1, w2;
auto&& v1 = w1;
decltype(w1)&& v2 = w2;
```

其中，v1 是一个 universal reference，它被一个左值初始化，所以它最终是一个左值；v2 是一个右值引用类型，但它被一个左值初始化，一个左值初始化一个右值引用类型是不合法的，所以会编译报错。但是，如果希望把一个左值赋给一个右值引用类型该怎么做呢？用 std::move：

```
decltype(w1)&& v2 = std::move(w2);
```

std::move 可以将一个左值转换成右值。关于 std::move 的内容将在下一节介绍。

编译器会将已命名的右值引用视为左值，而将未命名的右值引用视为右值。看下面的例子：

```
void PrintValue(int& i)
{
        std::cout<<"lvalue : "<<i<<std::endl;
}
void PrintValue(int&& i)
{
        std::cout<<"rvalue : "<<i<<std::endl;
}

void Forward(int&& i)
{
        PrintValue(i);
}

int main()
{
        int i = 0;
        PrintValue(i);
        PrintValue(1);
        Forward (2);
}
```

将输出如下结果：

```
lvalue : 0
rvalue : 1
lvaue : 2
```

Forward 函数接收的是一个右值，但在转发给 PrintValue 时又变成了左值，因为在 Forward 中调用 PrintValue 时，右值 i 变成了一个命名的对象，编译器会将其当作左值处理。

对于 T&& 作为参数的时候需要注意 T 的类型，如代码清单 2-1 所示。

代码清单 2-1　输出引用类型的示例

```
#include <type_traits>
#include <typeinfo>
#ifndef _MSC_VER
#include <cxxabi.h>
#endif
#include <memory>
#include <string>
#include <cstdlib>

template <class T>
std::string type_name()
{
        typedef typename std::remove_reference<T>::type TR;
        std::unique_ptr<char, void(*)(void*)> own
                (
#ifndef __GNUC__
```

```
    nullptr,
    #else
    abi::__cxa_demangle(typeid(TR).name(), nullptr,
                    nullptr, nullptr),
    #endif
                    std::free
                    );
            std::string r = own != nullptr ? own.get() : typeid(TR).name();
            if (std::is_const<TR>::value)
                    r += " const";
            if (std::is_volatile<TR>::value)
                    r += " volatile";
            if (std::is_lvalue_reference<T>::value)
                    r += "&";
            else if (std::is_rvalue_reference<T>::value)
                    r += "&&";
            return r;
    }

    template<typename T>
    void Func(T&& t)
    {
            cout<<type_name<T>()<<endl;
    }
    void TestReference()
    {
            string str = "test";
            Func(str);
            Func(std::move(str));
    }
```

上述例子将输出：

```
std::string&
std::string
```

在上述例子中，abi::__cxa_demangle 将低级符号名解码（demangle）成用户级名字，使得 C++ 类型名具有可读性。可以通过一个例子展示 abi::__cxa_demangle 的作用，代码如下：

```
#include <iostream>
#include <typeinfo>
class Foo {};

int main()
{
        class Foo {};
        std::cout << typeid(Foo*[10]).name() << std::endl;
        class Foo {};
        std::cout << typeid(Foo*[10]).name() << std::endl;
}
```

结果如下：

```
class Foo * [10] //vc
A10_P3Foo        //gcc
```

在 gcc 中，类型名很难看明白，为了输出让用户能看明白的类型，在 gcc 中需作如下修改：

```
#include <iostream>
#include <typeinfo>
#ifndef _MSC_VER
#include <cxxabi.h>
#endif

class Foo {};

int main() {
   char* name = abi::__cxa_demangle(typeid(Foo*[10]).name(), nullptr, nullptr,
       nullptr);
   std::cout << name << std::endl;
   free(name);

   return 0;
}
```

结果如下：

```
Foo* [10]
```

这个例子中的 std::unique_ptr 是智能指针，相关内容将在第 4 章中介绍。可以看到，当 T&& 为模板参数时，如果输入左值，它会变成左值引用，而输入右值时则变为无引用的类型。这是因为 T&& 作为模板参数时，如果被左值 X 初始化，则 T 的类型为 X& ；如果被右值 X 初始化，则 T 的类型为 X。

&& 的总结如下：

1）左值和右值是独立于它们的类型的，右值引用类型可能是左值也可能是右值。

2）auto&& 或函数参数类型自动推导的 T&& 是一个未定的引用类型，被称为 universal references，它可能是左值引用也可能是右值引用类型，取决于初始化的值类型。

3）所有的右值引用叠加到右值引用上仍然是一个右值引用，其他引用折叠都为左值引用。当 T&& 为模板参数时，输入左值，它会变成左值引用，而输入右值时则变为具名的右值引用。

4）编译器会将已命名的右值引用视为左值，而将未命名的右值引用视为右值。

2.1.2　右值引用优化性能，避免深拷贝

对于含有堆内存的类，我们需要提供深拷贝的拷贝构造函数，如果使用默认构造函数，

会导致堆内存的重复删除，比如下面的代码：

