C++ Primer 5th 笔记

目录

开始

万事开头难，开始意味着做出改变，而接下来的任何动作都是延续之前的行为，因为人的强大适应能力而一旦适应了想要再改变就会引起大脑的强烈反应。这有利于人适应环境，也会导致人的思维固化，所以偶尔的外来刺激会导致大脑获得新的认知。这是创造的必要条件。

接下来的几个月时间，我将会把主要精力放在为 《C++ Primer 5th》 这本书的笔记上。C++ 在我多年以来断断续续学了好几遍，之前用《C++ 编程思想》也学过，但那本书过于陈旧了，用《C++ Primer 4th》也学过一段时间，由于书本过于沉溺细节，导致最终没法坚持下来。第五版我去年已经看过一遍了，对于里边的内容也有一个大概的印象，我的一个感受就是第五版的内容安排变好了，而且很多过于细节化的东西被隐藏了。这是非常好的，对于初学者来说更重要的是看到语言的全貌而不是抠细节，我们得知道语言的边界从而为要解决的问题提供良好定义的方案。

我之前得到一些启发，说学习 C 不应该一开始就将 printf 的各种参数细节，这太过于机械记忆。只要用到的时候去查阅对应的文档即可。学太多这种东西，反而妨碍了了解语言的能力和限制，以及怎样设计具有良好结构的程序。所以，在所有这些笔记中如果是过于细节化的东西，我会省略掉，当需要时阅读原文即可。我希望笔记达到的功能是提醒所有想用好 C++ 语言的程序员这门语言的核心是什么。细枝末节将会被忽略。

好，以下是 《C++ Primer 5th》的笔记正文，供各位慢慢阅读。

前言

C++ 语言经过几十年的发展，已经从当初仅仅关注机器效率渐渐开始更加关注开发效率。这很大程度是由于别的语言的冲击，Python 、Javascript 这些语言的开发效率相对来说要高不少，渐渐 C++ 也往这方面考虑。由于 C++ 即想保持机器效率，又想最大化开发效率，导致现在语言特性繁多而且还在不断增加，这是不得已的妥协。

C++11 是目前最广为接受的现代 C++ 标准，近些年还发展出了 C++14 和 C++17 标准，都是在 C++11 上进行小幅扩展的。这本书所讲的内容都是关于 C++11 的。

C++11 具有语言更加统一，标准库更易用更安全并且更高效，动手写自己的库也更加容易的特点。比如 auto 关键字的广泛使用导致程序员可以忽略类型细节，将精力放在解决问题上而不是语言细节上。而智能指针和移动容器（move-enabled containers）使得程序员可以更少关注资源管理（resource management）方面的细节，从而可以更加安全高效的写复杂的库。

本书会用 **C++11** 标记所有 C++11 添加的新特性，为的是在需要区分这些特性时更容易找到。尽管新标准增添了许多特性，C++ 语言的核心并没有改变，并构成了本书的主要内容。

为什么是这一本?

现代 C++ 语言由以下三部分组成：

1. 从 C 语言继承而来的低层（low-level）语言细节；
2. 允许定义自己的类型的 OOP 编程方式，从而构建大的程序和系统；
3. 丰富的标准库；

很多别的书一开始就将很多关于 C 语言的细节，让读者学一堆动态内存管理和指针的内容，导致读者深陷低层语言的细节从而导致在挫败中放弃。这不是正确的学习 C++ 的方式，正确的方式是一开始就利用 C++ 语言丰富的标准库，而忽略掉低层语言细节，这样写出来的程序更容易写、更容易懂、更健壮。本书将帮助读者从一开始就养成正确的习惯。本书将强调惯用的技巧，提醒读者避免 C++ 中常见的坑，本书也将解释规则后面的原因，知道了为什么是这样读者将更快掌握语言的精髓。

C++ 是一门很大的语言，包含了为各种问题定制的方案。其中一些仅适用于大的项目，而小项目用不上。所以并不是所有程序员需要了解每个特性的所有细节。每个人都需要了解语言的核心部分，而一些高级以及很偏门（special-purpose）的主题可以快速浏览，并在真正需要时才用心研究这些特性。C++ 语言中还有一些概念对于理解整个语言有至关重要的意义，这些部分值的好好理解。

本书的第一、二部分讲的是语言的基础和库的使用，这两个部分是语言的核心，知道了这些部分就可以阅读和写出很多重要的软件。绝大部分程序要都需要这两部分的所有细节。C++ 中的库虽然实现复杂，但全部都是以 C++ 语言本身写就的，在体验库的强大的同时，也了解到语言的威力。学习 C++ 更为重要一点并不在于沉溺于 C 语言的细节，而在于其强大的抽象能力。想想 C++ 的库比 C 库大多少就知道了，C++ 是一门比 C 抽象程度更高的语言。

本书的第三部分教大家怎么写自己的类，其中涉及到拷贝控制（copy control），以及其它使得自定义类像内置类型一样易用的特性。第三部分也会将关于泛型编程（generic programming）的知识。第四部分则将在大型程序中用到的技术。
最后：学习任何语言都需要多写代码，用它去解决实际的问题，才能做到有效学习。

最简单的 C++ 程序

```
int main()
{
    return 0;
}
```

所有 C++ 程序都包含一个以上的函数，其中最重要的 main 函数，操作系统通过调用 main 函数来运行程序。函数包含四个元素：返回类型、函数名字、形参列表、函数体。main 函数被指定返回 int 类型，int 类型是内置类型，也就是语言本身提供的类型。通常 main 函数返回 0 表示程序运行正常，返回非 0 值表示遇到错误。函数体是以 { 开头的语句块。函数体中的 return 语句将终止函数的执行，并返回一个值给调用者，返回的值的类型必须与函数的返回类型一致。对于返回类型为 void 的函数，`return;` 将直接将控制权返回调用者，而执行到函数末尾也将隐式的返回。
最开始学习 C++ 语言时闹出过一个笑话，当时写了一小段代码，怎么都编译不通过，当时就慌了。找了好几天都找不出问题，还认为自己根本学不好编程。后来问了素未相识的朋友，人家一下就指出是我的 main 写成了 mian 了。直到现在我还印象非常深刻，当时简直是要怀疑人生了。
C++ 中有需要地方需要用到分号，而分号也非常容易被忽略。可能仅仅因为缺少一个分号，编译器就会报出一大堆的错误。在这些细节上不能马虎，当然坑踩的多了自然就会小心。
上面所讲的函数的要素如今在任何类 C 的语言中都一样，只要了解过任何一门此类语言肯定能够知道我在将什么。

关键概念：类型

类型在任何一门语言中都具有极其重要的地位，类型定义了数据内容和作用于这些数据上的操作，没有一门语言是没有类型的。差异在于有的语言允许自定义类型，有的语言只能使用语言内置的类型，有的语言允许变量是动态类型，有的语言要求变量只能是固定的类型。我们定义的数据保存在**变量**中，而所有变量都必须要有一个类型。
书中提到在学习语言时不应该过于使用的 IDE，因为 IDE 本身需要花大量的时间来学习如何使用。如果通过命令行去执行 C++ 的编译器来编译简单的程序，就可以将注意力集中在 C++ 语言本身。C++ 源文件的后缀名最常用的有：`.cc`，`.cxx`，`.cpp`，`.cp`，`.C` 。
如何编译程序请参考 GCC Basic ，为了开启 C++11 特性需要使用 `-std=c++0x` 编译选项。

输入输出

C++ 语言本身是不包含输入输出的。C++ 跟很多别的语言一样通过提供一个 IO 库来实现输入输出。本书中使用的输入输出库时 iostream 库，iostream 库有两个基本的输入输出类型，istream 用于输入，ostream 用于输出。流（stream）是从 IO 设备读取或写入的一连串字符。术语 stream 的含义就是字符以序列（sequence）的形式被产生和消费。C++ IO 库定义了 istream 类型的标准输入对象 cin，ostream 类型的标准输出对象 cout，另外两个 ostream 对象 cerr 表示标准误，clog 产生程序执行的通用信息。一般 clog 用的比较少。标准输入/输出从命令行窗口读取和写入，是最基本的输入输出方式。
`#include` 是预编译指令，通常用来包含程序需要的头文件，通常 `#include` 放在源文件的顶部。

```
std::cout << "Enter two numbers:" << std::endl;
```

C++ 中的表达式由至少一个操作数和一个操作符组成，通常会产生一个结果。表达式和表达式可以组成新的表达式，所以说表达式的定义是递归的。`<<` 操作符有两个操作数，左边的是 ostream 类型对象，右边的是需要输出的值，这个表达式的结果是左边的值，也就是 std::cout 对象本身。由于这样的设计多个 `<<` 串行，这种操作也称之为流式操作（flow operation），操作返回操作对象本身。
`std::endl` 是一个特殊的值，称之为操纵器（manipulator），这种数据可以对流本身进行设置，会影响流的状态。写入 `endl` 将会插入一个换行符到流中，并且刷新流缓冲。刷新缓冲将保证写入的数据被真正写入到输出流中，而不是暂存在内存中。其实呢，刷新流只是将写的数据提交给操作系统，操作系统本身也有一个文件缓冲，只有当文件缓冲积累到一定程度时才会真正写入到文件中，这是对文件系统的优化，避免每次写入都对磁盘进行操作。要了解这些细节需要参考相关的书籍。

在写代码过程中我们经常会在关键点埋一些调试打印语句，这些打印语句应当经常刷新缓冲，因为一旦程序 crash 掉时，运行时几乎会立即停止，保留在缓冲中的数据和可能会缺失，这样就很可能根本找不到程序在哪里导致了异常。

C++ 有更令人头疼的地方，因为 C++ 强调了程序员的责任，并且将资源管理交给了程序员。因此很多 crash 的地方与引发错误的地方相距非常远，要发现这种 bug 非常困难。C++ 发展出了很多好的设计程序的方法来应付这种情况，比如防御性编程。虽然防御性编程对 Java 等其它高级语言来说很重要，但相对来说对于 C++ 语言更为重要。有时候你根本不知道错误出现在哪里，而重现有极其困难，此时才是最令人烦恼的时刻。

C++ 对 C 的一个重要改进就是名称空间（namespace），C 中所有的外部可见名字都在同一个名称空间中，所以 C 发展出了一个简单有效的方式来避免名称冲突，就在以 `<lib>_` 为前缀，lib 就是你开发的库名字。名称空间像作用域一样将名字限制住，不同名称空间中的相同名字不会冲突。所有的标准库的名字都定义在 std 名称空间中。要使用名称空间中的名字必须加上名称限定符（:: operator），如 `std::cout` 引用标准库中的 cout 名字。

```
int v1 = 0, v2 = 0;
```

这条语句涉及到变量的**定义**和**初始化**，C++ 中的初始化比 C 复杂不少。C++ 的一大目标就是使得自定义的类型与内置类型的操作方式统一，C++ 在这方面做了许多努力同时这些方面的确也是很复杂的。当定义一个变量时同时提供一个初始的值（这里是 0）就将变量给初始化了。

```
std::cin >> v1 >> v2;
```

>> 操作符与上面的 << 操作符很类似，这里以 istream 类型对象为左操作数，以 v1 和 v2 变量为右操作数，功能是读取用户的输入并将值保存在 v1 和 v2 变量中。这个操作符也返回左操作数，所以可以将多个 >> 串在一起组成一个单一语句。当然分开写也是可以的。

```
std::cout << "The sum of " << v1 << " and " << v2
          << " is " << v1+v2 << std::endl;
```

这个语句与输出提示的语句一样，唯一值得注意的是现在 << 处理了两种不同类型：字符串类型和 int 类型。C++ 的库函数定义很多重载操作符来应对不同类型的操作数。

1.3 注释

注释是程序员对程序语句的解释，注释会被编译器完全过滤掉，所以对程序运行本身没什么影响。注释的主要功能是提供给程序员看的，可能是别的 review code 的同事也可能就是作者自己。注释是人与人的一种沟通方式。注释通常用来解释一个复杂的算法，解释变量的含义以及解释某段晦涩代码的执行过程。但是注释不易过多，过度的注释反而妨碍阅读者的视线和思维，真正好的代码应该是自明的，从阅读代码中就可以清晰明白代码的意图，如果代码需要注释才能让人看懂通常意味着代码很可能隐藏着 bug 。而且必须保证注释的含义与代码真实的含义一致，否则就是一种误导，必然会招致阅读者的咒骂，所以在更新代码时务必更新注释。要么干脆就把代码写的足够简单，从而省略掉注释。

C++ 中有两种注释：单行（//）注释和跨行注释（/**/）。单行注释将在行末结束，其中可以包含任何字符包括 // 符号本身。跨行注释是从 C 中继承过来的，这种跨行注释内容是 /* 和对应匹配的 */ 间的内容，内容中不能包含结束符，否则就是语法错误。这种注释最大的好处是可包含换行符，因而也成为块注释符。很多跨行注释都会在行首写一个前导 * 字符来标示多行注释。开始时 C 语言只支持这种块注释，后来受到了 C++ 的影响也开始支持单行注释。这里抠一个细节，块注释可以放在任何空白符可以放的地方，这些地方不包括字符串内部、变量名内部、别的注释内部。而单行注释将会导致行的后半段全部变成注释，所以通常放在行尾或占据一整行。

上面的细节并不重要，重要的领会注释的功能以及恰当使用注释的哲学。

1.4 控制流

一般代码的执行都是顺序的，而控制流可以改变程序执行的顺序。只有极少的程序不需要控制流。控制流包括循环、分支、返回、跳出等。下面会简单介绍几个控制流语句。

```
while (condition)
    statement
```

while 循环将在给定条件为 true 时重复执行，直到条件变为 false。所谓条件（condition）是一个表达式，这个表达式会产生一个 true 或 false 的值。while 循环会先测试条件是否为真，并在条件为真的情况下执行语句块，然后再测试条件，在决定是继续执行还是结束循环。语句块由一条或多条在大括号中的

语句组成。语句块也是语句，可以放在任何语句可以存在的地方。通常，程序会在语句块中改变条件测试的变量的状态。

```
for (init-statement; condition; update-expresion)
    statement
```

for 循环将控制变量的初始化、测试和更新放在了同一个位置，方便程序员查看修改。for 循环也用来遍历容器对象，这是绝大部分应用程序都会用到的特性。因而，在实际运用中 for 循环会比 while 循环运用的更多。for 语句由控制部分和主体部分组成。其中控制部分有三部分：初始化语句、条件语句和更新表达式。初始化语句中定义的控制变量只存在于 for 循环中，出了循环这个控制变量将不会存在。初始化语句只会在开始执行 for 循环时执行一次，条件将在每次将要进入主体语句前测试，测试如果为 true 将执行语句，否则将直接退出 for 循环。每次执行完主体语句之后就执行更新表达式来更新控制变量。然后接着重复测试条件与执行主体语句，直到最后测试条件为 false 。C++ 并没有规定这三个部分需要包含什么特定语句，其实你可以放任何表达式在里边，甚至所有部分留空都可以，如果留空将执行 forever loop 。

```
while (std::cin >> value)
```

当我们将 istream 对象作为条件时，效果是测试流的状态。如果流没有遇到任何错误或者到达文件末尾（end-of-file），测试将返回 true 否则将返回 false 。以这种方式测试 istream 实质上是在 istream 类中设置到 bool 内置类型的转换函数。这个函数是隐式调用的，具体在后面章节会讲到。

编译器的工作

编译器的工作是将源文件代码重新生成为操作系统可以识别的格式，在 Linux 下是 ELF 格式。编译器不能识别程序作者的意图，但是可以发现程序的语法错误、类型匹配错误以及声明错误。

4. 语法错误：任何该有分号的地方没有分号，括号不匹配之类的错误；
5. 类型匹配错误：给 int 类型变量赋值字符串类型值之类的错误；
6. 声明错误：没有声明就使用变量或者重复声明变量；

编译很可能在程序中有错误的时候产生非常的错误信息，这些信息过多几乎可以肯定没办法定位到所有错误。正确的做法是从上到下进行修复，每次修复一个就重新编译一下，然后看看是否还有错误。这个循环叫 edit-compile-debug .

Warning: 在 C++ 中 = 号用于赋值，== 用于比较。两个操作符都可以放在条件表达式中。如果在条件语句中混淆使用了 = 和 == 我相信你会非常头疼的，而且这种 bug 还非常常见。

程序风格

程序风格是一种极具个人特色的东西，而且特别容易引起程序员之间的争执。程序员为了该用空格还是 Tab 字符来缩进都能争吵一上午。如果用空格的话还可以为用 2 个还是 4 个空格争吵。总之是其乐无穷。但是所有优秀的程序都必定具有统一而漂亮的程序风格。如果你还没有形成一种风格，那么模仿其中一种是一个很好的选择。书中给出几个建议。函数体的 { 可以单独一行。而云风的代码中返回值是单独一行的。C# 将所有 { 单独放一行。具体怎么做取决于你自己，选择一个比较通用的风格，然后一直坚持就行。也请包容其它的程序风格。

1.5 类的介绍

在 C++ 中通过定义 **类** 来定义用户自定义数据结构。类定义了一个类型以及与类型相关的操作，这是 C++ 的数据封装和抽象的核心。类是 C++ 最重要的特性之一，事实上设计 C++ 的一个主要目的就是使得类类型表现得跟内置类型一样自然，写法上一致并且行为上也一致。这个 Java 很不一样，Java 没有以表现一致为目的而是为了高效进行对象资源管理。相对来说 Java 会安全的更多。

类包括名字、结构和操作。通常结构会被放在头文件中，而操作定义在 cpp 文件中。通常头文件名字与类型一致。头文件后缀有以下一些选择：.h 、 .hpp 、 .hxx 。标准库的头文件通常没有任何后缀，这是编译器所不关注的，而 IDE 可能会关注头文件后缀。

为了使用类，不必关心内部实现细节，这是类的实现者需要关注的。用户要关注的是此类对象所能提供的操作。所有的类（class）定义了一个类型（type），并且类型名与类名一致。如本章的例子：Sales_item 类定义了 Sales_item 类型。当定义了类之后就可以像内置类一样使用。

```
Sales_item item;
```

以上语句声明了一个类型为 Sales_item 的对象 item ，在 C++ 和 C 语言倾向于认为所有的变量都是对象，包括内置类型的变量，而不仅限于自定义的类型变量。所有这些变量可以作为函数的参数，被输入输出符读写，被等号赋值，使用加号对两个对象相加，当然也可以用 += 符号进行操作。

关键概念：类定义行为

在阅读文本中的程序时需要注意的是 Sales_item 的作者定义了这个自定义类型对象的所有行为（action），Sales_item 类定了当对象定义时发生了什么，赋值时发生了什么，以及做加法、输入输出时发生了什么。总的来说，类的作者定义了自定义类型对象的所有操作，从而所有类的操作可以从类的结构看出。

```
#include <iostream>
#include "Sales_item.h"

int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    std::cout << item1+item2 << std::endl;
    return 0;
}
```

来自标准库的头文件用尖括号（< >）包围，来自程序自定义的头文件用双引号（" "）包围。值得注意的是这里对两个 Sales_item 对象进行输入输出以及做加法。加法的实际意义由类的作者对 Sales_item 对象进行定义。

```
item1.isbn() == item2.isbn();
```

这里的成员函数是 isbn，成员函数（member function）是被定义为类一部分的函数，有时也被称为方法（methods）。通常用对象去调用成员函数，**item1.isbn** 中使用点号操作符来说明“取 item1 对象中的 isbn 成员”，点号操作符（.）只能作用于类的对象。其左操作数必须是类的对象，右操作数必须是类的一个成员名，结果就是取对象的一个成员。当用点号操作符访问一个成员函数时，通常是想调用该函数。我们使用调用运算符（()）来调用函数，调用运算符是一对圆括号，里边放置实参(argument)列表（可能为空）。成员函数 isbn 并不接收参数。

因而 **item1.isbn()** 调用对象 item1 的成员函数 isbn，函数返回 item1 中保存的 ISBN 书号。

关键术语

7. buffer 缓冲： 用来存储数据的一段内存，IO 通常都会有缓冲用于输入和输出，对于应用程序来说缓冲是透明的。输出缓冲可以被刷新，从而强制写入到目的地。cin 的读入会刷新 cout 缓冲，cout 的缓冲也会在程序结束时刷新。
8. built-in type 内置类型：由语言定义的类型，比如 int 、long、double 等；
9. class 类：一种用于定义自己的数据结构及相关操作的机制。类是 C++ 中最基本的特性之一。
10. class type 类类型：类定义的类型，类名就是类型名；
11. data structure 数据结构：数据及其上所允许的操作的一种逻辑组合；
12. expression 表达式：计算的最小单元。一个表达式由一个或多个操作数以及一个或多个操作符组成。表达式通常会产生一个结果。
13. function 函数：命名的计算单元；
14. initialize 初始化：当一个对象创建时同时赋予值；
15. standrad library 标准库：每一个 C++ 编译器必须支持的类型和函数集合。
16. statement 语句：程序的一部分，指定了在当程序执行时进行什么动作。一个表达式接一个分号就是一条语句。其它类型的语句包括 if 、for 、 while 语句，所有这些语句又可以包含别的语句。
17. uninitialized variable 未初始化变量：没有给初始值的变量。当类类型没有给初始值时其初始化行为由类自己定义。函数内部的内置类型初始化在未显式初始化时其值是未定义的。尝试使用未初始化的值是错误的，并且是 bug 的常见原因。
18. variable 变量：具名对象；

C++ 最基本特性

每一个广泛使用的编程语言都提供一些共通的特性，虽然它们之间的细节有差别。理解这些特性细节是理解语言的第一步。几乎所有语言都提供如下特性：

19. 内置类型，如整数、字符等；
20. 变量；

21. 表达式和语句；
22. 控制结构，如 if 和 while 来控制动作的条件执行或循环执行；
23. 函数，用于定义可调用的计算单元；

大部分语言在这些基础特性上提供两种扩展特性：1. 使得程序员可以定义自己的类型类扩展语言；2. 提供标准库来提供有用的函数和类型而不是内建在语言中，IO 库就是一个例子；

在 C++ 中类型规定了对象可以执行的操作。一个表达式是否是合法的取决表达式中的对象的类型。一些语言提供运行时类型检查，相反，C++ 是静态语言，类型在编译时已经作出检查。因而，编译器了解每个名字的类型。

由于 C++ 允许程序员定义新的数据结构，其表达力得到了极大的提升。程序员因此可以将语言塑造成适合将要解决的问题，而不需要语言设计者提前知晓这些问题的存在。C++ 中最重要的特性就是类，类允许程序员定义自己的类型。这种类型也被称为“类类型”，从而与内置类型区分开来。一些语言只能让程序员定义类型的数据内容，而 C++ 还允许程序员定义类型的操作。C++ 的一个主要设计目标就是让程序员定义自己的类型，从而与内置类型一样容易使用。C++ 标准库就是这方面的一个典范。

内置类型

类型是语言的基础，类型告诉我们数据的含义和可执行的操作。C++ 提供类型的扩展机制。语言仅定义了若干基础类型（字符、整数、浮点数），并且提供了让我们定义自己的类型的机制。标准库使用这些机制提供了功能复杂的类，如：可变长度字符串 string 类，向量 vector 类等。

本章描述 C++ 的内置类型以及开始描述 C++ 提供的定义复杂类型的机制。

```
i = i + j;
```

因为 C++ 的设计目标是为了类类型与内置类型表现一致。所以以上语句当类型不同时其含义也不同。对于算术类型就是执行算术加法，对于 Sales_item 类型就是执行 Sales_item class 所定义的行为。

2.1 内置类型

C++ 的内置类型集合几乎与 C 完全一致，除了多了一个 bool 类型，C 从 C99 开始通过头文件引入了 bool 类型。除此之外，包括字符类型、各种有符号无符号的整数类型以及浮点数类型，以及特殊类型 void。void 主要用于指示函数不返回任何值，不接收任何参数以及作为 C 中的通用指针。

2.1.1 算术类型

算术类型包含两种形式：整数类型（包含字符和布尔类型）和浮点数类型。内置类型的大小在不同的机器上很可能是不一致的，标准只说明了编译器必须保证的最小尺寸，但是编译器可以提供更大的尺寸，不同的尺寸导致可表示的数字范围不一样。bool 类型仅用于表示真值 true 和 false，我们不关注其尺寸。char 类型保证容纳机器的基本字符集中的字符，通常是 ASCII 字符集，char 长度是一个机器字节。int 类型表示宿主机器的整型自然尺寸（the natural size），通常机器在这个大小上做算术运算是最快的。short 和 long 以及 long long 用于修饰 int 表示各种不同长度的整型。标准保证 short 的长度不长于 int 长度，int 长度不长于 long 类型，long 类型则不长于 long long 类型，现代计算机中 short 通常为 16 位，int 为 32 位，long 在 32 位机器上是 32 位，在 64 位机器上是 64 位。long long 是由新标准引入的，用的比较少。

除此之外，C++ 还支持扩展的字符类型，wchar_t 保证可以容纳机器的最大扩展字符集中的任何字符。char16_t 和 char32_t 则用于 Unicode 字符集，它们的长度如类型名中的数字所示。

浮点数类型常用的有两种：单精度 float 类型和双精度 double 类型还有扩展精度 long double 类型。标准指定了最少有效位比特，大部分编译器会提供更高的精度，但程序员不应该依赖编译器实现细节。目前所有现代计算机都遵循 IEEE 754 浮点数标准。long double 可能是 96 位或 128 位，通常用于容纳特殊目的的浮点数硬件，且精度在不同实现之间不同。

内置类型的机器级表示

任何数据在机器中都是一串 bit，每个比特装载一个 0 或 1，对于机器本身来说这些数据是没有固定的意义的，这些指代整数、那些指代浮点数、这里是数据段、那里是代码段，机器中没有定义这些。机器可操作的最小数据块是 byte （字节），但是字节不是机器最自然的处理方式，如果对汇编有所了解会知道为了处理字节，32 位 CPU 是先处理成 32 位整型，再截断成字节。因而 int 类型也被成为机器字（word），现在更为广泛使用的是 CPU 以 64 位为机器字。所谓机器字就是处理器的寄存器大小。在任何机器中每个字节都会有自己的地址，从某个地址开始的连续字节可以被解释为不同的类型，关键在于看程序格式如何

处理。如：连续的 1 个字节可以处理成 char 类型，4 个字节可以处理成整数类型或者单精度浮点数，内存内容的具体含义取决于程序赋予地址的类型。类型决定了需要使用多少比特以及怎样解释比特的含义。

有符号和无符号类型

在讲解之前先提示有符号和无符号类型混用是 bug 的常见原因。在相互的类型转换混杂算术运算容易导致超出期望的结果，所以建议不要混用这两种类型。在后面会讲解有符号和无符号类型之间的转换规则。除了 bool 和扩展字符类型（wchar_t, char16_t, char32_t）外，别的整数类型（short, int, long, long long）既可以是 signed 或者 unsigned，其默认是有符号类型，在类型前面加上 unsigned 就变成无符号类型。unsigned int 可以缩写为 unsigned 。字符类型比较特殊，分为三种明确区分的类型：char, signed char, unsigned char 。char 可能是 signed char 或 unsigned char 中的一种，具体取决于编译器实现。
目前所有现代计算机都用二进制补码来表示有符号整数类型。

决定使用何种内置类型的建议

C++ 和 C 一样将内置类型设计的尽可能贴近硬件，因而算术类型被定义的很宽泛，为的就是满足不同硬件的特性。这种定义规则遵循最小可用原则，尽可能适用于尽可能多的硬件。然后，建议是程序员在编写程序时应该通过限定具体的类型来规避这种复杂性。有以下几点建议：

24. 当知道值不可能为负数时用 unsigned 类型；
25. 在算数运算中使用 int，short 的运算速度和容量都不及 int，而 long 在 32 位机器下雨 int 大小一致。如果值超出 int 的最小保证范围，使用 long long 。
26. 不要将 char 和 bool 类型用于算术表达式，将它们用于专用的场景。因为 char 可能是有符号或者无符号的，所以真的要使用的话就明确指定 signed char 或者 unsigned char 。这里需要说明一下：标准保证了 ASCII 中的字符数值都是正数，如果只是用 char 提供值而不存储值的话，是可以直接使用 char 的。
27. 将 double 用于浮点运算， float 通常精度不够而且 double 的运算时间可能还优于 float，long double 除非在特殊场景下几乎不会使用到。

2.1.2 类型转换

相关的算术类型可以进行转换，这些转换是隐式的，意味着不需要强转（cast）就能实现。当我们需要一种类型的对象但是提供了另外一种类型的对象，这时候就会发生自动转型。转换过程将发生什么定义规则如下：

28. 当给一个非布尔值算术类型赋值一个 bool 类型对象时，false 就取值 0， true 将取值 1 。
29. 给 bool 类型对象赋值一个非布尔值时，0 将取值 false，非 0 值取值 true 。
30. 将一个浮点数赋值给整型时，小数点后的值被截断，只保留小数点前的数值；
31. 将整数赋值给一个浮点数时，小数部分为 0，如果这个整数超出了浮点数的精度范围，精度将丢失；
32. 将一个超出范围的值赋值给无符号值，结果值将取此值对无符号值的最大范围的模，如 :unsigned char 的取值范围是 0~255，如果被赋的值大于等于 256，则将对 256 取模再赋值，-1 取模为 255；
33. 当将一个超出范围的值赋值给有符号类型时，结果是未定义的；

《C 语言编程》一书中的附录 A6 给出了类型转换的详细信息，这里简单描述一下。

A6.1 类型提升

字符类型、short 类型或者枚举在表达式中使用，如果 int 可以表示原始值将转为 int 值，否则转为 unsigned int 值，这个过程称为整型提升。

A6.2 整型转换

任何一个整数转为指定的无符号类型是通过在无符号可容纳的范围中找到最小的合适值，简单来说就是对无符号值的最大值加一取模，上面已经描述过，取模的结果一定是非负数。对于二进制补码实现来说，就是在无符号类型范围更小时将原始值的左边的位截断，而在无符号类型范围更大时，如果原始值是无符号类型就对左边位进行 0 填充，如果是原始值是有符号类型就对左边位进行符号填充。
当将任何整数转为有符号类型时，如果目的类型可以表示原始值，值将不变，否则结果由编译器实现决定。

A6.3 整数和浮点数

当将浮点数转为整型时，小数点后的部分将被截断。如果结果值无法被此整型表示，结果是未定义的。特别是将负的浮点数转为无符号整型时，结果未定义。当将整数转为浮点数时，如果值在浮点数的范围内，但没法达到对应的精度时，结果要么是最接近的更大值要么是最接近的更小值。如果值超出了范围，结果是未定义的。

A6.4 浮点类型

将精度更小的浮点数转为精度更大的浮点数，值不变。将精度更大的浮点数转为精度更小的浮点数，结果遵循整数转为浮点数的规则。

A6.5 算术转换

所有算术运算符都有可能引起类型转换，结果是将操作数转为一个相同的类型，同时也是结果的类型，这种行为成为算术转换。遵循规则如下：

34. 如果任何一个操作数是 long double 时，其它的操作数转为 long double;
35. 否则，如果一个操作数是 double 时，其它的操作数转为 double;
36. 否则，如果一个操作数是 float 时，其它操作数转为 float;
37. 否则，先执行整型提升，如果一个操作数是 unsigned long int 时，其它的类型转为 unsigned long int;
38. 否则，如果一个操作数是 long int 而另一个是 unsigned int 时，结果取决于 long int 是否能够表示 unsigned int 的所有值，如果可以则 unsigned int 转为 long int，否则两者都转为 unsigned long int;
39. 否则，如果一个操作数是 long int，其它的操作数转为 long int；
40. 否则，如果一个操作数是 unsigned int，其它操作数转为 unsigned int；
41. 否则，操作数都是 int 类型；

建议：避免未定义或者实现定义（implementation-defined）的行为

未定义行为通常是编译器不需要检查或者无法检查的错误导致的。通常代码是可以编译通过的，但是程序会得到一个错误的结果。更为严重的是，未定义行为是不可预期的，也就是每次得到的结果都不一致，甚至同一个程序的不同编译版本或者同一个编译版本的不同执行时期都会不同。同样，程序应该避免实现定义行为，比如假定 int 的尺寸是固定的某个字节数，这样的程序是不可移植的，当程序移植到其它机器时通常会失败。

算术表达式中有无符号类型

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; //32
std::cout << u2 - u1 << std::endl; //4294967264
```

需要记住的是当有无符号值参与运算时，通常结果就是无符号值，此时即便在我们的直觉中值应该是负数，数值会被解释为一个很大的无符号值。

注意：不要混用有符号和无符号类型

混用有符号和无符号类型可能会出现出乎意料的结果，特别是当有符号值是负数时。需要记住的是有符号值会自动转为无符号值。如：`a = -1, b = 1` 在 a b 是有符号时的结果与无符号时的结果是不一样的。

2.1.3 字面量

字面量用来描述数字、字符和字符串的值，字面量是常量。每个字面量都有类型，字面量的形式和值决定了其类型。

整型可以写做十进制、八进制或者十六进制。以 0 开头的是八进制，是 0x 或 0X 开头的是十六进制。如：20（十进制） 024（八进制）0x14（十六进制）。通常，十进制是有符号的，而八进制和十六进制可以是无符号或者有符号的。十进制数字顺序从 int, long 或 long long 中选择最小可容纳数值的类型。八进制和十六进制则顺序从 int, unsigned, long, unsigned long, long long 或 unsigned long long 中查找适合的类型。如果数值大于最大的类型的范围则会产生错误。没有 short 类型的字面量。可以在值后加上 L 或 l 明确表示值为 long 类型，或者后缀为 U 或 u 表示无符号类型，ul 或 UL 则明确指示 unsigned long，后缀 ll 或 LL 表示 long long 类型。以上后缀适用于十进制、八进制和十六进制。如：0XFUL 是 unsigned long 类型的值 15，1234L 则是 long 类型的值 1234。

浮点数字面量既可以包含小数点（如：123.4）也可以包含指数（1e-2），指数可以用 E 或 e 指示，指数值为 -N 表示小数点左移 N 位，正数表示右移。当浮点数没有后缀时表示的 double 类型值。只有当明确加了 f 或 F 后缀时才表示 float 常量。l 或 L 后缀表示 long double 类型。

字符常量值是一个整数。字符写做单引号中的单个字符如：'x'，值是字符在机器字符集中的数字表示值。通常不会直接在代码中写明字符的数字表示值，因为可能每个字符集的数字表示不太一样。字符常量在算术运算就像别的整数一样，尽管它们最常用的场景是和别的字符进行比较。C++ 和 C 中定义了几个可以

的字符，这些字符通常是不可打印或者在字符串中有特殊含义。如：\n \r \t \\ \? \' \"。任意字符可以用八进制转义序列 \ooo 或十六进制转义序列 \xhh 表示。特殊的转义 \0 表示值为 0 的字符，也就是空字符。通常写做 \0 而不是数字 0 是为了强调其字符属性。
字符串是双引号中的 0 个或多个字符。如: “I am a string” 和 “” 。字符串字面量就是字符数组，并且编译器会在字符串的末尾隐式加上一个 \0 字符。所以字符串的真正长度比看起来多了一个字符。如： “A” 有两个字符。以上描述的转义同样适用于字符串。两个相邻的字符串（中间只有空白符）会在编译期间拼接成一个字符串，通常如果字符串太长时会这么做。
true 和 false 是 bool 类型的常量。nullptr 是指针的常量，在 C 中一般写做 NULL 宏。
字面量一定是常量，只有常量参与运算的表达式为常量表达式。常量表达式可以在编译期间就求值，并且可以在常量出现的地方使用。

枚举

还有一种常量在 《C++ primer》 这本书中介绍的很晚，就是枚举常量。枚举是一系列整数常量。常量值从 0, 1, 2, … 这样顺下去，除非明确指定每个值。如果只有部分指定了值，没有指定的就从已经指定的开始顺下去。不同枚举中的名字必须是不同的，对于值则没有这样的要求。如：

```
enum boolean { NO, YES };
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t', NEWLINE = '\n', RETURN = '\r'};
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL };
```

枚举提供将常量和名字绑定的便利机制。并且便于调试器进行打印。

2.2 变量

变量提供程序可操作的具名内存。每个 C++ 变量都有类型，类型决定了变量内存的尺寸和布局以及可保存的值的范围，类型还决定了变量可以进行的操作。在 C++ 中变量和对象是可互换的。

2.2.1 变量定义

变量定义包含类型名和其后的一个或多个变量名，变量名之间用逗号分割，并且以分号结束。定义可以为一个或多个变量提供初始值。

```
int sum = 0, value, units_sold = 0;
Sales_item item;
std::string book("0-201-78345-X");
```

上面的定义包含了内置类型和用户自定义类型，并且都有包含初始化值。对象在创建时赋予特定的值成为初始化。初始化值可以是任意复杂的表达式，只要值是对应的类型即可。当在同一条定义语句中定义两个或以上的变量时，前面的变量将马上对后面的变量可见，因而，可以用前面的变量对后面的变量进行初始化。如：

```
double price = 109.99, discount = price * 0.16;
```

在 C++ 中初始化是一个非常之复杂的话题，这本书中会一次次的提及。究其原因在于 C++ 中在定义变量时可以对用户类型进行初始化，C 语言中由于没有这种用户定义类型所以初始化直接而简单，Java 中的用户类型对象只存在于堆中，用户定义时仅仅定义了一个对真正的对象的引用，因而都没有 C++ 这么复杂。在 C++ 中许多程序员无法理解 = 号用于初始化变量，大家都倾向于认为这种形式的初始化是赋值，但是在 C++ 中初始化和赋值是完全不同的操作。其实对于 C++ 的内置类型这两种操作区别并不明显，真正区别在于用户定义类型，当初始化用户定义类型对象时调用的是构造函数，而赋值时调用的是赋值操作函数。
初始化不是赋值。初始化发生在变量创建时给定一个值，而赋值是将对象原有的值擦除并替换成新值。

列初始化

C++ 中的初始化变得更复杂的原因还在于 C++ 定义了多种初始化方式。如一下均为初始化 units_sold 的方式：

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

新标准允许初始化使用 {} 大括号的形式，这种形式的初始化在之前的版本仅允许在初始化列表时使用，此种方式成为列初始化（list initialization）。大括号形式的初始化子（initializer）现在可以用于初始化对象，

或者在一些情况下用于对对象赋予新值。用这种方式来初始化内置类型有一个重要特性，就是编译不允许丢失信息的初始化，如：

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; //不允许的
```

术语：什么是对象？

C++ 程序员倾向于在各个场景使用 object 一词。对象在最广泛的含义上指代内存中的一块包含数据并且有类型的区域。不管对象是内置类型和类类型，是具名（named）还是匿名的（unnamed），是不可改变（immutable）的还是可改变(mutable)，都称为对象。

默认初始化

当定义变量时不显式进行初始化，将执行默认初始化（default initialized），默认初始化变量将获得默认值。至于这个默认值是什么则取决于变量类型以及变量在何处定义。

没有显式初始化的内置类型，如果定义在函数外面将初始化为 0，如果定义在函数内部则值是未定义的。使用未定义的值进行任何计算或者赋值给其它变量是错误的，而且是非常常见的错误。

类类型控制本类型对象的初始化过程，不管对象定义在函数内部还是外部。如果类没有定义默认构造函数，就无法构造不显式初始化的对象。大部分类都会定义默认构造函数，类会给对象提供合适的默认值，如：string 类默认初始化为空字符串。有一些类无法提供合适的默认值，所以必须显式提供初始值。

总结来说：函数内的未初始化内置类型变量的值是未定义的，没显式初始化的类对象初始化值由类的默认构造函数控制。

注意：未初始化是一个很严重的 bug

未初始化变量具有未决议的值，使用未初始化的值是一种很难定位的错误，并且这种错误是运行时错误，是编译器不保证检查的错误，尽管大部分编译器会提醒使用了未初始化变量。使用未初始化的变量的后果是未定义的，这是 C++ 中最困难的地方，在 Java 中任何错误的用法都会引发程序抛出异常，但在 C++ 中结果是未定义的。未定义的含义是不知道会发生什么。可能会运行正常、可能会立即 crash 、有时候会错误运行但会等到很久之后才 crash 掉，此时挂掉的位置跟错误真正发生的位置相距甚远，这是真正难以定位的问题，也是 C++ 和 C 难学的重要原因之一，因为当你用错了的时候无法立即得到反馈，你必须用心智去校验是否正确，这是真正的负担。

然后想要正确初始化变量并不是一件容易的事，很多时候当定义变量时，根本不知道什么样的值是合适的，定义任何值似乎都是错误的。

《C++ primer》花了很大的篇幅来将初始化的事，而 K&R 一书中涉及的比较少，我想了一下发现，在 C 中根本无法定义新的类型，所有的初始化机制都是内置的，而且 C 中真正的类型就只有各种数字类型，它们唯一的操作就是算术运算（像数组、指针、结构、union 都在这个框内，并没有增加新的运算），而 C++ 因为可以自定义类型，而且目标是尽可能贴合内置类型，所以 C++ 说类型包含了数据和操作，说初始化是一件复杂的事，因为它在任何时候考虑的范围都包含了类类型。而在 C 中只能定义抽象数据类型（ADT），是一种头脑中的类型，而非真正表现在代码中类型。

2.2.2 变量声明和定义

C++ 沿用了 C 的分离编译方式，分离编译允许我们将程序拆分成多个源文件，而且可以单独编译，最后再将所有的编译出来的 .o 文件链接在一起。当将程序拆分成多个文件后，需要一种能够在多个文件中共享变量的机制。定义在任何函数外部的函数、类、变量对于整个程序来说都是可见的。

C++ 在很多方面类似于 C ，但更为复杂，而且不少不兼容的地方。C++ 文件组织方式和实体（这里函数、类、变量统称为程序中实体）可见性规则与 C 是一样的。C++ 也支持声明和定义的分离。声明的意思是让实体名字对程序来说是可见的，一个文件想要使用另一个文件中名字需要包含一个对这个名字的声明。C++ 为了将所有声明集中到一个地方提供了 .h 头文件以及 #include 给各个源文件包含，被包含的头文件其内容将会被预处理器复制到对应的源文件中去，这就造成了声明天然的有很多份拷贝。

而定义则除了使得名字可见之外，还创建名字，创建的含义对应于变量则是分配内存以及进行可行的初始化，对应于函数就是定义函数体本身。声明和定义一样都必须指定类型名称和变量名。