```
class A
{
public:
        A() :m_ptr(new int(0))
        {
        }

        ~A()
        {
                delete m_ptr;
        }

private:
        int* m_ptr;
};


// 为了避免返回值优化，此函数故意这样写
A Get(bool flag)
{
        A a;
        A b;
        if (flag)
                return a;
        else
                return b;
}

int main()
{
        A a = Get(false);                    // 运行报错
}
```

在上面的代码中，默认构造函数是浅拷贝，a 和 b 会指向同一个指针 m_ptr，在析构的时候会导致重复删除该指针。正确的做法是提供深拷贝的拷贝构造函数，比如下面的代码（关闭返回值优化的情况下）：

```
class A
{
public:
        A() :m_ptr(new int(0))
        {
                cout << "construct" << endl;
        }

        A(const A& a):m_ptr(new int(*a.m_ptr)) // 深拷贝
        {
                cout << "copy construct" << endl;
        }
```

```
        ~A()
        {
                cout << "destruct" << endl;
                delete m_ptr;
        }

private:
        int* m_ptr;
};

int main()
{
        A a = Get(false);                    //运行正确
}
```

上面的代码将输出：

```
construct
construct
copy construct
destruct
destruct
destruct
```

这样就可以保证拷贝构造时的安全性，但有时这种拷贝构造却是不必要的，比如上面代码中的拷贝构造就是不必要的。上面代码中的 Get 函数会返回临时变量，然后通过这个临时变量拷贝构造了一个新的对象 b，临时变量在拷贝构造完成之后就销毁了，如果堆内存很大，那么，这个拷贝构造的代价会很大，带来了额外的性能损耗。有没有办法避免临时对象的拷贝构造呢？答案是肯定的。看下面的代码：

```
class A
{
public:
        A() :m_ptr(new int(0))
        {
                cout << "construct" << endl;
        }

        A(const A& a):m_ptr(new int(*a.m_ptr))//深拷贝
        {
                cout << "copy construct" << endl;
        }

        A(A&& a) :m_ptr(a.m_ptr)
        {
                a.m_ptr = nullptr;
                cout << "move construct: " << endl;
        }
```

```
        ~A()
        {
                cout << "destruct" << endl;
                delete m_ptr;
        }

private:
        int* m_ptr;
};

int main()
{
        A a = Get(false); //运行正确
}
```

上面的代码将输出：

```
construct
construct
move construct
destruct
destruct
destruct
```

上面的代码中没有了拷贝构造，取而代之的是移动构造（Move Construct）。从移动构造函数的实现中可以看到，它的参数是一个右值引用类型的参数 A&&，这里没有深拷贝，只有浅拷贝，这样就避免了对临时对象的深拷贝，提高了性能。这里的 A&& 用来根据参数是左值还是右值来建立分支，如果是临时值，则会选择移动构造函数。移动构造函数只是将临时对象的资源做了浅拷贝，不需要对其进行深拷贝，从而避免了额外的拷贝，提高性能。这也就是所谓的移动语义（move 语义），右值引用的一个重要目的是用来支持移动语义的。

移动语义可以将资源（堆、系统对象等）通过浅拷贝方式从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝以及销毁，可以大幅度提高 C++ 应用程序的性能，消除临时对象的维护（创建和销毁）对性能的影响。

以代码清单 2-2 所示为示例，实现拷贝构造函数和拷贝赋值操作符。

代码清单 2-2 MyString 类的实现

```
class MyString {
private:
        char* m_data;
        size_t   m_len;
        void copy_data(constchar *s) {
                m_data = newchar[m_len+1];
                memcpy(_data, s, m_len);
                m_data[_len] = '\0';
        }
```

```
public:
          MyString(){
                    m_data = NULL;
                    m_len = 0;
          }

          MyString(const char* p) {
                    m_len = strlen (p);
                    copy_data(p);
          }

          MyString(const MyString& str) {
                    m_len = str.m_len;
                    copy_data(str.m_data);
                   std::cout <<"Copy Constructor is called! source: "<< str.m_data
                       << std::endl;
          }