为了使用别地方定义的名字，必须得使用 extern 关键字，并且不能提供初始值。如果提供了初始值就变成了定义。同一个变量只能定义一次，但可以声明多次，对于函数是一样的。为了多个文件中使用同一的变量，必须只在一个文件中定义它，使用此变量的文件必须声明但是一定不能重复定义它。编译器会帮我们做这方面的检查，所以不用怎么担心。

概念：静态类型

语言分为静态类型和动态类型语言。静态类型的意思是在变量声明时得给出类型，并且在后面的使用中遵循此类型的行为，如果违反了约定的话编译器会帮助我们找到错误并且拒绝生成可执行文件。这个过程叫做类型检查。动态类型则不在编译期间检查类型，而将类型检查放到运行时检查，使用动态类型的语言大多会实现为“鸭子类型”，意思是只要行为像鸭子，我们就可以安全的认为它是鸭子。具体到程序中则是如果对象具有这个成员函数或者成员字段，则可以认为它是符合要求的。那么如果多个类类型具有这些函数和字段，它们就可以合法的传入到对应的位置而不会导致错误，Python 和 Javascript 是这方面的典范。动态语言中的类型是动词类型，而静态语言中的类型是名词类型，也就是说类型只跟类型的名字挂钩，只要名字不一样即便行为一样也被认为是不同的类型。对于小型程序来说动态类型提供了易修改的特性，因而适合快速原型开发或者面对需求易变的场景，而对于大型程序来说静态类型提供了易定位问题的特性，这是一种取舍和权衡，并无哪一种更优。

静态语言强制任何类型在使用之前必须可见，因而强制程序员在使用一个类型时，必须先声明它。

2.2.3 标识符

程序中的名字都是标识符，遵循标识符的规则。大部分的标识符都继承自 C 语言，具体说就是标识符（identifier）必须由字母、数字或下划线组成，其它字符都是非法的。早期的语言实现对于标识符长度有一个限定，超出的部分将会被编译器忽略，而标准规定了这个限定的长度至少得是多少。C++ 对于标识符的长度没有限定。标识符必须以字母或者下划线开头，而且是大小写敏感的。语言中包含了关键字集合，这些关键字是程序结构的骨架，不允许用户定义标识符跟关键字一样。C++ 有许多关键字，查看 C++ 标准检查整个列表。

C++ 还定义了一个操作符名字集合，如：and，bitand，compl，not_eq，and_eq，not，or，xor，这种关键字很少使用。C++ 中还规定了一些名字规则是专用于库的，因而在应用程序中尽量不要使用它们，如：以两个连续下划线开头的名字，下划线紧接着一个大写字母开头的名字。C++ 建议在函数外定义的名字最好不要以下划线开头。

标识符有一些通用的命名规范：

42. 标识符应该反映其含义；
43. 变量名通常以小写字母打头；
44. 类名以大写字母打头；
45. 包含多个单词的标识符需要将每个单词区分开，C 语言更常用下划线，C++ 还会使用驼峰法；

不论以何种方式命名最好就是遵循一致的规范。

2.2.4 名字的域

作用域（scope）是程序的一部分，名字在其中拥有特定的含义。C++ 中的作用域用大括号 {} 来分割各个作用域。相同的名字可以在不同的作用域中指代不同的实体（函数、类、变量）。名字从定义的位置直到声明它的作用域结束的位置都是可见的。C++ 中有多种级别的作用域：在任何其它作用域之外定义的名字具有 global 作用域，定义在类中的名字具有 class 作用域，定义在名称空间中的名字具有 namespace 作用域，定义在块中的名字具有 block 作用域。其中全局作用域中的名字可以在任何地方访问。

最好的方式是将变量尽量定义在靠近第一次使用的位置，这样可以变量容易找到，从而提高程序的可读性。更为重要的是在靠近使用的地方定义将更容易将变量初始化为有用的值。

作用域是可以嵌套的。一旦名字被定义在外部作用域中，就可以被接下来的内部作用域访问。同时内部作用域中定义的相同名字会遮蔽外部作用域的名字，这样做容易导致 bug 不推荐这样做。全局作用域是没有名字的，`::var` 这个作用域操作符就是访问全局作用域中的名字。

2.3 复合类型

复合类型指的是用别的类型定义的类型。C++ 中有好几种复合类型，这里重点描述引用（reference）和指针（pointer）。这里给出一个更通用的声明的含义，声明指的是基础类型（base type）加上一系列声明符（declarator）。基础类型是普通类型，是自己可以描述自己的类型，如：内置类型和类类型。声明符是变量名字以及可选的类型描述符，如：指针描述符 *，引用描述符 &，数组描述符 []。

2.3.1 引用

C++ 中的引用（reference）与 Java 中的引用虽然都叫引用，但是有很大的区别，Java 中的引用更像 C++ 的指针。区别在于 Java 中所有对象都是用引用来指代，所以对象之间的赋值其实就是单纯的引用交

换。而 C++ 中的引用赋值是真的改变了被引用的对象的值。C++ 的引用只有在作为参数传递给函数时才与 Java 一致，此时函数内的参数就是传递过来的实参的别名。
因而可以总结为，C++ 中的引用仅仅在初始化时对对象进行引用，而其后的任何操作都是操作其引用的对象。Java 中的引用更像是 C++ 中的指针，Java 中的引用可以改变从而引用别的对象，C++ 中的引用只要初始化后就只能指向一个对象。
当前讨论的 C++ 引用指的是左值引用（lvalue reference），左值是表示可改变的值，具有明确的内存地址，右值通常没有地址。新标准中定义了右值引用（rvalue reference），右值引用主要用于类的内部，这将在十三章描述。这里描述的都是左值引用。
引用的声明符是 **&d** 形式的，其中 d 是变量名。通常初始化变量时是将初始值复制到变量内存中，而初始化引用时是将引用绑定（bind）到初始值上。所谓绑定的含义是将名字与给定实体关联，这样使用这个名字就相当于使用此实体。一旦引用绑定到初始对象上就不能再令其绑定到另外一个对象上，因而，任何一个引用都必须初始化。引用不是对象，相反，引用是已存在的对象的另外一个名字。当引用被定义后，任何对引用的操作都最终操作在引用绑定的对象上。当我们给引用赋值时，其实是给引用绑定的对象赋值，这与 Java 是决然不同的。当读取一个引用时，其实是从那个对象读取值，当引用用于初始化时也是使用绑定的对象。如以下代码：

```
int &refVal3 = refVal; //refVal3 绑定到 refVal 绑定的对象 ival 上，而不是绑定到 refVal 本身。
int i = refVal; //将 refVal 绑定对象 ival 上读取值并用于初始化 i 变量。
```

因为引用不是对象，程序是不允许定义引用的引用的（reference to reference）。
如前所述，声明是基础类型加一列声明符。因而当在同一个定义语句中定义多个引用时，需要在每个引用前面加上 & 符号。这与指针（pointer）的定义方式是一样的。

```
int &r3 = i3, &r4 = i2;
```

除了引用可以定义为指向 const 修饰的类型以及父类引用指向子类外，所有的引用的类型都必须与绑定的对象类型完全一致（match exactly），即便是可以类型转换或整型提升也是不可以的，原因是任何转型都会使得引用绑定的对象与想要绑定的对象一样。同时，引用必须是对象，而不能是字面量或者返回右值的表达式，原因在于只有对象有地址。

2.3.2 指针

指针（pointer）是 C 和 C++ 中特有的东西，类似于 Java 语言中的引用，除了表示形式不一样之外。指针跟引用一样都是用来间接访问别的对象的，不同于引用的是指针是对象，会分配内存和地址给指针。指针可以被赋值以及读取，一个指针在其生命周期中可以指向多个不同的对象。与引用不同的是，指针在定义时不是必须初始化的，如果定义为自动变量而没有初始化则指针的内容是任意值。
C++ 的观点是指针通常难以理解，调试与指针相关的错误通常是非常难的，所以 C++ 创造了引用。
指针的声明符是 ***d**，d 是指针的名字，星号（*）必须在每个指针声明处重复。如：

```
int *ip1, *ip2;
```

指针保存另外一个对象的地址，通过取地址操作符（address-of operator &）可以得到一个对象的地址。由于引用不是对象，它们没有地址，因而不能定义引用的指针。除了可以定义 const 修饰的类型指针，以及定义父类指针指向子类外，指针类型必须与被指向的对象类型完全一致。原因在于指针的类型用于推断被指向的对象的类型，以及确定其能够提供的行为，如果错误使用了类型，几乎可以肯定操作会失败。

指针的值

指针有三种有效值，其中只有指向对象的指针是可以解引用的（dereference）。指针的另外两种特殊值：空指针和指向对象的下一个地址（just immediately past the end of an object）是不可解引用的。空指针很好理解，在 C 中已经非常常见了。而指向对象的下一个地址是 C++ 的惯用法，通常用来作为哨兵（sentinel），作为 for 循环的末尾检查点，当讲到 C++ 迭代器时会详述这方面的内容。除此之外的任何值都是无效的，读取无效指针的值或者访问无效指针指向的对象是错误的，这跟使用未初始化的变量一样，是未定义的，而且这种错误编译器是不保证检查的。因此，程序员必须时刻谨记指针是否有效。由于指针的两个特殊值并不指向任何对象，因而访问它们指向的对象是一种未定义（undefined）的行为。
C++ 使用解引用操作符（dereference operator *）来访问指向的对象，解引用将产生指针指向的对象。对此结果进行赋值就是对指向的对象进行赋值，对结果进行的任何操作都是对指向的对象进行操作。**只能对指向对象的指针进行解引用。**

& 和 * 在不同的语境中具有不同的语义。在声明中分别作为引用和指针的声明，而在表达式中则作为取地址和解引用操作。C++ 往往乐于复用这些符号到不同的地方，因为它们的含义毫无关联，所以忽视它们的相同外形而将其作为完全不同的符号是值得做的。

空指针（null pointer）不指向任何对象，程序应该在使用指针之前校验其是否为空指针。有三种做法来获取空指针，如：

```
int *p1 = nullptr;
int *p2 = 0;
int *p3 = NULL; //需要包含 <cstdlib> 头文件
```

在新标准中引入了 nullptr 常量初始化空指针，nullptr 是特殊类型的指针字面量，它可以转为任何其它的指针类型。旧的程序与 C 程序往往使用预处理器变量 NULL，其值在 cstdlib 中被定义为 0 。现代 C++ 程序应该使用 nullptr 而不是 NULL 的原因在于，NULL 由处理器控制，在被编译处理之前已经被全部替换为 0 ，因而编译器将无法获取 NULL 的符号，使用 nullptr 可以弥补这个缺点，nullptr 是可以被编译检查的具有类型的值。同时 NULL 不是 std 名称空间的一部分，因而不需要使用 std:: 来引用。

不能使用 int 类型的变量来初始化指针，即便其值恰好为 0 也是不允许的。

建议：初始化所有指针

未初始化的指针是运行时错误的常见来源，使用任何未初始化的指针都是未定义的。使用未初始化的指针几乎总是导致运行时崩溃，然而，调试程序的崩溃是非常难的。在大部分编译器下，当使用未初始化指针时，指针所占据的内存空间中任何值都会被当做地址。使用这样的一个未初始化的指针相当于访问一个不存在的地址上的不存在的对象。并且还没有一种有效的方法可以区分有效地址和无效地址。所以，初始化指针和初始化任何变量一样重要，甚至更重要。尽可能在定义了对象后再定义指向它的指针，或者将指针初始化为 nullptr 或 0，这样程序可以知道指针没有指向一个有效的对象。

赋值和指针

指针和引用都能间接访问其它对象，但是它们之间有重要的区别。最重要的是引用不是对象。一旦定义了引用，没有任何办法使得引用指向另外一个对象，引用只能指向其初始化时绑定的对象。指针则不一样，指针可以被赋予新的值，使得指针指向一个不同的对象。跟踪一个赋值是改变了指针还是改变了指针指向的对象是困难的，重要的是记住赋值总是改变左操作数。

```
pi = &ival; // pi 被改变为指向 ival
*pi = 0; // ival 的值被改变，而 pi 的值不变；
```

其它的指针操作

有效的指针值可以用于条件判断，空指针被判断为 false，任何其它值被判断为 true 。给定两个相同类型的有效指针，如果它们指向同一个地址，那么它们就被认为是相同的，== 将返回 true。两个同时为空指针的指针也被认为是相同的。由于条件判断或者比较操作用到了指针的值，因而，无效指针将导致以上行为未定义。关于指针的加减法和与数组的关系将在后面介绍。

void* 指针

void* 指针是一种特殊类型的指针，这种指针可以包含任何对象的地址。就像任何别的指针一样，void* 指针值是地址，但是我们不知道具体指向的对象是什么类型。void* 指针只能做几种操作 :与其它指针比较、传递给函数或者从函数中返回，赋值给其它的 void* 指针。void* 指针不能解引用，因而也不能对操作其指向的对象，因为我们不知道对象的类型，也不知道对象拥有的操作。要对指向的对象进行操作，必须将指针的类型进行强转，这将在后面提到。

void* 指针的作用在于引用一块内存，而不是使用指针来访问指向的对象。

2.3.3 理解复合类型声明

当在一个声明语句中声明多个变量时，每个声明符都说明它对应的变量与基础类型的关系，这与别的声明符是独立的，因而，一个声明语句中可以定义多个不同类型的变量。如：

```
int i = 1024, *p = &i, &r = i; //i 是 int 型，p 是指针，r 是引用
```

很多程序员对于基础类型与声明符中的类型修饰的组合很疑惑，通常很容易误解认为类型修饰（type modifier）（* 和 &） 是运用于整个变量定义语句的，这与我们有时将空白符放在了类型修饰符和名字之间有关。如：int* p; 容易让人认为 int* 将成为定义语句中所有变量的类型。然而真实的情况是 int 才是基础类型，* 是用来修饰 p 的，对于同一条语句中的别的名字没有作用。如：int* p1, p2; 其中 p1 是指针，p2 是 int 类型。在本书中倾向于将类型修饰符和名字紧靠在一起。如： int *p1, *p2;

C++ 对于指针的层级没有做限定，使用 ** 表示指针的指针，*** 表示指针的指针的指针。如：

```
int ival = 1024;
int *pi = &ival;
int **ppi = &pi;
```

指针的引用

引用不是对象，因此，不能定义引用的指针。而，指针是对象，因而可以定义指针的引用。如：

```
int i = 42;
int *p;
int *&r = p;
r = &i; // &i 赋值给 r，使得 p 指向 i
*r = 0; // 解引用 r 返回 i 变量，从而将 0 赋值给 i 变量
```

要理解 *&r，需要将声明符从右往左读，最靠近名字的修饰符是真正的类型。因而，r 是引用，下一个修饰符* 表示 r 引用的是一个指针，合起来就是 r is a reference to a pointer to an int，英文的表示更符合顺序，中文则刚好反过来。

2.4 const 限定符

使用 const 可以定义值不可变的变量，这种值是常量，这样程序就不至于不小心而改变了本不该改变的值。通常，常量用于缓冲大小、最大限度等对于程序来说有特殊含义的值，主要是为了取代 C 程序中的 #define 定义的常量。由于不能改变常量的值，常量必须初始化，初始化与变量初始化一样，特别是可以将常量初始化为运行时值，而 #define 只能让你定义字面量的常量。C 中只有几个基本内置类型所以对于常量的使用比较简单，而 C++ 由于引入了类的概念，而且类对象也可以定义为 const ，那么 const 类对象只能支持类的 const 操作，简单来说就是不改变对象的操作。

对于内置类型来说只要不涉及到赋值就不会改变其值，所以常量也能进行所有这些操作。这里非常值得一提的是当使用对象去初始化另外一个对象时，不管是常量还是非常量都是可以的。初始化别的对象并不会改变当前对象。复制对象并不会改变原始对象，一旦复制完成，新的对象就不能够再访问原始对象了。

```
int i = 42;
const int ci = i;
int j = ci;
```

默认情况下，const 变量不是全局变量

默认情况下 const 对象就像是 static 变量一样是存在于文件作用域中的。如：

```
const int bufSize = 512;
```

编译器会将这种编译时常量在每个使用的地方替换为对应的值，为了这种做编译器必须在单独编译文件时看到 const 对象的初始值，那么就必须将每个文件中都定义此对象。而为了避免重复定义的问题，const 变量默认是非全局的。因而，当我们在不同文件中定义相同名字的 const 对象时，与 static 变量，我们定义的是不同的变量。这个规则对于将 const 对象初始化为非常量值一样有效。

而有时，我们会希望多个文件访问同一个 const 变量，我们必须在定义和声明的地方都加上 extern 关键字。如：

```
//file_1.cpp 定义并初始化一个可以被其它文件访问的 const 对象
extern const int bufSize = fcn();
//file_1.h 当其它文件包含时将会声明此 const 对象
extern const int bufSize;
```

2.4.1 const 的引用

我们可以让引用绑定一个 const 类型对象，这样的引用不能改变绑定对象的值。const 对象只能被 const 引用绑定。如：

```
const int ci = 1024;
const int &r1 = ci;
r1 = 42; //错误！ const 引用不能用于改变常量的值
int &r2 = ci; // 非 const 不能绑定 const 对象
```

const 引用只是对绑定 const 对象的引用的缩写，并不存在引用自己是 const 的引用。两点原因：一、引用不是对象，const 只能修饰对象；二、引用一经初始化就不能在绑定到别的对象，所以严格说所有的

引用自己都是 const 的。引用的 const 属性决定的是是否可以通过引用改变其绑定的对象，而与引用本身无关系。
const 引用可以绑定到任何可以转换到引用类型的表达式上，包括非 const 对象、字面量和通用表达式。如：

```
int i = 42;
const int &r1 = i;  //非 const 对象
const int &r2 = 42; //字面量
const int &r3 = r1 * 2; //通用表达式
```

非 const 引用必须与绑定的对象类型严格匹配，而 const 是允许转换的。所以 int 类型的 const 引用可以绑定 double 类型的值。如：

```
double dval = 3.14;
const int &ri = dval;
```

原因在于 ri 不是真正绑定到 dval 对象上，而是绑定到一个编译器生成的临时对象上。所谓临时对象就是编译器在需要一个内存块来存储表达式求值时所创建的对象。dval 并没有直接绑定到 ri 引用上，而是转换并存储到临时对象上。如果允许将非 const 引用绑定到需要转换的对象上，那么对引用的操作将改变临时对象而不是真正的对象，这肯定不是我们想要的结果。所以，C++ 不允许这种操作。
const 引用可以绑定到非 const 对象上。绑定到 const 对象上的引用只是限制了不能通过引用来改变对象值，而没有限制底层的对象本身是否是 const 的。底层对象可以是非 const ，完全可以通过直接访问和别的引用来改变它的值。如：

```
int i = 42;
int &r1 = i;
const int &r2 = i; //const 引用绑定到非 const 对象上
r1 = 0; //通过非 const 引用可改变对象值
r2 = 0; //错误!! const 引用不能改变值
```

2.4.2 指针和 const

指针可以定义为指向 const 对象（point to const）的指针，指向 const 对象的指针不能用于改变指向对象的值。const 对象的地址只能保存在指向 const 对象的指针中。而非 const 对象的地址既可以保存在指向非 const 对象指针中，也可以保存在指向 const 对象指针中。跟引用一样，指向 const 对象的指针与对象本身是否是 const 没有关系，定义指向 const 的指针只是说不能通过指针改变对象的值，如果对象是可变的，那么可以通过别的方式改变对象值。
指向 const 的指针只是“认为”它们指向的对象是 const 的。

const 指针

指针跟引用的区别在于指针是对象，所以可以定义 const 指针（const pointer），所谓 const 指针就是指针本身不可变。const 指针跟别的 const 对象一样必须初始化，并且初始化只有其值不能改变。定义 const 指针得在 * 后面放置 const 限定符，在 * 前或者基础类型前放置 const 都是定义指向 const 对象的指针。如：

```
int errNumb = 0;
int *const curErr = &errNumb; //指向 int 对象的 const 指针
const double pi = 3.14159;
const double *const pip1 = &pi; //指向 const 对象的 const 指针
double const *const pip2 = &pi; //与上面含义完全一致的指针
```

要理解这种复杂定义的指针需要按英语方式从右往左读，pip is a const pointer to an object of type const double 。指针自身的 const 与是否可以使用指针改变底层对象无关，是否可以改变底层对象取决于是否指向 const 对象，比如可以用 curErr 改变 errNumb 的值，虽然 curErr 是 const 指针。

2.4.3 顶层 const（Top-Level const）

指针可以分开独立讨论指针本身是否为 const 的和指针指向的对象是否为 const 。我们称指针本身是 const 为顶层 const（top-level const），称指针指向一个 const 对象为底层 const（low-level const）。顶层 const 说明对象本身是 const 的，顶层 const 可以出现在任何对象类型。底层 const 只能出现在复合类型的基础类型中，指针声明可以同时包含顶层 const 和底层 const 。const 引用总是底层 const 。顶层 const 和 底层 const 区别在于拷贝一个对象时，其顶层 const 会被忽略。如：

```
int i = 0;
const int ci = 42;
const int *p2 = &ci;
const int *const p3 = p2;
i = ci; //ci 的顶层 const 被忽略
p2 = p3; //p3 的顶层 const 被忽略，但是底层 const 必须匹配
int *p = p3; //错误!! 原因是底层 const 必须匹配
p2 = &i; //可将 int* 转为 const int*
int &r = ci; //错误!! 引用中的 const 总是底层 const，不能忽略
const int &r2 = i; //可将 const int& 绑定到 int 类型
```

复制对象不会改变被复制的对象，所以对象本身的 const 即顶层 const 可以被忽略。然而底层 const 是不能被忽略的，两个对象间必须有相同的底层 const 限定符。或者将非 const 转为 const，但不能做相反的转换。

2.4.4 constexpr 和常量表达式

常量表达式（constant expression）是值在编译期求值，并且值不可变的表达式。字面是常量表达式，由常量表达式初始化的 const 对象也是常量表达式。非 const 对象或者不是由常量表达式初始化的 const 对象都不是常量表达式。

constexpr 变量

在大系统中往往程序员自己去判断常量表达式是很困难的，而在某些情况下又必须使用常量表达式，通常定义和使用的地方是分离的。在 C++11 中可以通过在定义变量时加上 constexpr 来检查此 const 变量是由常量表达式初始化的。如：

```
constexpr int mf = 20;
constexpr int limit = mf + 1;
constexpr int sz = size(); //仅当 size 函数是常量函数（constexpr function）时才合法
```

在第 6 章中将会描述如何将函数定义为 constexpr 函数，这样的函数必须足够简单使得可以在编译期间求值。常量函数可以用于初始化 constexpr 变量。建议将所有的常量表达式初始化的变量都定义为 constexpr ，强制要求编译器进行检查。

字面类型（Literal Types）

因为常量表达式必须在编译期进行求值，所以 constexpr 变量的类型必须符合这一限制，这些类型必须足够简单使得可以在编译期进行求值，所以称为字面类型。这些类型包括算术类型、引用和指针，但是不包括常见的类类型。第 7 章将介绍字面类（Literal Classes）和 19 章介绍的枚举也是字面类型。

对于引用和指针定义为 constexpr 是有限制的，指针可以初始化为 nullptr 和 0 字面量，两者也可以初始化为绑定固定地址的对象。在函数内部定义的变量不是固定地址对象，而定义在函数外的或者定义为 static 的变量是固定地址变量，这种变量从程序一启动就存在，并且持续到程序结束。只有这种变量才能用于初始化 constexpr 引用和指针。即便是定义在函数内部的 const 或 constexpr 变量地址也不是固定的。

当将 constexpr 运用于指针时，需要特别注意 constexpr 是运用于指针本身而不是指针所指对象。constexpr 隐含的意思是顶层 const。如以下声明是完全不同的：

```
const int *p = nullptr;  //p 指向 const 对象
constexpr int *q = nullptr; //q 本身是 const 的
```

constexpr 指针可以指向 const 对象。如：

```
constexpr int i = 42; //必须定义在函数外面，i 是 const int 类型
constexpr const int *p = &i; // p 是 const int *const 类型，指向固定地址对象 i
```

2.5 类型处理

本节将当类型变得复杂而难以理解时的解决方案：类型别名（Type Aliases）、auto、decltype 关键字。C++ 和 C 的关键字都很容易变得复杂而难以理解，关键就在于使用了很多符号（* & () []）进行嵌套组合，导致要真正理解类型的含义需要费一番功夫，而且过长的类型名很容易拼写错误。过于沉溺于类型是什么，而不是将精力集中在需要解决的问题上是一种舍近求远的措施。动态语言用编译器类型检查交换省略程序员拼写类型名的工作，各有取舍。近年来静态类型语言的发展就是用语法糖减轻程序员拼写类型的工作，Java，C++，Scala 都在这方面做出了努力。

2.5.1 类型别名

类型别名分为两种方式：旧式 C 的 **typedef** 和新式 C++ 的 using 。类型别名就是给长的易错的难以理解的类型名字定义一个短的名字。使用这个短名字跟使用原来的长名字效果是一样的。短名字的好处在于简化类型定义、易于使用并且强调了类型的作用。

```
typedef double wages;
typedef wages base, *p;
```

以上将 wages 定义为 double 的同义词，将 base 定义为 wages 同义词，进而是 double 的同义词。p 则是类型 double* 。**typedef** 是基础类型的一部分，有 typedef 的声明是在定义类型别名而不是变量，与定义变量一样，声明符中的 * 不会对所有名字起作用。

这里的例子其实很差，并不能反映真实项目代码中的用法，也是我在写笔记时觉得很恶心的地方，相比于 C 语言编程那本书来说给出的示例代码太过于弱智，没有什么实际的用处。为了讲语法而给出一大堆如此不合常理的例子，并且没有一个是完整的例子，也就是说讲到这里还不能让人写出一个完整的小程序，解决一些实际的问题。原因可能是语言过于庞大，但事实是 C++ 编程思想一书很早就开始给出很多可以实际跑起来的例子，那本书相对来说写的更好，除了它没有跟上标准之外。C++ primer 会将一些 C 语言编程不讲的内容进行细化，如：基础类型（base type）和声明符（declarator）的概念，有好有坏，好处是懂的更深入，坏处是记忆量变大了。

不得不说的是 C++ 确实是一门很大的语言，从书的厚度就可以看出，C 是一本不到三百页的小册子，而 C++ 是一本上千页的砖头。

C++11 定义了新的定义别名方式，语法：**using name = type** 将定义 name 为 type 的别名。使用类型别名跟类型名的效果是一样的，因而类型别名可以出现任何类型名出现的地方。

定义指针的类型别名时需要注意，如果用 const 修饰类型别名将导致指针本身是 const 的，而不是指针所指向对象是 const。如：

```
typedef char *pstring;
const pstring cstr = 0; //指向 char 类型的 const 指针
const char *astr = 0; //指向 const char 类型的指针
```

这里 pstring 是指向 char 类型的指针，const 修饰的是 pstring，因而，**const pstring** 是指向 char 的 const 指针。这是反直觉的。其实，纵观整个 C++ 就这一个特例，记住就行。

2.5.2 auto 类型限定符

C++ 中可以使用 auto 语法糖让编译器去推断变量的类型，使用 auto 的变量必须具有初始值。如：

```
auto item = val1 + val2;
```

以上如果 val1 和 val2 是 Sales_item 的话，那么 item 是 Sales_item。如果是 double 的话，item 就是 double 类型。auto 可以定义多个变量，声明中的所有初始值必须拥有一致的类型，特别是当涉及到 auto 推断的引用、指针、const 混杂在一起时会变得尤其复杂。如：

```
auto i = 0, *p = &i; // i 是 int 类型，p 是指向 int 的指针
auto sz = 0, pi = 3.14; //错误!!! sz 和 pi 的类型不一致
```

编译器推断的 auto 的类型并不总是与初始值完全匹配的。编译器会将 auto 的类型调整到与通常的初始化规则一致，原因在于没有顶层 const 和引用类型是更为通用的初始化。使用引用作为初始值，auto 推断的是引用绑定的对象类型。auto 会忽略顶层 const ，但是底层 const 会保留。

```
int i = 0, &r = i;
auto a = r; // a 是一个 int
const int ci = i, &cr = ci;
auto b = ci; // int
auto c = cr; // int
auto d = &i; // int*
auto e = &ci; // const int*
```

如果需要顶层 const，需要显式写出。如：

```
const auto f = ci; //const int
```

还可以用 auto 声明引用，如：

```
auto &g = ci; //const int&
auto &h = 42; //错误!! 不能将普通引用绑定到字面量上
const auto &j = 42; //const int&
```

当声明一个 auto 推断类型的引用时，初始值中的 const 不会被忽略。

```
auto k = ci, &l = i;  //k 是 int, l 是 int&
auto &m = ci, *p = &ci; //m 是 const int&, p 是 const int*
auto &n = i, *p2 = &ci; //错误!!! n 是 int, p2 是 const int*
```

2.5.3 decltype 类型说明符

C++11 中引入了 decltype 类型说明，其作用在于由编译器从表达式中推断类型，编译器将对表达式进行分析得出结果的类型但不会真正求值。如：

```
decltype(f()) sum = x; //sum 的类型是 f() 的返回值类型，但不会真的调用 f()
```

当将 decltype 运用于变量时，将会保留变量的顶层 const 和引用类型。只有在 decltype 表达式中引用不被当做其绑定的对象的别名。

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; //const int
decltype(cj) y = x; //const int&
decltype(cj) z; //错误!!! 引用必须初始化
```

当将 decltype 运用于表达式时，如果表达式返回的是左值则 decltype 返回的类型是引用类型，解引用操作符就是特别典型的例子。如：

```
int i = 42, *p = &i, &r = i;
decltype(r+0) b; //int
decltype(*p) c; //错误!!! c 是 int&
```

为了得到变量的引用类型有一种简单的方式就是 `decltype((variable))` ，在变量名外加上括号就成为一个返回变量的表达式，并且求值结果是左值。因而，decltype 返回的是引用。`decltype(variable)` 仅当变量本身是引用时才会返回引用类型。

```
decltype((i)) d; //错误!!! int& 类型必须得初始化
decltype(i) e; // int
```

2.6 自定义数据结构

数据结构（data structure）是一种聚合相关数据元素和使用数据的策略的方式。在 C++ 中通过定义类来自定义数据结构。本章将描述没有任何方法的类。如：

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

C++ 中可以用 struct 和 class 关键字定义类。struct 定义的类默认访问权限是 public ，class 定义的类默认访问权限是 private 。除了与别的语言一样的特性（类作用域、类实例字段是每个对象有一份拷贝、类体可以为空、用点号（.）访问成员）外，C++ 的类以分号（;）结尾，原因在于 C++ 允许在类体后定义变量，分号用来结束声明符（通常为空）。

```
struct Sales_data {} accum, trans, *salesptr;
```

建议不要将类定义和变量定义放在同一条语句中。

在 C++11 中允许为数据成员类内初始值（in-class initializer），没有初始值的成员将执行默认的初始化（内置类型的值是未定义的，类执行类的默认构造函数）。类内初始值的形式必须是列初始化或者等号初始化，类似于函数调用的括号形式的初始化是不允许的。类内初始值并没有限制哪些类型的成员可以被初始化，事实上，即便是类类型亦是可以在类中初始化。

C++ 规定在同一个文件中只能有一个同一个类的定义，这类似于 static 变量或者 const 变量。同时，如果类定义存在于多个文件中，它们必须保持一致。所以，类定义一般放在头文件中。如果头文件更新了，包含头文件的源文件必须重新编译。

术语

46. 基础类型（base type）：在声明语句中处于声明符（declarators）的前面的类型说明符，基础类型可以被 const 修饰。基础类型为声明语句提供共同的类型，声明符将以此为基础进行声明；
47. 绑定（bind）：将一个名字和给定实体相关联，因而，使用名字就是使用底层的实体。引用就是将名字绑定到对象上；
48. 常量表达式（constant expression）：可以在编译期间进行求值的表达式；
49. 数据成员（data member）：组成对象的数据元素，对象占据内存的元素。每个对象有自己独立的数据成员拷贝，数据成员可以在类内进行初始化；
50. 声明符（declarator）：声明的一部分，包含声明的名字和类型修饰符（* & []）；
51. 默认初始化（default initialization）：当没有显式给出初始值时如何进行初始化。类对象初始化取决于类本身，定义在全局作用域中的内置类型值初始化为 0，定义在函数或类中的内置类型值初始化为未定义值；
52. 类内初始化（in-class initializer）：类定义时提供数据成员的初始化值，类内初始化必须是等号符号或者大括号形式初始化；
53. 列初始化（list initialization）：用括号形式进行初始化，括号中放油一个或多个初始值；
54. 底层 const（low-level const）：在复合类型中与基础类型相结合的 const，初始化时不会被忽略；
55. 顶层 const（top-level const）：修饰对象本身的 const；
56. 临时对象（temporary）：由编译器在对表达式求值时定义的未命名对象。临时对象将存活到生成临时对象的表达式结束的地方；
57. 类型说明符（type specifier）：类型的名字；
58. 未定义（undefined）：语言没有明确说明含义的地方，通常依赖于机器、编译器实现，并且成为难以定位的问题来源；
59. 未初始化（uninitialized）：没有给出初始值的变量定义，尝试使用未初始化变量的结果是未定义的；
60. 字（word）：机器执行整数运算的自然尺寸，通常也是 int 类型的长度 4 字节；

C++ 除了定义了各种内置类型，还定义了很多实用抽象数据类型以及与之相关的类型如：迭代器（iterator）和大小（size_type）。本章将描述 string 和 vector 类型。string 支持可变长度字符串，vector 支持可变长度的集合。两者都定义了迭代器类型来遍历其中的元素，这个功能类似于 C++ 内置的指针类型。string 本身则是对 C 风格字符串的简化和模拟，vector 则是对数组的简化和抽象。

内置类型是对硬件的较低抽象，当需要操作底层时很方便使用。而对于大型项目来说就显得抽象程度太低了，因而，C++ 的标准库定义了很多实用的更高抽象的数据类型来提高大型项目的开发效率。

C++ 在设计语言的时候就致力于让类类型与内置类型一样易于使用。因而，定义了其它语言中没有的操作符重载、拷贝构造函数，以及在栈上初始化对象（事实上，C++ 是主流语言中唯一可以在栈上进行初始化对象的语言）。

本章还将介绍数组，特别是数组与指针之间的关系，这一部分将使用的《C 语言编程》中的内容。数组亦是对硬件的较低抽象，就其灵活性而言将无法媲美 vector 类型。

3.1 名称空间的 using 声明

using 声明的格式 `using namespace::name;` 使得在程序中不再需要名称空间前缀就可以直接访问名字。不要在程序中滥用此特性，会引起名字冲突。即便是使用了 using 声明，也可以用全限定的方式引用名字，确保使用是我们想要的名字。必须为每个希望直接使用的名字加上 using 声明，并且用分号结尾。并且不能在头文件中加入 using 声明，这将会导致所有包含的源文件中都有此 using 声明，冲突的危险将加大。

3.2 string 类型

string 表示可变长度的字符串。string 类型包含在 `<string>` 头文件中。std::string 类型被标准库的实现者实现的性能足够好，因而可以适用于绝大多数场景。我们应该记住很重要的一点：相对于内置类型并不总是初始化，所有的类类型都会调用类自己构造函数进行初始化。一个类会定义多个构造函数来执行初始化。

string 中的元素是顺序存储的，对于字符串 s 在范围 `[0, s.size())` 内满足 `&*(s.begin() + n) == &*s.begin()+ n` 相等性。这就是说可以将 `s[0]` 的指针传递给任何期待一个字符串数组头元素指针的函数。（since C++11）

以下是最基本的 string 的构造函数：

61. `string s1` 默认初始化，s1 是空字符串；
62. `string s2(s1)` 将 s2 初始化为 s1 的副本；
63. `string s2 = s1` 与 `s2(s1)` 一样；
64. `string s3("value")` s3 是字符串字面量，不包括末尾的 `\0` 字符；
65. `string s3 = "value"` 与 `s3("value")` 相同；
66. `string s4(n, 'c')` 将 s4 初始化为 n 个字符 c ；

在上面的构造函数中有两种不同的形式：（copy initialize）是以等号形式将右边的初始值复制到左边的对象中去。这里 `"value"` 并不是 string 类型，这种方式其实是构造了一个 string 临时对象并以此临时对象代替了 s3，如果考察 `string str = string("value")` 就会更清楚，因为这个语句跟 `s3 = "value"` 的调用过程是一模一样的。其实这里边的复制过程被优化掉了，而 `s2 = s1` 则执行了复制构造函数，其原因在于 s1 的内存已经被分配给了另一个对象，不能直接替换，相反临时对象可以替换。还有一种初始化形式即直接初始化（direct initialization）。

当只有一个初始值时使用直接初始化和拷贝初始化的效果是一样的。当有多个参数时只能使用直接初始化，或者用 `string s8 = string(10, 'c');` 的形式先创建一个临时对象再执行拷贝初始化，这里同样是直接将临时对象替换过去的，因为临时对象在声明语句后消失，因而可以安全替换。

3.2.2 string 可执行的操作

类除了定义了如何创建和初始化之外还定义了可以执行哪些操作。C++ 中的类既可以定义按名字调用的成员函数，也可以对操作符进行重载，事实上在 C++ 中操作符是一种特殊的函数。Lua 中提供了元表以及元方法跟 C++ 的操作符重载非常类似。如以下列出最常用的几个 string 类方法：

67. `os << s` 将字符串 s 写入到输出流 os 中，返回 os ；
68. `is >> a` 从输入流 is 中读取以空白符分割字符串到 s 中，前导空白也将略过，返回 is ；
69. `getline(is, s)` 从输入流 is 中读取一行到 s 中，读取的行将排除掉换行符，返回 is ；
70. `s.empty()` 判断 s 是否为空串；
71. `s.size()` 返回字符串 s 的长度；
72. `s[n]` [] 操作符重载，返回位置 n 的字符引用，索引从 0 开始；
73. `s1 + s2` + 操作符重载，返回 s1 和 s2 拼接后的字符串，这重载不改变 s1 和 s2 而是生成一个新的字符串；
74. `s1 = s2` 按照字面意思将 s2 拷贝到 s1 中，将调用 string 的拷贝赋值操作符函数；
75. `s1 == s2` == 操作符重载，比较字符串 s1 和 s2 是否完全一致；
76. `s1 != s2` != 操作符重载，当字符串 s1 与 s2 不一致时返回 true ；
77. `<,<=,>,>=` 都是操作符重载，按照字典顺序对字符串进行比较；

用 `<< >> getline` 方法进行读取时，可以检查返回的流状态来判断是否成功，当输入流遇到 eof 或者非法输入时将使条件判断失败。

需要注意的是 size 函数返回的类型是 `string::size_type` 而不是 int，此类型定义在 string 内部作为其补充类型，补充类型使得标准库在不同机器上可以进行不同的实现，从而满足可移植的目的。

`string::size_type` 被定义为一种无符号类型并且足够长用以容纳任何字符串的长度。在 C++ 中应该尽可能使用此类型，为了避免繁杂的拼写，可以用 auto 或者 decltype 关键字。

注意不要将 `string::size_type` 与 int 混用，因为负数会被转化为非常大的无符号值。

string 的比较策略是依次比较每一个字符，直到遇到不一样的字符，或者其中一个字符串结束，比较结果是不一致字符在字母表中位置较后的字符串较大。长短比较可以实现为 `\0` 与字符比较，而 `\0` 小于任何字符，因而，短字符串肯定更小。

将字面量与 string 字符串相加

注意在 + 操作符重载中，函数定义希望接收的参数是 string 类型，但是可以传递字符串字面量。当 C++ 需要一种类型，而另外一种类型可以转换为此类型时，可以将那一种类型的值传入。string 类允许将字符字面量和字符串字面量转为 string 对象。因而，我们可以混用 string 和字面量于 + 操作符重载函数中。但是不能完全使用字面量，原因在于完全使用字面量将会调用字符数组的加法运算，而数组不支持此运算，将无法通过编译。

string 类的这种转换事实上是由对应接收单个参数的构造函数定义的，C++ 语言会隐式调用此构造函数来执行转换。
必须说明的是由于 C++ 必须兼容 C 的原因，字符串字面量并不是 string 类型，而是以空字符 `\0` 结尾的字符数组。但进行混用时应当有所理解。

3.2.3 处理 string 中的字符

cctype 头文件中定义与字符相关的函数如：isalnum、isalpha、iscntrl、isdigit、ispunct 以及 tolower、toupper 。
C++ 除了使用其自己定义的标准库外，还会使用 C 的标准库，C++ 版本的 C 标准库头文件名字被替换为 c 开头，然后去掉 .h 后缀，如：ctype.h 在 C++ 中即为 cctype 。这样做的好处在于新的头文件中所有函数都在 std 名称空间中（虽然不使用 std:: 限定也是可以的），推荐的做法是使用新的头文件。
为了处理 string 中的字符有三种方法：范围 for（range for）语句、迭代器以及下标运算符。
范围 for 语句是 C++11 标准中引入的，形如：