          MyString&operator=(const MyString& str) {
                   if (this != &str) {
                               m_len = str.m_len;
                               copy_data(str._data);
                   }
                  std::cout <<"Copy Assignment is called! source: "<< str.m_data
                     << std::endl;
                   return *this;
          }

          virtual ~MyString() {
                    if (m_data) free(m_data);
          }
};
void test() {
          MyString a;
          a = MyString("Hello");
          std::vector<MyString> vec;
          vec.push_back(MyString("World"));
}
```

实现了调用拷贝构造函数的操作和拷贝赋值操作符的操作。MyString("Hello") 和 MyString("World") 都是临时对象，也就是右值。虽然它们是临时的，但程序仍然调用了拷贝构造和拷贝赋值函数，造成了没有意义的资源申请和释放的操作。如果能够直接使用临时对象已经申请的资源，既能节省资源，又能节省资源申请和释放的时间。这正是定义移动语义的目的。

用 C++11 的右值引用来定义这两个函数，如代码清单 2-3 所示。

代码清单 2-3　MyString 的移动构造函数和移动赋值函数

```
MyString(MyString&& str) {
          std::cout <<"Move Constructor is called! source: "<< str._data << std::endl;
          _len = str._len;
```

```
        _data = str._data;                          //避免了不必要的拷贝
        str._len = 0;
        str._data = NULL;
}

MyString&operator=(MyString&& str) {
        std::cout <<"Move Assignment is called! source: "<< str._data << std::endl;
        if (this != &str) {
                _len = str._len;
                _data = str._data;          //避免了不必要的拷贝
                str._len = 0;
                str._data = NULL;
        }
        return *this;
}
```

再看一个简单的例子，代码如下：

```
struct Element
{
        Element(){}
        //右值版本的拷贝构造函数
        Element(Element&& other) : m_children(std::move(other.m_children)){}
        Element(const Element& other) : m_children(other.m_children){}
private:
        vector<ptree> m_children;
};
```

这个 Element 类提供了一个右值版本的构造函数。这个右值版本的构造函数的一个典型应用场景如下：

```
void Test()
{
        Element t1 = Init();
        vector<Element> v;
        v.push_back(t1);
        v.push_back(std::move(t1));
}
```

先构造了一个临时对象 t1，这个对象中一个存放了很多 Element 对象，数量可能很多，如果直接将这个 t1 用 push_back 插入到 vector 中，没有右值版本的构造函数时，会引起大量的拷贝，这种拷贝会造成额外的严重的性能损耗。通过定义右值版本的构造函数以及 std::move(t1) 就可以避免这种额外的拷贝，从而大幅提高性能。

有了右值引用和移动语义，在设计和实现类时，对于需要动态申请大量资源的类，应该设计右值引用的拷贝构造函数和赋值函数，以提高应用程序的效率。需要注意的是，我们一般在提供右值引用的构造函数的同时，也会提供常量左值引用的拷贝构造函数，以保证移动不成还可以使用拷贝构造。

需要注意的一个细节是，我们提供移动构造函数的同时也会提供一个拷贝构造函数，以防止移动不成功的时候还能拷贝构造，使我们的代码更安全。

2.2　move 语义

我们知道移动语义是通过右值引用来匹配临时值的，那么，普通的左值是否也能借助移动语义来优化性能呢，那该怎么做呢？事实上 C++11 为了解决这个问题，提供了 std::move 方法来将左值转换为右值，从而方便应用移动语义。move 是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存拷贝。深拷贝和 move 的区别如图 2-1 所示。

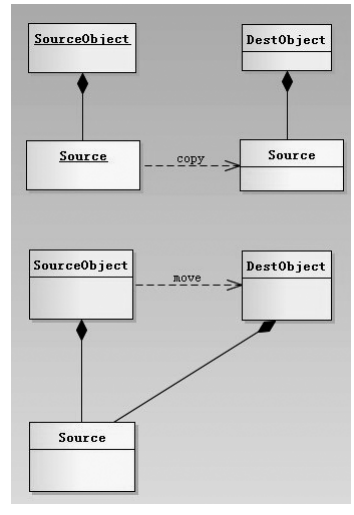


图 2-1　深拷贝和 move 的区别

在图 2-1 中，对象 SourceObject 中有一个 Source 资源对象，如果是深拷贝，要将 SourceObject 拷贝到 DestObject 对象中，需要将 Source 拷贝到 DestObject 中；如果是 move 语义，要将 SourceObject 移动到 DestObject 中，只需要将 Source 资源的控制权从 SourceObject 转移到 DestObject 中，无须拷贝。

move 实际上并不能移动任何东西，它唯一的功能是将一个左值强制转换为一个右值引用[㊀]，使我们可以通过右值引用使用该值，以用于移动语义。强制转换为右值的目的是为了方便实现移动构造。

这种 move 语义是很有用的，比如一个对象中有一些指针资源或者动态数组，在对象的赋值或者拷贝时就不需要拷贝这些资源了。在 C++11 之前拷贝构造函数和赋值函数可能要像下面这样定义。假设一个 A 对象内部有一个资源 m_ptr：

```
A& A::operator=(const A& rhs)
{
// 销毁 m_ptr 指向的资源
// 复制 rhs.m_ptr 所指的资源，并使 m_ptr 指向它
}
```

同样 A 的拷贝构造函数也是这样。假设这样来使用 A：

```
A foo(); // foo 是一个返回值为 X 的函数
A a;
a = foo();
```

最后一行将会发生如下操作：

- ❑ 销毁 a 所持有的资源。
- ❑ 复制 foo 返回的临时对象所拥有的资源。
- ❑ 销毁临时对象，释放其资源。

㊀ 《深入理解 C++11：C++ 新特性解析与应用》3.3.4 节。

上面的过程是可行的，但是更有效率的办法是直接交换 a 和临时对象中的资源指针，然后让临时对象的析构函数去销毁 a 原来拥有的资源。换句话说，当赋值操作符的右边是右值的时候，我们希望赋值操作符被定义成下面这样：

```
A& A::operator=(const A&& rhs)
{
        //转移资源的控制权，无须复制
}
```

仅仅转移资源的所有者，将资源的拥有者改为被赋值者，这就是所谓的 move 语义。再看一个例子，假设一个临时容器很大，赋值给另一个容器。

```
{
   std::list< std::string> tokens;      //省略初始化……
   std::list< std::string> t = tokens;
}
std::list< std::string> tokens;
std::list< std::string> t = std::move(tokens);
```

如果不用 std::move，拷贝的代价很大，性能较低。使用 move 几乎没有任何代价，只是转换了资源的所有权。实际上是将左值变成右值引用，然后应用 move 语义调用构造函数，就避免了拷贝，提高了程序性能。当一个对象内部有较大的堆内存或者动态数组时，很有必要写 move 语义的拷贝构造函数和赋值函数，避免无谓的深拷贝，以提高性能。事实上，C++ 中所有的容器都实现了 move 语义，方便我们实现性能优化。

这里也要注意对 move 语义的误解，move 只是转移了资源的控制权，本质上是将左值强制转换为右值引用，以用于 move 语义，避免含有资源的对象发生无谓的拷贝。move 对于拥有形如对内存、文件句柄等资源的成员的对象有效。如果是一些基本类型，比如 int 和 char[10] 数组等，如果使用 move，仍然会发生拷贝（因为没有对应的移动构造函数），所以说 move 对于含资源的对象来说更有意义。

2.3 forward 和完美转发

在 2.2 节中介绍的右值引用类型是独立于值的，一个右值引用参数作为函数的形参，在函数内部再转发该参数的时候它已经变成一个左值了，并不是它原来的类型了。比如：

```
template <typename T>
void forwardValue(T& val)
{
   processValue(val);                    //右值参数会变成左值
}
template <typename T>
void forwardValue(const T& val)
{
   processValue(val);                    //参数都变成常量左值引用了
}
```

都不能按照参数的本来的类型进行转发。

因此，我们需要一种方法能按照参数原来的类型转发到另一个函数，这种转发被称为完美转发。所谓完美转发（Perfect Forwarding），是指在函数模板中，完全依照模板的参数的类型[㊀]（即保持参数的左值、右值特征），将参数传递给函数模板中调用的另外一个函数。C++11 中提供了这样的一个函数 std::forward，它是为转发而生的，不管参数是 T&& 这种未定的引用还是明确的左值引用或者右值引用，它会按照参数本来的类型转发。看看这个例子：

代码清单 2-4　参数转发的示例

```
template<typename T>
void PrintT(T& t)
{
   cout <<"lvaue"<< endl;
}

template<typename T>
void PrintT(T && t)
{
   cout <<"rvalue"<< endl;
}

template<typename T>
void TestForward(T && v)
{
   PrintT(v);
   PrintT(std::forward<T>(v));
   PrintT(std::move(v));
}