```
for (declaration : expression)
    statement
```

其中 declaration 跟普通的声明一样，可以使用 auto 或者 decltype ，如果希望改变 expression 所表示对象中值或者避免拷贝，将声明中的变量指定为引用。范围 for 会顺序遍历中的 expression 中的元素，并将值拷贝到控制变量中。
下标运算 [] 返回的是元素的引用，C++ 中的所有下标都是从 0 开始。如果下标超出了范围得到的结果是未定义的，因而对空字符串进行下标操作是未定义的。这种数组越界的错误在 C++ 中是不检查的，程序员必须自己保证下标不会越界。值得注意的是 string 的下标操作要求索引必须是 string::size_type 类型的，也就是无符号类型的，所以并不支持负数的索引，这与数组的索引允许负数是不一致的。string 的下标运算允许随机访问。

3.3 vector 类型

vector 是一种类似与数组的容器，容器中的对象都有相同的类型，并且有唯一的索引与之对应。所谓容器就是用来包含其它对象的对象。同时，vector 是一个类模板（class template），C++ 中同时有类和函数模板。模板本身不是函数或者类，但是当提供实例化参数时，编译器会帮助生成新的函数或者类，这个过程叫做实例化（instantiation）。实例化参数包含在模板名称后的尖括号中。如：`vector<int>` 和 `vector<vector<string>>` ，可以将 vector 实例化为包含绝大多数类型，甚至元素可以是 vector ，但不能实例化引用的 vector 。
vector 中的元素是顺序存储的，意味着元素不仅仅可以通过迭代器进行访问，并且可以使用元素指针的偏移进行访问。这就是说可以将 vector 元素的指针传递给一个期待数组元素指针的函数。（since C++ 03）
值得说明的是早期的模板实例化语法不允许两个相邻的 >，因为会被解释为右移操作，如：`vector<vector<string> >` 而不是上面的写法。

3.3.1 定义和初始化 vector

78. `vector<T> v1` 默认初始化 v1 为包含 T 类型变量的空向量；
79. `vector<T> v2(v1)` v2 拥有 v1 中每个元素的拷贝；
80. `vector<T> v2 = v1` 同上，拷贝初始化形式；
81. `vector<T> v3(n, val)` 将 v3 初始化包含 n 个 val 值；
82. `vector<T> v4(n)` v4 初始化为包含 n 个值初始化 T 类型对象；
83. `vector<T> v5{a,b,c,d}` 使用列初始化将 v5 初始化为包含全部初始列表中的值；
84. `vector<T> v5={a,b,c,d}` 与上面的初始化形式一致；
85. `vector<T> v6(begin(T_arr), end(T_arr))` 将 v6 初始化为由两个迭代器指定的范围，范围内的元素拷贝到 v6 中；

一一解释下，空的 vector 看起来没什么用，可其实是最常用的方式。惯用的方式是先定义一个空的 vector，然后在运行时不断添加元素。添加元素本身是一个高效的操作。
当只提供个数而不提供初始值，vector 会对元素进行值初始化（value-initialized），值初始化对于类类型是调用类的默认构造函数，对于内置类型则初始化为 0 。值初始化保证一定有值，而默认初始化（default initialized）对于类类型将调用类的默认构造函数，对于内置类型则在函数外初始化为 0 ，在函数内作为自动变量则是未定义值。

值初始化和默认初始化的一个有意思的例子是，在函数内定义数组，如果不进行初始化则为默认初始化，其中的所有元素都是未定义值，而如果提供少量的初始值，其它元素则执行值初始化为 0 。如：

```
int arr1[10];
int arr2[10] = {1};
```

打印以上数组就会发现不同。

如果类类型的元素没有默认构造函数，则无法定义只包含个数的 vector 。

对于无法执行列初始化时，会转而直接调用构造函数。如：

```
vector<string> v7{10}; //v7 有 10 个元素，每个元素执行值初始化
vector<string> v8{10, "hi"}; //v8 有 10 个元素，每个元素都是 hi 字符串
```

为了达到列初始化，必须使得大括号中的值的类型与元素的类型匹配。

3.3.2 添加元素到 vector 中

预先给 vector 分配个数是低效的，而且 vector 的应用场景就是在不确定有多少元素时使用。事实上，对 vector 进行添加元素是非常高效的。vector 使用 `push_back` 函数进行添加元素。通常，在 C++ 中不会预先给 vector 分配个数。

将一个元素添加到 vector 中，事实上是将其拷贝一份然后添加到 vector 中。值得注意是在范围 for 中不能改变 vector 的元素个数。

3.3.3 vector 的其它操作

vector 中包含另外一些类似于 string 类的成员函数。如：

86. `v.empty()` 判断向量是否为空向量；
87. `v.size()` 返回向量的个数；
88. `v.push_back(t)` 在向量的尾部增加元素 t 的副本；
89. `v[n]` 下标操作将得到向量中的索引为 n 的元素引用，索引从 0 开始；
90. `v1 = v2` 将 v2 中的元素拷贝到 v1 中并替换掉 v1 中的旧值；
91. `v1 = {a,b,c,d}` 将大括号中的值替换掉 v1 中的旧值；
92. `v1 == v2` 判断 v1 与 v2 中的元素个数和值完全一致；
93. `v1 != v2` 判断 v1 与 v2 中的元素不完全一致；
94. `<,<=,>,>=` 重载后的关系操作符，执行字典比较，下面会解释；

size 函数返回的类型需要包含元素类型，如：`vector<int>::size_type` 。字典比较的意思是依次比较向量中的每个元素，当遇到第一个不一致的元素时，较大的元素的向量较大，如果所有元素都是一样，则根据长度判断，较长的向量较大。只有当可以比较向量的元素值时可以比较向量。

vector 和 string 的下标运算一样，索引是 size_type 类型是一个无符号整数，当索引超出范围时，其行为是未定义的。这种越界是非常常见而且难以定位的问题，称之为缓冲溢出（buffer overflow），但是 C++ 并不试图阻止程序员这样做，甚至在标准库容器中也不阻止，string 和 vector 的下标操作同样会产生未定义行为而不是抛出异常。

3.4 介绍迭代器

在 C++ 中更常用的访问容器中的元素方式是通过迭代器（iterator），由于有些容器事实上是根本无法通过下标来访问的，这种容器是不可随机访问的。所有容器都可以通过迭代器来访问元素。虽然 string 不是容器，但是 string 支持很多容器的操作，包括迭代器和下标访问。迭代器是对语言中的指针的一种抽象和模拟，用来间接访问元素。与指针一样，通过迭代器可以从一个元素移动到另一个元素，可以通过解引用返回元素的引用，可以通过箭头符调用其成员函数。迭代器也有非法迭代器（invalid iterator），只有确实指向元素或者指向尾元素的下一个位置才是合法迭代器，其它所有迭代器都是非法的。

3.4.1 使用迭代器

使用成员函数 begin 和 end 分别返回指向头元素的迭代器和指向尾元素的下一个位置（a position one past the end）的迭代器。尾元素的下一个位置是一个不存在的元素称为尾后（off-the-end）作为表示已经处理了所有元素的标记。当迭代器一直递增直到与尾后迭代器相等时应该停止迭代。空容器的 begin 和 end 返回相等的迭代器。

迭代器只支持比较少的操作，只有支持随机访问的容器支持其迭代器与整数的加减法来用快速步进迭代器。以下是所有迭代器都支持的操作：

95. `*iter` 解引用迭代器返回所指向的元素的引用，不能对尾后迭代器进行解引用；

96. `iter->mem` 直接调用迭代器所指向元素的成员函数 mem，与 `(*iter).mem` 完全一致；
97. `++iter` 自增 iter 使其指向下一个元素，不应该对尾后迭代器进行自增操作；
98. `--iter` 自减 iter 使其指向前一个元素；
99. `iter1 == iter2` 判断两个迭代器是否相等，只有指向同一个容器中的同一个元素（尾后元素也是一个元素）的迭代器才会相等；
100. `iter1 != iter2` 判断两个迭代器不相等；

与指针一样解引用一个非法迭代器和尾后迭代器的行为是未定义的。在 C++ 中常用的比较迭代器的方式是 `!=` 而不是 `<`，其原因在于很多迭代器根本没有 `<` 操作，为了支持所有迭代器，通用的做法是使用 `!=` 来比较。如：

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it);
```

迭代器类型

通常我们不需要知道迭代器的确切类型。标准库为每个容器类定义两个迭代器类型：`iterator` 和 `const_iterator`。所有的 const 对象返回的的迭代器都是 const_iterator，const 迭代器和 const 指针一样不能用于改变所指向的元素值。非 const 对象可以通过 cbegin 和 cend 成员函数返回 const 迭代器。值得一提的是迭代器是一组概念相关的类型，它们都支持一类操作，并且行为是类似的。同样容器也是一组相关的类型。

记住：任何改变 vector 长度的操作（如 push_back）会使得之前返回的迭代器失效。所以，不要在使用迭代器的循环中改变 vector 的长度。

迭代器算术运算

只有少部分容器支持迭代器的算术运算，支持算术运算的容器通常是可随机访问的容器，意思是可以在任何时刻访问容器中的任意元素，而其它容器则必须一个一个向前移动访问。这种区别就跟数组与链表的区别。迭代器算术运算（iterator arithmetic）指的是对迭代器加减整数返回的是一个跨越若干元素的另外一个迭代器，加则向后移动，减则向前移动。两个迭代器相减返回两者之间的距离。这样的迭代器通常还支持关系运算，可以比较大小，而其它的迭代器则不可以。以下表格：

101. `iter + n` `iter - n` 加减运算将迭代器向前或向后移动 n 个位置，得到的迭代器需要程序员自己确保是合法的；
102. `iter += n` `iter -= n` 支持复合运算；
103. `iter1 - iter2` 产生两个迭代器间的距离，两个迭代器必须指向同一个容器，否则结果是未定义的。返回结果加上右操作数返回左操作数；
104. `>,>=,<,<=` 关系运算，只有当两个迭代器都指向同一个容器时才有意义，否则结果是未定义的。当一个迭代器在另一个之前时，我们说此迭代器较小。

`iter1 - iter2` 的结果类型是各自容器的 difference_type，如：`vector<int> difference_type`，这是一个有符号整数。

3.5 数组

数组（array）由语言定义，用于容纳一系列紧靠的无名对象，数组的大小是不可变的。通常使用数组是由于其优于 vector 的运行性能。建议除非有充分的理由使用数组，尽可能是在任何场景使用 vector。数组是第三种复合类型，而且数组的维度是其类型的一部分，意味着 `int[3]` 和 `int[10]` 不是同一种类型。因而，C++ 要求数组的维度在编译时就得确定，因而，数组的维度必须是常量表达式。但需要了解的是某些编译器是允许定义可变长度数组（VLA），这是编译器自己的扩展行为并不属于语言标准。如：

```
constexpr unsigned sz = 42;
int *parr[sz];
```

在不给定初始值时，数组是默认初始化（default initialized）的，上面讲过与值初始化的区别。内置类型数组如果定义在函数中，其默认初始化是未定义值。数组的定义是不允许使用 auto 关键字对其元素类型进行推断，也没有元素为引用的数组。

显式初始化数组元素

数组可以用列初始化，这跟所有的容器类的列初始化是一样的。如果进行列初始化时不提供维度，则编译器从初始值列表中推断。如果指定了维度，则初始列表的个数一定不能超过维度，否则将是编译错误。

如果初始值列表长度不足维度数，则剩余的元素将执行值初始化，对于内置类型来说就是都初始化为 0 。如：

```
int a3[5] = {0,1,2}; //相当于{0,1,2,0,0}
string a4[3] = {"hi","bye"};//相当于{"hi","bye",""}
```

需要特别指出的是 C++ 继承了 C 的字符串字面量，它们是以空字符 \0 结尾的字符数组。因而，当用字符串字面量初始化数组时，实际的维度比看到的字符数多 1，这个字符串包括结尾空字符都复制到字符数组中去。如：

```
char a1[] = "C++"; //相当于{'C','+','+','\0'}
```

不能将数组初始化为另外一个数组，也不能将数组赋值给另外一个数组。

理解复杂数组声明

需要理解以下一些复杂的数组声明：

```
int *ptrs[10]; //ptrs is an array of ten pointers to int
int (*Parray)[10]; //Parray points to an array of ten ints
int (&arrRef)[10]; //arrRef refers to an array of ten ints
```

3.5.2 访问数组元素

通过范围 for 和下标操作可以访问数组的元素，索引是从 0 开始的。下标值通常被定义为 size_t 类型，size_t 是机器相关的，被保证足够容纳所有类型的对象内存大小。但是数组的索引还可以用 int 类型并被指定为负数，表示从数组的某个位置向前 n 个位置。新标准建议现在的程序最好使用范围 for 来遍历整个数组。

记住一点：数组的类型包含了其维度。这在指针、范围 for 和引用中都会用到此特性。

重申一下，访问数组的数组越界行为同样是未定义行为，所谓未定义行为就是即便出错了，编译器也不协助检查，一切全靠程序员自己检查。而且，在运行时未定义行为可能会正确，可能会在很久之后引起系统崩溃，但绝不会抛出异常。而这正是 C/C++ 这两门语言难学之处。缓冲溢出几乎是所有重要的程序中最严重的问题。

3.5.3 指针和数组

指针和数组在很多时候可以相互替换使用，原因在于数组名其实是数组首元素的指针。但是，它们之间还是存在一些细微而且易错的却别。如：

对数组进行 sizeof 操作得到是整个数组的所占的内存的大小，而对指针的 sizeof 操作得到的是指针所占内存的大小。

```
int arr[] = {10,20,30,40,50,60};
int *ptr = arr;
cout << sizeof(arr) << endl;//24
cout << sizeof(ptr) << endl;//8
```

语言不允许对数组进行直接赋值，但是指针可以，对指针赋值使得指针指向别的位置。

```
int x = 10;
arr = &x; //错误!!!
ptr = &x; //指向 x
```

对指针进行取地址得到是指针的指针，对数组进行取地址得到是包含数组维度的数组指针。

```
int ** pptr = &ptr;
int (*parray)[6] = &arr;
```

用数组初始化的字符串常量可以改变其元素，用指针初始化的字符串常量改变其元素将是未定义行为，原因在于前者拷贝了字符串常量，而后者指向的是只读存储字符串常量的只读内存位置（称为字符串常量表）。

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

除此之外指针和数组就可以完全替换使用，特别是数组名可以赋值给指针变量，指向数组元素的指针可以用下标访问别的元素。通常，编译器会将数组转为一个指向首元素的指针。如：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
auto ia2(ia); //int*
```

然而， decltype(ia) 返回的是数组 int[10]，这是例外的地方，并且定义数组的引用时，不会自动变成指针的引用，因而以上差异存在与此引用与对应的指针之间。

指针是语言定义的迭代器

事实上迭代器是对指针的模拟和抽象。指针支持自增、自减和算术运算。如第二章所说，指向数组元素的指针中有一个特殊指针即指向数组尾元素下一个位置（one past the last element）的指针，这个指针叫尾后（off-the-end）指针。通过将索引指定为数组长度得到的就是尾后指针，如：

```
int *e = &ia[10]; //ia 是长度为 10 的 int 数组
```

上面 ia[10] 是一个不存在的元素，对其唯一允许的操作就是取地址，除此之外的任何操作都是未定义的。尾后指针不能解引用，向后移动亦是非法的。

标准库 begin 和 end 函数

新标准中在 <iterator> 头文件中定义了 begin 和 end 函数用于返回数组的头指针和尾后指针，行为与容器的同名函数一样。这两个方法以数组为参数。这样就将迭代器和指针统一了，范围 for 以及泛型方法就是利用了这个特点得以以统一的方式对它们进行操作。

指针算术运算

毫无疑问数组是支持随机访问的，程序员可以在任何时候访问数组中的任意位置上的元素。指针支持与整数的加减法，确保结果指针依然指向数组中的元素的工作交给了程序员完成。指向相同数组的指针间的减法将得到两者之间的距离，结果类型是 ptrdiff_t ，此类型是机器相关的有符号整数，并且保证容纳任何地址差。同样指针支持关系运算符，然而将其运用于两个不相关的对象指针上结果是未定义的。

指针和下标操作

对数组进行下标操作和对指针进行下标操作的效果是等同的，意味着可以在对指针进行下标操作。如以下方式都是等同的：

```
int i = ia[2];
int *p = ia;
i = *(p+2);
i = p[2];
```

甚至可以像如下代码这样做，将索引指定为负数，只要取出的元素确实存在于数组中。如：

```
int *p = &ia[2];
int j = p[-1];
int k = p[-2];
```

vector 和 string 的下标要求一定是无符号整数，而数组的下标可以是负数。这是它们之间的重大区别。

3.5.4 C 风格字符串

C 风格字符串并不是一种新的类型而是保存在字符数组中并且以空字符结束。通常是使用指针来操作这种形式的字符串。C 风格字符串函数定义在 <cstring> 头文件中。strlen 获取字符串长度，当遇到空字符时停止计数。strcmp(p1,p2) 比较字符串 p1 和 p2，相等时返回 0，strcat(p1,p2) 则是将 p2 拼接到 p1 上，需要程序员保证 p1 的内存足够容纳下这些字符，否则行为将是未定义的。strcpy(p1, p2) 将字符串 p2 复制到 p1 所在内存位置，亦需程序员保证内存足够。

以上函数中的参数必须是以空字符结尾的字符数组的首元素指针。否则行为是未定义的。使用 string 类型时可以直接用关系操作符进行比较，而 C 风格字符串却是必须调用函数进行比较，如果用关系操作符比较的则是指针的值。如：

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) //错误!!!
```

strcat 和 strcpy 都需要程序员保证内存不会溢出，如果使用 string 则没有这样的担心，这正是应该使用 string 的原因所在：更加安全而且更加高效。缓冲溢出已经成为了 C 语言中最严重的安全问题。

3.5.5 提供给旧代码的接口

在 C++ 标准订立前 C++ 语言就已经横空出世了，那时很多程序根本没有 string 类可用，并且很多时候 C++ 程序必须给 C 程序提供接口，因而需要将 string 字符串转为 C 风格字符串。C++ 语言中 C 风格字符串可以自动转为 string 类型，但是并没有相反的转换，string 提供了 c_str 成员函数用来返回其内容的 C 风格字符串。返回结果是 const char* 类型。并且，不保证这个 C 风格字符串一直是有效的，当原始 string 改变时，此字符串就很可能会失效。所以要求使用者每次都调用 c_str 函数。

现代 C++ 程序更加推荐使用更为抽象的 string vector iterator ，好处在于更加安全且方便。

3.6 多维数组

多维数组的用途比较受限，因而较少使用。事实上，多维数组的所有元素都是顺序排列在一起的，所以用相同长度的一维数组构建是一样的。唯一的区别在于类型上的不一样：多维数组的元素是指定维度的数组。多维数组因而又称为数组的数组。

定义多维数组是一件很简单的事，多加一个中括号维度即是。如：`int ia[3][4]` `int arr[10][20][30]`。这里第一个数组的元素是 `int[4]` 类型，第二个数组是 `int[20][30]` 类型。ia 可容纳 12 个 int 值，arr 可容纳 600 个 int 值。

多维数组的初始化很有意思，根据前面的描述：所有元素都是依次排列成一线。所以内部的 {} 可以消除。如：

```
int ia[3][4] = {
    {0,1,2,3},
    {4,5,6,7},
    {8,9,10,11}
};
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11}; //两者是完全一致的
```

如果不提供全所有值的话，每个内嵌的初始列表将初始化那一列，未给出的值均为 0 。

```
int ia[3][4] = {{0}, {4}, {8}}; //每一行后面的 3 个元素初始化为 0
```

而不给出内嵌大括号，将只初始化整个数组的前几个元素。如：

```
int ia[3][4] = {0,3,6,9}; //后面的元素都是 0
```

多维数组的下标引用

多维数组进行下标引用将得到多种不同类型的元素，具体看给出了多少个下标值。如：

```
int arr[10][20][30];
arr[0][0][0]; //int
arr[1][3]; //int[30]
arr[2]; //int[20][30]
```

这同样会影响到指针的类型，指向数组的指针会带上数组的维度。如：

```
int (*ptrarr)[20][30] = arr;
int (*ptr2)[30] = arr[0]; //或者 = *arr，但是下标形式更加易于理解
int *ptr3 = arr[0][0]; // or = **arr
```

如果将多维数组运用于范围 for，外部循环中的控制变量必须使用引用形式，否则得到的将是指针而不是数组，而指针是不能遍历的。如：

```
for (auto &row : ia)
    for (auto &col : row)
        //do something
```

关键术语

105. 缓冲溢出（buffer overflow）：一种严重的程序 bug ，由容器或数组的越界导致的，在 C++ 最值得注意的问题之一；
106. 类模板（class template）：关于怎样由编译器生成指定类型的蓝图，定义了通用的函数和数据，只要填入对应模板参数即可；
107. 编译器扩展（compiler extension）：特定编译器支持的特性，通常是语言标准的超集；
108. 容器（container）：用于容纳其它对象的对象，现代语言的基石；
109. 拷贝初始化（copy initialization）：= 形式的初始化，将给定的初始值拷贝到即将创建的对象中去；
110. 直接初始化（direct initialization）：直接调用类的构造函数的初始化；
111. 实例化（instantiation）：从模板中产生一个特定的类和函数的编译器行为；
112. 尾后迭代器（off-the-end iterator）：指向容器尾元素的下一个位置的迭代器；
113. 范围 for（range for）：有编译器控制的用于迭代容器或数组的特殊 for；
114. 值初始化（value initialization）：一种初始化，内置类型将初始化为 0，类类型将调用默认构造函数进行初始化；

C++ 中的操作符（operator）可以进行重载，于是类对象也可以方便的用操作符表达自己的含义。如，string 对象的 [] 下标操作符，迭代器的 * 解引用操作符。本章的内容主要讲解内置类型的操作符。
表达式（expression）是由多个操作数（operand）拼接操作符组成，其求值可得到一个结果。最简单的表达式是字面量或变量，其值就是字面量或变量值。复杂的表达式由一个操作符和一个或多个操作数组成。表达式是最基本的计算单元。

4.1 fundamentals

有一些基本概念将影响表达式如何求值。如：操作符优先级、结合性以及操作数的求值顺序这些关键概念，以及操作数类型转换、操作符重载以及左值右值区分，都是理解表达式需要理解的更为基础的概念。

4.1.1 基础概念

操作符有一元操作符（unary operators）和二元操作符（binary operators）以及三元操作符（ternay）以及函数调用（function all），其中函数调用可以有无限制的操作数。有些操作符有不同的含义，如：* 既可以是解引用也可以是乘号，应当将他们当作独立的符号，依据上下文进行判定。
多个操作符构成的表达式需要理解操作符的优先级（precedence）和结合性（associativity）以及在某些情况下表达式依赖于操作数的求值顺序（order of evaluation）。
当表达式中的操作数不一样时，会将操作数转为相同的类型。如：整型提升，将整数转为浮点数的转换，在赋值表达式中将浮点数转为整数。
对于类类型，可以定义类自己的操作符含义，使得它们可以跟内置类型一样使用，称为操作符重载（overloaded operators）。操作符重载可以改变操作符的操作数类型和结果值类型，但是不能改变操作符的操作数个数以及优先级和结合性。
C++ 中左值（lvalue）和右值（rvalue）是很重要的概念。这两个概念是从 C 中继承过来的，C 定义的很简单，在等号左边就是左值，右边就是右。C++ 的左值表达式生成对象或函数，const 左值是不能放在等号左边的，一些表达式返回临时对象只能作为右值。通常来说，右值使用的是对象的值（内容），左值使用的对象本身（内存中的位置）。操符在需要左右值作为操作数以及返回左右值有重大区别。而需要右值的地方可以使用左值，当这样使用左值时，用的是对象的值。以下列举最常见的左右值操作符解释：
115. = 等号的左操作数必须是左值，并且将赋值后的左操作数作为左值返回；
116. & 取地址符需要左值，并且以右值形式返回指针；
117. * 解引用和 [] 下标操作符都将产生左值；
118. 自增操作符和自减操作符都需要左值作为操作数，前置形式返回左值，后置形式返回右值；
将 decltype 运用与产生左值的表达式时，返回的是结果类型的引用。

4.1.2 优先级和结合性

表达式中有两个或多个操作符是一个复合表达式（compound expression），优先级和结合性决定了符合表达式中的操作符与操作数怎样进行组合，决定了哪些部分先进行求值。可以用括号覆盖这些规则。优先级高的操作符比低的先求值，结合性决定了相同优先级的操作符谁先进行求值。

4.1.3 求值顺序

大部分的操作符不会要求操作数的求值顺序，可以以任何顺序对操作数进行求值。如：`int i = f1() * f2();` 就能以任何顺序对 f1 和 f2 函数进行调用。对于不强制要求操作数顺序的操作符，如果对一个操作数又改变又取值就是错误，通常导致未定义行为。如：

```
int i = 0;
cout << i << " " << ++i << endl;
```

编译器可能先去 i 的值在进行 ++i，也可能先进行 ++i 再取 i 值，或者编译器可以同时运算。三种结果是完全不同的。程序应当避免这样做。
只有 4 个操作符是保证求值顺序的。&& 逻辑与、|| 逻辑或、?: 条件操作符、，逗号操作符。
而且操作数的求值顺序与优先级和结合性是完全独立的。
建议：当对操作符的优先级有怀疑时使用括号进行标明，或许别人有跟你一样的疑惑。不要在一个表达式中又改变操作数值，又取值。

4.2 算数运算符

包括：+ 一元加，- 一元减，* 乘号，/ 除号，% 取模，+ 加号，- 减号。

所有算数运算符（arithmetic operator）都是左结合的，操作数和结果都是右值。一元加号返回操作数的一个值副本。一元负号返回负的值副本。记住：对于绝大部分操作符，bool 值会被提升为 int ，特别注意的是整数与布尔值进行相等比较时，会将布尔值转为整数进行比较。
算数运算符需要注意的是值溢出，分为上溢和下溢。特别是乘法与除法结合时，虽然最后结果在范围内，但是乘法先计算是有可能导致溢出。此时可以先计算除法再计算乘法来避免。现代计算机的整数表示都是补码，补码溢出时会回绕，上溢之后值变成最小值，下溢则变成最大。
整数除法需要注意的是当遇到负数时应当如何舍入，取模运算（%）只能运用于整数类型，并且其舍入方式与除法是一致的。新标准要求舍入方向是向 0 取整，意味着截断商结果的小数点。由于模运算符合 `(m/n)*n + m%n` 因而，模的结果与被除数 m 的符号一致。通常，极少会对负数进行取模。如：

```
-21 % -8 = -5      -21 / -8 = 2
21  % -5 = 1       21  / -5 = -4
```

4.3 逻辑和关系操作符

包括：`!`逻辑非，`<` 小于号，`>` 大于号，`<=` 小于等于，`>=` 大于等于，`==` 等于，`!=` 不等于，`&&` 逻辑与，`||` 逻辑或。
逻辑和关系操作符的操作数是右值，结果是右值。所有这些操作符都返回 bool 值，算术值和指针值中的 0 被认为是 false，其它值被认为是 true 。逻辑与和逻辑或执行短路求值（short-circuit evaluation），意味着它们在决出了结果之后其后面的操作数不再计算。&& 只有在左操作数的求值为真时，才会对右操作数求值。|| 只有在左操作数为假的情况下，才会对右操作数求值。因此，可以让左操作数作为右操作数的守卫，使得右操作数的计算是安全的。如：`index != s.size() && !isspace(s[index])` 只有当 index != s.size() 时，对 s 进行下标操作才是安全的。
关系操作符需要注意的是不要将多个关系符号串在一起，结果肯定是不对的。

相等测试与 bool 字面量

需要注意的是测试算术或指针对象的最好方式是直接在条件语句中测试，而不要让它们与 bool 值进行比较，这样会引起意外的结果。如：

```
if (val) {/*...*/}
if (!val) {/*...*/}
if (val == true) { /* ... */ }  //只有当 val 为 1 时才会相等
```

原因在于第三个语句中 true 会被整型提升为 int 类型，从而 true 转为 1，从而变成 `if (val == 1)` 这肯定不是我们想要的。在 C++ 中 bool 其实是一种小整型。bool 字面量须与 bool 类型值比较才是正确的做法。

4.4 赋值操作符

赋值操作符的左操作数必须是可修改的左值，结果是其左操作数，并且是一个左值。当左右类型不一样时，右操作数类型转为左操作数类型。在新标准下，可以提供一个大括弧初始值列表，如果提供一个空列表，左操作数将被赋值一个值初始化的值。列表中的值不能执行精度变小的转换。如：

```
int k;
k = {3.14}; //错误!! 精度变小
k = {}; //k 等于 0
vector<int> vi;
vi = {};
vi = {0,1,2,3,4};
```

赋值是右结合的。赋值语句的结果是最右端的操作数的值，整个链上的值都是一样的。多赋值表达式的类型必须与其右边的操作数一致，或者可以互相转换。
赋值操作的优先级通常较低，特别是赋值比关系操作优先级还低，所以经常需要用括号将赋值操作括起来。注意不要将赋值操作和等于操作混淆，GCC 编译器会对括号中的没有用括号括起来的赋值语句进行报警，混淆赋值和等于操作的 bug 是非常难定位的。

复合赋值运算符

复合赋值运算就是将左边操作数与右边操作数进行运算然后将结果值赋值到左操作数中。如：`+= -= *= /= %= >>= <<= &= ^= |=`，复合赋值语句相当于 `a = a op b`，当使用复合赋值时，左操作数只求值一次，而常规赋值将求值两次。仅当求值会影响时，才在考虑的范围内。

4.5 自增和自减操作符

自增和自减操作符可用于指针、迭代器和算术类型。这两个运算符都要求操作数是左值。自增和自减有两种形式：前置和后置。通常在 C++ 中使用前置形式，因为前置返回改变后的对象，返回的是左值。而后置形式返回的改变前的对象，因而必须是对象的副本，是一个右值。对于内置类型来说是无关紧要的，但是对于类类型来说则减少了拷贝。相对的，在 C 中两个操作符返回的都是右值。C++ 建议：仅在必要使用后置形式。

在 C 和 C++ 中 `*pbeg++` 是一种非常常见的模式，程序必须掌握这种地道的写法，绝大多数 C++ 程序都使用这种写法。

4.6 成员运算符

C++ 跟 C 一样提供了两种形式的成员运算符，类或结构对象可以直接用 `.` 点号访问成员，对象指针可用 `->` 箭头访问成员。`ptr->mem` 相当于 `(*ptr).mem`，箭头操作符要求一个指针操作数，并且返回一个左值方能访问其成员。点号操作符当对象是左值是返回左值，当对象是右值时返回右值。

4.7 条件操作符

`cond ? expr1 : expr2` 条件操作符的优先级非常低，并且求值顺序是有要求的，当条件为真时，? 后的 expr1 求值，否则对 expr2 求值。要求 expr1 和 expr2 的结果是相同类型或者可以相互转换。如果两个表达式都是左值，则整个条件操作的结果是左值，否则将是右值。

条件操作可以嵌套，但最好不要嵌套多于两层。条件操作是右结合的，意味着多个条件操作符嵌套时将从右边开始分析。

由于条件操作符的优先级非常低，所以很多时候必须将条件操作符用括弧括起来。

4.8 位（bitwise）操作符

位操作符：~ 位取反，<< 位左移，>> 位右移，& 位与，| 位或，^ 位异或。 位操作符只能作用于整型操作数，并将整数作为位的集合，位操作符可以操作单个位。如果是小整数首先将会执行整形提升。操作数则既可以是有符号也又可以是无符号整数。如果操作数是负数，则如何处理符号位是机器相关的。特别是那些有可能会改变符号位的操作，比如左移操作。由于语言没有保证如何处理符号位，因而强烈建议只使用无符号类型。

位移（bitwise shift）操作符

位移操作符返回的值是左边操作数的一个副本，它们并不会改变左操作数而是返回一个新值。位移操作将左操作数按照右操作数指定的位数进行左移或者右移。左移时将以 0 填充右边的位，而右移如果是无符号整数则用 0 填充左边的位，如果是有符号数则是机器相关的，可能用符号位填充左边的位，也可能用 0 填充。

右边操作数要求必须是非负数的，并且值必须严格小于结果的位数，否则结果是未定义的。

位移操作符是最常见的重载操作符之一，如：输入输出操作符就是重载过的位移操作符。重载后的操作符的优先级以及结合性与内置版本是一样的。由于这两个操作符的优先级属于中等，意味着经常需要使用括号来保证优先级的正确。

位与、或、异或操作符

通常位于、或、异或用来检查和设置掩码（mask）。如：

```
quiz1 |= 1UL << 27; //将 quiz1 的第 27 位设置为 1
quiz1 &= ~(1UL << 27); //将 quiz1 的第 27 位设置位 0
bool status = quiz1 & (1UL << 27); //测试 27 位是否位 1
```

sizeof 操作符

sizeof 操作符返回表达式结果类型或类型名所表示类型的内存尺寸，返回的大小是固定尺寸不会包含动态分配的内存。因而，对于 vector 类型不会包含分配的元素的内存大小。sizeof 是右结合的，sizeof 是一个结果为 size_t 类型的一个常量表达式。操作符形式如：

```
sizeof (type)
sizeof expr
```

sizeof 操作符并不会对表达式进行求值，而是推断其结果类型，进而计算所需的内存大小。甚至不理会表达式本身是否求值出来是未定义的，如：解引用一个未初始化的指针。对解引用一个非法指针就行 sizeof 求值是安全的，因为 sizeof 根本不会真正对指针进行解引用，只要它的返回值类型。如：

```
Sales_data *p;
sizeof *p; //获取 p 所指对象类型的内存大小
```

在新标准中，可以使用 :: 作用域操作符来获取类的成员的大小。如：`sizeof Sales_data::revenue` 。需要注意的是当对数组进行 sizeof 操作时返回的是数组占的内存大小，此时数组不会转为指针。对指针进行 sizeof 得到的就是指针本身所占内存大小，在 64 位机器上是 8 字节。对 string 或 vector 进行 sizeof 操作只返回对象所占的固定部分，不会返回为元素分配的动态内存。

4.10 逗号操作符

逗号操作符（comma operator）有两个操作数，而且是从左到右顺序求值的。逗号操作的结果是其右边操作数的值，如果右操作数是左值返回的结果就是左值。逗号操作符常用于 for 的条件部分。

4.11 类型转换

C++ 中的类型之间存在转换（conversion）时被认为是类型相关，当需要其中一种类型值时，可以用另一种类型值替换。在第二章中讲述了内置类型间的转换规则，这里不再累述。当不需要程序员干预就可以自动发生的转换称为隐式转换（implicit conversion），隐式转换保证不丢失原有类型的精度。如果转换发生在初始化时，被初始化的类型占据主要地位，意味着别的类型必须转换过去，如：`int ival = 3.541;` 将 double 类型值转换为 int 类型值，即使此时精度会丢失。

以下情况将发生隐式转换：

119. 在几乎所有的表达式中，小整型将首先进行整型提升；
120. 在条件部分，非 bool 值将首先转为 bool 值；
121. 在初始化或者赋值表达式中，右边的值类型转为左边值类型；
122. 在算数和条件表达式中，操作数的类型转为一个共同的不丢失精度的类型；
123. 函数调用时实参类型转为形参类型；

4.11.2 其它隐式转换

124. 数组转为指针，在某些条件下不转为指针参考 CH4 描述。
125. 指针转化：0 和 nullptr 转为任何类型的指针，非 const 指针可以转为 void，*任何类型的指针可以转为 const void*，子类的指针可以转为父类的指针；
126. 转为 bool：任何非 0 值都将转为 true，0 转为 false；
127. 转为 const：指向非 const 对象的指针可以转为指向 const 对象指针，于引用是一样的，而不能做相反操作；
128. 由类定义的转换：类可以通过定义单参数的构造函数来实现由其参数类型到类类型的转换，除非显式禁止，编译器会自动进行如此转换。编译器类的直接初始化（direct initialization）转换，如果需要通过两步以上的转换才能转为需要的类型，编译器会拒绝；

4.11.3 显式转换

显式转换（cast）被 C++ 重新定义为更加精细的多种不同强转，当然 C 风格的强转也可以使用。C++ 中定义了多种具名强转，形如：`cast-name<type>(expression)` 。其中 type 是目标类型，expression 是需要被转换的值，如果 type 是一个引用，则结果是一个左值。强转名有：`static_cast`、`dynamic_cast`、`const_cast`、`reinterpret_cast`。

static_cast

这种强转主要用于定义良好的转换，如：算数类型之间的转换，将 void* 指针转为实际的类型指针，以及指向基类的指针转为指向子类的指针，或者可以用 expression 直接初始化 type ，将任何类型转为 void 类型，将枚举转为整数或者相反。如果本身不能这样转换，但是使用了的话则行为是未定义的。对于这种转换，C++ 语言本身不做任何运行时安全检查。相对的，dynamic_cast 将会对转换类型进行运行时检查，确保确实可以从基类指针转为子类指针。

const_cast

用于改变操作数的底层 const，最常用的是去掉 const 修饰。一旦去掉了 const ，编译器将不做任何检查，如果对象本身就是一个 const 的，试图改写这个对象将是未定义的。只能通过 const_cast 来改变对象或表达式的 const 属性，如果用别的具名强转将产生编译时错误，同样不能用 const_cast 改变表达式类型。

reinterpret_cast

reinterpret_cast 将执行位模式的重新解释，这样可以将一个类型转为任何别的类型。主要用于内置类型的转换，运用于非内置类型可以肯定是错误的。如：

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

接下来的操作将会假设 pc 指向一个 char 类型值，编译器将无法对真实类型做检查。好处在于灵活性，特别是处理网络层读写时经常用到。坏错则是这种操作非常之危险，一朝不慎，整个程序崩溃。

C 风格强转

C 风格强转形如：

```
type (expr);  //函数风格
(type) expr;  //C 语言风格
```

C 风格强转根据语境可以是以上四种具名强转的任何一种，在 C++ 中不推荐如此使用。

4.12 操作符优先级

参考网页：https://en.cppreference.com/w/cpp/language/operator_precedence

C++ 定义了表达式语句（expression statement）与声明语句（declaration statement）和控制语句（flow-of-control statement）以及空语句（empty statement）。

表达式语句就是表达式后跟一个分号，其对表达式进行求值并将结果抛弃，通常表达式语句会产生副作用（size effect）如：赋值、输入输出等。

声明语句则是声明各种变量，这在几章中进行过详细的描述。

控制语句包括：if 语句、switch 语句、while 语句、do-while 语句以及 for 语句。这些语句跟 C 、Java 语言中的结构几乎完全一致，一致的部分不再给出。

空语句就是一个分号，本身不具有任何功能，当语言要求必须有一条语句而又没有可执行的操作时就可以写一条空语句。

以上语句加上 {} 大括号构成复合语句（compound statement）或者称为块（block）。块引入一个新的作用域，定义在此作用域中的名字只能在此作用域或其子作用域中可见。名字从声明位置到块结束的右括号为止。这种复合语句与简单语句是可以互换的，通常当逻辑上需要多条语句，而语言只允许一条语句时就会使用复合语句。

if 语句

if 语句需要注意悬决 else（dangling else） 问题，即 else 只与最近的 if 匹配，如果不是这样，需要使用括号强制匹配。

switch 语句

case 标签中的值必须是整型常量表达式，并且不能有任何两个 case 具有相同的值。当 switch 选中一个 case 后会一直执行到最后，如果需要只执行一个 case 则需要主动使用 break 。switch 语句中涉及到跳过代码，C++ 语言要求一定不能跳过变量的定义而直接跳到使用的地方。如：

```
int val = 1;
switch(val) {
    case 0:
        int passby = 0;
        break;
    case 1:
        ++passby;  //错误!! 跳过了定义，而直接使用变量 passby
        break;
}
```

while 语句

注意 while 条件或循环体中定义的变量，在每次循环时会重新定义和销毁。

do-while 语句

需要注意在 do 循环体中定义的变量，无法在条件中访问。do-while 语句至少会执行一次。

for 语句

for 分为两种：传统 for 和范围 for。传统 for 就是 C 风格的 for，头部形如：

`for(initializer;condition;expression)`其中 for 头部中的三个部分都可以省略，此时循环体就会

无限执行。在 C++ 中 initializer 中定义的变量当循环结束时会被销毁，而在 C89 中并不会，C99 行为与 C++ 一样。

范围 for 形如：`for (declaration : expression)` 其中表达式可以是大括号中的初始列表（initializer list）、数组、或者容器类或 string 类等能够被遍历的序列类型。declaration 中定义的变量类型必须与序列中元素类型一致，最常用的方式是用 auto 声明变量，如果希望改变序列中的值，将循环变量类型定义为引用类型。每次循环时，循环变量都会被初始化为下一个元素的值。

goto 语句

goto 语句用于在同一个函数内跳转。goto 跳转需要给出一个标签，这个标签在一条语句的头部，形成标签语句（labeled statement）。形如：

```
goto label;
label: statement;
```

其中标签独立于语言中的任何别的类别的标识符，即便是与它们重名亦是可以的。与 switch 一样，不可以跳过变量的定义而直接到使用的位置。如果是向后跳转则将跳过部分的变量销毁。

异常处理

在 C++ 中进行异常处理是很难做到干净而简单的，其根本原因在于 C++ 中的对象是由程序员管理的。而抛出异常之后需要将申请的资源以及内存回收，C++ 除了会回收局部对象外并不做额外的工作。C++ 的异常机制没有 finally 子句，原因在于 C++ 有 RAII（Resource Acquisition Is Initialization）获取资源即初始化，当对象销毁时将自动释放资源。

C++ 允许抛出任何类型的对象而不仅仅是 exception 及其子类。C++ 抛出的对象不需要 new，因而，编译器会将其放到一个特殊的位置中，从而可以在不同的调用栈层次进行匹配检查。throw 子句通常构造一个对象，将这个对象放到编译器的特殊位置中。catch 子句中包含一个对象声明，当匹配时编译器将抛出的对象复制到声明对象中，如果声明为引用则将其初始化为抛出对象的引用。当 catch 子句结束时，声明的对象将先被销毁，然后抛出对象也被销毁。重新抛出对象只要 `throw;` 即可，此时会将同一个异常对象再次抛出，如果声明为引用并且改变了此异常对象，那么再次抛出的就是改变后的对象。

查看：throw.cpp 了解示例代码。

C++ 的异常匹配与 Java 中的一样，将最匹配最先找到的 catch 子句，而不是精确匹配，所以应当将最具体的异常类放在最前面。另外需要注意的一点是在 try 块中声明的对象是不能在 catch 中访问的。

警告：C++ 中很难达到异常安全

当异常发生时将打断程序的正常执行流，此时某些计算还未完成，这样便将一些对象置于非法或者不完全状态，或者资源没有被释放。程序必须正确的清理所有这些才算是异常安全。需要处理异常的程序需要时刻谨记可能出现的异常，以及必须采取的步骤来保证对象处于安全的状态以及资源不会泄露，并且将程序恢复到正确的状态。

标准异常

exception 是所有标准异常的基类，定义在 exception 头文件中。stdexcept 头文件中定义了诸如：

129. `runtime_error` 运行时错误；
130. `range_error` 运行时错误，产生的结果超出有效值范围；
131. `overflow_error` 运行时错误，计算上溢；
132. `underflow_error` 运行时错误，计算下溢；
133. `logic_error` 程序逻辑错误；
134. `domain_error` 逻辑错误，在给定参数的情况下无法生成结果；
135. `invalid_argument` 逻辑错误，非法参数；
136. `length_error` 逻辑错误，试图生成一个操作最大尺寸的独享；
137. `out_of_range` 逻辑错误，参数超出合法值的范围；

具体参考网页：https://en.cppreference.com/w/cpp/error/exception

标准库中的异常只有少数操作：创建、复制、赋值，其中 `exception` `bad_alloc` `bad_cast` 只有默认构造函数。其它类型的异常只有一个接收字符串的构造函数用来说明异常信息，这些异常信息可以通过调用异常的 `what()` 成员函数来取得。

函数（function）是具名的计算单元（named compute unit），在 C 和 C++ 中与子程序（subroutine）是不加区别的，指的就是执行特定任务的程序指令，它们被打包成一个单元。函数可以用在此任务需要执行的任何地方。
函数可以定义在程序中或者分离定义在库（library）中（定义在库中的代码可以被多个程序复用）。在不同的编程语言中，它具有不同的名字 —— 过程（procedure）、函数（function）、子程序（routine）、方法（method）、子程序（subprogram），以及更加通用的术语**调用单元**（callable unit）。
函数可以在程序的一次执行中从多个不同的地方以及多个不同的时间被调用（call），当计算完成时又返回到调用的下一条指令。Maurice Wilkes 等人称之为闭合子程序（closed subroutine）以区别于宏（macro）或者叫开放子程序（open subroutine），它们之间的区别在于函数有一个单独的作用域，调用时将实参进行求值然后初始化这个独立计算区域中的形参。而宏则是执行文本替换，实际上不存在调用的概念，将实参表达式替换每一个对应的形参，因而也会求值多次。只有闭合子程序才能支持递归调用。
函数的好处在于提高代码的密度（code density），复用代码可以有效的降低开发和维护大程序的成本，同时提高代码的质量和可靠性。面向对象编程（object-oriented programming）是建立在将函数与数据拼合在一起的原则上的。
从编译器的角度来看，程序就是一系列的函数的调用。
函数有两种功用，其一是数学函数（mathematical functions）的概念，就是纯计算，计算的结果完全由输入的参数决定，如：计算对数。其二是调用产生副作用（side effect）如：修改数据结构或者读写 I/O 设备，创建文件等。相同的参数在不同时期的调用结果很可能是不一样的，具体则依赖于当时的程序状态。
函数可以递归的调用的特性，使得可以直接将数学归纳（mathematical induction）和分治算法（divide and conquer algorithms）实现为程序代码。

6.1 函数基础

函数由返回类型、名字和形参（parameters）列表以及函数体构成。参数列表可能有一个或多个参数，也可能没有参数，参数之间用逗号分隔。函数的指令放在一个语句块中，被称为函数体（function body）。
执行函数用调用操作符（call operator），调用操作符是一对括号，调用操作符以函数或者函数指针为操作数，在括号则指定逗号分隔的实参（arguments），实参被用于初始化形参，初始化的过程与变量的初始化是一样的。调用表达式的类型是返回类型。
函数调用时将形参初始化为实参，并且将控制权转移到被调用的函数，此时调用函数（calling function）的执行被暂停，被调用函数（called function）开始执行。被调用函数执行的第一步是隐式定义并初始化参数。然后执行函数体。最后当函数遇到 return 语句或者执行到函数体尾部时自动返回控制权到调用函数，并且返回 return 语句中的值，这个值被用于初始化调用表达式的结果。

6.1.1 形参和实参

实参用于初始化形参，C++ 是按照参数顺序进行初始化的，并且不保证实参的求值顺序的。传入的参数必须与形参的类型匹配，并且个数也是必须一致的，因而，所有形参都保证一定会初始化。与初始化一样，允许从参数到形参的转换，只要这种转换是合法的。
C++ 中可以用一对空括号表示来表示没有参数，C 中早年必须在括号中写 void 来表示没有参数。如

```
void f() {/*...*/}
void f(void) {/*...*/}
```

形参列表就是一系列用逗号分隔的变量声明，变量声明之间是独立的，因而，即便是类型相同的参数也需要重复书写类型名。没有形参的名字是一样的，函数体内的顶层作用域中的本地变量也不会与形参同名。这与同一作用域中不允许同名变量是一样的。
偶尔函数的参数不使用，这种参数可以不命名，即便此参数不命名，当调用时还是必须传入对应的参数。

6.1.2 返回类型

返回类型可以是除数组类型和函数类型外的任何类型，可以是 void 类型表示没有没有返回值。虽然不能返回数组和函数，但是可以返回数组的指针和引用，以及函数指针。

6.1.3 本地变量

C++ 中的名字有作用域和生命周期（lifetimes）。作用域是程序文本中哪些位置可以看到此名字。生命周期是对象在程序运行期间存活的时期。函数体是一个语句块，因而生成一个新的作用域。形参和在此作用域中定义的变量被称为本地变量（local variables），本地变量会隐藏外部定义的变量。定义在任何函数外的变量在程序的整个执行过程中都存在，这种变量当程序启动时创建，直到程序中止时销毁。

函数中有一种本地变量是自动对象（automatic objects），这种变量当函数控制通过此变量定义时创建，当离开定义变量的语句块时销毁。当离开语句块时，其中定义的自动对象值是未定义的。参数是自动对象，当函数开始时将分配内存给参数，当函数结束时销毁。自动对象中的函数参数被初始化为函数实参，自动对象中的本地变量被初始化为定义时给出的初始值，或者被默认初始化，意味着未初始化的内置类型对象具有未定义值。

将本地变量定义为 static 将得到一个本地静态对象（local static object），本地静态对象将在第一次遇到对象的定义时初始化，当函数结束时对象不会被销毁，只有当程序结束时才会销毁。在多次调用函数之间，变量的值沿用上次调用时的最终值。当本地静态对象没有显式初始化时，将执行值初始化，意味着内置类型的本地静态变量将被初始化为 0。

6.1.4 函数声明

函数在使用前必须先声明，这样函数才会可见。函数与变量一样只能定义一次，但是可以声明多次。一个函数可以在没有定义的情况下声明，但想要调用此函数必须得有定义才行。函数声明与函数定义一样除了没有函数体之外。函数声明可以省略参数的名字只留下类型。函数声明分为三个部分：返回类型、函数名字和参数类型，这是调用函数需要知道的全部信息，这个三个元素合起来称为函数原型（function prototype）。

函数声明最好是放在头文件中，以保证所有引入的地方都保持一致，改变声明也仅需改变一个地方。实现函数的源文件也需要包含函数声明，这样可以校验定义和声明是否一致。

6.1.5 分离编译（Separate Compilation）

随着程序不断变大，以及为了支持库的开发。C++ 提供了类似于 C 语言一样的分离编译机制。将程序分为逻辑上的多个部分并且允许将这些部分放到不同的文件中去单独编译。编译出来的文件为对象文件（obj file），当最终需要可执行文件时再链接（link）成可执行文件。由于支持单独编译，因而，当只改动了一个文件时，可以对那一个文件进行编译然后链接在一起形成可执行文件。

6.2 参数传递

每次调用函数时都会创建形参并且用实参值进行初始化。对形参的初始化与对普通变量初始化是一样的。如果形参是引用，那么形参就会被绑定到实参上，这种方式的传递称为按引用传递（passed by reference），此函数调用称为按引用调用（called by reference），因而引用形参就是实参的别名。形参不是引用时就会将实参拷贝到形参中，此时形参和实参是相互独立的对象，这种方式的传递称为按值传递（passed by value），此函数称为按值调用（called by value）。

6.2.1 按值传递

当按值传递时，对形参做出的任何改变都不会影响到实参造成任何影响。以指针为形参需要记住一点就是指针间接访问对象，所以可以通过指针来改变对象值，在直观上与按值传递是不一致的。C 程序员通常用指针来访问调用函数中的对象，C++ 程序员则更多使用引用来访问。

6.2.2 按引用传递

按引用传递的一个作用就是使得函数可以改变其实参。而且很多对象拷贝很耗时，甚至有对象是不允许拷贝的。如：I/O 类型对象就不允许拷贝。C++ 的原则是对于不想改变值的对象使用 const 引用作为形参。

C++ 的函数只能返回一个值，当需要返回多个值时，引用使得可以有效返回多个结果。

6.2.3 const 形参和实参

与变量初始化一样。当用形参去初始化实参时，顶层 const 将被忽略，所以形参的顶层 const 不影响怎样传递实参，我们可以传递 const 和非 const 对象给有顶层 const 的形参。如果重载的函数只是形参的顶层 const 修饰不一样，可以用相同的参数去调用这两个函数，编译器没有足够的信息来区分这两个函数。这种情况下将被认为是重复定义。如：

```
void fcn(const int i) { /* ... */}
void fcn(int i) { /* ... */ } //error: 重复定义 fcn
```

对于指针和引用的底层 const 修饰，与变量初始化一样，可以将非 const 对象用于初始化 const 对象，但不能执行相反的过程。对于非 const 引用，用于初始化的实参必须是相同类型的。所以这里有一个原则就是尽可能使用 const 引用，原因是非 const 引用导致函数只能接收类型精确匹配的左值对象，任何 const 对象、字面量或者需要转型的对象都被排除在外。除非是真正想要改变引用对应的实参值才会使用非 const 引用。

6.2.4 数组形参

数组是比较特殊的类型，原因在于数组不能够被复制，作为参数传递时数组会自动转换为指向头元素的指针。事实上即便是将形参形式写成数组的格式，接收的参数依然是指针。如：

```
void print(const int*)
void print(const int[]);
void print(const int[10]);
```

以上三个函数是完全一样的，参数都是 const int*，注意最后一个带了函数的大小，但是这个值会被编译器完全忽略掉。事实上，如果给上面几个函数进行定义，编译器会认为是重复定义。而且用 sizeof 对这几个参数进行求大小得到的结果是指针的大小而不是数组的大小。程序员必须自己保证传入的数组不会越界，这需要接口实现者和调用者协调保证，语言本身不会提供任何保护机制，如果真的发生了数组越界结果将是未定义的。因而，这种函数常常会有第二参数来传入数组的长度。

有三种方式来定义数组的边界：其一是设置一个结束的标记，如：C 风格字符串末尾的空字符。其二是传入头指针和尾后指针。其三是 C 语言和旧式 C++ 程序最常用的方式：传入额外的 size 参数。

与引用参数一样，除非是希望改写数组中的元素，应该将参数定义为指向 const 对象的指针。

除了以上传入数组首元素指针的形式，可以定义数组引用的形参。此时，数组的大小将称为类型的一部分，因而函数内可以依赖于此大小。然而由于函数接口定死了数组大小也限制了数组的运用范围。如：

```
void print(int (&arr)[10]);
```

以上函数必须将 & 号放在括号内部。

在第二章中介绍过多维数组，如果以多维数组作为参数传给函数，那么其实传的是指向二级数组的指针，并且子数组的长度是类型的一部分，因而必须指定其长度。如：

```
void print(int(*matrix)[10], int rowSize) { /*...*/ }
```

以上部分中 * 号必须放在括号内部，表示指针的优先级高于数组。并且与以下定义完全一样：

```
void print(int matrix[][10], int rowSize) { /*...*/ }
```

数组形式的参数与指针形式的参数是完全一致的。

6.2.5 main：处理命令行参数

C++ 程序允许从命令行传递选项到程序中去。方式是定义 main 函数是指定两个额外的参数。以下为其形式：

```
int main(int argc, char *argv[]) { /*...*/ }
```

其中第一个参数是选项的个数，第二个参数是指针数组，其中每个指针指向一个表示选项的 C 风格字符串。以下函数定义是完全一样的，将 argv 定义指向 char* 的指针。

```
int main(int argc, char **argv) { /*...*/ }
```

main 函数中 argv 指向的第一个元素是函数的名字或者空字符串，真正的选项是从 1 号位开始的。尾后元素被保证是空指针。如：

```
//prog -d -o ofile data0
argv[0] = "prog";
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```

6.2.6 不定形参

当不确定参数有多少个以及这些参数是什么类型时，需要用到不定形参。其实除了参数不一样外，行为都是一样的，重载多个函数完全不现实。C++ 中逐渐遗弃了 C 风格的省略号形参（Ellipsis Parameters），转而使用两种类型校验更强的方式。如果参数的类型是一样的，那么可以使用类 initializer_list，如果类型都不一样那么使用可变参数模板（variadic template）。而省略号形式的不定形参被建议仅用于提供给 C 函数的接口。

initializer_list 是一个类模板，表示某种类型的数组，被定义在头文件 initializer_list 头文件中。以下简单列举此类对象可以执行的操作：

138. initializer_list<T> lst; 包含元素类型 T 的空列表；
139. initializer_list<T> lst{a,b,c...}; 将初始化列表中的值拷贝到列表中，最终列表中的值是 const 的；

140. lst2(lst) lst2 = lst 这种形式的初始化或赋值不会拷贝元素，而是共享相同的元素；
141. lst.size() 返回列表的元素个数；
142. lst.begin() lst.end() 返回头元素和尾后元素的迭代器；

需要记住的是 initializer_list 中的元素总是 const 的，不能改变其中元素的值。当传递给函数 initializer_list 参数时需要将序列放在大括弧中。如：

```
void error_msg(initializer_list<string> il);
error_msg({"functionX", "okay", "expected", "actual"});
```

省略号形如：void foo(parm_list, ...); ，其中省略号只能出现在参数列表的末尾。其中 parm_list 后的逗号可以省略，但最好不要这样做以免引起歧义。有几个函数来帮助访问省略号形式的可变参数（variadic arguments）。

143. va_start 使得可以开始访问可变参数；
144. va_arg 访问下一个可变参数；
145. va_list 保存供 va_start va_arg va_end 访问的信息，必须首先调用；
146. va_end 结束访问可变参数；

```
#include <iostream>
#include <cstdarg>

//其 fmt 后省略了逗号
void simple_printf(const char* fmt...)
{
    va_list args;
    va_start(args, fmt);

    while (*fmt != '\0') {
        if (*fmt == 'd') {
            int i = va_arg(args, int);
            std::cout << i << '\n';
        } else if (*fmt == 'c') {
            // note automatic conversion to integral type
            int c = va_arg(args, int);
            std::cout << static_cast<char>(c) << '\n';
        } else if (*fmt == 'f') {
            double d = va_arg(args, double);
            std::cout << d << '\n';
        }
        ++fmt;
    }

    va_end(args);
}

int main()
{
    simple_printf("dcff", 3, 'a', 1.999, 42.5);
}
```

在 C++ 中省略号形式的参数中如果有类类型，很可能不能正确的进行拷贝。并且省略号形式的参数是不做类型检查的。在 C 语言中要求在省略号前必须要有一个具名形参（named parameter），C++ 中如果没有具名参数虽然是合法的，但是无法访问可变参数。

6.3 返回类型和 return 语句

return 语句将终止函数的执行并返回控制权到函数调用点，将立即执行调用点之后的代码，这个地方叫做返回地址（return address），返回地址在执行调用时被保存在调用栈上。C++ 语言 return 语句返回一个返回值（return value）给调用者，而一些语言则没有提供返回值的语言特性，它们转而提供了出参数（output parameter），还有另外一些语言则默认函数的最后一条语句是函数的返回值，如：scala 。

当函数执行到最后一条语句时会自动返回。曾经有争议是否应当在函数的中间返回（称为早期退出 “early exit”），支持的观点有证据显示从中间返回使得程序员书写起来更不易出错，也更容易理解。而反对观点则认为中间返回将导致资源得不到有效释放，而从函数的底部退出就不会跳过释放的代码。C++ 通过在栈展开（stack unwinding）时由对象释放的析构函数自动调用进行资源释放，这种方式也被称为 RAII（resource acquisition is initialization）资源获取即初始化，很多人认为这是一个非常糟糕的术语用于描述对 C++ 来说几乎是最重要的概念，有人甚至认为应当用 DIRR （Destruction is Resource Relinquishment）析构即资源释放，或者叫 SBRM（Scope Bound Resource Management）局部绑定资源管理。

有两种形式进行函数返回：

```
return;
return expression;
```

6.3.1 函数没有返回值

如以下例子：

```
void swap(int &v1, int &v2)
{
    if (v1 == v2)
        return;
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

没有返回值的函数的返回类型被定义为 void，其可以在函数中间用 `return;` 语句返回，这个 return 后是不带返回值的。如果返回了任何值便是编译错误，对于有返回值的函数也是一样的，如果返回的类型不一致并且无法转换便是编译错误。

如果没有返回语句，函数将在最后一条语句隐式返回。

void 返回类型函数可以用第二种形式的返回，但需要保证 expression 是对一个返回 void 类型的函数的调用，返回其它类型的表达式是编译时错误。

6.3.2 函数有返回值

当函数有返回值时只能用第二种形式的 return 语句，并且返回的值的类型必须与函数声明的返回值类型一致或者可以隐式转换过去，编译器会强制要求这一点。与 Java 不同的是声明为具有返回值的函数底部可以没有 return 语句，没有给出 return 语句是编译器不保证检查的错误，并且结果是未定义的，通常会导致程序异常退出。

返回值与变量和参数初始化方式是一样的。返回的值被用来初始化一个调用函数中的临时量（temporary），这个临时量就是被调用函数返回的结果。因此，如果返回不是引用，那么返回的值是什么对象都无关紧要，编译器会负责正确地进行拷贝。而如果返回的是引用，就必须遵守引用初始化的规则。

考虑以下函数：

```
const string& shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

这里无论是调用时还是返回时都是 `const string &`，避免了复制对象的消耗，结果总是实参的一个别名。这里需要注意的一点是 s1 和 s2 不能是 C 风格字符串，原因在于编译器会自动将其转为 string 类型的临时量，而当函数调用完毕时这些临时量会被销毁，导致的结果就是返回的引用其实绑定到了一个不存在的对象上，其行为是未定义的。

记住：千万不要返回本地对象的指针和引用。当函数执行完时，为函数分配的任何本地变量内存都会被销毁。如果使用这样的值结果将是未定义的。

返回值如果是类类型，可以继续调用结果值的成员函数。如：`shorterString(s1, s2).size()` 。

C++ 的函数是否为左值取决于返回值是否为非 const 引用，返回非 const 引用则结果是左值，返回其它形式的值则是右值。返回引用的函数调用可以像别的左值一样操作，特别是可以给结果值赋予别的值。如：

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix];
```

```
}
string s("a value");
get_val(s, 0) = 'A';
```

这也许会令人惊奇，但 C++ 中是允许这种操作的，而且是完全符合逻辑的。

在新标准中，函数可以返回置于括弧中的一列值，这个过程与列初始化是完全一样的。列表中的值被用于初始化函数的返回值，如果列表为空则返回值被值初始化。如果返回值是内置类型，那么括弧中最多只能有一个值，并且值不能执行精度变小的转换（narrowing conversion），如果返回的是类类型，则有类自己定义如何进行初始化。

从 main 函数中返回有一个例外：允许 main 函数不返回任何值，此时编译器会隐式返回 0 。main 的返回值被 Unix 认为是程序状态，任何非 0 值都被认为是发生了错误。

6.3.3 函数返回指向数组的指针

C++ 中不允许复制数组，函数因而不能声明为返回数组，但可以返回一个指向数组的指针或者数组引用。然而定义返回数组指针或者数组引用的函数有点难懂，因为引用和指针的优先级低于数组索引符。如：

```
int (*func(int i))[10]; //返回指向 int[10] 的指针
int (&func(int i))[10]; //返回 int[10] 的引用
```

为了改善这种难懂的声明式，C 语言提供了 typedef ，C++ 提供了新的 using 声明。如：

```
typedef int arrT[10];
using arrT = int[10];
arrT* func(int i);
```

新标准可以使用尾部返回类型（trailing return type）来简化函数声明，尾部返回类型的格式是原来的返回类型地方用 auto 占位，真正的类型放在参数列表后的 -> 之后，这种方式最常用的地方就是像返回数组指针或数组引用的复杂声明。如：

```
auto func(int i) -> int(*)[10];
```

另外一种做法是用 decltype 来推断返回类型，如：

```
int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
decltype(odd)* arrPtr(int i);
```

前面说过 decltype 不会将数组自动转型为指向头元素的指针，因而，返回的就是数组本身的类型，而且数组的长度是类型的一部分。所以，声明时需要加上 * 来说明返回的是数组的指针。

6.4 函数重载（Overloaded Functions）

C 语言是不支持函数重载的，即不允许具有同名函数。C++ 允许具有不同参数列表的函数有相同的名字，当它们出现在同一个作用域时就是重载（overloaded）。重载函数在不同的参数上执行类似的任务，编译器通过我们传递的实参来推断调用哪个函数。函数重载使得不再需要发明并记住更多名字，但只应该在重载函数执行的任务类似时才重载。需要记住的是 main 函数不能够被重载。

重载的函数必须在参数的个数和类型上有所区别，如果函数仅仅是返回值类型不一致将不符合重载的条件。如果一个函数仅仅是某些参数的顶层 const 不一样也不符合重载条件。如：

```
Record lookup(Phone phone);
Record lookup(const Phone phone);
```

以上两个函数是一样的，因而，是重复定义。

另一方面可以定义底层 const 不一致的重载，即指针是否指向 const 对象，引用是否绑定到 const 对象。如：

```
Record lookup(Account& account);
Record lookup(const Account& account);
Record lookup(Account* account);
Record lookup(const Account* account);
```

由于不能从 const 对象转为非 const 对象，而可以从非 const 对象转为 const 对象。const 对象只能用于调用 const 版本的函数，而非 const 对象可以用于调用这两个版本的函数。如果同时存在两个版本，当使用非 const 对象进行调用时，编译器会调用非 const 版本的函数。

调用重载函数

调用重载函数时编译器会进行函数匹配（function matching）或者叫重载解析（overload resolution），通过比较实参和形参来决定调用哪个重载的函数。在绝大部分时候函数匹配是简单而直接的，因为大部分重

载函数的参数数量不同或者参数类型不相关。真正困难的地方在参数数量一样而且类型相关的时候。6.6 节将描述怎样进行不同重载函数的参数类型可以转换的函数匹配，这里给出三个函数匹配可能的结果：1. 当存在一个最优匹配时选用这个最优匹配 ;2. 当没有重载的函数可以匹配函数调用时，产生编译时错误 ;3. 当有多于一个重载匹配函数调用，并且没有一个是优于其它的，是模糊调用（ambiguous call），产生编译时错误；

6.4.1 重载和作用域

函数重载只存在于同一个作用域内，不同的作用域不存在重载，如果在内嵌作用域中定义了相同的名字会隐藏外部作用域中的相同名字。重载不能跨作用域。这个道理可以运用到子类隐藏基类中的相同函数名字。如：

```
void print(const string&);
void fooBar(int ival)
{
    string s = read();
    void print(int); //将隐藏外部的 print 函数，不推荐在函数内声明函数
    print("Value: "); //编译时错误，此时外部函数已经被隐藏了
}
```

在 C++ 中名称查找发生在类型匹配之前，被隐藏的名字不在函数调用的考虑范围内。

6.5 C++ 函数特殊特性

C++ 有三种函数相关的特性，这些特性并不需要运用于所有函数，而仅当需要时使用。它们分别是默认实参（default argument）、内联函数和 constexpr 函数，以及在程序调试过程中常用的一些功能。

6.5.1 默认实参（Default Arguments）

默认实参用于当函数参数在绝大多数时候值都是某个特定值时，当调用函数时可以不提供这个参数，函数参数进行初始化时就使用声明时给出的默认参数，当调用者也可以明确给出此参数的值。默认实参在形参列表中以初始值的形式给出，可以给一个或多个形参给出默认实参，如果某个参数被赋予了默认值，其后的所有参数都必须给出默认值。如：

```
string screen(size_t ht = 24, size_t wid = 80, char backgrnd = '*');
```

调用函数时可以省略带有默认实参的参数，需要注意的是如果省略了某个参数，其后所有参数也必须省略。如果为某个已经有默认实参的参数提供值，其前面的所有参数都必须给出值，原因在于 C++ 的函数只支持按位置初始化参数，而 Python 可以按形参名字给出对应实参。如：

```
string window;
window = screen(); //screen(24, 80, '*');
window = screen(66); //screen(66, 80, '*');
window = screen(24, 80, '#'); //为覆盖最后的 backgrnd 参数，前面的参数必须给出
```

设计带有默认实参的函数的一个原则是将最不可能改变的参数放在最右边。

虽然函数可以声明多次，但对带有默认实参的函数有限制：同一作用域内的多次函数声明，只能提供一次默认值，所有的默认值将叠加起来形成最终的函数声明，即便是相同的多次给出默认值都是错误。如：

```
string screen(sz, sz, char = ' ');
string screen(sz, sz, char = ' '); //声明最后一个参数的默认值多次，即便是相同也是错误的
string screen(sz, sz, char = '*'); //改变更加是错误的
string screen(sz=24, sz=80, char); //增加是可以的，最终将是 string screen(sz=24, sz=80, char=' ');
```

通常是将默认实参放在函数声明中，虽然放在函数定义的参数列表中也是可以的，但并不常用，说到底其实提供默认值的时候看的依然是当时可见的函数声明，函数定义的头部也是一种函数声明。

用于初始化默认实参的值除了可以是常量表达式之外，还可以是全局的变量、函数返回值，但不能是本地变量。作为函数默认实参的表达式中的名字将在函数声明时进行名字解析，而求值发生在函数调用时。另外，如果是内嵌的相同函数声明，将隐藏外部作用的函数声明。具体查看：

https://github.com/chuenlungwang/cppprimer-note/blob/master/code/default_arguments.cpp

6.5.2 内联函数和 constexpr 函数

内联函数是 C++ 为了改进 C 中类函数宏所作出的努力，内联函数具有闭合子程序的所有特性，然而编译器会将函数调用过程展开为执行代码。优雅地融合了函数和宏的特性。好处在于：更加容易阅读和理

解程序意图、行为统一、易于修改和重用，展开执行代码将减轻了函数调用的额外工作：保存寄存器并恢复、赋值参数、程序跳转。由于内联函数将在每一处调用时进行展开，所以，要求要函数将是非常小的，不然生成的执行文件将很大，如果函数是递归的更是不应该内联。而且由于内联展开发生在编译期间，所以，应当将内联函数代码放在头文件中，才能在编译其它文件时进行展开，原因是编译器必须看到执行的代码才能展开。内联函数通常是给编译器的一种提示，编译器会根据实际情况内联或者忽略。如：

```
inline const string&
shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

通常来说，内联函数都是非常小的，常常只有一个表达式。

constexpr 函数是通常是非常简单的函数，当接收常量表达式作为参数时，返回的值必定是常量值，其函数体所执行的操作亦必须是编译器可以在编译期间执行的操作。这样的情况下，constexpr 函数就可以作为常量表达式（constant expression）用于需要常量表达式的场景中。因而，constexpr 函数需要满足以下限制：返回类型和参数类型都必须是字面类型（literal type）（参考第二章内容），函数体内必须只包含一个 return 语句，不能包含任何别的语句，但可以包含 typedef 和 using 声明语句或空语句，只要它们不执行任何运行时的操作。如：

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz();
```

编译可以验证 constexpr 函数返回的值是常量表达式，并且可以用于初始化 constexpr 变量，如果不符合限制，编译器会报错，并不允许编译通过。在符合条件的情况下编译器会将所有的 constexpr 函数调用都替换为求值结果。为了让编译器能够求值成功，编译器必须看到所有的 constexpr 的函数体代码，所以，隐式要求 constexpr 为内联的。

constexpr 函数可以接收字面类型的参数，并对其执行运算，但所有的运算都必须是编译器可以完成的，也就是说一定不可以包含内存分配、对象初始化、IO 等操作，即没有运行时操作。但可以包括类型别名和 using 声明，这些是不涉及运行时操作的。

接收参数的 constexpr 函数当用常量表达式参数进行调用时结果是常量表达式，当用运行时变量调用时结果不是常量表达式。语言是允许这样的操作的，仅当确实需要常量表达式时编译器才会要求 constexpr 函数返回常量表达式。如：

```
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
int arr[scale(2)]; //常量表达式
int i = 2;
int a2[scale(i)]; //错误，scale(i)不是常量表达式
```

6.5.3 辅助 Debugging 的特性

C++ 沿用了 C 中使用 assert 宏来断言某些不可能的状态，当 assert 中表达式求值结果为 false 将导致程序打印错误信息并退出。使用 assert 宏的文件不能再定义名为 assert 的函数、变量，否则将会被认为是 assert 宏，因为，宏替换发生在编译器编译之前。assert 宏的行为只有在没有定义宏 NDEBUG 时才会执行，可以通过在编译时提供编译选项 -D NDEBUG 来禁用掉 assert 宏的执行。通常，我们在开发时定义 assert，在生产上线时关闭 assert。

C++ 编译器定义几个特殊的预处理变量用于辅助调试：

147. __func__ 当前所在函数名；
148. __FILE__ 当前所在文件名；
149. __LINE__ 当前行号；
150. __TIME__ 文件编译时的时间；
151. __DATE__ 文件编译时的日期；

6.6 函数调用匹配

函数匹配（function matching）发生在函数调用时，编译器依据函数调用的参数类型和个数选择哪个重载函数是最优的，通常这个过程是简单而直接的，仅当函数参数的个数一致而且类型又相关时会比较复杂。如：

```
void f(int);
void f(int, int);
void f(double, double=3.14)
f(5.6); //f(double);
f(5); //f(int);
f(5, 5); //f(int, int);
f(5, 5.6); //错误，调用模糊
```

以上究竟调用哪个函数是很难分辨的，甚至有函数调用是错误的，因为，编译器亦不知该调用哪一个函数。

编译器进行函数调用匹配分为三步 :选择候选函数(candidate functions)和决定可行函数(viable functions)，从可行函数中选择最优匹配。候选函数是与调用函数同名的可见重载函数集合。可行函数则是与调用匹配的函数，其参数数目一致并且类型要么精确匹配要么可以进行转换。如果没有可行函数则是调用不匹配错误，如果有多于一个可行函数，而没有一个是绝对优于另外一个的，则是调用模糊（ambiguous）错误。所谓优于意思是没有一个参数匹配是比别的可行函数差的，并且有至少一个参数匹配是优于别的可行函数的。所谓参数匹配的优只的是形参与实参之间的类型更加靠近。

`f(5, 5.6);` 的前一个参数可已经精确匹配 `f(int, int);` 而第二个参数需要转换。对于 `f(double, double);` 则第二个参数精确匹配，但第一个参数需要转换。这里有一个观点：`int -> double` 的转型并不优于 `double -> int` 转换更优。

通常不应该重载 `f(double, double)` 和 `f(int, int)` ，这并不是好的设计。同时定义引用参数和非引用参数也是非常不好的设计，函数调用必定产生调用模糊错误。如：`f(int)` 和 `f(int&)` 。

6.6.1 实参类型转换

编译器为了决定出最佳匹配，对以下转换进行了排序，从上到下依次变差：

152. 精确匹配：包括完全一致，由数组或函数转为对应的指针类型，顶层 const 的增加或去除；
153. const 转换：指向非 const 对象指针转为指向 const 对象指针，非 const 引用转为 const 引用（第四章有描述）；
154. 整型提升；
155. 算数转换或指针转换（第四章有描述）；
156. 类类型转换；

6.7 函数指针

函数指针就是指向函数的指针，函数指针的类型由函数的签名决定：返回值类型和参数列表类型，名字则被忽略。如：

```
bool lengthCompare(const string&, const string&);
```

的类型为

```
bool(const string&, const string&)
```

定义函数指针与定义数组指针类似，都需要用到括号来强制优先级。

```
bool (*pf)(const string&, const string&);
```

如果定义为

```
bool *pf(const string&, const string&);
```

其含义是声明一个函数，返回 bool 的指针。

当需要函数指针时，函数名会自动转为指针。如：

```
pf = lengthCompare;
pf = &lengthCompare; //与以上完全一致
```

可以使用函数指针进行函数调用，如：

```
bool b1 = pf("hello", "goodbye");
bool b2 = (*pf)("hello", "goodbye");
bool b3 = lengthCompare("hello", "goodbye"); //三个函数调用是完全一致的
```

函数指针之间不存在转换，不能令一个类型的函数指针指向别的不同类型的函数，即使是重载函数或者只有参数类型存在转换关系或者仅仅返回值类型不一样也是不可以的。不过，可以将 nullptr 或字面量 0 赋值给函数指针用于表示不指向任何函数。

对于用重载的函数名进行初始化，编译器会依据指针的类型识别是哪一个重载函数，只有完全精确匹配才是正确的函数。

C++ 的函数不能以函数作为参数，但可以将函数指针作为参数，这与数组是一样的。即便将形参写作函数形式，其实质也是一个函数指针。

```
void useBigger(const string &s1, const string &s2, bool pf(const string&, const string&));
void useBigger(const string &s1, const string &s2, bool (*pf)(const string&, const string&));
//以上两个是完全一致的
```

调用时将函数名作为实参，将自动转为函数指针。如：`useBigger(s1, s2, lengthCompare);`

使用类型别名和 decltype 可以简化函数指针的声明。如：

```
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2;  //声明 Func 和 Func2 为函数类型
typedef bool (*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2; //声明 FuncP 和 FuncP2 为函数指针
```

需要注意的是 decltype 并不会将函数类型自动转为指针。按以上声明之后，可以将 Func 和 FuncP 作为函数参数。如：

```
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP);
```

C++ 函数不能返回函数，但可以返回函数指针，这与数组特性一致。并且必须书写为返回函数指针，写成返回函数并不会自动转为返回函数指针。函数返回函数指针的书写格式是难以理解的，如：

```
int (*f1(int))(int*, int);
```

表示一个函数 f1 返回 `int (*)(int*, int)` 函数指针。可以用别名的方式来简化返回类型，如：

```
using F = int(int*, int); //F 是函数类型
using PF = int(*)(int*, int); //PF 是指针类型
PF f1(int);
F f1(int); //错误
F *f1(int);
```

除此之外还可以使用尾置返回类型的方式。如：

```
auto f1(int) -> int(*)(int*, int);
```

还有就是直接使用 decltype 来推断返回类型。如：

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
decltype(sumLength) *getFcn(const string&);
```

这与前面的返回数组指针是一样的，需要了解的是当将 decltype 运用于函数时，返回的是函数类型，并且不会自动转为函数指针。

关键术语

157. 调用模糊（ambiguous call）：一种编译期错误，当函数调用进行匹配时发现两个以上相同优良的匹配所产生的错误；
158. 自动对象（automatic objects）：仅当函数执行时存在的对象，当程序执行到对象定义时创建，当离开其所在块时销毁；
159. 最佳匹配（best match）：从一系列重载函数中选择一个函数，此函数与其它函数对比，至少有一个参数是优于其它函数的，并且没有一个参数是差于别的函数的；
160. 函数原型（function prototype）：函数声明，包括函数名字、返回类型和参数类型。要想调用一个函数，必须在调用之前声明函数原型；
161. 隐藏名字（hidden names）：在内嵌的作用域中声明的名字将隐藏之前在外部作用域中声明的名字；
162. 本地变量（local variables）：在块中定义的变量；
163. 对象声明周期（object lifetime）：每个对象都有自己的声明周期。自动对象从定义的位置创建，在退出所在块时被销毁。全局对象在函数执行即存在，一直存活到程序结束。本地 static 变量将在第一次执行定义时创建，并与全局变量一样一直存活到程序结束；

164. 递归循环（recursion looop）：描述缺少终止条件的递归函数，如果调用这种函数将最终耗尽程序的调用栈；

C++ 中最重要的特性之一就是类，通过类可以有效表达代解决的问题中的概念。类讲求数据抽象（data abstraction），即是将类的实现与类所能表达的功能分离开来。类背后的哲学就是数据抽象和封装（encapsulation）。

数据抽象意在将不必要的细节剥离出来以期更加接近问题的本质特性。数据抽象往往是设计系统的第一步，原因在于完整的系统非常复杂，几乎不能在没有一个简单抽象的情况下一步完成。数据抽象使得可以从最本质的特性开始然后一步步丰富到最终的系统。有一些别的术语与此相近，比如：建模（modeling）、泛化（generalization）。抽象是设计一些软件的开端，亦是最重要的武器。

封装则是两个相关但不同概念的组合：将对象状态隐藏使得外部无法直接访问；将数据和方法打包到一起；在一些语言中隐藏组件不是自动发生的，甚至不可能隐藏，因而，在这些语言中只有第二种含义。而第一种含义有另外一个术语表示：信息隐藏（information hiding）。

类使用数据抽象和封装来定义抽象数据类型（abstract data type，ADT），抽象数据类型使得可以以抽象的方式来思考，即仅需要知道类可以做什么，而不必深究类是怎么做到的，这些实现的细节留给设计者去考虑。

7.1 定义抽象数据类型

只有具有行为或者说接口的类型才能称为数据类型，如果只有数据而没有行为，所有行为都需要用户自行编写，那便不能称为抽象的数据类型。“行为”的含义了抽象机器的公理语义（axiomatic semantics）和操作语义（operational semantics），有些语境下还会包含计算复杂度（computational complexity）包括时间复杂度和空间复杂度。现实中的很多数据类型不是完美的抽象数据类型，原因在于机器现实，如：算数溢出等。抽象数据类型是计算机科学中的理论概念，不对应于任何特定的语言特性。这个概念被运用于设计和分析算法，数据结构和软件系统。许多概念类似于抽象数据类型：抽象类型（abstract types）、透传数据类型（opaque data types）、协议（protocols）和按约定设计（design by contract）。

https://en.wikipedia.org/wiki/Abstract_data_type

7.1.1 设计类

设计类时需要分清楚两种不同的编程角色：设计者和用户。类的设计者负责设计接口和实现。用户则负责使用这些接口完成别的任务。不论在任何时候都需要清楚分离设计者和用户角色，这样能够很好的解耦从而使得程序演变更加容易。在一些简单的应用中，常常类的用户和设计者时同一个人，作为设计者应该努力思考让类更加容易使用，从而使得用户必须了解类的实现就能够很好使用类。设计良好的类通常接口时直观且易用的，并且实现地足够高效。

7.1.2 定义类

类中有一些函数是实现的一部分，这些函数不会作为接口的一部分，这些函数需要被定义为私有的。C++ 中会包含一些作为接口的非成员函数，这些函数将定义在类的外部。定义和声明成员函数（member functions）于普通函数类型，成员函数必须在类中声明，定义在类内或者类外。定义在类内的函数隐式称为内联函数。

示例代码：use_sales_data.cc

当调用一个对象的成员函数时，所有操作都作用在此对象上。如：

```
string isbn() const { return bookNo; }
```

当调用 `total.isbn()` 时，返回的是 `total.bookNo`，成员函数通过一个额外的隐式参数 this 来访问调用对象。this 被默认初始化为函数调用对象的地址。此处 this 是 total 对象的地址。就如以下代码：

```
isbn(&total)
```

在成员函数内可以直接使用成员名字而不需要使用成员访问符去访问，这种直接使用成员名字会被编译器隐式转换为 `this->bookNo` 形式。

this 参数由编译器隐式定义，并且不允许程序员定义参数或者变量名为 this，this 总是指向当前调用的对象，所以，this 是一个 const 指针，我们不能改变 this 使其指向别的对象。

参数列表后的 const 用来说明 this 指针指向 const 对象。通常情况下 this 是指向非 const 对象的 const 指针，尽管 this 是隐式定义的，它也遵循初始化原则，即不能将非 const 的 this 指向 const 对象，从而不能在 const 对象上调用非 const 的函数。为了在 const 对象上调用成员函数，必须使得 this 指向 const 对象，如果 this 指针可以在参数列表中，那么它将形如：`const Sales_data *const`，然而 this

是隐式定义的并且不可能出现在参数列表中，所以语言通过在参数列表后放置 const 来表明 this 指针指向 const 对象，从而使得此成员函数称为 const 成员函数（const member function）。isbn 函数可以描述为：

```
string Sales_data::isbn(const Sales_data *const this)
{ return this->isbn; }
```

this 指针指向 const 对象，说明 const 成员函数不能改变调用对象的状态。const 对象、指向 const 对象的指针和绑定 const 对象的引用只能调用 const 成员函数。

类作用域和成员函数

类本身就是一个作用域，所以，成员函数的定义处于类作用域中，因而，当 isbn 不加任何限定符使用 bookNo 时，将被解析为使用 Sales_data 中的 isbn。值得注意的是这种使用甚至可以在 bookNo 的定义之前。编译器处理类是分两步走的：成员定义先处理，然后才是成员函数体。因此，成员函数可以使用其它成员而不必纠结它们出现的顺序。

在类外部定义成员函数

与任何别的函数一样，当在类外部定义成员函数时需要定义与声明完全一致。特别是参数列表后的 const 需要一致。除此之外，在类外定义的成员名字必须用类名加以限定。如：

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

Sales_data::avg_price 的含义就是说函数名字 avg_price 是声明在作用域 Sales_data 类中的。一旦编译器看到函数名字，剩下的代码将被解释为处于类作用域中。因而，revenue，units_sold 都是隐式解释为 Sales_data 的成员。

定义函数返回“当前”对象

成员函数返回当前调用对象的关键在于设置返回类型和书写返回语句。返回当前对象意味着需要返回当前对象的引用，而且需要在返回时对 this 进行解引用。如：

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

调用 `total.combine(trans);` 将返回 total 对象的引用。

7.1.3 定义类相关的非成员函数

类作者可以定义一些辅助函数，这些函数定义的操作在概念上属于类接口的一部分，但是本身不在类内。这些函数通常与类定义在同一个头文件中，使得用户只需要包含一个文件就可能够使用与类相关的任何接口部分。这些函数与普通的函数并没有什么区别，它们并不在类的作用域内，因而，没有隐式的 this 指针以及不能直接使用成员，甚至调用时是直接写出函数名，而不是由对象进行调用。

7.1.4 构造函数

每个类都定义该类的对象如何进行初始化，类通过定义一个或者多个特殊成员函数来控制初始化，称为构造函数（constructors）。构造函数的工作就是初始化类的数据成员，每当对象创建时构造函数就会调用。

构造函数和类的初始化是很复杂的，有一个总原则就是初始化所有数据成员，而不是依赖于编译器设置默认值。原因在于，编译器在不同的情况下会不会设置默认值是不一样的，而记住所有不同情况下的行为是很累人的。这些复杂的初始化参考：https://en.cppreference.com/w/cpp/language/initialization 中值初始化（value initialization）、默认初始化（default initialization）和零初始化（zero initialization）。

构造函数与类名相同而且没有返回值，除此之外与常规函数没有别样。构造函数与常规函数一样可以进行重载。构造函数不能被声明为 const 的，原因在于即便是 const 对象亦需要在构造函数完成初始化之后才能变成常量。

合成默认构造函数

当定义对象时不给定初始值将执行默认初始化。类控制默认初始化的构造函数称为默认构造函数（default constructo），这个构造函数是没有任何参数的构造函数。其特殊性在于如果类没有显式定义任何构造函数，编译器会隐式定义一个默认构造函数，这个生成的默认构造函数称为合成默认构造函数（synthesized default constructor）。合成构造函数初始化每个数据成员的方式是：

165. 如果有类内初始值，就用那个初始值进行初始化；

166. 没有，则使用默认初始化，对于类成员将调用其默认构造函数，对于内置类型则不进行任何初始化；

事实上，默认初始化远远比上面提到的两句话要复杂。在 Initialization in C++ is bonkers 一文中做出了详尽的解释，是目前阅读过的最好的一篇关于初始化的文章。如果要更加理解初始化过程中的内容，建议阅读 C++ 标准中相关章节。

一些类不能依赖于合成默认构造函数

只有十分简单的类可以依赖于合成的默认构造函数。最常见的原因在于编译器只在程序员没有为此类定义任何构造函数时才会生成。如果定义了任何构造函数，将必须手动定义默认构造函数，编译器不会再协助生成了。原因在于编译器认为如果程序员在一种情形下控制了对象的初始化，多半会需要在所有情况下控制对象的初始化。

第二个原因是合成的构造函数不能达到预想的效果。内置类型和复合类型（数组、指针）作为类成员，在默认初始化时不做任何事情。所以，内置类型或复合类型的成员都应该在类内初始化或者定义自己的默认构造函数。只有当内置类型或复合类型有类内初始值时才能依赖于合成默认构造函数。

第三个原因是有些类编译器根本无法合成默认构造函数。原因在于，一个类可以有一个没有默认构造函数的类类型成员，那么编译器将无法初始化此成员。

** =default 的含义**