Test()
{
   TestForward(1);
   int x = 1;
   TestForward(x);
   TestForward(std::forward<int>(x));
}
```

测试结果如图 2-2 所示。

图 2-2　std::forward 示例测试结果

我们来分析一下测试结果。

- TestForward(1)：由于 1 是右值，所以未定的引用类型 T && v 被一个右值初始化后变成了一个右值引用，但是在 TestForward 函数体内部，调用 PrintT(v) 时，v 又变成了一个左值（因为在这里它已经变成了一个具名的变量，所以它是一个左值），因此，

㊀《深入理解 C++11：C++11 新特性解析与应用》85 页。

第一个 PrintT 被调用，打印出“ lvaue”。调用 PrintT(std::forward<T>(v)) 时，由于 std::forward 会按参数原来的类型转发，因此，它还是一个右值（这里已经发生了类型推导，所以这里的 T&& 不是一个未定的引用类型（关于这点可以参考 2.1 节），会调用 void PrintT(T &&t) 函数。调用 PrintT(std::move(v)) 是将 v 变成一个右值（ v 本身也是右值），因此，它将输出 rvalue。

- TestForward(x) 未定的引用类型 T && v 被一个左值初始化后变成了一个左值引用，因此，在调用 PrintT(std::forward<T>(v)) 时它会被转发到 void PrintT(T& t)。

右值引用、完美转发再结合可变模板参数，我们可以写一个万能的函数包装器（可变模板参数将在 3.2 节中介绍，读者不妨读完下一节再回头来看这个函数），带返回值的、不带返回值的、带参数的和不带参数的函数都可以委托这个万能的函数包装器执行。下面看看这个万能的函数包装器。

```
template<class Function, class... Args>
inline auto FuncWrapper(Function && f, Args && ... args) -> decltype(f(std::for
    ward<Args>(args)...))
{
         return f(std::forward<Args>(args)...);
}
```

测试代码如下：

```
void test0()
{
         cout <<"void"<< endl;
}

int test1()
{
         return 1;
}

int test2(int x)
{
         return x;
}

string test3(string s1, string s2)
{
         return s1 + s2;
}

test()
{
   FuncWrapper(test0);                    //没有返回值，打印 1
   FuncWrapper(test1);                    //返回 1
```

```
    FuncWrapper(test2, 1);                  //返回 1
    FuncWrapper(test3, "aa", "bb");         //返回 "aabb"
}
```

2.4　emplace_back 减少内存拷贝和移动

emplace_back 能就地通过参数构造对象，不需要拷贝或者移动内存，相比 push_back 能更好地避免内存的拷贝与移动，使容器插入元素的性能得到进一步提升。在大多数情况下应该优先使用 emplace_back 来代替 push_back。所有的标准库容器（array 除外，因为它的长度不可改变，不能插入元素）都增加了类似的方法：emplace、emplace_hint、emplace_front、emplace_after 和 emplace_back，关于它们的具体用法可以参考 cppreference.com。这里仅列举典型的示例。

vector 的 emplace_back 的基本用法如下：

```
#include <vector>
#include <iostream>
using namespace std;
struct A
{
        int x;
        double y;
        A(int a, double b):x(a),y(b){}
};

int main() {
        vector<A> v;
        v.emplace_back(1, 2);
        cout<<v.size()<<endl;
        return 0;
}
```

可以看出，emplace_back 的用法比较简单，直接通过构造函数的参数就可以构造对象，因此，也要求对象有对应的构造函数，如果没有对应的构造函数，编译器会报错。如果把上面的构造函数注释掉，在 vs2013 下编译会报如下错误：

```
error C2661: “A::A”：没有重载函数接受 2 个参数
```

其他容器相应的 emplace 方法也是类似的。

相对 push_back 而言，emplace_back 更具性能优势。下面通过一个例子来看 emplace_back 和 push_back 的性能差异，如代码清单 2-5 所示。

代码清单 2-5 emplace_back 和 push_back 的比较

```
#include <vector>
#include <map>
#include <string>
#include <iostream>
using namespace std;

struct Complicated
{
        int year;
        double country;
        std::string name;

        Complicated(int a, double b, string c):year(a),country(b),name(c)
        {
                cout<<"is constucted"<<endl;
        }