```
struct Sales_data {
    Sales_data() = default;
};
Sales_data::Sales_data() = default;
```

这种形式的默认构造函数执行与编译器合成的默认构造函数一样的功能。通过在参数列表后加上 = default 将要求编译器为我们生成默认构造函数。=default 可以出现在类内的声明处，或者出现在函数定义处。定义在类内则合成的默认构造函数自动称为内联的，定义在类外则是非内联的。这两种形式的差异还在于定义在类外的函数是用户提供（user-provided）的构造函数。用户提供的构造函数在对象是值初始化时，将执行默认初始化。而编译器生成的默认构造函数，在值初始化时，将先执行零初始化再执行默认初始化。

在 C++11 之后才提供类内初始化，在之前的 C++ 版本是不提供的，此时必须使用构造初始值列表（constructor initializer list）来初始化所有成员。

构造初始值列表

```
Sales_data(const string &s, unsigned n, double p):
    bookNo(s), units_sold(n), revenue(p*n) {}
```

在参数列表后的冒号直到函数体的括弧之间的代码称为构造初始值列表（constructor initializer list），它的作用是给对象的成员提供初始值，它是一系列的成员名，在每个成员名后跟随括号或括弧中的初始值，多个成员初始值之间用逗号分隔。

当一个成员从构造初始值列表中省去时，它是默认初始化的，使用合成构造函数的初始化规则。

使用类内初始值的好处在于不必每次都在构造初始值列表中提供值。如果语言版本不支持的话，则需要在每个构造函数中初始化每一个内置类型的成员。

以上构造函数的函数体是空的，这在 C++ 中是很常见的，这个构造函数所做的工作就是给数据成员提供值。

在类外定义构造函数

```
Sales_data::Sales_data(std::istream &is)
{
    read(is, *this);
}
```

定义在类外的构造函数没有返回类型，直接从名字开始，亦无返回语句。与其它成员函数一样，函数名字需要用类名进行限定。尽管此处构造初始值列表是空的，类成员依然在构造函数体执行之前先初始化，此时这些没有出现在构造初始值列表中的成员将被默认初始化或被类内初始值进行初始化。

7.1.5 拷贝、赋值和析构

除了定义类对象如何初始化，类还可以控制对象赋值（copy）、赋值（assign）和销毁（destroy）时的操作。当初始化变量或者传递参数、返回值时将发生对象复制。当使用赋值操作符时发生赋值操作。当离开定义对象所在的块时将销毁对象，当 vector 或数组被销毁时，其中元素亦被销毁。

如果程序员不定义这些操作的函数，编译器将合成这些函数。通常，编译器生成的版本就是将对象的每个成员分别进行拷贝、赋值和销毁。如编译器生成的 Sales_data 的赋值函数如：

```
total = trans;
////////////////
total.bookNo = trans.bookNo;
total.units_sold = trans.units_sold;
total.revenue = trans.revenue;
```

有些类不能依赖于合成版本

尽管编译器可以合成拷贝、赋值和析构函数，但需要记住的是默认版本的行为不一定是符合要求的。特别是当类在类对象本身外部分配了资源时（动态内存、数据库连接、TCP 连接），合成版本通常无法正确工作。通常如果类有分配动态内存，是不能依赖合成的版本。

值得提醒的时，很多需要动态内存管理的类可以使用 string 或 vector 来管理需要内存。使用这些组件的类避免了分配和回收内存的复杂度，而且合成的函数可以在此之上正确工作。当拷贝带有 vector 成员的对象时，vector 类会正确处理其中元素的拷贝和赋值。当销毁时，vector 将被销毁，其中的元素亦被销毁。

直到你知道如何正确定义以上三种函数时，尽量不要在类所占据内存外分配资源。

7.2 访问控制和封装

在 C++ 中使用访问说明符（access specifiers）来强制封装，即信息隐藏：

167. 定义在 public 说明符后的成员程序的所有部分都可以访问的，public 成员是类的接口（interface）；
168. 定义在 private 说明符后的成员只能由本类中其它部分访问，不能被使用类的代码使用，private 成员封装了类的实现；

一个类可以有零个或多个访问说明符，对于一个访问说明符出现的频率并没有限制。每个访问说明符说明接下来的成员访问级别，这些访问级别持续到下一个访问说明符的出现或者类的结尾。

使用 class 或 struct 关键字

使用 class 关键字或者 struct 关键字唯一的区别在于默认的访问级别。对于 struct 关键字而言，定义在第一个访问说明符前的成员是共有的，对于 class 而言则是私有的。即 class 的默认访问说明符是 private，而 struct 的默认访问说明符是 public 。

7.2.1 友元

尽管类相关的非类成员函数在概念上是类的一部分，但是依然需要遵循常规函数的规则，即不能访问类的私有成员。一个类可以通过将另一个类或函数定义为 friend 来使得其可以访问此类的非 public 成员。这样做只需要简单地将类或函数声明放在类内，并在之前加上 friend 关键字即可。如：

```
class Sales_data {
    friend Sales_data add(const Sales_data&, const Sales_data&);
    friend istream &read(istream&, Sales_data&);
};
Sales_data add(const Sales_data&, const Sales_data&);
istream &read(istream&, Sales_data&);
```

友元声明需要放在类定义内，它们可以出现在类的任何地方。友元不是类的成员，因而，不受访问控制的影响。将所有的友元声明集中放在一起，或置于类定义头部或尾部，是一个好的处理方式。

封装的益处

使用封装可以防止被封装的对象状态被破坏；被封装的类的实现可以随着时间的推移进行修改而不需要涉及到用户代码的修改，api 的改变往往会导致用户的怨声载道。将数据定义为 private 的，类的作者可以按照自己的意愿改变其中数据。实现的概念只会影响到类中使用到此部分代码的行为，测试仅限于这些部分。

而改变接口往往会涉及到用户代码的改变，所有依赖于旧接口的代码都会被打破，导致要重写所有这些使用旧接口的代码。
将数据设置为 private 的另一个好处是数据不会被用户无意中改变。如果有一个 bug 破坏了对象的状态，查找 bug 的地方将仅限于实现代码，从而极大简化了维护成本。
尽管封装避免了用户代码的改变，但改变类实现代码依然需要编译所有用到此类的源文件必须重新编译。

声明友元

友元声明只说明了访问权限，它并不是一个通用的函数声明。为了调用此友元函数，依然需要在类外部声明函数本身，如上面所做的那样。为了使友元函数为用户所见，通常在同一个头文件中再次声明这些函数。
许多编译器并不强制要求友元函数必须在类外再次声明，但并不是所有编译器都支持。为了最大的兼容性，最好在类外部再次声明函数。

7.3 其它类特性

本节将介绍类型成员，类类型成员的类内初始化，mutable 数据成员以及内联成员函数，从成员函数中返回 `*this` ，以及更多关于如何定义和使用类类型，以及友元类的知识。

7.3.1 类成员

定义类型成员（Type Member）

类可以使用 typedef 或 using 为类型定义自己的本地名字，由类定义的类型名字可以像其它成员一样定义其访问权限。如：

```
class Screen {
public:
    typedef std::string::size_type pos;
};
```

在 public 部分定义 pos 类型，这样用户代码亦可以通过 `Screen::pos` 使用此名字。通过定义这种类型别名将隐藏类是如何实现的。其实，`std::string::size_type` 亦是一种类型别名，其真实类型将依据不同的平台定义，客户代码只需要知道此类型即可。
另外用 using 声明是一样的效果。如：

```
class Screen {
public:
    using pos = std::string::size_type;
};
```

需要注意的一点是，与常规成员不同，类型别名定义需要出现在任何其被使用之前。因而，类型成员通常出现在类定义的顶部。

将成员函数设为 inline

类经常有一些小的成员函数可以从内联中获益，定义在类定义内的成员函数自动是内联的，这可以运用于构造函数以及常规成员函数。可以显式将成员函数声明为 inline 的，有两种方式：一种是在类定义内的成员函数声明处声明，另一种是在类外面的成员函数定义处声明。同时在两处都声明内联是合法的。书中认为仅在类外的定义处声明内联可以使得类更加易读。
内联的成员函数应该定义在头文件中，这与常规内联函数遵从一致规则。且应当和与其对应的类定义放在同一个头文件中。

重载成员函数

与常规函数一样，成员函数亦是可以重载的。而且函数匹配的规则与常规函数的匹配是一样的。

mutable 数据成员

偶尔 C++ 中的类的数据成员需要即便是在 const 成员函数中亦是可以改变的，这种数据成员需要用 mutable 关键字进行声明。mutable 数据成员永远不是 const 的，即便其在 const 对象中，因而，const 成员函数可以改变 mutable 成员。如：

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr;
};
void Screen::some_member() const
```

```
{
    ++access_ctr;
}
```

类类型数据成员的初始值

在新标准中，除了可以为内置类型数据成员设定类内初始值外，亦可以为类类型数据成员设定初始值。而且，在新标准中最好的设置默认值的方式就是类内初始值。如：

```
class Window_mgr {
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};
```

当初始化类类型成员时，其实是通过提供实参给那个类的构造函数并调用而进行初始化。

类内初始值遵循一种格式规范即：要么用等号（=）号形式，要么用括弧形式进行初始化。除此之外的形式都是不合法的。

7.3.2 返回 *this 的函数

```
inline Screen &Screen::set(pos r, pos col, char ch)
{
    contents[r*width+col] = ch;
    return *this;
}
```

函数返回引用即其结果为左值，意味着返回的是对象本身而不是对象的拷贝。如果将操作串连起来就是连续对此对象进行操作。如：

```
myScreen.move(4, 0).set('#');
```

如果返回的是对象，而不是引用，那么以上操作中的 set 就是对返回的临时量进行操作，而不是 myScree，其形如：

```
Screen temp = myScreen.move(4, 0);
temp.set('#');
```

从 const 成员函数中返回 *this

const 成员函数返回的 *this 是一个 const 引用，因为，this 是一个指向 const 对象的指针。这将导致其不能与返回 *this 的非 const 成员函数串连在一起。

基于 const 进行重载

```
class Screen {
public:
    Screen &display(std::ostream &os);
    const Screen &display(std::ostream &os);
private:
    void do_display(std::ostream &os);
};
```

可以按照一个成员是否是 const 来进行重载，原因在于，this 指针指向的对象是否为 const 可以进行正确函数匹配。非 const 成员函数不能被 const 对象调用，const 对象只能调用 const 成员函数。然而，在非 const 对象上可以调用任意版本，但非 const 版本是更合适的，意味着如何同时存在 const 和非 const 成员函数，非 const 对象将调用非 const 版本，而 const 对象将只能调用 const 版本。

当一个成员函数调用另外一个成员函数时，this 指针被隐式传递。当一个非 const 成员函数调用 const 成员函数时，其 this 指针将被隐式转为指向 const 对象的指针。非 const 成员返回 *this 是常规引用，const 成员返回 *this 则是 const 引用。通过调用对象是否为 const 来决定哪个版本的函数被调用，以及返回什么样的引用。如：

```
Screen myScreen(5, 3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); //调用非 const 版本
blank.display(cout); //调用 const 版本
```

建议：使用私有 utility 函数来提供公共功能

在设计良好的 C++ 程序中有许多类似于 do_display 的小好函数来做真正的函数工作，接口则将其功能指派给这些工具函数。有几点好处：

169. 避免了在超过一个地方书写相同的代码；
170. 当类随着时间推移变得愈加复杂时，其功能也愈加复杂，将代码写在的一处的好处将更加明显；
171. 将代码写在一处对于添加和删除调试代码更加简单；
172. 通过将工具函数定义为 inline 的，消除了调用函数的额外运行时负担；

7.3.3 类类型

C++ 中的类按名字进行区分，每个类就是一个独特的类型。两个不同的类即便有完全一致的代码亦是不同类型。两个类中的成员是完全独立的。

使用类类型有两种方式：直接使用类的名字，或者在类名前面加上 class 或 struct，如：

```
Sales_data item1;
class Sales_data item1;
```

以上两种使用类类型的方式是完全一样的。第二种方式是从 C 中继承过来的。

类声明

可以声明一个类而先不定义它，如：`class Screen;` 这种声明有时成为前向声明（forward declaration），将名字 Screen 引入到程序中，并指示 Screen 是一个类类型。当发生声明出现在定义之前时，类型 Screen 是未完成类型（incomplete type），它仅仅知道 Screen 是一个类类型，但不知道其成员是什么。未完成类型的使用有诸多限制：可以定义这种类型的指针或引用，可以声明使用未完成类型作为参数或返回值的函数。

只有当知道类的定义时，才能书写创建这种类型对象的代码，否则，编译器不知应当如何存储对象。同样，在用指针或者引用访问类的成员时，必须知道类的定义。总之，如果类没有被定义，编译器便不知道类的成员是什么。

除了静态成员可以是未完成类型外，所有的数据成员只能被设定为已经定义了的类类型。只有当类的定义完成时，编译器才知道如何存储其数据成员。一个类只有到达类体的末尾时才算是完成了定义，一个类不能定义其本身类型的数据成员。然而，只要看到类名字，就算是声明了类，因而，可以定义本身类型的指针或引用。

7.3.4 友元再探

一个类可以使得另一个类作为其友元类，或者将指定特定的成员函数作为其友元。另外，一个友元函数可以定义在类的内部，这个函数则隐式成为内联的。

类之间的友元关系

友元类的成员函数可以访问授权类的所有成员，包括所有非公有的成员。需要记住的是友元关系是不可传递的。一个类是另一个类的友元并不意味着这个类自己的友元可以访问那个类，每个类控制着哪些函数或类是自己的友元。如：

```
class Screen {
friend class Window_mgr;
};
```

使成员函数成为友元

除了可以使得整个类作为友元，类还可以使得特定的成员函数作为友元。当指定成员函数作为友元时需要指定是哪一个类的成员。如：

```
class Screen {
friend void Window_mgr::clear(ScreenIndex);
};
```

为了使得一个成员函数作为友元，有一些需要注意的地方：小心的组合程序结构使得定义和声明之间的相互依赖得以合法。如：先得定义 Window_mgr 类，并在类中声明 clear 函数，但不定义 clear 函数体。这是由于 clear 会用到 Screen 的成员，所以需要先定义 Screen，而声明成员函数作为友元又需要先完成那个类的定义。此后就是定义 Screen 类并声明 clear 函数作为友元。最后是定义 clear 函数所做的事，其中会使用到 Screen 的成员。

函数重载与友元

尽管重载的函数使用的是同一个名字，它们依然是不同的函数。因而，声明友元函数时必须显式指定其中想要的函数原型，单单声明一个并不会将整个重载的函数集加入进来。如：

```
class Screen {
friend std::ostream &storeOn(std::ostream&, Screen&);
```

```
friend BitMap& storeOn(BitMap&, Screen&);
};
```

友元声明和作用域

类和非成员函数在被用作声明友元之前不需要提前声明。当名字第一次出现在友元声明中时，这个名字就已经是可见的了。然而友元并不是通用的声明形式。即便是在类内定义了友元函数，亦需要在类外提供一个函数声明，使得函数可见。即便是在授权类的成员函数中调用，这种单独声明亦是必不可少的。如：

```
struct X {
friend void f() { /* function body */ }
    X() { f(); }  //错误!!未声明函数 f
    void g();
    void h();
};
void X::g() { return f(); }  //错误!! f 未声明
void f();
void X::h() { return f(); }  //正确，此时 f 已经被声明了
```

需要理解的是友元声明仅仅影响访问权限，其并不是真正的声明形式。而，有一些编译器并不强制要求这种查找规则，友元函数可以不在类外再单独声明一次。

7.4 类作用域

每个类定义其自己的新作用域。在类作用域外，必须通过对象、引用或指针以成员访问符来访问数据和函数成员，而类型成员（typedef 、using）则通过作用域运算符访问。

定义在类外的成员及其与类作用域的关系

存在类作用域的事实解释了为何在类外定义成员函数时必须同时提供类名和函数名。在类外，其成员的名字将被隐藏。

一旦遇到了类名，剩下的定义部分（包括形参列表和函数体）就在类作用域内，因此，不需要类名进行限定就可以使用其它类成员。如：

```
void Window_mgr::clear(ScreenIndex i)
{ /* 函数定义 */}
```

在遇到类名 Window_mgr 后，引用类的类型成员时不再需要用类名进行限定。

而另一方面，当一个成员函数在类外定义时，由于返回类型出现函数名字前，此时作用域并不在类作用域内，因而，必须加上类的名字进行限定。如：

```
Window_mgr::ScreenIndex
Window_mgr::addScreen(const Screen &s);
```

实质上，仅在类成员函数定义时才能够在类名后进入类作用域，返回值类型并不会使得进入限定其的类名的作用域。

7.4.1 名称查找和类作用域

涉及到定义在类体内的类成员函数时进行名称查找（name lookup）将复杂一点。类定义分为两步：其一，成员的声明被编译；其二，当整个类定义完成时，成员函数定义将被编译；请记得，成员函数定义的编译发生在类的所有成员的声明之后。

由于此，成员函数体可以使用类中的任何名字。特别是，可以使用在成员函数体后才声明的名字。否则，处理成员声明的顺序将是脆弱而复杂的。

类成员声明的名称查找

类定义的两步走仅仅对于类成员函数使用的类体中的名字有效。对于运用于声明中的名字，包括返回值类型和参数列表中的类型则必须在使用前是可见的。如果成员声明使用了类此刻未见的名字，编译器将在类定义的外部作用域寻找此名字。如：

```
typedef double Money;
string bal;
class Account {
public:
    //Money 使用的是外部声明的，bal 则使用的后面声明的。这是两步走的缘故。
    Money balance() { return bal; }
private:
```

```
    Money bal;
};
```

类型名字是特殊的

通常，内部作用域可以重定义外部作用域中的名字，即便那个名字已经被用于内部作用域。然而，在类中如果一个成员使用了外部作用域中的名字，而此名字是一个类型，那么在类中就不能重定义此名字。如：

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }
private:
    typedef double Money; //错误!!不能重定义 Money 类型
    Money bal;
};
```

尽管此重定义类型名字是一种错误，编译器并不一定识别此错误。一些编译会静默的接受这样的代码，即便这样的程序是错的。类型定义应该放在类的头部，这样其它成员就可以看到此类型定义。

成员函数体内的名字查找规则

成员函数体中使用的名字通过如下方式进行查找：

173. 在成员函数内查找声明，只有出现在使用之前的声明才是可见的；
174. 如果未找到，在类定义内查找声明，所有的成员的名字都可见；
175. 如果未找到，继续从定义成员函数定义所在的作用域内向上查找，如果成员函数定义在类中则从类外部作用域查找，如果成员函数定义在类外，则从类外的位置向上查找；

通常定义参数名字与别的成员名字一样是不好的戏关，如果这样的话，函数内部使用的名字将是参数名字而不是成员名字。可以通过加上类名进行限定而访问成员名字，或者使用 this 指针进行访问。然而最好的方式是给参数取一个不同的名字。如：

```
int height;
class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height)
    {
        cursor = width * height; //访问参数而非成员
        cursor = width * this->height; // 用 this 访问
        cursor = width * Screen::height; //用类名进行限定
        cursor = width * ::height; //访问全局作用域中的 height
    }
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};
```

在类作用域之后，于外部作用域中查找

如果外部作用域中的名字被类作用域中相同名字给隐藏了，如：height ，可以通过作用域操作符（scope operator）显式访问全局中的名字。

名字在它们出现在文件中的位置进行解析

当成员定义在类外时，名字查找的第三步会检查函数定义所在的地方，也会继续往上查找穿过类定义的位置。如：

```
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0;
};
Screen::pos verify(Screen::pos);
```

```
void Screen::setHeight(pos var) {
    height = verify(var);
}
```

这里的 verify 是不为类定义 Screen 所见的，因而不能运用于类定义中。但是，类外定义的成员函数 setHeight 却可以看到此名字，函数体内的名字并不是从类体处开始查找的，而是从函数定义处开始查找的。

7.5 构造函数再探

构造函数是任何 C++ 中都至关重要的一部分。

7.5.1 构造初始值列表

当定义变量时通常是马上进行初始化而不是定义了然后再进行赋值。这种区别同样运用于数据成员的初始化和赋值。如果不在构造初始值列表中显式初始化，成员将会被默认初始化，然后才进入构造函数体中。如：

```
Sales_data::Sales_data(const string &s, unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

这个版本的构造函数将先默认初始化数据成员，然后在函数体内进行赋值。如果数据成员是类类型，那么则会先调用其默认构造函数，再调用其重载后的赋值操作符，这可能造成低效或者某些类甚至是错误的。

构造初始值在某些情况下是必需的

初始化和赋值之前的区别并不总是可以忽略。const 成员和引用成员必须被初始化。同样的，没有默认构造函数的类类型必须被初始化。省略了这些构造初始值将是错误，当构造函数体开始执行时，初始化已经完成，初始化 const 成员和引用成员的唯一机会就是在构造初始值中。因而，必须在构造初始值列表中给 const 、引用、没有构造函数的类类型数据成员提供初始值。

数据成员的初始化顺序

构造初始值列表只给成员提供初始值，并不决定初始化的顺序。成员的初始化顺序由它们出现在类定义中顺序决定，先出现先初始化，后出现的后初始化。在构造初始值列表中的顺序并不会影响初始化顺序。

通常初始化顺序无关紧要。然而，如果一个成员的初始化依赖于另一个成员，此时成员的初始化顺序就很重要。如：

```
class X {
    int i;
    int j;
public:
    //错误!! i 将先初始化，然后才是 j
    X(int val):j(val), i(j) {}
};
```

以上代码的结果是 i 将被初始化为未定义值 j ，然后 j 被初始化为值 val 。一些编译器足够聪明可以为这种情况产生一个警告，通过 -Wall 选项可以让 g++ 产生这个警告。将构造初始值的顺序书写的与类定义中的声明的顺序一致是一个好的习惯。并且，如果可能，不要用别的成员来初始化另一个成员，这样可以不必思考成员初始化顺序。如：

```
X(int val): i(val), j(val) {}
```

默认实参和构造函数

构造函数与类的成员函数可以设置默认实参。如：

```
class Sales_data {
public:
    Sales_data(std::string s = ""):
        bookNo(s) { }
};
```

如果定义了带默认实参的构造函数，就不能再定义默认构造函数了，不然编译器就无法进行函数匹配了。即：一个构造函数如果提供了所有默认实参就相当于隐式定义了一个默认构造函数。

7.5.2 代理构造函数

在新标准中，扩展了构造初始值使得可以定义所谓的代理构造函数（delegating constructors），代理构造函数使用本类的另一个构造函数才执行初始化。即将一些或者全部工作“代理”给其它构造函数完成。代理构造函数的构造初始值列表中只有一项就是另外一个构造函数的调用，实参列表必须匹配另外一个构造函数的签名。如：

```
class Sales_data {
public:
    Sales_data(std::string s, unsigned cnt, double price):
        bookNo(s), units_sold(cnt), revenue(cnt*price) {}
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0, 0) {}
};
```

当一个构造函数代理给另外一个构造函数时，那个函数的构造初始值列表和函数体都会先于代理构造函数执行，意味着在代理构造函数体开始执行之前，提供代理那个构造函数体将先执行完。

7.5.3 默认构造函数的角色

当执行默认初始化或值初始化时，默认构造函数将被自动调用。

默认初始化发生在：

176. 当在块作用域中以未初始化的方式定义非静态变量或非静态数组；
177. 且类初始化时使用合成构造函数，而且其包含类类型成员；
178. 且此类类型成员没有显式地在构造初始值列表中初始化；

值初始化发生在：

179. 当数组初始化时提供的值少于数组的长度，将进行值初始化；
180. 定义本地静态变量（local static object）而不提供初始值；
181. 当显式用 `T()` 对类 T 的对象进行初始化，想一下 vector 就是用这种方式对元素进行至初始化的；

值初始化所做的事：如果类有用户提供（user-provided）默认构造函数，则直接调用此构造函数。如果默认构造函数是编译器隐式定义的，那么所有的非静态成员将递归值初始化。这里隐含的意思在于用户提供的默认构造函数如果不对某些内置类型的数据成员进行初始化，那么它将是未定义值。而由编译器隐式定义的默认构造函数，在值初始化时会对整个对象进行零初始化。

建议：如果定义了任何构造函数，好的做法是同时提供一个默认构造函数。并且，构造函数应当初始化所有成员。

使用默认构造函数

以下使用默认构造函数的形式是错误的，如：`Sales_data obj();` 以上定义了一个返回 Sales_data 的函数 obj，我之前就用错过。要正确使用应当写做 `Sales_data obj{};` 或 `Sales_data obj = Sales_data();`，这将会导致 obj 进行值初始化。而 `Sales_data obj` 进行默认初始化。

7.5.4 隐式类类型转换

除了语言定义的内置类型之间的自动转换。程序员可以定义类之间的隐式转换。每个只有一个参数的构造函数都定义了一个从参数到类类型之间的隐式转换。这种构造函数被称为转换构造函数（converting constructors），除此之外，还可以在类中直接定义转换函数，而从此类转为其它类型。如：

```
//Sales_data
string null_book = "9-999-99999-9";
item.combine(null_book);
```

combine 成员函数的参数是 `const Sales_data&`，而 Sales_data 有一个以 string 为参数的构造函数。因而，这种形式的 combine 调用，会以 null_book 创建一个 Sales_data 的临时量，再以此临时量调用 combine 函数。由于，combine 的参数是 const 对象的引用，所以传递一个临时量过去。

只允许一个类类型转换

编译器只允许一次类类型的自动转换。如：`item.combine("9-999-99999-9");` 是不能编译通过的。原因在于，从字符指针到 Sales_data 对象需要经过两步转换。为了进行此调用，必须进行显式的转换。如：

```
item.combine(string("9-999-99999-9"));
item.combine(Sales_data("9-999-99999-9"));
```

类类型自动转换并不总是有用

有时这种类类型的自动转换根本不是程序员的意图，这样做只会增加理解程序的负担，而且产生的错误还可能是难以发现的。

抑制由构造函数定义的隐式类类型转换

通过在构造函数前加上 explicit 关键字，来抑制此构造函数被用于隐式转换。如：

```
class Sales_data {
public:
    explicit Sales_data(std::istream&);
    explicit Sales_data(const std::string&);
};
```

explicit 关键只有对仅有一个参数的构造函数有效，其它形式的构造函数因为并不用于隐式类型转换，所以并不需要指定为 explicit 。explicit 关键只能用于构造函数声明处，不必在类外部函数定义处进行重复。

explicit 构造函数只能用于直接初始化

需要隐式转换的场景之一便是拷贝形式的初始化，explicit 构造函数不能用于这种形式，其必须使用直接初始化。如：

```
Sales_data item1(null_book);
Sales_data item2 = null_book; //explicit 构造函数不能用于拷贝形式的初始化
```

显式使用构造函数进行转换

尽管不能将 explicit 构造函数进行隐式转换，但可以用于显式转换。如：

```
item.combine(Sales_data(null_book)); //直接调用构造函数
item.combine(static_cast<Sales_data>(cin)); //使用 static_cast 进行显式转换
```

标准库中的 string 类中以 `const char*` 作为参数的构造函数是可以进行隐式转换的。而 vector 类以长度为参数的构造函数是 explicit，不允许隐式转换。

7.5.5 聚合类

聚合类（aggregate class）可以直接让用户访问其成员，并且有特殊的初始化语法。聚合类有点类似 POD（Plain Old Data）结构，满足以下条件的类可以被称为是聚合的：

182. 所有的数据成员是 public 的；
183. 没有定义任何构造函数；
184. 没有类内初始值；
185. 没有基类或者虚函数（virtual）；

如以下类便是聚合的：

```
struct Data {
    int ival;
    string s;
};
```

可通过括弧列表中的初始值来初始化聚合类的成员，如：`Data val1 = {0, "Anna"};` 要求是括弧中的初始值需要与声明的数据成员的顺序一致。

与初始化数组元素一样，如果初始值列表中的值少于类的成员数，剩余的成员将被值初始化。初始值列表中包含的值一定不能多于类成员数。

聚合类有一些缺陷：

186. 将正确初始化所有成员的责任强加在用户身上，这种初始化是冗长而易错的；
187. 当类的成员被添加或移除时，所有的初始化的地方都需要更新；

7.5.6 字面类

constexpr 函数的参数和返回值都必须时字面类型。除了算数类型、指针和引用外，还可以定义字面类（literal classes）。字面类中的成员函数可以是 constexpr 的，这些成员函数必须满足常量表达式函数的所有要求。这些成员函数隐式是 const 的。

一个聚合类如果其所有数据成员都是字面类型是一个字面类。一个非聚合类，如果满足以下条件亦是一个字面类。

188. 所有数据成员都是字面类型；
189. 类至少有一个 constexpr 构造函数；

190. 如果数据成员有类内初始值，内置类型的初始值必须是常量表达式，如果是类类型，初始值必须是调用 constexpr 构造函数产生的；
191. 类必须使用编译器生成的析构函数；

constexpr 构造函数

字面类必须至少包含一个 constexpr 构造函数。constexpr 构造函数可以被声明为 =default 或者为 deleted 函数。否则，这种构造函数通常是空的，原因在于构造函数没有 return 语句，而 constexpr 函数唯一的语句便是 return 语句。如：

```
class Debug {
public:
    constexpr Debug(bool b = true): hw(b), io(b), other(b) {}
private:
    bool hw;
    bool io;
    bool other;
};
```

constexpr 构造函数必须初始化所有数据成员，初始值要么是调用 constexpr 构造函数，要么是一个常量表达式。constexpr 构造函数常用于给 constexpr 函数构建参数或返回值。

7.6 static 成员

类有时需要一些成员是与整个类关联的，而不是与类的单个对象关联。从效率的角度来看，不必为每个对象存储一样的值，而且将值存储为类相关，当值发生改变时所有对象都可以马上使用此新值。

声明静态成员

通过在成员前加上 static 关键可以使得其与整个类关联。与其它成员类似，static 成员可以是 public 或 private 的，static 数据成员可以是 const 、引用、数组或类类型。

static_class_member.cc 包含详细示例代码。

静态成员存在于任何对象之外，对象中不包含静态成员数据。

类似的，静态成员函数不与任何对象关联，它们没有 this 指针。因而，静态成员函数不能被定义为 const 的，并且在函数体内不能使用 this 指针，包括显式使用 this 指针或者通过访问非静态成员数据或调用非静态成员函数隐式使用。

使用静态成员

可以通过作用域操作符直接访问，或者通过类对象、指针、引用访问静态成员，虽然静态成员并不是类对象的一部分。如：

```
double r = Account::rate();  //通过作用域操作符
Account ac1;
Account *ac2 = &ac1;
r = ac1.rate(); //对象访问
r = ac2->rate(); //指针访问
```

静态成员可以被任何成员函数直接访问，不管是静态的还是常规的，这些访问是不需要作用域操作符进行限定的。

定义静态成员

可以在类的内部或外部定义静态成员函数。当在外部定义时，不需要重复 static 关键字，此关键字只应该放在类内的声明处。静态数据成员不能在类内进行初始化。所有的静态数据成员都必须在类外进行定义和初始化，并且只能被定义一次。如全局对象一样，静态数据成员在任何对象之外定义，因而，一旦它们被定义了，它们将持续到程序完全退出。

定义静态数据成员需要在成员名之前加上类名进行限定。如：

```
double Account::interestRate = initRate();
```

以上需要注意的是一旦定义中看到类名之后，定义的其余部分就处于类作用域中，因而使用 initRate 成员就不必用类名进行限定，即便 initRate 是私有成员依然是可以访问的，原因是任何成员的定义都可以访问类的私有成员。

通常将定义静态数据成员的代码放在与定义类的非内联函数一起的源文件中。

静态数据成员的类内初始化

可以在类内为 const 整数类型的静态成员进行初始化，并且必须给 constexpr 字面类型的静态成员进行类内初始化。初始值必须是常量表达式，这些成员本身亦是常量表达式，它们可以被用于需要常量表达式的地方。如可以用于定义数组成员的维度。如：

```
class Account {
    static constexpr int period = 30;
    double daily_tbl[period];
};
```

如果常量字面类型静态成员只运用于编译器可以将成员替换为其值的地方，那么这个静态成员不需要额外定义。如果此静态成员被用于一个不可以进行替换的场景，那么必须为此成员进行定义。例如，将 Account::period 传递给一个接收 const int& 的函数，那么 period 必须进行定义。

如果在类内提供了初始值，类外的成员定义不需要指定初始值。如：

```
constexpr int Account::period;  //初始值已经在类内提供了
```

通常在类外部定义静态常量成员是一个好习惯。

以 const 修饰的静态成员，如果在类内初始化则是常量表达式，如果在类外进行初始化则为常规的 const 常量，而非常量表达式。

静态成员可以被用于常规成员不能运用的场景

静态数据成员可以使用未完成的类型。特别是，静态数据成员可以使用当前定义的类本身的类型，而非静态数据成员只能被声明为指针或引用。

另外要给区别在于可以将静态数据成员作为默认参数，而常规数据成员是不可以的。如：

```
class Screen {
public:
    Screen& clear(char = bkground);
private:
    static const char bkground;
};
```

关键概念

192. 抽象数据类型（abstract data type）：封装了实现的数据结构；
193. 类声明（class declaration）：形如：`class T;`，如果一个类只被声明而没有定义，它是一个未完成类型（incomplete type）；
194. 类作用域（class scope）：每个类都定义一个作用域。类作用域比其它作用域跟复杂，成员函数甚至可以使用出现在其后的声明的成员；
195. 常量成员函数（const member function）：不改变对象状态的成员函数，其函数体内的 this 指针指向 const 对象，成员函数可以通过是否为 const 进行重载；
196. 构造函数（constructor）：用于初始化对象的特殊成员函数，每个构造函数都应该给每个数据成员一个良好定义的初始值；
197. 构造初始值列表（constructor initializer list）：为每个数据成员指定初始值，成员被初始化为初始值列表中指定的值，这个过程发生在函数体执行之前。未出现在构造初始值列表中的成员将被默认初始化；
198. 转换构造函数（converting constructor）：非 explicit 构造函数的单一参数构造函数可以被用于隐式转换其参数类型到类类型；
199. 数据抽象（data abstraction）：一种聚焦于类型接口的编程技术。数据抽象使得程序员可以忽略类型的实现细节，只关注于类型可以执行的操作。数据抽象是面向对象和泛型编程的基础；
200. 封装（encapsulation）：将实现和接口分开，封装隐藏了实现的细节；
201. 前置声明（forward declaration）：声明一个尚未定义的名字。通常用于在表示在定义一个类之前先声明类，它是一个未完成类型；
202. 未完成类型（incomplete type）：只有声明而没有定义的类型，不能用未完成类型定义变量或者成员，但是可以定义此类型的指针或引用；
203. 名称查找（name lookup）：将名字的使用与其声明相匹配的过程；
204. this 指针（this pointer）：隐式传递给每个非 static 成员函数的实参，this 指针指向当前调用对象；

C++ 语言本身并不直接处理输入和输出，相反是在标准库中定义一族类型来处理 IO。这些类型支持从文件和控制台中进行输入和输出。还有额外的类型允许对内存中的 string 进行 IO。
IO 库已经定义如何对内置类型值进行读写，同样，类类型可以定义自己的 IO 操作符来进行输入输出。本章主要聚焦于基本的 IO 操作，14 章将介绍如何定义类类型的操作符，17 章将介绍如何控制格式化以及随机访问文件。
最基本的 IO 库设施包括：
205. istream 用于输入操作；
206. ostream 用于输出操作；
207. cin 是标准输入对象（istream 类型）；
208. cout 是标准输出对象（ostream 类型）；
209. cerr 是标准错误对象（ostream 类型）；
210. >> 用于从 istream 对象中读取输入；
211. << 用于向 ostream 对象写入输出；
212. getline 函数用于从 istream 对象中读取一行输入并存入 string 中去；

8.1 IO 类

直接介绍过 cin、cout、cerr 对象用于在控制台上进行输入输出，而真实应用还需要从文件和 string 中进行输入输出，同时需要支持宽字符的输入输出。为了支持所有这些不同类型的 IO 操作，标准库定义了一系列 IO 类型用于补充 istream 和 ostream 类型。
它们被定义在三个不同的头文件中，iostream 定义基础类型，fstream 定义针对文件的输入输出，sstream 定义针对 string 的输入输出。如：
213. iostream 头文件：istream、wistream 从流中读取，ostream、wostream 写入流，iostream、wiostream 对流进行读写；
214. fstream 头文件：ifstream、wifstream 从文件中读取，ofstream、wofstream 写文件，fstream、wfstream 对文件进行读写；
215. sstream 头文件：istringstream、wistringstream 从字符串中读取，ostringstream、wostringstream 写入到字符串中，stringstream、wstringstream 对字符串进行读写；

为了支持宽字符集，标准库定义了处理 `wchat_t` 数据的类型和对象。宽字符版本通常以 w 开头，如 wcin，wcout，wcerr 是 cin，cout，cerr 的款字符对应对象。宽字符类型与对象和对应的常规的字符类型输入输出版本定义在同一个头文件中。

IO 类型之间的关系

不论是哪一种方式的 IO，从控制台或者文件或者字符串中进行输入输出，或者是字符宽度的不一样都不会影响操作的使用形式，都是使用 << 和 >>。之所以能够这样的做的原因在于不同种类的 stream 使用了继承（inheritance）。使用继承时不需要理解继承工作的细节，可以将派生类的对象完全当做基类对象来使用。ifstream 和 istringstream 都继承自 istream，所以可以将 ifstream 或 istringstream 类型对象当做 istream 对象来使用，这样可以将 >> 用于从 ifstream 和 istringstream 中读取数据。
本节中后面介绍的内容可以无差异的用于所有的 stream 类型。

8.1.1 不能拷贝或赋值 IO 对象

IO 类型的对象是不可以拷贝或赋值的，函数只能传递或者返回流对象的引用。读取或写入 IO 对象会改变起状态，所以引用必须不是 const 的。

8.1.2 条件状态

IO 操作不可避免地会出现错误，有些错误是可以恢复的，如格式错误；有些错误则深入到系统中，并且超出了程序可以修正的范围。IO 类定义函数和标记来访问和修改流的条件状态（condition state）。如：
216. strm::iostate strm 是一个 IO 类型，iostate 是一个机器相关的整形类型用于表示 stream 的条件状态；
217. strm::badbit 常量值，可以赋值给 strm::iostate，用于表示流被损坏了；
218. strm::failbit 常量值，可以赋值给 strm::iostate 用于表示 IO 操作失败了；
219. strm::eofbit 常量值，可以赋值给 strm::iostate 用于表示流到了 end-of-file；
220. strm::goodbit 常量值，可以赋值给 strm::iostate，用于表示流没有遇到错误，这个值保证是 0；
221. s.eof() 当流对象的 eofbit 被设置时返回 true；

222. s.fail() 如果流对象的 failbit 或 badbit 被设置时返回 true；
223. s.bad() 如果流对象的 badbit 被设置时返回 true；
224. s.good() 如果流对象状态没有任何错误（valid state）时返回 true；
225. s.clear() 将流对象状态设置为没有错误，返回 void；
226. s.clear(flags) 将流对象的状态设置为 flags 所表示的值，它是 strm::iostate 类型；
227. s.setstate(flags) 在 s 上添加特定的条件状态 flags，flags 的类型时 strm::iostate，返回 void；
228. s.rdstate() 返回 s 的当前条件状态，返回类型是 strm::iostate；

如以下代码：

```
int ival;
std::cin >> ival;
```

如果输入 Boo 将会读取失败，输入操作符想要获取一个 int 类型的值，但是遇到了字符 B，这时 cin 对象中将会产生一个错误状态。同样，如果输入 end-of-file 时，cin 也会处于错误状态。

一旦发生了错误，此流对象接下来的 IO 操作将会全部失败，只有当流对象没有错误时才能对其进行读取和写入。由于流可能处于错误的状态，通常需要在使用前检查流是否处于正确状态。检查流对象的状态的最简单的方式就是将其放到一个条件中去：

```
while (cin >> word)
    // ok: read operation successful
```

while 条件检查 >> 表达式返回的流的状态，如果操作成功，这个状态将保持有效，并且条件检查将会返回 true。

检查流的状态

将流放到条件中只是告知我们流是否是有效的，但是并没有告知发生了什么。有时需要知道更详细的信息。IO 对象使用一个机器相关的整形 iostate 来承载状态信息。这个类型是一个 bit 的集合。IO 类同时定义了 4 个常量值来表示不同的 bit 模式，它们用于表示不同的 IO 状态。这些值可以通过位操作符来测试和设置 iostate 值。

badbit 表示系统级的错误，如不可恢复的读写错误。通常如果 badbit 被设置的话，这个流对象就不再可用了。failbit 在一个可恢复的错误后设置，如，当需要一个数字时读取一个字符。通常是可以修正错误，并且继续使用流的。当到达文件尾部时会同时设置 eofbit 和 failbit。goodbit 则保证位 0，用于表示流没有错误。如果 badbit，failbit 或 eofbit 中任何一个被设置，那么对流进行条件求值将会失败。

标准库还定义了一系列函数来检查这些标记的状态。如果 good 函数返回 true 则表示没有任何一个错误位被设置，bad，fail 和 eof 函数返回 true 则是当对应的位被设置时。另外，fail 在 bad 位被设置时返回 true。

查看流的整体状态使用 good 或 fail 函数，将流对象用于条件中时，相当于调用 `!stream.fail()`，eof 和 bad 操作则仅仅只揭示特定的错误是否发生。

管理条件状态

rdstate 成员返回流的当前状态的 iostate 值，setstate 在流上设置给定的状态位来表示发生了某些问题。而 clear 成员中不带参数则清除所有的失败位，在这之后调用 good 将返回 true。如：

```
auto old_state = cin.rdstate();
cin.clear();
process_input(cin);
cin.setstate(old_state);
```

clear 还有一个有一个实参的版本，这个实参的类型是 iostate 值用于表示新的流状态，它是将参数中的值全部替换掉流中的状态值。为了关闭一个单独的状态，使用 rdstate 成员函数读取状态值，并用位操作符生成想要的新状态。如以下代码将 failbit 和 badbit 关掉，但是不改变 eofbit 的值：

```
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

8.1.3 管理输出缓冲

每个输出流都管理着一个缓冲，这个缓冲用于装载程序读写的数据。程序调用输出可能会立即打印出去，也可能操作系统会将数据缓存在 buffer 中，在之后 buffer 积累满了之后再打印出去。使用缓冲允许操作系统将多次输出操作拼接起来，最后调用一次系统级的写操作。这是因为写入到设备中是很耗时的操作，让操作系统将多次输出操作拼接成一次写操作可以提供很大的性能提升。

有多种条件将导致缓冲被刷新——将数据写入到真正的输出设备或者文件中去：

229. 程序正常结束，所有的程序中使用到的输出缓冲都会被写入到设备中去；
230. 在某些中间时刻如果缓冲已经满了的时候，会在下一次写操作前将其内容刷新到设备中去；
231. 通过 endl 操纵子（manipulator）显式将刷新缓冲；
232. 使用 unitbuf 操纵子在每次输出操作之后将流的内部状态设为空的缓冲。默认情况下，只有 cerr 的 unitbuf 是设置了的，所以 cerr 的刷新是立即的；
233. 一个输出流被绑定到另外一个流上，在这种情况下，被绑定的流将会在绑定的流进行读写时刷新。默认情况下，cout 被绑定到 cin 和 cerr 上，意味着，当任何时候读 cin 或者写 cerr 时都会刷新 cout 中的缓冲；

刷新输出缓冲

endl 在缓冲中增加一个换行符并刷新，flush 不添加任何字符进行刷新，ends 添加一个 NULL 字符然后刷新。如：

```
std::cout << "hi!" << std::endl;
std::cout << "hi!" << std::flush;
std::cout << "hi!" << std::ends;
```

unitbuf 操纵子

如果想在每次输出之后都进行刷新，就需要用到 unitbuf 操纵子。这个操纵子告知流在每次写入时都调用 flush，nounitbuf 操纵子则将流的状态恢复到正常，由操作系统控制缓冲刷新：如：

```
cout << unitbuf; // all writes will be flushed immediatedly
cout << nounitbuf; // returns to normal buffering
```

注意：当程序崩溃时缓冲不会被刷新

当程序意外终止时输出缓冲是不会刷新的，此时程序写入的数据很可能还在缓冲中等待被打印。当你调试一个崩溃的程序时，确保你认为应该输出的数据确实刷新了。由于这种问题已经导致了不计其数的调试时间被浪费了。

将输入流和输出流绑在一起

当一个输入流和输出流绑在一起时，当尝试去读取输入流时将会先刷新输出流中的缓冲。ite 函数有两个重载的版本：一个没有参数并返回一个输出流的指针，如果它绑定了一个输出流的话。如果没有绑定则返回一个 nullptr。第二个版本接收一个 ostream 类型的指针，并将其绑定到这个流上，如：`x.tie(&o)` 将流 x 绑定到输出流 o 上，这样 x 的任何操作将导致 o 的输出缓冲被刷新。可以将 istream 或 ostream 对象绑定到另外一个 ostream 上。如：

```
cin.tie(&cout);
ostream *old_tie = cin.tie(nullptr); // cin is no longer tied
cin.tie(&cerr); // reading cin flushes cerr
cin.tie(old_tie);
```

交互式系统应该将其输入流绑定到输出流上，这样做意味着所有的读操作之前都会将缓冲中的数据刷新出去。

传给 tie 一个输出流的指针就可以将调用的流对象绑定到其上，如果传空指针则将其解绑。每个流每次只能绑定到一个输出流上。然而多个流对象可以同时绑定到一个输出流对象上。

8.2 文件输入输出

在 fstream 头文件中定义了三个支持文件 IO 的类型：ifstream 来读取给定文件，ofstream 来写入给定文件，fstream 可以同时读写文件。17 章中将介绍如何同时进行同一个文件的读写。

这几个对象提供了 cin 和 cout 提供的操作，如：<< >> 用于读写，可以用 getline 来读一个 ifstream，并且 8.1 节介绍的内容都适用于这几个类型。

除了从 iostream 类型继承来的行为之外，fstream 头文件中的类型还定义了管理文件相关的成员，这些操作可以被 fstream 、ifstream 或 ofstream 调用，但不能在其它 IO 类型上调用。如：

234. `fstream fstrm;` 创建一个没有关联文件的文件流，fstream 是定义在 fstream 头文件中的一个类型；
235. `fstream fstrm(s);` 创建一个 fstream 并打开名为 s 的文件，s 可以是 string 类型或者是一个 C 风格字符串指针，这个构造函数是 explicit 的，默认的文件模式取决于 fstream 的类型；
236. `fstream fstrm(s, mode);` 与上一个构造函数类似，但是以给定的模式 mode 打开 s 文件；

237. `fstrm.open(s)` 打开名为 s 的文件，将其关联到 fstrm 对象上，s 可以是 string 或者 C 风格字符串指针，默认的文件模式取决于 fstrm 的类型，返回 void；
238. `fstrm.open(s.mode)` 与上面一样，但是由 open 方法提供文件开打模式 mode；
239. `fstrm.close()` 关闭与 fstrm 关联的文件，返回 void；
240. `fstrm.is_open()` 返回一个 bool 值告知是否此 fstrm 关联的文件已经成功打开，并且没有被关闭；

8.2.1 使用文件流对象

当我们想要读写文件时，我们通常会定义个文件流对象，并且将之关联到一个此文件上。每个文件流对象都定义了 open 成员函数，它会做与系统相关的操作来定位给定，并且打开它进行读或写操作。
当创建文件流对象时可以选择性的提供一个文件名，如果提供了文件名，那么 open 就会自动调用。如：

```
ifstream in(ifile); // construct an ifstream and open the given file
ofstream out; // output file stream that is not associated with any file
```

在新标准下，文件名字可以是 string 类型或者 C 风格字符数组。之前的版本只支持 C 风格字符数组。

open 和 close 成员

当定义一个空的文件流对象时，可以接着在后面通过 open 将其关联到一个文件上。如：

```
ifstream in(ifile);
ofstream out;
out.open(ifile + ".copy");
```

当调用 open 失败时，会设置 failbit，由于 open 可能会失败，所以最好需要验证一下 open 是否成功。如：

```
if (out) // check that the open succeeded
    // the open succeeded, so we can use the file
```

如果打开失败，条件将会失败，我们就不能使用 out 对象。
一旦一个文件流对象被打开，它将与给定的文件持续关联。如果在一个已经打开的文件流对象上调用 open 将会失败，并且设置 failbit。接下来尝试使用这个文件流将会失败。为了将文件流对象关联到一个不同的文件上，需要选将之前的文件关闭，才能打开新的文件。如：

```
in.close();
in.open(ifile + "2");
```

如果 open 成功的话，open 会将流的状态设置为 0，这样 good() 将返回 true。

自动构建和析构

代码如：

```
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p);
    if (input) {
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
}
```

每次迭代时都会自动创建一个新的 ifstream 对象并打开给定文件。由于 input 是 while 中的本地对象，它将在每次迭代时自动的创建和销毁。当 fstream 对象离开作用域之后，与其关联的文件会自动关闭。在下一次迭代时会创建一个新的。

8.2.2 文件模式

每个流都有与之关联的文件模式（file mode）来代表文件可以如何被使用。以下列举文件模式和它们的含义：

241. in 打开作为输入；
242. out 打开作为输出；
243. app 在每次写入前都跳到文件的尾部；
244. ate 在打开后直接跳到文件的尾部；
245. trunc 将文件截断；
246. binary 以二进制模式进行 IO 操作；

当打开文件时可以提供一个文件模式。打开文件可能是直接调用 open 打开，通过初始化文件流关联到一个特定文件上也会间接打开这个文件。可以指定的文件模式有一些限制：

247. out 只能给 ofstream 或者 fstream 对象设置；
248. in 只能给 ifstream 或者 fstream 对象设置；
249. trunc 只能设置了 out 的时候进行设置；
250. app 只要在没有设置 trunc 时就可以设置，如果指定了 app，那么文件总是在输出模式下打开，即便 out 没有显式指定；
251. 默认情况下，一个文件如果是在 out 模式下会被截断，即便没有显式指定 trunc。为了保留文件的内容，可以通过指定 app 或者同时指定 in 同时用于输入输出；
252. ate 和 binary 模式可以在任何文件流类型上设置，可以与任何别的文件模式组合使用；

每个文件流类型都定义了默认的文件模式。ifstream 默认以 in 模式打开文件，ofstream 默认以 out 模式打开文件；fstream 关联的文件则同时以 in 和 out 打开文件；

8.3 string 流

sstream 头文件中定义了三个支持内存中 IO 的类型；这些类型将 string 当作一个 IO 流进行读写。istringstream 可以读取一个 string，ostringstream 可以写一个 string，stringstream 可以对 string 进行读写。sstream 头文件中的类型继承自 iostream 头文件中的类型，除了继承来的成员之外，sstream 还定义了成员来管理与之相关的 string。它们不能在其它 IO 类型上调用，如：

253. `sstream strm` strm 是没有关联的 stringstream。*sstream* 是定义在 sstream 头文件中的一个类型；
254. `sstream strm(s)` strm 是关联到字符串 s 副本的 *sstream* 对象，这个构造函数是 explicit 的；

尽管 fstream 和 sstream 共享相同的 iostream 接口，它们之间没有任何别的关系，特别是不能用 open 和 close 在 stringstream 上，也不能将 str 用于 fstream 上。

8.3.1 使用 istringstream 对象

8.3.2 使用 ostringstream 对象

如代码：stringstream.cc

使用 sstream 可以很方便的将 string 作为输入输出源，可以做到很精细化的操作。

关键概念

255. **条件状态（condition state）：**流类中用于查询流对象是否可用的标志以及相关的函数；
256. **文件模式（file mode）：**fstream 定义的标志用于控制文件如何被打开；

本章内容是对第三章的一个扩展，并且是对标准库顺序容器的一个完整讨论。顺序容器（sequential containers）中的元素顺序就是其被添加到容器中的位置。标准库还定义了几个关联容器（associative containers），关联容器中元素顺序由每个元素的 key 决定。

容器类共享一些共同的接口，然后在对其进行自己的扩展。共同接口使得容器更容易学习；在一个容器上所学的东西可以运用到另外一个容器上，除了每种容器提供不同的性能和功能性上的抉择。

容器（container）可以容纳一系列特定类型的对象，顺序容器使得程序员可以控制元素被存储和访问的顺序。这些顺序不由元素的值决定，而是根据其对应的置入容器中的位置决定的。相反，有序和无序关联容器基于键值来存储器元素。

标准库还提供了三个容器适配器，适配器通过为容器类型的操作定义不同的接口来达到不同的语义。

以下内容是基于 §3.2 §3.3 §3.4 的内容的，在阅读下面的材料之前请先阅读这三节。

9.1 顺序容器概述

所有下面的容器都能够快速顺序访问其元素。它们之前的区别在于以下功能的性能取舍：

257. 给容器添加和删除带来的消耗；
258. 非顺序访问容器中的元素带来的消耗；

顺序容器列表如下：

259. vector：灵活尺寸的数组，支持快速随机访问，除了在尾部插入或删除元素外会很慢；
260. deque：双端队列，支持快速随机访问，支持首部和尾部的快速插入和删除；
261. list：双链表，只支持双向顺序访问，在任何位置上都支持快速插入和删除；
262. `forward_list`：单链表，只支持单一方向的顺序访问，在任何位置上都支持快速插入和删除；
263. array：固定尺寸的数组，支持快速随机访问，但是不能添加和删除元素；
264. sring：包含字符的特殊容器，类似于 vector，支持快速随机访问，以及在尾部的快速插入和删除；

除了 array 是固定尺寸（fixed-size）的容器，所有容器都提供了高效而灵活的内存管理。我们可以添加和删除元素，可以增加（growing）或缩减（shrinking）容器的大小。容器使用的存储元素的策略对于这些操作的性能有固有和重要的影响。在一些情况下甚至会影响一个特定的容器是否会提供特定的操作。
如：string 和 vector 将其元素放在连续的内存中，由于元素是连续的，能够快速按照其索引计算元素的地址。然而，在容器的中间添加或移除元素就是很耗时的操作了：所有位置在插入或删除之后的元素都必须移动以保持连续性。更甚者，添加元素有时会导致需要分配额外的内存，此时，所有元素都需要移动到新的内存中，并销毁原来使用的内存。
list 和 `forward_list` 被设计使得可以在容器的任何位置快速添加和删除，作为交换，这些类型不支持元素的随机访问：只能通过迭代的方式访问元素。并且相较于 vector, deque 和 array 消耗更多的内存。
deque 是一个更为复杂的数据结构。与 string 和 vector 一样，deque 支持快速随机访问（fast random access），并且在 deque 的中间添加和移除元素是很耗时的。然而，在 deque 的两端进行插入和删除则是很快速的操作，可以比得上在 `list` 或 `foward_list` 中添加元素。
`forwad_list` 和 array 是在新标准中添加进来的。array 是内置数组的一个更安全且易用的替代品。与内置数组一样, array 是固定尺寸的。因而, array 不支持添加和移除元素, 或者改变容器的大小。`forward_list` 则是为了提供了性能上比得上最好的手写单链表。`forward_list` 没有提供 size 操作，这是由于存储和计算大小会导致其性能比不上手写的单链表。其它容器的 size 操作则保证是快速且是固定时间（constant-time）的操作。
由于移动操作的出现，现在的库容器的性能已经比之前的版本有了戏剧性的提升。库容器几乎肯定比绝大多数手写版本性能要好，现代 C++ 程序应该更多使用库容器而不是更基础的结构如内置数组。
决定使用哪个顺序容器
有一些选择使用哪个容器的首要原则：
265. 除非了更好的理由，优先使用 vector；
266. 如果程序有许多小的元素并且对空间消耗很敏感，不要使用 list 和 `forward_list`；
267. 需要随机访问，使用 vector 或 deque；
268. 需要在容器中间部分进行快速插入和删除，使用 list 或 `forward_list`；
269. 如果程序需要是首部和尾部插入或删除元素，而不是中间部分，使用 deque；
270. 如果程序在读取输入时需要在中间部分插入元素，而接下来则需要随机访问元素：
    1. 先看是否真的需要在中间部分插入元素，通常更容易的做法是添加到 vector 中，然后调用库函数 sort 进行排序；
    2. 如果确实需要在中间部分插入，考虑在输入阶段使用 list，一旦完成了输入，将其复制到 vector 中去；

如果同时需要随机访问和在中间部分插入或删除元素？此时需要看占主要的操作是什么（更多的是插入或删除，还是更多的是访问），来决定使用哪种类型的容器。在这种情况下，使用多种容器进行性能测试是必不可少的。
最佳实践：如果你不知道该选择哪种容器，那就进使用 vector 和 list 共同的接口：迭代器而不是下标，避免随机访问元素。这样就可以在需要时很容易的替换。

9.2 容器库概述

容器类型上的操作组成了一种层级结构。一些操作是所有容器类型都提供的，其它一些则是顺序容器特有的，或者关联容器或无序容器特有的。而最后还有一些只有少部分的容器类型拥有。
在本节，我们将描述所有类型的容器都拥有的共同部分。§9.3 描述顺序容器特有的操作。¶11 描述关联容器特有的操作。
通常每个容器定义在与其名字相同的头文件中。deque 定义在 `<deque>` 头文件中, list 定义在 `<list>` 头文件中。容器都是类模板。与 vector 一样，使用时必须提供额外的信息来生成特定的容器类型。对于绝大部分容器来说，提供的是元素的类型，如：`list<Sales_data>` `deque<double>` 。
以下是所有容器共同的操作：
类型别名 - iterator ：每个容器类型自己的迭代器类型； - `const_iterator` ：可以读取但不能修改元素的迭代器类型； - `size_type` ：容器类型足够容纳最大的容器的尺寸的无符号整形值类型； - `difference_type` ：两个迭代器之间的距离的有符号整形类型； - `value_type` ：元素类型； -

reference ：元素的左值类型，与 value_type& 是同义词； - const_reference 元素的 const 左值类型；

赋值和 swap - c1 = 2 将 c1 中的元素替换为 c2 中的元素； - c1 = {a,b,c...} 将 c1 中的元素替换为列表中的值（不适用于 array）； - a.swap(b) 将 a、b 中的值进行交换； - swap(a, b) 等于交换 a.swap(b)；

Size - c.size() c 中元素个数（不适用于 forward_list）； - c.max_size() c 所处的容器类型可以容纳的元素的最大数目； - c.empty() 如果 c 有元素返回 false，否则返回 true；

添加、移除元素（不适用于 array） Note：每个容器的接口参数不一样 - c.insert(args) 将 args 中的元素拷贝到 c 中； - c.emplace(inits) 使用 inits 来构建 c 中的一个元素； - c.erase(args) 从 c 中移除由 args 所表示的元素； - c.clear() 移除 c 中的所有元素；返回 void；

相等性和关系操作符 - == != 相等性操作符使用所有容器类型； - < <= > >= 关系操作符（对于无序关联容器不适用）；

获取迭代器 - c.begin() c.end() 返回首元素和尾后元素（one past the last element）的迭代器； - c.cbegin() c.cend() 返回 const_iterator；

可反转的容器的额外成员（不适用于 forward_list） - reverse_iterator 以相反方向访问元素的迭代器； - const_reverse_iterator 不能写元素的反向迭代器； - c.rbegin() c.rend() 返回尾元素和首前元素（one past the first element）； - c.crbegin() c.crend() 返回 const_reverse_iterator；

以下是定义和初始化容器的方式：

271. C c; 默认构造函数，如果 C 是 array，那么 c 中的元素默认初始化的；否则 c 是空的；
272. C c1(c2); c1 是 c2 的副本。c1 和 c2 必须是相同的类型（容器类型和元素类型都必须一样，对于 array 还必须有相同的 size）；
273. C c{a,b,c...} C c={a,b,c...} c 是初始化列表元素的一个副本；列表中元素的类型必须与 C 的元素类型相互兼容，如果是 array，列表中的数目必须小于等于 array 中的大小，所有缺失的元素是值初始化的；
274. C c(b,e) c 是迭代器 b 和 e 所表示的范围的元素的副本，其元素的类型必须与 c 中的元素类型互相兼容（不适用于 array）；

以 size 初始化顺序容器（不适用于 array） - C c(n); c 具有 n 个值初始化元素，此构造函数是 explicit 的； - C c(n,t); c 中具有 n 个值 t；

容器可以容纳的元素类型

几乎任何类型都可以作为顺序容器的元素类型。特别是可以定义容器的元素类型是另外一个容器。如：vector<vector<string>> lines; 在一些旧式的编译器中可能会要求相连的两个尖括号间用空格隔开。如：vector<vector<string> >。

尽管可以将几乎任何类型存储在容器中，一些容器操作对元素类型是有要求的。我们可以定义不支持某些操作要求的容器类型，但是想要使用这些操作就必须让元素类型满足这些操作的需求。

比如，接收一个 size 参数的顺序容器构造函数将使用元素类型的默认构造函数。一些类型是没有默认构造函数的，可以定义容器包含这种类型的对象，但是不能只使用元素数目去构建这样的容器。如：

```
vector<noDefault> v1(10, init); // 正确：提供了元素初始值
vector<noDefault> v2(10); // 错误：必须提供元素的初始值
```

在接下来的介绍中会逐步描述其它操作对于容器元素类型的要求。

9.2.1 迭代器

与容器一样，迭代器（iterator）本身也是有共同的接口的。如果一个迭代器提供了某种操作，那么所有迭代器类型都提供的这种操作的行为都是一致的。如所有的标准容器都提供解引用操作符以访问容器元素，所有的标准库容器迭代器都提供自增运算符从一个元素移动到下一个元素。

forward_list 容器的迭代器不支持自减操作符，只有 string，vector，deque 和 array 支持迭代器的算术运算。

迭代器范围

迭代器范围的概念是标准库的一个基石。迭代器范围（iterator range）是由同一个容器中的一对迭代器来表示的，其中一个第一个元素，第二个指向最后的一个元素的下一个位置（one past the last element）。这两个迭代器通常用 begin 和 end 或者 first 和 last 来表示一个容器中的元素范围。

名字 last 带有一点误导的作用，第二个迭代器并不是指向最后一个元素，而是指向最后一个元素的下一个位置。元素范围包括 first 和一直到 last 之前的所有元素。这种元素范围成为左包含区间（left-inclusive interval），用数学表示就是 **[begin, end)** ，表示范围从 begin 开始，以 end 结束但是不包括 end。迭代器 begin 和 end 必须指向同一个容器。
end 一定不能指向 begin 之前的元素，只有这样才能通过自增 begin 迭代器而到达 end ，否则，所表示的范围就会无效。编译器并不能强制要求这种需求，只能有程序自己去保证这种约定。

左包含范围的编程隐喻

标准库使用左包含范围是由于这种范围有三个遍历的属性：
275. 如果 begin 和 end 相等，那么范围是空的；
276. 如果 begin 与 end 不相等，那么范围中至少有一个元素，并且 begin 表示范围内的第一个元素；
277. 通过自增 begin 几次之后会达到 **begin == end**；

这几个属性使得我们可以这样书写循环，如：

```
while (begin != end) {
    *begin = val;
    ++begin;
}
```

9.2.2 容器类型的成员

每个容器都定义了几个类型。如：**size_type** ，iterator 和 **const_iterator** 。绝大多数的容器还提供反向迭代器，反向迭代器的 ++ 将使得迭代器指向前一个元素，从而达到反向遍历容器元素的目的。
剩下的类型别名使得我们使用容器的元素类型时不需要真正知道其类型是什么。如果我们需要元素类型，我们只要使用 **value_type** 就可以，如果需要值的引用类型，就使用 reference 或者 **const_reference** 就可以。这些元素相关的类型别名在泛型编程中十分有用处。
使用这些类型需要用容器的类型加以限定，如：

```
list<string>::iterator iter;
vector<int>::difference_type count;
```

9.2.3 begin 和 end 成员

begin 和 end 操作能够产生指向首元素和尾后元素的迭代器。它们通常用于表示容器中全部元素的迭代器范围。
标准库中有多个版本的 begin 和 end：r 版本返回反向迭代器，c 返回 const 的迭代器。如：

```
list<string> a = {"Milton", "Shakespeare","Austen"};
auto it1 = a.begin();
auto it2 = a.rbegin();
auto it3 = a.cbegin();
auto it4 = a.crbegin();
```

没有以 c 开头的函数是重载的，在 const 成员返回的是 **const_iterator**，非 const 版本返回的 iterator 。这适用于 begin 、rbegin 、 end 和 rend。当在非 const 对象上调用时返回的是 iterator，只有当在 const 对象上调用时调用才是 const 版本。与指针和引用一样，可以将 iterator 转为 **const_iterator** ，但不能反着操作。
c 版本是新标准引入的，这样做是为了支持 auto 声明变量。在早期只能使用指明需要哪个类型如：

```
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6 = a.begin();
auto it7 = a.begin();
auto it8 = a.cbegin();
```

如果使用 auto 那么 begin 和 end 返回的迭代器类型就由容器类型决定。使用 c 则可以不管是什么类型的容器都可以得到 **const_iterator**，当不需要写入时，使用 cbegin 和 cend。

9.2.4 定义和初始化容器

每个容器类型都定义了默认构造函数。除了 array ，其它容器的默认构造函数都创建一个空的容器。除了 array 之外，其它的容器可以接收一个 size 的参数，将初始化为有 size 个元素被值初始化的容器。

初始化容器作为另外一个容器的副本

有两种方式可以将一个容器初始化为别的容器的副本：调用拷贝构造函数或者（除 array 之外）拷贝由一对迭代器表示元素范围。

如果通过拷贝构造函数那么容器和元素的类型都必须一样。如果通过迭代器则没有这样的要求，只要元素类型之间是可以转换的。如：

```
list<string> authors = {"Milton", "Shakespeare","Austen"};
vector<const char*> articles = {"a","an","the"};
list<string> list2(authors);
deque<string> authList(authors); // 错误：容器类型不匹配
vector<string> words(articles); // 错误：元素类型不匹配
// 将 const char* 转为 string
forward_list<string> words(articles.begin(), articles.end());
```

接收两个迭代器的构造函数使用它们来表示想要拷贝的元素范围。其中一个表示首元素另一个表示尾后元素。新的容器的大小与范围中的大小一致。新容器中的每个元素都是用范围内元素进行初始化的。

由于迭代器表示一个范围，可以使用构造函数拷贝一个容器的子序列。如 it 表示 authors 中的一个元素：

```
deque<string> authList(authors.begin(), it);
```

以上将拷贝从 authors 的首元素直到 it 之前的所有元素。

列表初始化

在新标准下，可以使用列表初始化一个容器，如：

```
list<string> authors = {"Milton","Shakespeare","Austen"};
vector<const char*> articles = {"a","an","the"};
```

列表初始化会指定容器中的每个元素的值。除了 array 之外，初始化列表同时暗含了容器的大小：容器大小与元素的初始化列表中的个数一样多。

顺序容器指定大小的构造函数

上面所介绍的所有构造函数同时适用于关联容器。除此之外顺序容器（除了 array 之外）还可以指定大小和可选的元素初始值。如果没有提供元素初始值，那么将创建一个值初始化的容器。如：

```
vector<int> ivec(10, -1);
list<string> svec(10, "hi!");
forward_list<int> ivec(10);
deque<string> svec(10);
```

如果元素类型是内置类型或者具有默认构造函数的类类型，那么就可以使用接收一个大小的参数的构造函数来构建容器。如果元素类型没有默认构造函数，在初始化时必须同时显式提供元素初始值。

注意：指定大小的构造函数只适用于顺序容器；并不适用于关联容器。

array 类是固定尺寸

标准库 array 类型与内置数组一样，其尺寸是类型的一部分。当定义数组时除了指定元素类型时，还要指定大小。如：

```
array<int, 42> arr1;
array<string, 10> arr2;
```

使用 array 类型同时需要指定元素类型和大小：

```
array<int, 10>::size_type i;
array<int>::size_type j; // 错误：array<int> 不是一个类型
```

由于大小是 array 的一部分，array 不支持通常的容器构造函数。这些构造函数或隐式或显式决定了容器的大小。给 array 的构造函数传递表示大小的参数本身就是不必要且易错的。

array 的固定尺寸的属性同样影响其构造函数的行为。与其它容器不一样，其默认构造的 array 不是空的，而是所有的元素都是默认初始化的，这与内置数组的行为一致。如果想要使用列表初始化 array，提供的初始值必须小于等于 array 的大小。如果初始值更少的话，前面的部分将会被初始值初始化，后面的元素将被值初始化。在这两种情况下，如果元素类型是类类型，都必须要有默认构造函数。如：

```
array<int, 10> ia1;
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> ia3 = {42};
```

值得注意的是尽管我们不能拷贝或赋值内置数组，但是标准库 array 是可以拷贝和赋值的。如：

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs; // 错误：内置数组不能拷贝和赋值
```

```
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits;
```

与任何容器一样，拷贝或赋值时类型必须是完全匹配的，对于 array 来说元素类型和尺寸都必须完全一样。

9.2.5 赋值和 swap

赋值相关的操作符作用域整个容器。其将左边容器的全部元素替换为右边容器中的元素的副本。如：

```
c1 = c2; // 将 c1 的内容替换为 c2 中的元素的副本
c1 = {a,b,c}; // 将 c1 的内容替换为初始化列表中的元素的副本
```

当执行完赋值后，左右边的容器的内容就完全一样了。

与内置数组不一样，标准库 array 类型支持赋值。左右边的操作数必须是相同的类型：

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0};
a1 = a2; // 元素类型和大小都一样
a1 = {0}; // !!!错误：array 不支持初始值列表的赋值操作
```

由于初始值列表中的元素个数可能与 array 中的元素个数不一样，array 不支持用初始值列表进行赋值。基于同样的原因 array 同样不支持 assign 操作。

容器的赋值操作

278. `c1 = c2` 将 c1 中的元素替换为 c2 中元素的副本。c1 和 c2 必须是相同类型；
279. `c = {a,b,c...}` 将 c1 中元素替换为初始值列表中的元素的副本（不适用于数组）；
280. `swap(c1, c2)` `c1.swap(c2)` 交换 c1 和 c2 中的元素。c1 和 c2 必须是相同的类型。swap 通常比 c2 到 c1 的赋值要*快很多*。

assign 操作不适用于关联容器和 array

281. `seq.assign(b,e)` 将 seq 中的元素替换为由迭代器表示的范围中的值的副本，b 和 e 不能指向 seq 中的元素；
282. `seq.assign(il)` 将 seq 中的元素替换为初始值列表 il 中的值的副本；
283. `seq.assign(n,t)` 将 seq 中的元素替换为 n 个值 t 的副本；

赋值操作将使得左边容器中的迭代器、引用和指针失效，除了 string 之外，swap 之后却保持有效并且其所指向的容器被改变了（注：就是之前指向左边容器的迭代器，在 swap 之后指向右边容器中的元素了）。

使用 assign（只有顺序容器支持）

赋值操作符需要左边和右边操作具有相同的类型，它将其右边操作数的所有元素拷贝到左边操作数。同时顺序容器还定义了一个名为 assign 的成员（array 没有定义），assign 的存在使得可以从一个不同但是兼容的容器类型进行赋值，或者从一个容器的子序列进行赋值。assign 将左边容器中的元素全部替换为其参数中的元素。如可以将 `vector<const char*>` 赋值给 `list<string>` ：

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // 错误：容器类型不匹配
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

注意：由于现存的元素被替换掉，传递给 assign 的迭代器一定不能指向调用 assign 的容器。

assign 的第二个版本接收一个整数以及一个元素值。它将容器中的元素全部替换为特定数目的元素，每个元素值都与参数中给出的元素值一样。如：

```
list<string> slist1(1);
slist1.assign(10, "Hiya!");
```

使用 swap

swap 操作交换两个类型相同的容器的内容，在 swap 之后两个容器的元素是互换的。如：

```
vector<string> svec1(10);
vector<string> svec2(24);
swap(svec1, svec2);
```

除了 array 之外，语言标准保证 swap 两个容器是很快的，元素本身不会被交换；内部数据结构被交换。

注意：除了 array，swap 不做任何拷贝、删除或者插入元素，因而保证在固定时间内完成。
元素不被移动意味着迭代器、引用和指针都不会失效（对于 string 不成立）。它们在 swap 之后指向与之前同一个元素，但是需要明白的一点是这些元素被交换到了不同的容器中。如 iter 本身指向 svec1[3] 的，swap 之后指向 svec2[3] 了。与容器不同，在 string 调用 swap 会导致迭代器、引用和指针失效。
array 的 swap 有特别之处，它会交换所有元素值。所以交换两个 array 需要与 array 元素数目成比例的时间。想要 swap 一个 array 它们的元素类型和大小都必须一致。
array 在 swap 之后，其迭代器、指针和引用都指向原来指向的元素不变。当然，其元素值已经被替换为了另外一个 array 中对应位置的元素值。
在新标准中，同时提供了一个成员版本和非成员版本的 swap。早期版本只提供了成员版本的 swap，非成员版本对于泛型编程来说很重要。所以，最好是使用费成员版本的 swap。

9.2.6 容器 size 操作

除了 **forward_list** ，所有容器类型都提供 size 成员返回容器中元素的数目；除此之外，所有容器类型还都提供 empty 返回 bool 表示容器是否为空；**max_size** 返回此容器类型能够容纳的元素数目的限制；
forwad_list 只提供了 **max_size** 和 empty 操作。

9.2.7 关系操作符

所有容器类型都提供相等操作符（== 和 !=）；除了无序关联容器所有容器都提供关系操作符（> >= < <=）。左边和右边操作数必须是相同类型的容器，并且其元素类型也必须是一样的。
比较两个容器执行的是两两之间的比较，这与 string 的关系操作是一样的：

284. 如果两个容器具有相同的大小，并且所有元素都是一样的，那么这两个容器就是相等的；否则他们就是不相等的 ‘
285. 如果两个容器的大小不一样，但是短一点的容器中的每个元素都与对应位置的长的容器中的元素值相等，那么短的就比长的小；
286. 如果没有一个容器是另一个的头部子序列，那么比较取决于第一个不相等的元素；

如以下代码：

```
vector<int> v1 = {1,3,5,7,9,12};
vector<int> v2 = {1,3,9};
vector<int> v3 = {1,3,5,7};
vector<int> v4 = {1,3,5,7,9,12};
v1 < v2; // true
v1 < v3; // false
v1 == v4; // true
v1 == v2; // false
```

注意：只有元素类型同样定义了关系操作符，我们才能使用容器的关系操作符进行比较。

关系操作符使用元素的关系操作符

容器的相等操作符使用元素的 == 操作符，关系操作符使用了元素的 < 操作符。如果元素类型不支持需要的操作符，那么我们就无法使用容器的对应的操作。如：**Sales_data** 不支持 == 和 < 操作。那么就不能比较两个 **Sales_data** 的容器元素：

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // 错误：Sales_data 没有小于操作符
```

9.3 顺序容器操作

顺序容器和关联容器的区别在于如何组织元素。这些区别会影响元素是如何被存储、访问、添加和移除的。前面的部分主要介绍的是所有容器共有的操作。
以下将介绍顺序容器特有的操作。

9.3.1 给顺序容器添加元素

除了 array 之外的容器都提供了灵活的内存管理机制。可以动态的添加或移除容器中的元素从而在运行时改变容器的大小。以下是给顺序容器添加元素的操作：
以下的操作都会改变容器的大小；array 不支持这些操作。**forward_list** 有自己的 insert 和 emplace 的版本；**forward_list** 并不支持 **push_back** 和 **emplace_back**。string 和 vector 不支持 **push_front** 和 **emplace_front**；

287. `c.push_back(t)` `c.emplace_back(args)` 在容器 c 的尾部创建一个元素，其值为 t 或者以 args 为参数进行构建，无返回值；
288. `c.push_front(t)` `c.emplace_front(args)` 在容器 c 的头部创建一个元素，其值为 t 或者以 args 为参数进行构建，无返回值；
289. `c.insert(p,t)` `c.emplace(p,args)` 在迭代器 p 所指向的元素之前插入元素，其值为 t 或者以 args 为参数构建，返回被添加元素的迭代器；
290. `c.insert(p,n,t)` 在迭代器 p 所指向的元素之前插入 n 个值为 t 的元素。返回指向第一个被添加的元素的迭代器；如果 n 是 0，返回 p；
291. `c.insert(p,b,e)` 在迭代器 p 所指向的元素之前插入由迭代器 b 和 e 所表示的范围中的元素。b 和 e 不能 c 中的元素。返回指向第一个被插入的元素的迭代器；如果范围是空，返回 p；
292. `c.insert(p,il)` il 是一个括号中的初始值列表。将给定值插入到由 p 所指向的元素之前；返回第一个被添加的元素的迭代；如果列表是空的，返回 p；

注意：添加元素到 vector、string 或者 deque 中有可能会使得所有容器中现存的迭代器、引用和指针失效；

当我们使用这些操作时，需要记住这些容器使用不同的策略来分配内存给元素，这将会影响到性能。除了在尾部给 vector 和 string 添加元素，或者除了在首尾给 deque 添加元素都会导致元素移动。而且，添加元素给 vector 和 string 可能会导致整个容器的重新分配。重新分配整个对象需要重新分配内存，然后将其元素从旧的位置移动到新的位置。

使用 push_back

`push_back` 可以在容器的尾部添加元素，除了 array 和 `forward_list` 所有的顺序容器都支持 `push_back`。如：

```
string word;
while (cin >> word)
    container.push_back(word);
```

由于 string 就是字符的容器，可以用 `push_back` 在 string 的尾部添加字符：

```
word.push_back('s');
```

关键概念：容器元素是拷贝的

当我们使用对象来初始化容器，或者插入一个对象到容器中时，放入容器中的是哪个对象值的拷贝，不是对象本身。这就像我们传递一个对象给非引用参数是一样的，容器中的元素与这个对象没有任何关系。接下来如果改变了元素的值并不会影响到原始值，反之亦然。

使用 push_front

list 、`forward_list` 和 deque 支持类似的操作 `push_front` ，这个操作在容器的首部插入一个新的元素：

```
list<int> ilist;
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

在执行完这个循环之后，ilist 中包含值 3,2,1,0

deque 和 vector 一样支持对其元素的快速随机访问，并且提供 `push_front` 成员。deque 保证在其首尾添加或删除元素是固定时间的。与 vector 一样，在中间部分插入元素将是很耗时的操作。

在容器的特定位置添加元素

`push_back` 和 `push_front` 操作提供了方便的方式在顺序容器的头尾插入单个元素。更为通用的是 insert 成员允许我们在容器的任何位置插入零个或更多元素。vector 、 deque 、 list 和 string 支持 insert 成员。 `forward_list` 提供了这些成员的特别版本。

每个 insert 函数都以一个迭代器作为其第一个参数。这个迭代器表示元素将被插入的位置，它可以是容器中的任何位置，包括容器的尾后位置。由于迭代器可能指向尾后的不存在的元素，并且需要支持在容器的头部插入元素，元素将被插入到迭代器表示的位置之前。如：

```
slist.insert(iter, "Hello!");
```

将插入一个字符串到 iter 所表示的元素之前。

尽管有些容器没有 `push_front` 操作，insert 并没有这种限制。可以在容器的前面 insert 元素而不用担心容器是否支持 `push_front`：

```
vector<string> svec;
list<string> slist;
slist.insert(slist.begin(), "Hello!");
svec.insert(svec.begin(), "Hello!");
```

注意：在 vector 、deque 、string 的任何位置插入元素都是合法的，但是这样做是很耗时的操作。

插入元素范围

insert 除了第一个参数后的参数的模式与构造函数具有相同的参数模式。

其中一个版本 insert 以元素数目和值作为参数，将添加指定数目的同一元素到给定的位置：

```
svec.insert(svec.end(), 10, "Anna");
```

这段代码添加 10 个元素到 svec 的尾部，每个元素的值都是 “Anna” 。

还有一个版本的 insert 以一对迭代器或者一个初始值列表作为参数，将给定范围内的值插入到指定位置：

```
vector<string> v = {"quasi","simba","frollo","scar"};
slist.insert(slist.begin(), v.end()-2, v.end());
slist.insert(slist.end(), {"these","words","will","go","at","the","end"});
// 运行时错误：表示拷贝源的迭代器范围不能指向插入的容器，这是因为迭代器会失效
slist.insert(slist.begin(), slist.begin(), slist.end());
```

当以一对迭代器作为拷贝源时，这些迭代器一定不能指向将要插入元素的容器。

在新标准下 insert 成员会返回指向被插入的第一个元素迭代器，之前的版本没有返回值。如果范围是空的，没有元素被插入，并且返回 insert 函数的第一个参数。

使用 insert 的返回值

使用 insert 的返回值可以重复的插入元素到容器中指定的位置：

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word);
```

使用 emplace 操作

新标准加入三个新的成员 emplace_front 、 emplace 和 emplace_back ，这三个成员对应于 push_front 、 insert 和 push_back，只是它们不是拷贝元素而是直接构建。

当我们调用 emplace 成员时，传递给元素类型的构造函数的参数，并以这些参数直接直接构建一个对象放在容器中。如在 c 中置入 Sales_data 元素：

```
c.emplace_back("987-0590353403", 25, 15.99); //(1)
c.push_back(Sales_data("987-0590353403", 25, 15.99)); //(2)
```

(1) 是直接在容器中构建对象，(2) 则是先创建一个本地临时量，然后将其置入容器中。

传递给 emplace 函数的参数与元素类型有关。参数必须与元素类型的一个构造函数参数列表匹配：

```
c.emplace_back(); // 调用 Sales_data 的默认构造函数
c.emplace_back(iter, "999-999999999");
c.emplace_back("987-0590353403", 25, 15.99);
```

9.3.2 访问元素

下面的列表中列举了可以用于访问顺序容器中元素的操作，这些操作在容器中没有元素时是*未定义*的：

at 和 [] 是适用于 string 、 vector 、 deque 和 array，back 不适用于 forward_list

293. c.back() 返回指向容器 c 的最后一个元素的引用，如果 c 是空的结果将是未定义的；
294. c.front() 返回指向容器 c 的首元素的引用，如果 c 是空的结果将是未定义的；
295. c[n] 返回由无符号整数 n 索引的元素，如果 n >= c.size() 结果是未定义的；
296. c.at(n) 返回由无符号整数 n 索引的元素，如果索引超出了范围，将抛出 out_of_range 的异常；

注意：在空的容器中调用 front 或 back 或者在使用下标时超出范围，是非常严重的编程错误。

每个顺序容器都有 front 成员（包括 array），除了 forward_list 之外的顺序容器都有 back 成员，这些操作分别返回首元素和尾元素的引用，如：

```
if (!c.empty()) {
    auto val = *c.begin(), val2 = c.front();
```

```
    auto last = c.end();
    auto val3 = *(--last);
    auto val4 = c.back();
}
```

以上程序用两种不同的方式获取容器 c 的首元素和尾元素。需要注意的 end 返回的迭代器指向的是一个不存在的元素，在解引用之前需要先递减一。而且在获取元素之前，需要先检查 c 是不是空的，如果容器是空的，那么 if 内的操作就是未定义的。

访问成员返回的是引用

访问成员返回的是容器中元素的引用，如果容器是 const 对象，那么返回的引用也是 const 的。如果容器是非 const 的，那么引用就是常规引用，我们可以使用其来改变元素的值：

```
if (!c.empty()) {
    c.front() = 42;
    auto &v = c.back();
    v = 1024;
    auto v2 = c.back();
    v2 = 0;
}
```