        Complicated(const Complicated&other):year(other.year),county(other.
            county),name(std::move(other.name))
        {
                cout<<"is moved"<<endl;
        }
};

int main()
{
        std::map<int, Complicated> m;
        int anInt = 4;
        double aDouble = 5.0;
        std::string aString = "C++";
        cout<<"—insert--"<<endl;
        m.insert(std::make_pair(4, Complicated(anInt, aDouble, aString)));

        cout<<"—emplace--"<<endl;
        // should be easier for the optimizer
        m.emplace(4, Complicated(anInt, aDouble, aString));

        cout<<"--emplace_back--"<<endl;
        vector<Complicated> v;
        v.emplace_back(anInt, aDouble, aString);
        cout<<"--push_back--"<<endl;
        v.push_back(Complicated(anInt, aDouble, aString));
}
```

输出如下：

```
--insert--
is constucted
is moved
is moved
--emplace--
is constucted
is moved
--emplace_back--
is constucted
--push_back--
is constucted
is moved
is moved
```

代码清单 2-5 测试了 map 的 emplace 和 vector 的 emplace_back，用 map 的 insert 方法插入元素时有两次内存移动，而用 emplace 时只有一次内存移动；用 vector 的 push_back 插入元素时有两次移动内存，而用 emplace_back 时没有内存移动，是直接构造的。

可以看到，emplace/emplace_back 的性能比之前的 insert 和 push_back 的性能要提高很多，我们应该尽量用 emplace/emplace_back 来代替原来的插入元素的接口以提高性能。需要注意的是，我们还不能完全用 emplace_back 来取代 push_back 等老接口，因为在某些场景下并不能直接使用 emplace 来进行就地构造，比如，当结构体中没有提供相应的构造函数时就不能用 emplace 了，这时就只能用 push_back。

2.5　unordered container 无序容器

C++11 增加了无序容器 unordered_map/unordered_multimap 和 unordered_set/unordered_multiset，由于这些容器中的元素是不排序的，因此，比有序容器 map/multimap 和 set/multiset 效率更高。map 和 set 内部是红黑树，在插入元素时会自动排序，而无序容器内部是散列表（Hash Table），通过哈希（Hash），而不是排序来快速操作元素，使得效率更高。由于无序容器内部是散列表，因此无序容器的 key 需要提供 hash_value 函数，其他用法和 map/set 的用法是一样的。不过对于自定义的 key，需要提供 Hash 函数和比较函数。代码清单 2-6 是无序容器的基本用法。

代码清单 2-6　无序容器的基本用法

```
#include <unordered_map>
#include <vector>
#include <bitset>
#include <string>
#include <utility>
```

```
struct Key {
        std::string first;
        std::string second;
};

struct KeyHash {
        std::size_t operator()(const Key& k) const
        {
                return std::hash<std::string>()(k.first) ^
                        (std::hash<std::string>()(k.second) << 1);
        }
};

struct KeyEqual {
        bool operator()(const Key& lhs, const Key& rhs) const
        {
                return lhs.first == rhs.first && lhs.second == rhs.second;
        }
};

int main()
{
        // default constructor: empty map
        std::unordered_map<std::string, std::string> m1;

        // list constructor
        std::unordered_map<int, std::string> m2 =
        {
                {1, "foo"},
                {3, "bar"},
                {2, "baz"},
        };

        // copy constructor
        std::unordered_map<int, std::string> m3 = m2;

        // move constructor
        std::unordered_map<int, std::string> m4 = std::move(m2);

        // range constructor
        std::vector<std::pair<std::bitset<8>, int>> v = { {0x12, 1}, {0x01,-1} };
        std::unordered_map<std::bitset<8>, double> m5(v.begin(), v.end());

        // constructor for a custom type
        std::unordered_map<Key, std::string, KeyHash, KeyEqual> m6 = {
```

```
                { {"John", "Doe"}, "example"},
                { {"Mary", "Sue"}, "another"}
        };
}
```

对于基本类型来说，不需要提供 Hash 函数和比较函数，用法上和 map/set 一样，对于自定义的结构体，就稍微复杂一些，需要提供函数和比较函数。

2.6　总结

C++11 在性能上做了很大的改进，最大程度减少了内存移动和复制，通过右值引用、forward、emplace 和一些无序容器我们可以大幅度改进程序性能。

- ❑ 右值引用仅仅是通过改变资源的所有者来避免内存的拷贝，能大幅度提高性能。
- ❑ forward 能根据参数的实际类型转发给正确的函数。
- ❑ emplace 系列函数通过直接构造对象的方式避免了内存的拷贝和移动。
- ❑ 无序容器在插入元素时不排序，提高了插入效率，不过对于自定义 key 时需要提供 hash 函数和比较函数。