用 auto 保存返回的引用，必须将 auto 定义引用类型。

下标操作和安全的随机访问

提供快速随机访问的容器（string、vector、deque 和 array）同时提供了下标操作。下标操作符用一个索引返回那个位置的元素的引用。索引必须是在安全范围内的（大于等于 0 并且小于容器的大小）。保证索引是合法的是程序的责任；下标操作不会检查索引是否在范围内。使用超出范围的索引是严重的编程错误，但是这个错误是编译器发现不了的。

如果想要保证索引是合法的，可以使用 at 成员。at 成员的行为与下表操作符类型，但是当索引是不合法的时候，at 将抛出 out_of_range 异常：

```
vector<string> svec;
cout << svec[0]; // 运行时错误，但是编译器发现不了
cout << svec.at(0); // 将抛出 out_of_range 异常
```

9.3.3 移除元素

正如有多种方式添加元素到容器中去，同样有多种方式移除元素。（array 不支持这些函数），以下是这些成员：

这些操作会改变容器的大小，所以 array 不支持这些操作。forward_list 有自己特殊的 erase 版本；pop_back 不适用于 forward_list；pop_front 不适用于 vector 和 string；

297. c.pop_back() 移除容器 c 中的尾元素，如果 c 是空的，结果将是未定义的，返回值是 void；
298. c.pop_front() 移除容器 c 的首元素，如果 c 是空的，结果将是未定义的，返回值是 void；
299. c.erase(p) 移除迭代器 p 指向的元素，并返回被删除元素之后的迭代器，如果 p 指向最后一个元素则返回尾后迭代器。如果 p 本身就是尾后迭代器，结果将是未定义的；
300. c.erase(b,e) 移除由迭代器 b 和 e 表示的范围中的元素，返回最后一个被删除的元素之后的迭代器，如果 e 本身就是尾后迭代器，返回尾后迭代器；
301. c.clear() 移除 c 中所有的元素，返回 void；

注意：除了在 deque 的首尾删除元素之外，其它位置的删除操作将会导致所有的迭代器、引用和指针失效。对于 vector 和 string，处于删除点之后的所有元素的迭代器、引用和指针将失效；

移除元素的成员不会检查其参数，保证元素在移除前存在是程序员的责任。

pop_front 和 pop_back 成员

pop_front 和 pop_back 分别移除首尾元素。就像 vector 和 string 没有 push_front 一样，它们也没有 pop_front 。类似的 forward_list 没有 pop_back 成员。就像元素访问成员一样，不要将 pop 操作用于空的容器。

这些操作返回 void，如果你需要被 pop 的值，你需要在 pop 之前先将其存储起来：

```
while (!ilist.empty()) {
    process(ilist.front());
```

```
    ilist.pop_front();
}
```

在容器的中间部分移除元素

erase 成员移除容器中特定位置的元素，可以删除由迭代器表示的单个元素或者由一对迭代器表示的元素范围。两种形式的 erase 都返回被移除的最后一个元素的之后的位置的迭代器。也就是说如果 j 是元素 i 之后的元素，那么 erase(i) 将会返回指向 j 的迭代器。

移除多个元素

以一对迭代器作为参数的 erase 版本允许我们删除范围内的元素，如：

```
elem1 = slist.erase(elem1, elem2); // elem2 将会失效
```

elem1 指向要删除的首元素，elem2 指向最后一个元素的下一个位置。

为了删除容器中的所有元素，可以调用 clear 或者将 begin 和 end 传给 erase ：

```
slist.clear();
slist.erase(slist.begin(), slist.end()); // 两者的作用是相等的
```

9.3.4 特定于 forward_list 的操作

为了理解为何 forward_list 有特别的添加和移除元素操作的版本，需要知道从单链表（singly linked list）中移除元素将会发生什么。如：

```
elem1 -> elem2 -> elem3 -> elem4
```

为了移除 elem3 必须改变 elem2 使得其 next 重新指向 elem4

当我们给单链表添加或删除元素时，被增加或删除的元素的之前那个元素的 next 改变了。所以，为了添加或移除元素，我们需要访问其前置元素，从而才能更新那个元素的 next 连接。然而，对于 forward_list 这种单链表来说很难获得一个元素的前置元素。基于这个原因，在 forward_list 中的添加或移除元素操作改变的是给定元素后面的那个元素。这样总能访问到由于改变而影响到的元素。

由于这些操作在 forward_list 上的不同表现，forward_list 没有定义 insert 、emplace 和 erase ，相反它定义 insert_after emplace_after 和 erase_after，如为了移除 elem3 ，我们可以在 elem2 的迭代器上调用 erase_after，为了支持这些操作，forward_list 定义了 before_begin 返回首前迭代器（off-the-beginning iterator）。这个迭代器允许我们在一个首元素之前的不存在的元素之后添加或移除一个元素。

forward_list 中插入和移除元素的操作：

302. `lst.before_begin()` `lst.cbefore_begin()` 返回一个迭代器表示链表首元素之前的不存在的元素，这个迭代器不能被解引用。 `cbefore_begin()` 返回一个 `const_iterator` ；
303. `lst.insert_after(p,t)` `lst.insert_after(p,n,t)` `lst.insert_after(p,b,e)` `lst.insert_afer(p,il)` 在由迭代器 p 表示的元素之后插入新的元素。t 是一个对象，n 是数目，b 和 e 是表示元素范围的迭代器（b 和 e 不能指向 lst），il 是一个括号中的值列表。返回插入的最后一个元素的迭代器。如果范围是空的，则返回 p，如果 p 是尾后迭代器，行为是未定义；
304. `emplace_after(p, args)` 使用 args 在 p 迭代器所表示的元素之后构建一个新的元素。返回新元素的迭代器，如果 p 是尾后迭代器，行为将是未定义的；
305. `lst.erase_after(p)` `lst.erase_after(b,e)` 移除 p 迭代器所表示的元素的后一个元素，或者 b 之后直到 e 但是不包括 e 范围内的元素。返回删除的元素之后的下一个元素的迭代器，如果不存在下一个元素则返回尾后迭代器。如果 p 表示链表中最后一个元素或者是尾后迭代器，行为将是未定义的；

当我们想要在 forward_list 中添加或移除元素时，我们需要记录两个迭代器，一个用于检查元素值 curr，一个是这个元素的前置迭代器 prev

9.3.5 resize 容器大小

除了 array 的容器可以使用 resize 操作来使得容器更大或更小。如果当前大小比请求的尺寸大的话，元素将从后面开始删除；如果当前大小比新的尺寸小的话，元素将被添加到容器的尾部，以下是顺序容器支持的改变大小的操作：

306. `c.resize(n)` 改变容器 c 的大小，使得其有 n 个元素，如果 `n < c.size()` 额外的元素将被删除。如果需要添加新的元素，它们将是值初始化的。
307. `c.resize(n,t)` 改变容器 c 的大小，使得其有 n 个元素，如果需要添加元素，其值将是 t ；

如果 resize 缩小了容器的大小，那么指向被删除的元素的迭代器、引用和指针将会失效；在 vector 、string 或者 deque 上调用 resize 将会使得所有的迭代器、指针和引用失效。

以下是一些实例：

```
list<int> ilist(10, 42);
ilist.resize(15); // 添加 5 个元素到 ilist 的尾部，其值为 0
ilist.resize(25, -1); // 添加 10 个元素到 ilist 的尾部，其值为 -1
ilist.resize(5); // 移除 ilist 尾部的 20 个元素
```

resize 操作以一个可选的元素值作为参数，用于初始化添加到容器中的元素。如果没有这个参数，添加的元素将是值初始化的。如果容器中的元素是类类型，那么当 resize 添加元素时，要么提供初始值要么元素类型有一个默认构造函数。

9.3.6 容器操作会使得迭代器失效

改变容器大小的操作将会使得迭代器、引用和指针失效。失效的迭代器、引用和指针将不再指向一个元素。使用失效的迭代器、引用和指针是一个严重的编程错误，这就像使用未初始化的指针引发的问题是一样的。

在添加元素到容器之后：

308. 如果 vector 或 string 重新分配了内存，迭代器、指针或引用将会失效。如果没有重新分配，插入位置之前的间接引用将保持有效；在插入点之后的元素的引用将会失效；
309. 在 deque 首尾位置之外的任何位置添加元素都将导致迭代器、指针和引用失效。如果在首尾添加元素，迭代器将失效，但是引用和指针将保持有效；
310. list 和 forward_list 的迭代器、指针和引用（包括尾后和首前迭代器）都将保持有效；

在移除一个元素之后，除了被移除的元素的迭代器、引用和指针失效之外：

311. list 和 forward_list 的所有其它元素的迭代器、引用和指针（包括尾后和首前迭代器）都将保持有效；
312. 除了在 deque 的首尾位置移除元素，所有其它迭代器、引用和指针都将失效。如果在 deque 的尾部移除元素，尾后迭代器将失效，但是其它迭代器、引用和指针将保持有效；如果是从首部移除元素，所有迭代器、引用和指针都将保持有效；
313. 在 vector 和 string 中，所有在移除点之前的迭代器、引用和指针都将保持有效。而尾后迭代器将总是失效；

注意：使用失效的迭代器、引用和指针是一个严重的运行时错误。

建议：管理迭代器

当使用容器元素的迭代器、引用或者指针，尽可能缩短需要迭代器、引用、指针保持有效的代码的范围。由于给容器添加或删除元素的代码会使得迭代器、引用和指针失效，需要在每次改变容器之后重新获取迭代器。这个建议对于 vector 、string 和 deque 特别重要。

书写改变容器大小的循环

给 vector 、string 和 deque 添加或移除元素的循环必须了解的一个事实是迭代器、引用和指针可能会失效。程序必须保证每次循环之后迭代器、引用和指针都会更新。如果循环调用的是 insert 或 erase 的话，更新迭代器将会很简单。这些操作都会返回迭代器，从而可以用于重置迭代器。

避免存储由 end 返回的迭代器

当在 vector 和 string 中添加和移除元素，或者在 deque 中添加元素或者移除非首元素时，其 end 返回的迭代器将总是失效。所以添加和移除元素的循环需要总是调用 end 而不是将获取到的迭代器存起来。部分由于这个原因，C++ 标准库将 end() 调用实现为一个很快的操作。

所以不要在会往 deque 、string 或 vector 中插入或删除元素的循环中缓存 end() 返回的迭代器。

9.4 vector 如何增长

为了支持快速随机访问，vector 的元素是连续存储的，每个元素都毗邻前一个元素。通常，我们不关心库类型是如何实现的；我们需要关心的仅仅是如何使用它。然而，对于 vector 和 string 其部分实现泄露到了接口中。

如果元素是连续的，而容器的尺寸是可变的，想象一下当添加元素到 vector 和 string 中会发生什么：如果没有用于存储新元素的空位置，容器必须要分配新的内存并将所有元素移动到新位置，并且添加新元素，然后释放掉旧的内存。如果 vector 每次添加元素分配和释放内存，那么性能将会很差。

为了避免这种消耗，库实现使用一种分配策略来减少容器需要重新分配内存的次数。当需要获取新的内存时，vector 和 string 实现通常会分配超出其所直接需要的内存。容器将保留这部分内存用于存储新添加的元素。这样容器就不需要每次添加一个新元素就重新分配内存。
这种分配策略比每次添加一个元素都重新分配更加高效。事实上，vector 的性能甚至比 list 或 deque 更加好，即便 vector 每次重新分配内存时都需要移动其所有元素。

管理容量的成员

vector 和 string 类型提供了一些成员用于与内存分配部分的实现进行交互。capacity 操作告知我们一个容器在需要分配更多内存之前可以容纳多少元素。reserve 操作则允许我们告知容器准备让它持有多少个元素。
`shrink_to_fit` 适用于 vector, string 和 deque。capacity 和 reserve 仅适用于 vector 和 string。

314. `c.shrink_to_fit()` 将 capacity() 减少到与 size() 一样；
315. `c.capacity()` 在重新分配内存之前 c 可以存储的元素数目；
316. `c.reserve(n)` 分配至少保存 n 个元素的内存；

reserve 不会改变容器中元素的数目；它仅仅影响 vector 预分配的内存的大小。

调用 reserve 只有在请求的空间大于当前容量时才会改变 vector 的容量。如果请求的大小待遇当前容量，reserve 将会分配至少相当于（甚至更多）请求的内存大小。
如果请求的大小小于或等于当前的容量，reserve 将不做任何事。特别是，如果 reserve 的参数所表示的大小比 capacity 小，不会导致容器收缩内存。所以，在调用 reserve 之后，capacity 将会大于等于传递给 reserve 的参数。
因而，调用 reserve 将永远不会减少容器使用的空间大小。而前面介绍的 resize 成员将仅仅改变容器中元素的数目，而不是其容量。我们不能使用 resize 来减少容器持有的内存。
在新标准下，可以使用 `shrink_to_fit` 来要求 deque, vector 或 string 归还未使用的内存。这个函数表明我们不需要超出范围的容量。然而，到底归不归还则由实现自己决定，并不保证 `shrink_to_fit` 会归还内存。

capacity 和 size

理解 capacity 和 size 之间的区别是很重要的。size 是容器中已经拥有的元素的数目，capacity 是这个容器在重新分配内存之前可以容纳多少元素。
当刚创建一个空的容器时，size 和 capacity 都是 0，随着不断添加元素，capacity 总是大于等于 size。容器只有在必须要分配内存时才会重新分配，而且似乎目前实现的策略是将容量扩大为原来的两倍，不过这是由实现决定的。
vector 只在 size 等于 capacity 时执行插入操作，或者调用 resize 或 reserve 时它们的参数值大于当前的 capacity 的情况下才会重新分配内存。至于分配多少内存则是由实现决定的。
每个实现都需要实现一种分配策略从而保证 `push_back` 元素到 vector 中是高效的。通过在一个原本是空的 vector 上调用 `push_back` n 次创建一个具有 n 个元素的 vector 的执行时间不应该超过一个常量乘以 n 。

9.5 额外的 string 操作

string 类型在共同的顺序容器操作之外还提供了很多额外的操作。其中的大部分操作要么是提供 sting 与 C 风格字符数组之间的交互，要么是提供使用索引而不是迭代器来操作 string 。
string 库提供了大量的函数，幸运的是，这些函数有着重复的模式。

9.5.1 构建 string 的其它方式

除了第三章中介绍构造函数以及前面介绍的与其它顺序容器共用的构造函数，string 类型还支持三个额外的构造函数。
n，len2 和 pos2 都是无符号值。

317. `string s(cp, n);` s 是 cp 所指向的字符串数组中的前 n 个字符的拷贝，这个数组至少要有 n 个字符；
318. `string s(s2, pos2);` s 是 string s2 从索引 pos2 开始的字符串的拷贝，如果 pos2 > s2.size() 结果将是未定义的；
319. `string s(s2, pos2, len2);` s 是 string s2 中从索引 pos2 开始最多 len2 个字符的拷贝，如果 pos2 > s2.size() 结果将是未定义的。不管 len2 的值是多少，最多拷贝 s2.size() - pos2 个字符；

以 string 或 const char* 为参数的构造函数，还会接收一个额外的参数来指定需要拷贝的字符。当我们传入 string 时，我们还需要指定开始拷贝的字符的索引，如：

```
const char *cp = "Hello World!!!";
char noNull[] = {'H', 'i'};
string s1(cp);
string s2(noNUll, 2);
string s3(noNull); // undefined: noNull not null terminated
string s4(cp + 6, 5);
string s5(s1, 6, 5);
string s6(s1, 6);
string s7(s1, 6, 20);
string s8(s1, 16); // out_of_range
```

通常如果我们从 const char* 创建一个 string，指针指向的字符数组必须是 NULL 结尾的；字符将被拷贝直到 NULL。如果同时传入 count ，数组不必必须以 NULL 结尾。如果没有传入 count 而数组没有以 NULL 结尾，或者给了 count 但是其值大于数组的长度，那么操作将是未定义的。

当我们从 string 拷贝时，可以提供额外的参数来指定开始拷贝的位置，以及一个 count。开始位置必须小于等于给定 string 的长度。如果起点大于长度，那么构造函数将抛出 out_of_range 异常。当传入 count，将从起点拷贝 count 个字符，不管 count 是多少，只会最多拷贝到 string 的结尾处，不可能更多字符。

substr 操作

substr 返回原始 string 的一部分或全部的拷贝 string。可以给 substr 传入可选的起点和计数器：

320. s.substr(pos, n); 返回 s 字符串中从 pos 位置开始的 n 个字符的 string 拷贝。pos 默认是 0，n 默认是从 pos 直到 string 结尾的总长度；

如：

```
string s("hello world");
string s2 = s.substr(0, 5);
string s3 = s.substr(6);
string s4 = s.substr(6, 11);
string s5 = s.substr(12); // 抛出 out_of_range
```

如果 pos 超出了 string 的长度，substr 函数将抛出 out_of_range 异常。如果 pos+count > s.size()，count 将被调整只拷贝到 string 的结尾。

9.5.2 改变 string 的其它方式

string 类型除了支持顺序容器的赋值操作符和 assign，insert 以及 erase 操作外。它自己还定义了额外的 insert 和 erase 版本。

除了定义接收迭代器的 insert 和 erase，string 提供了接收索引的版本。索引表示开始移除的元素，或者表示插入值到指定位置前。如：

```
s.insert(s.size(), 5, '!');
s.erase(s.size() - 5, 5);
```

string 还提供 insert 和 assign C 风格字符数组的版本。如：

```
const char *cp = "Stately, plump Buck";
s.assign(cp, 7);
s.insert(s.size(), cp+7);
```

首先将 s 的内容通过调用 assign 来替换，赋值给 s 的字符时从由 cp 开始的 7 个字符。请求的字符数必须小于等于从 cp 开始直接数组尾部的长度（不包括 NULL 字符）。

当我们调用 insert 时，我们是想在 s 的尾部（一个不存在的字符 s[s.size()] 之前）插入字符，在这种情况下我们从 cp 之后的第 7 个字符开始拷贝知道数组的 NULL 字符为止（不包括 NULL 字符）。

我们还以指定让 insert 或 assign 的字符来自于另外一个 string 或者它的子串：

```
string  s = "some string", s2 = "some other string";
s.insert(0, s2);
// 在 s[0] 之前插入从 s2[0] 开始的 s2.size() 个字符
s.insert(0, s2, 0, s2.size());
```

append 和 repalce 函数

string 类还定义了两个额外的成员 append 和 replace 来改变 string 的内容。append 操作是一种在字符串尾部插入字符的快捷方式：

```
string s("C++ Primer"), s2 = s;
s.insert(s.size(), " 4th Ed."); // (2)
s2.append(" 4th Ed."); // (3) 两种方式是相同的
```

replace 操作是调用 erase 和 insert 的快捷方式：

```
s.erase(11, 3);
s.insert(11, "5th");
s2.replace(11, 3, "5th");
```

replace 替换的 string 可以与移除的字符数目不一致，`s.repalce(11, 3, "Fifth");` 这里移除了三个字符串但是插入了五个字符。

改变 string 的多种重载方式

append, assign, insert 和 replace 函数有多个重载版本。这个参数随着指定添加字符和改变 string 的哪个部分的不同方式而改变。

replace 函数提供两种方式来指定要移除的字符范围。可以通过指定位置和长度来表示范围，或者通过迭代器范围。insert 函数提供了两种方式来指定插入点：用索引或者迭代器。在两种方式中都是新元素都被插入到给定索引或者迭代器之前。

有多种方式来指定添加到 string 中的字符。新的字符可以来自于另外一个 string，来自于字符指针或者来自于花括号中的字符列表，或者一个字符加计数器。当字符来自于 string 或字符指针时，可以传递额外的参数来告知是拷贝参数中的部分还是全部字符。

9.5.3 string 搜索操作

string 类提供了六个不同的搜索函数，每个有四个重载版本。以下是这些成员函数以及它们的参数，每个搜索都返回一个 `string::size_type` 来表明匹配发生的索引。如果没有匹配，函数将返回一个 static 成员 `string::npos`，库定义 npos 的值为 -1，由于 `string::size_type` 是无符号值，意味着 npos 将是最大的整数值。

搜索操作返回期待的字符的索引或者在没有找到的情况下返回 npos 。

321. `s.find(args)` 查找 args 在 s 中第一次出现的位置；
322. `s.rfind(args)` 查找 args 在 s 中最后一次出现的位置；
323. `s.find_first_of(args)` 查找 args 中任一字符在 s 中第一次出现的位置；
324. `s.find_last_of(args)` 查找 args 中任一字符在 s 中最后一次出现的位置；
325. `s.find_first_not_of(args)` 查找 s 中第一个不是在 args 中的字符；
326. `s.find_last_not_of(args)` 查找 s 中最后一个不是在 args 中的字符；

args 必须是其中之一：

327. `c, pos` 在 s 中从位置 pos 开始查找字符 c，pos 默认是 0；
328. `s2, pos` 在 s 中从位置 pos 开始查找 string s2，pos 默认是 0；
329. `cp, pos` 在 s 中从位置 pos 开始查找由指针 cp 指向的 C 风格字符串(以 NULL 结尾的字符数组)，pos 默认是 0；
330. `cp, pos, n` 在 s 中从位置 pos 开始查找由指针 cp 指向的字符数组的前 n 个字符。pos 和 n 都没有默认值；

string 搜索函数返回 `string::size_type`，这是一个无符号类型。因而，用 int 或者其它带符号的类型来接收返回值不是一个好的想法。

find 函数是最简单的，它在 string 中搜索参数，然后返回第一个匹配的索引，如果没有找到就返回 npos：

```
string name("AnnaBelle");
auto pos1 = name.find("Anna"); // pos1 == 0
```

string 的搜索操作是大小敏感的。

一个稍微复杂的问题是在 string 中查找参数字符串中的任何一个字符，如以下查找 name 中的第一个数字：

```
string numbers("0123456789"), name("r2d2");
auto pos = name.find_first_of(numbers);
```

与之相反，我们可以调用 find_first_not_of 来查找第一个不存在于搜索参数中的字符，如以下查找第一个非数字字符：

```
string dept("03714p3");
auto pos = dept.find_first_not_of(numbers);
```

指定从哪里开始搜索

可以传递额外的起始位置给 find 操作，这个额外的参数告知从哪里开始搜索。默认的情况下开始搜索的位置是 0。这个额外的参数常用于循环查找所有的匹配位置：

```
string::size_type pos = 0;
while ((pos = name.find_first_of(numbers, pos)) != string::npos) {
    cout << "found number at index: " << pos
         << " element is " << name[pos] << endl;
    ++pos; // 如果忽略了自增，将导致进入无限循环
}
```

发现搜索

find 操作是自左向右进行搜索的，库中还提供了类似的操作进行自右向左的搜索。rfind 成员查找最后一个也就是最右边的匹配，如：

```
string river("Mississippi");
auto first_pos = river.find("is");
auto last_post = river.rfind("is");
```

find_last_of 搜索参数中任一一个字符出现在 string 中的最后的位置；find_last_not_of 搜索不匹配搜索参数中任何字符的最后一个位置；

以上这些操作都有第二个参数用于表示在 string 中开始搜索的位置。

9.5.4 compare 函数

除了关系操作符，string 库还提供一系列 compare 函数，这些函数类似于 C 库中的 strcmp 函数。与 strcmp 一样，s.compare 返回 0 或者正数或者负数分别表示 s 等于,大于,小于其参数。

这六个版本的 compare 函数不同之处在于其参数是 string 或者字符数组，是比较全部还是部分：

331. `s.compare(s2);` 比较 string s 和 string s2；
332. `s.compare(pos1, n1, s2);` 比较 string s 中从 pos1 开始的 n1 个字符和 string s2；
333. `s.compare(pos1, n1, s2, pos2, n2)` 比较 string s 中从 pos1 开始的 n1 个字符和 string s2 中从 pos2 开始的 n2 个字符；
334. `s.compare(cp)` 比较 string s 和 cp 所表示的 C 风格字符串；
335. `s.compare(pos1, n1, cp);` 比较 string s 中从 pos1 开始的 n1 个字符和 cp 所表示的 C 风格字符串；
336. `s.compare(pos1, n1, cp, n2);` 比较 string s 中从 pos1 开始的 n1 个字符和 cp 所表示的 C 风格字符串中的前 n2 个字符；

9.5.5 数字转换

string 中经常包含表示数字的字符。在新标准中引入了多个函数可以在数字与 string 之间进行转换：

```
int i = 42;
string s = to_string(i);
double d = stod(s);
```

337. `to_string(val)` 返回 val 的字符串表现形式，val 可以是任何算术类型；
338. `stoi(s,p,b) stol(s,p,b) stoul(s,p,b) stoll(s,p,b) stoull(s,p,b)` 如果 s 的最开始的部分是数字，将它转换为 int, long, unsigned long, long long, unsigned long long。b 表示进制；b 默认是 10 进制；p 是一个 size_t 类型值的指针，用于存储第一个非数字字符的索引；p 默认是 0，表示不存储这个索引；
339. `stof(s,p) stod(s,p) stold(s,p)` 将 s 最开始部分的数字内容转为 float, double 或 long double。p 用于存储非数字字符的索引；

为了将 string 转为数字，其第一个非空白符字符必须可以存在于数字中：

```
string s2 = "pi = 3.14";
d = stod(s2.substr(s2.find_first_of("+-.0123456789")));
```

find_first_of 先获取第一个可以出现在数字中的字符位置，然后传递那个位置开始的子串给 stod，std 函数读取 string 直到找到一个不是数字一部分的字符，然后将前面部分的数字字符转为一个双精度浮点数。
如果是十六进制的数，string 可以以 0x 或 0X 开始，转为浮点数的 string 可以以小数点开始，并且可能包含 e 或 E 表示指数。根据基数的不同，string 中可以包含对应的字母用于表示超出 9 的数字。
注意：如果 string 不能转为数字，这些函数将抛出 invalid_argument 异常，如果转换出来的值不能被表示，将抛出 out_of_range 异常。

9.6 容器适配器

除了顺序容器，标准库还定义了三个顺序容器适配器(adaptor)：stack, queue 和 priority_queue。适配器是标准库中的通用概念。其中包含容器适配器、迭代器适配器和函数适配器。本质上，适配器是一种使得让一个物件表现得像另外一个物件的机制。容器适配器（container adaptor）使得一个现存的容器类型在行为上看起来像是一个不同的类型。如 stack 适配器让一个顺序容器（非 array 或 forward_list）的行为看起来好像是一个 stack。
以下是所有的容器适配器共有的操作和类型：

340. size_type 足够容纳此类型中最大对象的尺寸类型；
341. value_type 元素类型；
342. container_type 实现适配器的底层容器类型；
343. A a; 创建一个空的适配器 a；
344. A a(c); 创建适配器 a，其内容是容器 c 的拷贝；
345. 关系操作符，每个适配器都支持所有的关系操作符：== != < <= > >= 这些操作符返回底层容器的比较结果；
346. a.empty() 如果 a 有任何元素返回 false，否则返回 true；
347. a.size() a 中元素的数目；
348. swap(a,b) a.swap(b) 交换 a 和 b 中的内容；a 和 b 必须是相同的类型，包括其底层实现的容器类型；

定义一个适配器

每个适配都定义了两个构造函数：默认构造函数以创建一个空的对象，以及接收一个容器的构造函数并将适配器初始化为给定容器的拷贝。如假设 deq 是 deque<int> 可以将 deq 用于初始化 stack：

```
stack<int> stk(deq);  // 将 deq 中的元素拷贝到 stk 中
```

默认情况下 stack 和 queue 都是将 deque 作为底层实现的，而 priority_queue 将 vector 作为底层实现。可以在创建适配器时将一个顺序容器作为第二个类型参数从而创建不同类型的适配器：

```
stack<string, vector<string>> str_stk;  // 以 vector 创建空的 stack
// 以 vector 创建 stack，此 stack 将 svec 中的元素值拷贝作为初始值；
stack<string, vector<string>> str_stk2(svec);
```

对于哪些容器可以被用于一个给定的适配器是有限制的。所有的适配器都需要有添加和移除元素的能力。因而不能将 array 用于适配器，同样不能使用 forward_list ，原因是所有适配器都需要能够添加、移除和访问容器中最后一个元素。stack 只需要 push_back, pop_back 和 back 操作，所以可以将所有的剩下的容器类型用于 stack。queue 适配器需要 back, push_back, front 和 push_front 操作，所以它可以建立在 list 或 deque 上而不能是 vector。priority_queue 除了 front, push_back 和 pop_back 操作以及快速随机访问，所以它可以建立在 vector 或 deque 上，而不能在 list 上。

stack 适配器

stack 类型定义在 stack 头文件中。以下是 stack 提供的操作：
stack 将 deque 作为默认的底层实现；同样可以实现在 list 或 vector 之上。

349. s.pop() 移除但是不返回 stack 的顶部元素；
350. s.push(item) s.emplace(args) 在 stack 的顶部创建一个新的元素，这个元素要么是拷贝或移动 item 或者直接从 args 中构建此元素；
351. s.top() 返回但是不移除 stack 的顶部元素；

实例如：

```
stack<int> intStack;
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix);
while (!intStack.empty()) {
    int value = intStack.top();
    intStack.pop();
}
```

虽然每个容器适配器的所有的操作都是用底层容器类型的操作进行定义的。但是我们只能使用适配器的操作，而不能使用底层容器类型的操作，如：`intStack.push(ix);` 调用 deque 对象的 `push_back` 。尽管 stack 是使用 deque 实现的，却不能直接使用 deque 的操作，即不能在 stack 上调用 `push_back`；相反，必须使用 stack 的 push 操作。

queue 适配器

queue 和 `priority_queue` 适配器都定义在 queue 头文件中。以下是所有操作：

默认情况下 queue 使用 deque，`priority_queue` 使用 vector；queue 可以使用 list 或 vector 作为底层容器类型，`priority_queue` 可以使用 deque。

352. `q.pop()` 移除但是不返回，queue 首元素或者 `priority_queue` 最高优先级的元素；
353. `q.front()` `q.back()` 返回但是不移除 q 的首元素或尾元素，只有 queue 支持这两个操作；
354. `q.top()` 返回但是不移除最高优先级的元素，只有 `priority_queue` 支持此操作；
355. `q.push(item)` `q.emplace(args)` 在 queue 的尾部创建一个元素，其值是 item 或者直接以 args 为参数调用构造函数构造一个元素，对于 `priority_queue` 则是在队列的合适的位置；

queue 类使用先进先出（FIFO）的存取策略。进入队列的对象被放在尾部，对象从队列的首部开始移除。`priority_queue` 让我们在元素之间建立一个优先级，新添加的元素被放在所有更低的优先级的元素之前。默认情况下，库使用小于操作符来比较元素类型。

总结

标准库中的容器都是模板类型，用来保存给定类型的对象。顺序容器的元素按照他们位置进行排序和访问。顺序容器共享一个共同的标准的接口：如果两个顺序容器提供了一个特定操作，那么在两个顺序容器中这两个操作的接口和函数都一样。

对于大多数人来说，容器定义了特别少的操作。容器定义了构造函数，用于添加和移除元素的操作，已经返回容器的大小的操作，返回特定元素的迭代器的操作。其它的操作，如搜索和排序都没有定义在容器类型中，而定定义在标准算法中。

关键术语

356. 适配器（adaptor）：一种标准库中的类型、函数或迭代器，将另外一种类型、函数或迭代器包装的像是另外一种类型、函数或迭代器。下一章会介绍迭代器的适配器；
357. array ：一种长度固定的顺序容器。想要定义 array 必须同时给定元素类型以及长度。array 中的元素可以通过他们的位置索引进行访问，支持元素的快速随机访问；
358. begin ：返回指向容器中首元素的迭代器的容器操作，如果容器是空的，返回尾后迭代器。返回的迭代器是否为 const 取决于容器本身的类型；
359. cbegin ： 行为如 begin 一样，不过返回的是 `const_iterator` 类型；
360. end ： 返回指向容器中尾元素的下一个位置的迭代器，返回的迭代器是否为 const 取决于容器的类型；
361. cend ： 行为与 end 一样，不过返回的是 `const_iterator` 类型；
362. 迭代器范围（iterator range）：由一对迭代器表示的元素范围。第一个迭代器表示序列中的首元素，第二个迭代器表示尾元素的下一个位置。如果范围是空的，那么两个迭代器相等（相反的，如果迭代器不相等意味着范围不为空）。如果范围不为空，那么可以通过重复的递增第一个迭代器，从而与第二个迭代器相等。通过递增迭代器，序列中的每个元素都可以被处理；
363. 左包含区间（left-inclusive interval）：一个值范围其中包含其首元素，但不是尾元素。形如 `[i,j)` 表示一个序列从 i 并且包含 i 直到 j 但是不包含 j；
364. 首前迭代器（off-the-beginning iterator）：在 `forward_list` 中表示一个在首元素之前的一个不存在元素的迭代器。由 `forward_list` 的 `before_begin` 返回，与 end() 迭代器一样，它不可以被解引用；

365. 尾后迭代器（off-the-end iterator）：表示序列中最后一个元素的下一个位置的迭代器，通常也被称为尾部迭代器；

标准库容器定义了相当少的操作，与其添加无数的功能到每一个容器中，标准库提供了一系列算法，其中大部分独立于任何特定的容器类型。这些算法是通用的（generic）：它们在不同类型的容器和元素上进行操作。

本章的内容主要包括通用算法（generci algorithms）和更加详细的介绍迭代器。

顺序容器定义了简约的操作：其中大部分用于添加移除元素，访问首尾元素，判断容器是否为空，以及获取首元素和尾后元素的迭代器。

我们可以想象还需要其它有用的操作：查找特定的元素，替换或移除特定值，重排序容器中的元素。

与其将这些操作定义为每个容器类型的成员，标准库定义了一系列通用算法："算法"意思是它们实现了常见的经典算法如排序和搜索，"通用"是由于它们操作与不同类型的元素以及跨越多种容器类型——不仅仅是标准库中的类型如 vector 和 list，还包括内置数组类型以及其它自定义的序列（sequences）类型。

10.1 概述

绝大部分算法定义在 algorithm 头文件中。标准库还在 numeric 头文件中定义了一小撮通用数字算法（generic numeric algorithms）。

通常来说，算法并不直接在容器上进行工作，而是通过遍历由两个迭代器组成的元素范围进行操作。当算法遍历整个范围时，它会对每个元素做一些事情。如要查找容器中的特定元素值最简单的方式是调用 find 算法：

```
int val = 42;
auto result = find(vec.cbegin(), vec.cend(), val);
```

find 的前两个参数是两个迭代器，它们组成了一个左包含的元素范围，第三个参数是一个值。find 将给定范围中的每个元素与待查找的值进行比较。它返回第一个等于这个值的第一个元素的迭代器。如果没有找到，find 将返回其第二个参数来表示失败。因而，我们可以通过比较返回值是否与第二个迭代器参数相等来判断是否找到。

由于 find 仅对迭代器操作，可以将 find 用于任何的容器类型。如将 find 用于 list<string>：

```
string val = "a value";
auto result = find(lst.cbegin(), lst.cend(), val);
```

由于内置数组中的指针与迭代器的行为非常类似，可以将 find 用于数组：

```
int ia[] = {27,210,12,47,109,83};
int val = 83;
int* result = find(begin(ia), end(ia), val);
```

此处值得注意的是调用了库函数 begin 和 end 来获取 ia 的首元素指针和尾后元素指针。

通过传递子范围的首元素和尾后元素的迭代器（或指针）从而仅对容器的子范围进行查找。如：

```
// 搜索元素从 a[1] 开始直到但不包括 ia[4]
auto result = find(ia+1, ia+4, val);
```

算法如何工作

通过考察 find 算法来了解算法是如何运用于不同的容器类型的。find 可以在未排序的一系列元素中查找特定的元素。以下是它采取的步骤：

366. 访问序列的首元素；
367. 将其与给定值进行比较；
368. 如果这个元素匹配给定的值，find 将返回迭代器或者指针来标识这个元素；
369. 否则 find 继续查找下一个元素，重复步骤 2 和 3；
370. find 必须在到达序列的末尾时结束；
371. 如果 find 到了序列的末尾，需要返回一个值来表示元素没有找到。返回的值需要与第 3 步中的类型相兼容；

以上所有操作都没有依赖于容器的类型，只要能够使用迭代器来访问元素，find 根本不需要依赖于容器的类型。

迭代器使得算法与容器互相独立

除了第二步之外的所有步骤都可以通过迭代器进行操作：迭代器解引用可以访问元素；如果找到了匹配的元素，find 将返回那个元素的迭代器；迭代器的自增操作将移动到下一个元素；“尾后”迭代器表示 find 已经到达了给定序列的尾部；find 返回尾后迭代器用于表示没有找到指定的值；

但是算法依赖于元素类型的操作

尽管迭代器使得算法和容器类型相互独立，大部分的算法使用一个或多个元素类型的操作。如：第 2 步用元素类型的 == 操作符比较每个元素和给定值。

其它的算法需要元素类型由 < 操作符。然而，绝大部分的算法还提供了一种方式允许我们提供自己的操作用于替换默认的行为。

关键概念：算法永不执行容器的操作

通用算法自己不会执行容器的操作。它们只会操作迭代器。算法不直接调用容器的成员函数有一个重要的隐喻：算法永远不会改变底层容器的长度。算法可能会改变元素的值，可能会在容器中移动元素，但是它们不会直接添加或移除元素。

我们将在后面看到，有一种特殊类别的迭代器——插入器（inserter），除了遍历序列之外。当我们给这个迭代器赋值时，它将在底层容器中执行插入操作。当算法操作在这种迭代器上时，迭代器具有加元素到容器中的效果。但是算法本身不会插入元素，甚至它都不知道插入元素这回事。

10.2 算法入门

标准库提供了超过 100 个算法。幸运的是算法有固定的结构。理解结构使得学习和使用算法比记住 100 个以上的算法要容易。在本章，我们将描述如何使用算法，并且讲解它们的组织原则。附录 A 中按照算法的操作进行分类列举。

除了一些小的例外，算法操作于一组元素。我们称这个范围为“输入范围”（input range）。带输入范围的算法总是用其前两个参数来表示范围。这两个参数分别表示要处理的首元素和尾后元素。

尽管绝大部分算法都处理一组输入范围，它们操作元素的方式不一样。理解算法的最基本的方法是直到它们是读元素、写元素还是对元素进行重排序。

10.2.1 只读算法

有很多算法对输入范围只读不写。find 和 count 算法就是一个例子。另外一个只读算法是 accumulate，它定义在 numeric 头文件中。accumulate 函数有三个参数，前两个指定输入范围，第三个参数是综合的初始值。假设 vec 是一个整数序列，那么：

```
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

第三个参数的类型决定了使用哪个加操作，并且是 accumulate 的返回值类型。

算法和元素类型

accumulate 使用其第三个参数作为求和的起点有一个重要的隐喻：元素类型必须与总和的类型可以相加。也就是说元素的类型必须与第三个参数的类型相匹配或者可以相互转换。在这个例子中 vec 中的元素类型可以是 int，double，long long 或者其它可以与 int 相加的类型。

另外一个例子是 string 有加操作符，可以通过调用 accumulate 将 vector 中的元素拼接起来：

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

这个调用 v 中的每个元素拼接到一个 string 上，这个 string 最开始是空字符串，请注意这里需要显示创建 string 作为第三个参数，如果传递的是空字符串字面量将会是编译时错误：

```
// error: const char* 上没有加操作符
string sum = accumulate(v.cbegin(), v.cend(), "");
```

如果传递 string 字面量，存储 sum 的对象类型将是 const char*，这个类型决定了使用哪个加号操作符。由于 const char* 上没有加操作符，这个调用将无法通过编译。

通常在只读算法上最好使用 cbegin() 和 cend()，如果打算使用返回的迭代器来改变元素的值就要传递 begin() 和 end() 。

在两个序列上进行操作的算法

另外一个只读算法是 equal，用于判断是否两个序列中的值完全一样。它将第一个序列中的每个元素与第二个序列中对应位置的元素进行比较，如果所有的对应位置的元素都相等，返回 true，否则返回 false。这个算法有三个迭代器：前两个表示第一个序列的范围，首元素和尾后迭代器；第三个是第二个序列的首元素：

```
// roster2 至少要有与 roster1 一样多的元素
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

由于 equal 在是迭代器上定义的，可以将 equal 用于比较不同类型的容器中的元素。甚至，元素类型也不需要完全一致，只要可以使用 == 进行元素值比较就行。如，roster1 可以是 `vector<string>`，roster2 可以是 `list<const char*>` 。

然而，equal 做出了一个严格的假设：它假设第二个序列至少跟第一个序列一样长。这个算法潜在地会遍历第一个序列中的所有元素。它假设第二个序列中一定会有对应的元素。

警告：以单个迭代器表示第二个序列的算法会*假设*第二个序列至少跟第一个序列一样长。

10.2.2 写容器元素的算法

一些算法会给序列中的元素赋予新值。当使用会对元素赋值的算法时需要注意，必须保证算法写入的序列其大小大于等于需要写入的数目。记住，算法不会直接调用任何容器的操作，所以它们本身没有任何办法改变容器的大小。

其中一部分算法只会对输入范围内的元素进行写入，它们没有上面谈到的危险，原因是它们只会对输入范围内的元素进行写入。

比如，fill 以一对迭代器表示范围，以及第三个参数表示要写入的值。fill 将给定值赋值给范围内的所有元素。如：

```
fill(vec.begin(), vec.end(), 0);
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

由于 fill 只会往给定的输入序列中写入值，只要传入的是合法的输入序列，那么写入将是安全的。

关键概念：迭代器参数

有些算法同时从两个序列中读取元素，构成这些序列的的元素可以被存储在不同类型的容器中。如第一个序列可能被存储在 vector 中，第二个序列则存储在 list 或者 deque ，内置数组或者别的类型中。甚至，两个列中的元素类型都不需要完全匹配。只要能够比较两个序列中的元素。比如，在 equal 算法中，元素类型不需要完全一致，但是需要可以使用 == 来比较两个序列中的元素。

操作两个序列的算法的不同之处在于如何传递第二个序列。有一些算法比如 equal，有三个迭代器：前两个表示第一个序列的范围，第三个迭代器表示第二序列的首元素。其它的算法以四个迭代器为参数：分别表示两个序列的范围。

使用一个迭代器表示第二个序列的算法假设第二个序列至少有第一个序列那么长。由程序员保证算法不会访问第二个序列中不存在的元素。比如 equal 有可能会将第一个序列中的每个元素与第二序列中的对应位置的元素进行比较。如果第二个序列是第一个序列的子序列，那么将会产生很严重的错误 —— equal 将会访问超出第二个序列尾部的元素。

算法不会检查输出操作

有些算法只有一个单独的迭代器表示写入目标。这些算法将新值赋值到以目标迭代器 (destination iterator) 表示的元素开始的序列的元素中。 `fill_n` 以一个迭代器，计数器和值作为参数。它将这个给定值赋值到从迭代器所表示的元素开始的指定数目的元素中去。如：

```
vecotr<int> vec;
fill_n(vec.begin(), vec.size(), 0);
```

`fill_n` 假设写入指定数目的元素是安全的。这是因为在调用 `fill_n(dest, n, val)` 中，`fill_n` 假设 dest 表示一个元素，并且从 dest 开始至少有 n 个元素。

对于初学者来说，在空容器上调用 `fill_n` （以及其它写入的算法）是一个常见的错误：

```
vector<int> vec;
// 试图写入 10 个不存在的元素是严重的错误
fill_n(vec.begin(), 10, 0);
```

这个调用想要写入 10 个元素，但是容器中根本没有元素。结果是未定义的。

警告： 写入到目标迭代器的算法假设目标序列足以容纳将要写入的元素。

介绍 back_inserter

有一种保证算法有足够的元素用以写入，那就是使用插入迭代器（insert iterator）。插入迭代器是往容器中添加元素的迭代器。平常，当我们给迭代器赋值时，我们是给迭代器所表示的元素赋值。当我们给插入迭代器赋值时，一个等于右边值的新元素被添加到容器中。

插入迭代器将在后面更为详细的介绍，现在我们先使用 back_inserter 来举几个例子，back_inserter 是定义在 iterator 头文件中的函数。
back_inserter 以容器引用为参数，返回一个绑定到容器上的插入迭代器。当通过这个迭代器给元素赋值时，将调用容器的 push_back 函数给容器添加元素。如：

```
vector<int> vec;
auto it = back_inserter(vec);
*it = 42;
```

我们经常用 back_inserter 创建的迭代器用作算法的目的迭代器。如：

```
vector<int> vec;
fill_n(back_inserter(vec), 10, 0);
```

在每次迭代时，fill_n 给序列中的元素赋值，由于我们传入的是 back_inserter 的返回值，每次赋值都将调用 vec 的 push_back ，因而每次赋值 fill_n 都会添加一个元素到容器的尾部。

拷贝算法

copy 算法是另一个将值写入到由目的迭代器表示的输出序列中的例子。这个算法只有三个参数。前两个表示输入序列；第三个表示输出序列的首元素。这个算法将输入范围内的元素拷贝输出序列中。有一点很重要的是传递给 copy 的输出序列至少要和输入序列一样长。如将值拷贝内置数组中去：

```
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)];
auto ret = copy(begin(a1), end(a1), a2);
```

copy 的返回值是自增后的目的迭代器。意味着 ret 将指向 a2 的尾后位置。
有多个算法提供了“拷贝”版本，相比于将计算后的值重新存入输入序列中，这个算法创建一个新的序列以容纳结果。比如 replace 算法接收四个参数：两个迭代器表示输入序列，以及两个值。它将序列中的每个等于第一个值的元素替换为第二个值。如：

```
replace(ilist.begin(), ilist.end(), 0, 42);
```

上面的调用将所有的 0 替换为 42。如果想要保持原始的容器不变的话，需要调用 replace_copy ，这个算法有第三个迭代器参数表示输出目的地。如：

```
replace_copy(ilist.cbegin(), ilist.cend(), back_inserter(ivec), 0, 42);
```

在此调用之后 ilist 将保持不变，而 ivec 的元素将是 ilist 拷贝，并将其中所有的 0 替换为 42。

10.2.3 对容器元素进行重排序的算法

有些算法对容器中的元素进行重排序，一个显著的例子就是 sort 算法。调用 sort 将使用元素的 < 操作符将输入范围内的元素排序。

消除元素

一个例子是消除容器中的相同的字符串。首先对这些字符串进行排序，然后调用 unique 将所有唯一的字符串放到容器的首部，并返回最后一个唯一字符串的下一个位置的迭代器。unique 本身是不改变容器的大小的，所以需要用容器的 erase 成员移除元素：

```
void elimDups(vector<string> &words)
{
    sort(words.begin(), words.end());
    auto end_unique = unique(words.begin(), words.end());
    words.erase(end_unique, words.end());
}
```

调用完 unique 之后，返回值迭代器后面的值到底是什么我们无法知道，它可能已经被算法给改写了。由于标准库算法是在迭代器而不是容器上进行操作，算法不能直接添加或移除元素。

使用容器操作来移除元素

为了移除无用的元素，必须使用容器的操作。

10.3 定制操作

大多数算法会比较容器中的元素。默认情况下算法使用的是 < 或 == 操作符。算法还定义了允许我们提供自己的操作来替换默认操作符的版本。
比如对于 sort 算法使用的是 < 操作符。可能我们的序列并不是按照 < 进行排序的，或者有些类型根本就没有 < 操作符，在这两种情况下都需要重载默认的 sort 行为。

10.3.1 传递函数给算法

对于字符串序列进行排序可以是先按照长度进行排序，对于长度一样的再按照字典顺序进行排序。为了这样做需要使用第二个版本的 sort，这个版本的 sort 有第三个参数称之为谓词（predicate）。

谓词

谓词是一个可以被调用然后返回一个值的表达式，返回值可以作为条件使用。标准库使用的谓词可以是一元谓词（unary predicate）（意思是只有一个参数）也可以是二元谓词（binary predicate）（意思是有两个参数）。带有谓词的算法在输入范围内的元素上调用这个谓词。因而，必须要可以将元素类型转为谓词的参数类型。

使用二元谓词的 sort 版本将给定谓词替换 < 操作用于比较元素。提供给 sort 算法的谓词必须满足 §11.2.2 中描述的限制，现在我们只需要知道这个操作必须在每一个元素之间提供稳定的顺序。单纯比较字符串的长度就是其中一个例子。如：

```
bool isShorter(const string &s1, const string &s2) {
    return s1.size() < s2.size();
}
sort(words.begin(), words.end(), isShorter);
```

这样所有的字符串就是按照长度进行排序的了。

排序算法

当使用长度进行排序时，我们依然希望保持相同长度的元素之间的字典排序。为了达到这样的效果，我们将使用 stable_sort 算法。稳定排序（stable sort）将保持原来相等的元素之间的顺序。

通常，我们并不关心相等元素之间的相对顺序，毕竟它们都是相等的。然而，现在的情况是我们定义相等是按照长度来定义的。相同长度的字符串的内容依然是有区别的。通过调用 stable_sort 可以保持相同长度的字符串的字典顺序：

```
elimDups(words);
stable_sort(words.begin(), words.end(), isShorter);
```

10.3.2 lambda 表达式

传递给算法的谓词必须有一个或两个参数。但是有时我们想传递多于算法的谓词需要的参数。为了查找字符串序列中大于等于给定长度的字符串，我们写了一个函数：

```
void biggies(vector<string> &words, vector<string>::size_type sz)
{
    elimDups(words);
    stable_sort(words.begin(), words.end(), isShorter);
    auto wc = find_if(words.begin(), words.end(), [sz](const string &a) {
        return a.size() >= sz;
    });
}
```

我们使用 find_if 标准库算法来查找一个元素大于等于给定长度的。find_if 接收一对迭代器表示输入范围，不同于 find 的是它接收的第三个参数是谓词。find_if 算法在每个元素上调用给定的谓词。它将返回第一个使得谓词的返回值为非 0 值的元素迭代器，或者当无法找到这样的元素时返回尾部迭代器。由于 find_if 接收的是一元谓词，任何传递给 find_if 的函数必须只有一个参数。所以想要传递可变的 size 参数给谓词必须使用 lambda 表达式。

介绍 lambda

我们可以传递任何可调用对象（callable object）给算法。如果可以给对象或表达式运用调用操作符，它就是可调用的（callable）。也就是说如果 e 是可调用的表达式，那么可以写作 e(args) 其中 args 是一个逗号分隔的零个或多个参数的列表。

目前我们使用的可调用对象就是函数和函数指针。还有两种可调用对象：重载了调用操作符的类将在 §14.8 中介绍，以及现在将要介绍的 lambda 表达式（lambda expressions）。

lambda 表达式表示可调用的代码单元。可以被视作匿名内联函数。与任何函数一样，lambda 有返回类型，参数列表和函数体。与函数不同的是，lambda 可以被定义在函数中。其有如下形式：

```
[capture list](parameter list) -> return type { function body}
```

其中捕获列表（capture list）（经常是空的）表示定义在外围函数中的一系列本地变量。返回值，参数列表和函数则与普通函数是一样的。然而，与普通函数不同的是，lambda 必须使用尾部返回（trailing return）来表示返回类型。

可以忽略参数列表和返回类型，但是必须总是包含捕获列表和函数体：

```
auto f = [] { return 42; };
```

我们将 f 定义为一个不接受任何参数返回 42 的可调用对象。使用 lambda 的方式与任何函数调用都是一样的。如：

```
cout << f() << endl;
```

省略 lambda 中的括号和参数列表表示其参数列表为空。如果省略返回值类型，lambda 通过函数体中语句进行推断。如果函数体中只有一个 return 语句，那么返回类型就是被返回的表达式的类型。否则，返回类型就是 void。

注意：lambda 中的函数体如果包含了除了返回语句之外的任何语句，都将被推断为返回 void。

传递参数给 lambda

与常规的函数调用一样，lambda 调用中的实参也是用于初始化其形参的。实参和形参的类型必须匹配。与常规函数不同的是，lambda 没有默认实参。因而，调用 lambda 给足实参。一旦形参被初始化，函数体就开始执行。

如下 lambda 需要传递参数：

```
[](const string &a, const string &b) {
    return a.size() < b.size();
}
```

空的捕获列表表示不使用任何外围函数的本地变量。lambda 的参数与常规函数的参数一样是 const string 的引用。我们可以重写 stable_sort 的调用从而使用 lambda：

```
stable_sort(words.begin(), words.end(),
    [](const string &a, const string &b) {
        return a.size() < b.size();
    });
```

当 stable_sort 需要比较元素值时，它将调用给定的 lambda 表达式。

使用捕获列表

注意上面使用的 find_if 中的 lambda 表达式中的捕获列表中的 sz。尽管 lambda 被定义在函数中，只有被指定了想要使用的外围函数本地变量才能使用。lambda 通过捕获列表来指定想要使用的变量。捕获列表中包含了 lambda 用于访问外围函数本地变量的信息。

这样 find_if 使用的 lambda 表达式捕获了 sz，并且只有一个参数，函数体中将给定 string 的长度与捕获值 sz 进行比较：

```
[sz](const string &a) {
    return a.size() >= sz;
};
```

在 [] 中是逗号分隔的捕获列表，里边的名字是外围函数中定义的本地变量。由于这个 lambda 捕获了 sz，函数体就可以使用 sz，而没有捕获 words 就不能访问此变量。

注意：lambda 只能使用出现在捕获列表中的外围函数本地变量。

for_each 算法

for_each 算法有一个可调用对象并在输入范围内的每个元素上调用此对象。如：

```
for_each(wc, words.end(), [](const string &s) { cout << s << " "; });
```

注意：这里并没有捕获 cout，捕获列表只用于捕获外围函数中定义的非静态变量。lambda 可以自由使用定义在函数外的变量，此处 cout 并不是一个本地变量；这个名字定义在 iostream 头文件中。只要包含了 iostream 头文件，这个 lambda 就可以使用 cout 。

捕获列表只用于捕获外围函数中定义的非静态变量；lambda 可以自由使用本地 static 变量或者定义在函数体外的变量。

10.3.3 lambda 捕获和返回

当我们定义 lambda 时，编译器为此 lambda 产生一个匿名的新类类型。在 14.8.1 节我们将看到这个类是怎样的。现在只需要知道当我们定义一个 lambda 时，我们同时定义了一个新类型和这个类型的对象：这个由编译器生成的类的匿名对象。

默认情况下，从 lambda 中生成的类包含有数据成员，这些数据成员与捕获列表中的变量一一对应。当 lambda 对象创建时其数据成员被初始化。

值捕获

与参数传递一样，可以通过值或引用捕获变量。目前我们使用到的是值捕获。值捕获中的变量必须是可拷贝的。与参数不同的是，捕获变量是在 lambda 创建时拷贝的，而不是在调用时：

```
void fcn1()
{
    size_t v1 = 42;
    auto f = [v1] {return v1;};
    v1 = 0;
    auto j = f(); // j 是 42；在创建 f 是拷贝了 v1 的值；
}
```

由于在创建 lambda 时已经拷贝了值，因而接下来对捕获变量的修改不会影响 lambda 中对应的值。以下是所有的捕获方式：

(2) `[]` 空的捕获列表。lambda 不适用外围函数中的变量；j

(3) `[names]` names 是逗号分隔的捕获列表。默认情况下，变量是值捕获的。在名字前加上 & 就是引用捕获；

(4) `[&]` 隐式引用捕获列表。在 lambda 函数体中所使用的外围函数的本地变量都被隐式地引用捕获；

(5) `[=]` 隐式值捕获列表。在 lambda 函数体中所使用的外围函数本地变量都被隐式地值捕获；

(6) `[&,identifier_list]` identifier_list 是逗号分隔的外围函数本地变量列表，这些变量是值捕获的，并且不能在名字前加上 &，其它的使用到外围本地变量是隐式引用捕获的；

(7) `[=,reference_list]` reference_list 是逗号分隔的外围函数的本地变量引用列表，这些变量是引用捕获的，所以必须在名字前加上 &，这个列表不能包括 this 指针。其它使用到的外围本地变量都是隐式值捕获的；

引用捕获

可以定义以引用捕获变量的 lambda，如：

```
void fcn2()
{
    size_t v1 = 42;
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2();
}
```

v1 前的 & 表示 v1 是引用捕获的。被引用捕获的变量与其它引用是一样的。当在 lambda 函数体中使用引用捕获的变量时，使用的是这个引用绑定的对象。这个例子中返回的 v1 其实是其绑定的外围函数中的本地变量的值。

使用引用捕获的变量跟常规的使用引用变量一样，当 lambda 被调用时需要确保引用捕获的变量确实存在。被 lambda 捕获的变量是本地变量，这些变量在函数完成时将消失。如果在函数结束后还可以调用 lambda，那么将是严重的编程错误。

引用捕获有时是必须的，比如不可复制的对象：

```
void biggies(vector<string> &words, vector<string>::size_type sz, ostream &os = cout,
char c = ' ')
{
    for_each(words.begin(), words.end(), [&os, c](const string &s) { os << s << c; });
}
```

我们可以从函数中返回 lambda ，函数可以直接返回可调用对象或者函数返回一个具有可调用对象数据成员类对象。如果函数返回 lambda ，那么与函数不能返回本地变量的引用一样，这个 lambda 一定不能包含引用捕获。

警告：当我们进行引用捕获时，一定要保证当 lambda 执行时变量是存在的。

建议：保持 lambda 的捕获尽量的简单

lambda 捕获的信息在创建时保存，在执行时进行访问。保证这些信息的可用性是程序员自己的责任。按值的方式捕获 int ，string 和其它非指针变量是很简单的。这个时候，我们只需要关心在捕获时变量是否有我们需要的值。

如果捕获指针或迭代器或者按引用捕获，我们需要保证绑定到迭代器，指针和引用的对象在 lambda 执行时依然存在。甚至需要保证这个对象值是合理的。在 lambda 创建和执行之间的代码可能会改变按引用捕获的对象值。可能在创建时的值的确是我们想要的，但是当执行时其值已经变得很不一样了。

为了避免由捕获带来的问题尽可能缩小捕获的变量，并且尽可能避免捕获指针，迭代器和引用。

隐式捕获

相比于显式列举需要捕获的变量，可以让编译器对 lambda 函数体的代码进行推断。为了引导编译去推断捕获列表，我们在捕获列表中使用 & 或 =，其中 & 告知编译器按引用捕获，= 告知编译按值捕获。如以下是重写的 find_if：

```
wc = find_if(words.begin(), words.end(), [=](const string &s) {
    return s.size() >= sz;
});
```

如果想让其中的一些变量按值捕获，可以昏庸隐式和显式捕获：

```
for_each(words.begin(), words.end(), [&, c](const string &s) {
    os << s << c;
});
for_each(words.begin(), words.end(), [=, &os](const string &s) {
    os << s << c;
});
```

Mutable Lambdas

默认情况下，lambda 不会改变按值捕获的变量值，如果想要改变捕获变量的值，必须在参数列表后加上关键词 mutable。有 mutable 关键词的 lambda 不能省略参数列表：

```
void fcn3()
{
    size_t v1 = 42;
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f();
}
```

按引用捕获的对象是否可以改变仅仅依赖于引用的对象是 const 还是非 const 的：

```
void fcn4()
{
    size_t v1 = 42;
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2();
}
```

指定 lambda 的返回类型

目前我们所些的 lambda 只有一个 return 语句。因而我们不需要指定返回类型。如果 lambda 函数体中包含了任何不是 return 语句的话，那么 lambda 将被推断为返回 void。推断为返回 void 的 lambda 不会返回任何值。如：

```
transform(vi.begin(), vi.end(), vi.begin(), [](int i) -> int {
    if (i < 0)
        return -i;
    else
```

```
        return i;
});
```

transform 将输入序列中的每个元素通过调用 lambda 进行转换，并将结果存储到其目的迭代器所表示的序列中。目的序列和输入序列可以是同一个。由于这里的 lambda 中语句多于一条，我们必须指定其返回值。

10.3.4 绑定实参

lambda 表达式对于只用于一两个地方的简单操作非常有用。如果我们需要将相同的操作用于多个地方的话，我们应该定义函数而不是重复相同的 lambda 表达式。同样，如果操作比较复杂的话，最好是定义函数。

对于没有捕获列表的 lambda 表达式很容器用函数来替换。然而，想用函数来替换捕获了本地变量的 lambda 就有难度了。如：

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

想将 check_size 函数用于 find_if 肯定是不行的，find_if 要求函数只能接收一个参数。为了能够传递 check_size 必须要进行参数绑定。

标准库 bind 函数

使用定义了 functional 头文件中的 bind 函数。bind 可以被认为是通用的函数适配器(function adaptor)。它接收一个可调用对象并产生一个新的可调用对象，其中改变了原始函数的参数列表。bind 的通用形式是：

```
auto newCallable = bind(callable, arg_list);
```

其中 newCallable 是一个可调用对象，arg_list 是逗号分隔的参数并与给定的 callable 中的参数一一对应。也就是说，当我们调用 newCallable 时，newCallable 将调用 callable，并传递参数列表 arg_list。参数列表 arg_list 可以包含形式为 _n 的名字，其中 n 是一个整数。这种形式的参数称之为 “占位符”（placeholder），表示 newCallable 的形参列表。意味着当我们调用 newCallable 并传递给其多个参数，这些参数会被填值到 _1 _2 _3 并以 arg_list 的顺序传递给 callable 可调用对象。传递给 newCallable 的参数个数与占位符的个数一样多。

将 sz 绑定到 check_size 上

以下是给 check_size 绑定一个固定的 sz 参数：

```
auto check6 = bind(check_size, _1, 6);
```

这个例子中 bind 只有一个占位符，意味着 check6 有一个参数。占位符出现在 arg_list 的第一个位置，意味着 check6 的参数对应于 check_size 的第一个参数。这个参数的类型是 const string&，意味着 check6 中的参数也是 const string&。那么当调用 check6 时就必须传递类型为 string 的参数，check6 会将其传递给 check_size 的第一个参数。

arg_list 中的第二个参数是值 6，这个值被绑定到 check_size 的第二个参数上，无论任何时候我们调用 check6，它都会将 6 传递给 check_sz 的第二个参数：

```
string s = "hello";
bool b1 = check6(s); // check6(s) 调用 check_size(s, 6)
```

使用 bind 可以替换原来的 lambda 版本的 find_if：

```
auto wc = find_if(words.begin(), words.end(), bind(check_size, _1, sz));
```

bind 的调用生成了一个新的可调用对象，并将 check_size 的第二个参数绑定到值 sz 上。当 find_if 调用此对象时将会用 string 和 sz 调用 check_size。

使用 placeholders 名称空间

名字 _n 定义在名称空间 placeholders 中，这个名称空间本身被嵌套在 std 名称空间中。用 using 声明可以简写占位符的名字：using std::placeholders::_1; 。

如果嫌麻烦的话，就在函数中使用 using 指令（using directive）：using namespace std::placeholders; ，using 指令将使得 std::placeholders 中的名字都可见，这样就不需要再使用完全限定名了。

与 bind 函数一样，placeholders 名称空间定义在 functional 头文件中。

bind 的参数

bind 函数可以用于固定参数的值。甚至，bind 还可以用于重排序参数。如：

```
auto g = bind(f, a, b, _2, c, _1);
```

将产生一个有两个参数的新的可调用对象 g。其中 g 的第一个参数被传递给 f 的第五个参数，g 的第二个参数将被传递给 f 的第四个参数。效果相当于 `g(_1, _2)` 与 `f(a, b, _2, c, _1)` 一样，当调用 `g(X, Y)` 时与 `f(a, b, Y, c, X)` 效果一样。

还有 bind 可以对一个函数的参数进行重排序，如：

```
sort(words.begin(), words.end(), isShorter);
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

在第一个调用中，sort 调用的正常的 isShorter ，其含义是较短的字符串在前面。第二个调用中，sort 调用的反向的 isShorter ，其含义是较长的字符串在前面。

绑定引用形参

通常调用 bind 时，非占位符参数时拷贝到生成的调用对象中的。然而有时我们希望这个参数是按引用绑定的。如：

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

由于 os 不能被拷贝，所以不能直接 bind 这个参数。如果想要以引用方式进行绑定需要调用 ref 函数：

```
for_each(words.begin(), words.end(), bind(print, ref(os), _1, ' '));
```

ref 将返回一个对象其本身是可以被拷贝的，然而它的内部包含了给定参数对象的引用。同时还有一个 cref 函数用于生成一个类以容纳 const 引用。ref 和 cref 都定义在 functional 头文件中。

向后兼容：绑定参数

早期的 C++ 版本对于绑定参数到函数有诸多限制，并且更加复杂。标准库定义了 bind1st 和 bind2nd，在新标准下应该使用 bind 函数。

10.4 再谈迭代器

除了定义在每个容器中的迭代器，标准库还在 iterator 头文件中定义了几种别的迭代器：

(8) 插入迭代器（Insert iterators）：这些迭代器绑定到容器上，并且可用于往容器中插入元素；

(9) 流迭代器（Stream iterators）：这些迭代器绑定到输入输出流上，并可用于迭代相关的 IO 流；

(10) 反向迭代器（Reverse iterators）：这些迭代器向后移动而不是向前移动，除了 `forward_list` 之外都提供反向迭代器；

(11) 移动迭代器（Move iterators）：特殊的迭代器用于移动元素而不是拷贝；

10.4.1 插入迭代器

插入器（inserter）是一个迭代器适配器，以一个容器为参数，生成一个可以插入元素的迭代器。当通过插入迭代器赋值时，迭代器将调用容器的操作添加一个元素到指定的位置。这些迭代器支持的操作包括：

(12) `it = t` 插入值到 it 所表示的当前位置，根据不同种类的插入迭代器，可能会调用容器的 `push_back` ，`push_front` 或 insert 方法；

(13) `*it ++it it++` 这些操作存在当不做任何事情，每次都是返回 it 本身；

有三种不同种类的插入器。它们之间的区别在于元素插入到哪个位置： - `back_inserter()` 这个迭代器调用容器的 `push_back`； - `front_inserter()` 这个迭代器调用容器的 `push_front`; - `inserter()` 这个迭代器调用 insert，`inserter` 有第二个参数，并且这个参数必须是容器中的迭代器。元素被插入到给定迭代器所表示的元素前面；

只有容器有 `push_front` 函数时才能使用 `front_inserter` ，同样只有容器有 `push_back` 时才能使用 `back_inserter`。

需要理解的是当调用 `inserter(c, iter)` 时，如果连续向其赋值会把元素顺序的插入到这个迭代器所表示的元素之前。而 `front_inserter` 生成的迭代器的行为与 inserter 生成的迭代器有很大的不同。当使用 `front_inserter` 时，元素总是被插入到首元素之前，意味着后面插入的元素在前面插入的元素之前。而 inserter 生成的迭代器刚好相反，后面插入的元素在前面插入的元素之后。如：

```
list<int> lst = {1,2,3,4};
list<int> lst2, lst3;
```

```
// copy 完成后，lst2 的内容是 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// copy 完成后，lst2 的内容是 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

10.4.2 iostream 迭代器

尽管 iostream 类型不是容器，依然可以在 IO 类型上使用迭代器。`istream_iterator` 读取输入流，`ostream_iterator` 写输出流。这些迭代器将其对应的流当作特定类型元素的序列。使用流迭代器，我们可以使用通用算法对流对象进行读写。

以下是 `istream_iterator` 的操作：

(14) `istream_iterator<T> in(is);` in 从 is 中读取类型 T 的值；

(15) `istream_iterator<T> end;` in 的尾后迭代器；

(16) `in1 == in2 in1 != in2` in1 和 in2 必须读取相同类型的值。它们只有都是 end 值或者绑定到同一个输入流上时才相等；

(17) `*in` 返回从流中读取到值；

(18) `in->mem` 与 `(*in).mem` 含义相同，访问读取到的值的成员 mem；

(19) `++in in++` 从流中读取下一个值，使用的操作符是元素成员的 `>>` 操作符。前置版本返回自增后的迭代器，后置版本返回旧的迭代器；

当创建流迭代器时，需要指定元素的类型。`istream_iterator` 使用 `>>` 读取流。因而这个元素类型必须要有 `>>` 操作符才行。在创建 `istream_iterator` 时我们可以将其绑定到一个流上，或者当默认初始化时，创建的迭代器被当作尾后值。如：

```
istream_iterator<int> int_it(cin);
istream_iterator<int> int_eof;
ifstream in("afile");
istream_iterator<string> str_it(in);
```

只有当要给迭代器绑定的流到达了文件尾部或者遇到了 IO 错误时才会等于 end 迭代器。以下是利用 `istream_iterator` 读取流中的数据并存储到 vector 中：

```
istream_iterator<int> in_iter(cin);
istream_iterator<int> eof;
while (in_iter != eof)
    vec.push_back(*in_iter++);
```

以及用 `istream_iterator` 用于初始化 vector：

```
istream_iterator<int> in_iter(cin), eof;
vector<int> vec(in_iter, eof);
```

将流迭代器用于算法

由于算法是在迭代器上进行的，并且流迭代器支持一些迭代器操作。算法是按照迭代器分类决定哪些迭代器是可以用于此算法的。如 accumulate 可以用于一对 `istream_iterator`：

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

`istream_iterator` 允许延迟计算

当将 `istream_iterator` 绑定到流时，并没有说会立即读取这个流。实现会推迟到使用迭代器时才读取流。标准库保证当第一次解引用迭代器前，流是被读取了的。对于大部分程序来说，是立即读取还是延迟读取是没有关系的。然而，如果在同一个流上同时绑定了两个迭代器，此时就要当心了。

以下是 `ostream_iterator` 的操作：

(20) `ostream_iterator<T> out(os);` out 将类型 T 的值写入到输出流 os 中；

(21) `ostream_iterator<T> out(os, d);` out 将类型 T 的值和 d 一起写入到输出流 os 中，d 是一个 C 风格字符串的指针，每次写入时 d 都在 T 值的前面；

(22) `out = val;` 将 val 通过 out 写入到其绑定的输出流中，使用的 val 的 `<<` 操作符。val 的类型必须与 out 可写的对象类型相兼容；

(23) `*out ++out out++` 这些操作存在但是没有做什么事，每个操作都返回 out；

ostream_iterator 可以定义在任何有 << 操作符的元素类型。创建 ostream_iterator 时可以提供第二参数是一个字符串，这个字符串必须是 C 风格字符串，它将在任何元素打印之前先打印。
ostream_iterator 必须与特定的流绑定，并且没有尾后迭代器。
使用 ostream_iterator 写入一系列的值：

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e;
cout << endl;
```

每次赋值给 out_iter 时都会提交一个写入操作。需要注意的是其实我们可以省略解引用和自增操作，这两个操作并没有什么作用。但是现在这种写法是更好的，原因是更符合习惯，并且可以很容器的替换为别的迭代器类型。
除了自己写循环之外，我们还可以调用 copy 算法，如：

```
copy(vec.begin(), vec.end(), out_iter);
```

10.4.3 反向迭代器

反向迭代器是一种反向遍历容器的迭代器。反向迭代器反转了自增和自减的含义。自增一个反向迭代器会将迭代器移动到前一个元素；自减则会将迭代器移动到下一个元素。
除了 forward_list 的容器都有反向迭代器。通过调用 rbegin，rend，crbegin 和 crend 成员函数来获取反向迭代器。这些成员返回指向最后一个元素和首前位置的迭代器。与正常的迭代器一样，反向迭代器分为 const 和非 const 的。
在反向迭代器上调用 sort 将容器中的元素按照相反的顺序排序。如：

```
sort(vec.begin(), vec.end());
sort(vec.rbegin(), vec.rend());
```

反向迭代器需要自减操作符
只有同时支持自减和自增操作符的迭代器才能定义反向迭代器。毕竟反向迭代器的目的就是为了让迭代器反向移动。除了 forward_list ，所有标准容器的迭代器都同时支持自减和自增操作。流迭代器不支持自减操作，毕竟它不能反向移动。
反向迭代器和其它迭代器之间的关系
reverse_iterator 有一个 base 成员，返回其对应的正常迭代器。反向迭代器的正常迭代器指向的位置与反向迭代器本身是不一样的，正常迭代器指向的位置是下一个位置。

10.5 通用算法的分类

算法的基本特性是它对于迭代器的要求。有些算法如 find 只需要能够通过迭代器进行元素访问、自增迭代以及比较两个迭代器相等性。有些算法如 sort 需要可以进行读、写和随机访问元素。算法要求的迭代器操作被分类为五种迭代器类别（iterator categories），每个算法都说明了其每个迭代器参数所属的类别：
(24) 输入迭代器（Input iterator）：读，但不写；单遍扫描（single-pass），只能递增；
(25) 输出迭代器（Output iterator）：写，但是不能读；单遍扫描，只能递增；
(26) 前向迭代器（Forward iterator）：可以读写；多遍扫描（multi-pass），只能递增；
(27) 双向迭代器（Bidirectional iterator）：可读写；多遍扫描；可递增递减；
(28) 随机访问迭代器（Random-access iterator）：可读写；多遍扫描，支持全部迭代器运算；
另一种对算法进行分类的方法是他们是否对元素进行读，写或者重排序。算法还共享一组参数传递规范和明明规范，这个会在后面介绍。

10.5.1 按照 5 种迭代器分类

与容器一样，迭代器定义一组公共操作。一些操作是所有迭代器都会提供的；其它一些操作则只有特定种类的迭代器会提供。如 ostream_iterator 只能进行递增，解引用和赋值。vector，string 和 deque 的迭代器除了支持这些操作外，还支持递减、关系和算术运算。
迭代器按照他们提供的操作进行分类，并且组成了某种形式的层级。除了输出迭代器，如果一个迭代器处于更高的层级那么它将提供低层级迭代器的所有操作。
标准说明了通用和数字算法对迭代器参数所要求的最低层级。如 find 最低要求需要输入迭代器。replace 函数需要一对迭代器至少是前向迭代器。同样，replace_copy 要求其前两个迭代器至少得是前向迭代器。其第三个迭代器至少得是输出迭代器。对于每个迭代器，至少得是要求的最低层级，传递更低层级的迭代器是一种错误。

警告：大多数迭代器在我们传递错误的迭代器类别时并不会识别这个错误。

迭代器类别

输入迭代器（Input iterators）：可以读取序列中的元素。输入迭代器必须提供：

(29) 相等和不等操作符（==，!=）用于比较两个迭代器；

(30) 前置和后置递增操作符（++）用于推进迭代器；

(31) 用于读取元素的解引用运算符（*）；解引用只能出现在赋值运算符的右边；

(32) 箭头运算符（->）等价于 `(*it).member`，意味着解引用迭代器，并从底层对象中获取一个成员；

输入迭代器只能用于顺序访问。`*it++` 保证是有效的，但是递增输入迭代器可能会导致流中的所有其它迭代器失效。结果就是，不能保证输入迭代器的状态可以保存，并且用于测试一个元素。输入迭代器因而只能用于单遍扫面算法，就是对元素值只访问一次，而不能多次访问。如果需要多次访问就要使用前向迭代器。find 和 accumulate 算法只要求输入迭代器；`istream_iterator` 是输入迭代器；

输出迭代器（Output iterators）：可以被认为是输入迭代器的补集，只能写不能读元素。输出迭代器必须提供：

(33) 用于推进迭代器的前置和后置递增运算符（++）；

(34) 解引用运算符（*），只能出现在赋值操作符的左侧（给输出迭代器赋值是将内容写入到底层元素）；

我们只能给输出迭代器写入给定值一次。与输入迭代器一样，输出迭代器只能被用于单遍扫描算法。用于目的迭代器基本都是输出迭代器。如 copy 的第三个参数是输出迭代器。`ostream_iterator` 是输出迭代器。

前向迭代器（Forward iterators）：可以读写给定序列。它们只能在序列的一个方向上移动。前向迭代器支持输入和输出迭代器的所有操作。而且，可以多次读写同一个元素。因此，我们可以保存前向迭代器的状态。使用前向迭代器的算法可以多次扫描序列。replace 算法需要一个前向迭代器；`forward_list` 上的迭代器就是前向迭代器；

双向迭代器（Bidirectional iterators）：可以正向和反向读写序列中的元素。除了支持前向迭代器中的所有操作，双向迭代器还支持前置和后置递减操作符（--）。reverse 算法需要双向迭代器，除了 `forward_list` 外的标准库容器都提供符合双向迭代器要求的迭代器。

随机访问迭代器（Random-access iterators）：提供固定时间内访问序列中任何位置的能力。这些迭代器双向迭代器的所有操作。除此之外，随机访问迭代器还支持：

(35) 关系操作符（< <= >= >）用于比较两个迭代器的相对位置；

(36) 加法和减法操作符（+ += - -=）用于迭代器和整数。结果是推进或回退给定整数个元素的位置；

(37) 减法操作符运用于两个迭代器，将返回两者之间的距离；

(38) 下标操作符（`iter[n]`）与 `*(iter+n)` 同义；

sort 算法要求随机访问迭代器。array，deque，string 和 vector 中的迭代器是随机访问迭代器，用于访问内置数组元素的指针也是。

10.5.2 按照算法的参数模式分配

算法的参数有自己的规范，理解这些规范对于理解算法本身很有帮助。绝大部分算法的参数形式是以下四种之一：

algorithm(beg, end, other args); algorithm(beg, end, dest, other args); algorithm(beg, end, beg2, other args); algorithm(beg, end, beg2, end2, other args);

其中 algorithm 就是算法的名字，beg 和 end 表示输入范围。尽管几乎所有的算法都有一个输入范围，其它参数的出现则依赖于算法所做的工作。dest，beg2 和 end2 都是迭代器，它们在算法中的角色是类似的。除了这些迭代器参数，有些算法还有额外的非迭代器参数。

有一个单一目的迭代器的算法

dest 参数表示一个目的地，这样算法可以将输出写入到这个 dest 所表示的元素中。算法总是假设能够写入足量的元素，保证目的迭代器能够容纳算法的输出是程序员自己的责任。如果 dest 直接指向容器中的元素，那么算法将其输出写入到容器中已经存在的元素中去。更为一般的是，dest 是一个插入迭代器或者 `ostream_iterator`。插入迭代器往容器中添加元素，因此可以保证容器中有足够的容量。`ostream_iterator` 将内容写入到输出流中，同样不用管需要写入多少个元素。

有第二个输入序列的算法

有单一的 beg2 或者 beg2 和 end2 参数的算法将这些迭代器当作第二个输入范围。这些算法将第二个范围内的元素与第一个范围内的元素联合起来进行运算。
如果一个算法的参数是 beg2 和 end2，那么这些参数有两个完全确定的范围：第一个输入范围 [beg, end) 和第二个输入范围 [beg2, end2) 。
如果算法只有一个 beg2 ，那么将 beg2 当作第二个输入范围的首元素。这个范围的尾部没有指定，相反，算法假定第二个范围从 beg2 开始，并且至少跟第一个输入范围一样大，保证这个条件成立是程序员自己的责任。

10.5.3 算法名字的约定

除了上面介绍的约定外，算法的名字和重载也有自己的约定。这些约定用于处理怎样提供操作以替代默认 < 或 == 操作符，以及算法是将输出写入到其输入序列还是一个独立的目的地。

有些算法使用重载来传递谓词

使用谓词来替换 < 或 == 操作符的算法，以及没有这个谓词参数的算法是重载的。其中一个版本使用元素的操作进行比较；另一个版本则有一个额外的参数表示谓词，如：

```
unique(beg, end); // 使用 == 操作符比较元素
unique(beg, end, comp); // 使用 comp 比较元素
```

两个调用都会移除相邻的重复元素。第一个使用元素的 == 操作符以检查重复元素；第二个使用 comp 决定两个元素是否相等。由于两个函数的参数数目不一样，调用哪个函数不存在模糊性。

_if 的算法版本

有一个元素值的算法通常会有第二个名字（而不是重载），第二个版本的参数是一个谓词而不是元素值。以谓词为参数的算法的后缀是 _if：

```
find(beg, end, val); // 查找输入范围内 val 的首次出现的位置
find_if(beg, end, pred); // 查找输入范围内 pred 首次为真的位置
```

这两个算法都是查找一个特定元素在输入序列中出现的位置。find 查找特定的值；find_if 查找使得 pred 为真的元素值；
这些算法提供要给额外的名字，而不是进行重载的原因是两个版本都具有相同数目的参数。重载可能会导致模糊性。

_copy 的算法版本

默认情况下，会调整元素的算法将调整后的元素写回到给定的输入范围中。这些算法提供了第二个版本将元素写入到给定的输出目的地，这样的版本以 _copy 为后缀：

```
reverse(beg, end); // 反转元素并将其写入到输入范围中
reverse_copy(beg, end, dest); // 将反转后的元素写入到 dest 中
```

一些算法同时提供了 _copy 和 _if 版本。这些版本有一个目的迭代器和谓词：

```
remove_if(v1.begin(), v1.end(),
    [](int i) { return i%2; });
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
    [](int i) { return i%2; });
```

两个调用都使用 lambda 来判断一个元素是否为偶数。第一个版本，直接在输入序列中移除。第二个版本，将符合条件的值拷贝到 v2 中。

10.6 特定于容器的算法

与别的容器不同的是，list 和 forward_list 将几个算法定义为自己的成员。如：链表类型定义自己的 sort，merge，remove，reverse 和 unique 版本。通用的 sort 算法需要随机访问迭代器。因而，sort 不能被用于 list 和 forward_list，原因是它们分别只提供双向迭代器和前向迭代器。
其它一些通用算法可用于链表类型，但是会损失性能。这些算法交换输入序列中的元素。链表可以通过交换链接来达到目的，而不是交换元素的值。因而，链表特定的版本会比通用版本有一个更好的性能。
以下是链表特定的算法，没有出现在下面的算法在链表和别的容器类型上表现一样好：

(39) lst.merge(lst2) lst.merge(lst2, comp) 将 lst2 上的元素合并到 lst 上。lst 和 lst2 都必须是排序了的。元素将从 lst2 中移除，在 merge 之后 lst2 将是空的。第一个版本使用 < 操作符；第二个版本使用给定的比较操作；

(40) `lst.remove(val)` `lst.remove_if(pred)` 使用 erase 移除每一个 `==` 给定值或者使得谓词为真的元素；
(41) `lst.reverse()` 反转 lst 中的元素顺序；
(42) `lst.sort` `lst.sort(comp)` 对 lst 中的元素进行排序，第一个函数使用 `<` 操作符，第二个使用给定的比较操作；
(43) `lst.unique()` `lst.unique(pred)` 调用 erase 移除相邻相等的元素值，第一个版本使用 `==`；第二个版本使用谓词；

以上操作返回 void，对于 list 和 `forward_list` 有限使用成员版本而不是通用算法版本。

splice 成员

链表类型还定义了 splice（拼接）算法，这是链表特有的算法 `list.splice(args)` 以及 `forward_list.splice_after(args)`：

(44) `(p, list2)` 将 lst2 中的元素移动到 p 迭代器所表示为位置前，并从 lst2 中移除元素。如果是 `splice_after` 则拼接到 p 之后，lst2 的类型必须与 list 和 `forward_list` 的类型相同，而且不能与调用者是同一个对象；
(45) `(p, lst2, p2)` p2 是 lst2 中的有效迭代器，将 p2 所表示的元素移动到 list 中，或者将 p2 后面的那个对象移动到 `forward_list` 中，lst2 可以与 list 或者 `forward_list` 是相同的对象；
(46) `(p, lst2, b, e)` 将 lst2 中的迭代器 b 和 e 所表示的范围中的元素移动到 list 或 `forward_list` 中，元素会从 lst2 中删除。lst2 与 list 或者 `forward_list` 可以是相同的对象，但 p 不能表示 b 和 e 范围内的元素；

链表特有的算法类似于其通用算法版本。然而一个重大的区别就是链表版本会改变其底层容器。如链表版本的 remove 会真的执行移除操作，链表版本的 unique 会移除第二个及之后的重复元素。
同样的 merge 和 splice 会改变其参数，而通用版本的 merge 将内容写入到给定的目的迭代器中；两个输入序列表示不变。链表 merge 函数会销毁给定链表，将其元素从参数中移除合并到调用者链表上。在 merge 之后所有的元素都在同一个链表上。

关联容器与顺序容器的本质区别就在于关联容器是按照键的来存取元素的，而顺序容器是按照元素在容器中的位置进行存取的。尽管关联容器和顺序容器共享了很多行为，然而在如何处理键（key）时有很大的区别。
关联容器（Associative containers）支持通过键进行高效的查找。其中最关键的两个关联容器是 map 和 set。map 中的元素是键值对(key-value)：键被当做 map 中的索引，值表示与这个索引相关联的数据。set 只包含键，set 支持高效查询一个键是否在在集合中。
标准库提供了八种关联容器，这八种容器在三个维度上有所不一样：(1)要么是集合(set)要么是映射(map)；（2）是要求键唯一和不唯一；（3）将元素按照有序方式进行存储还是没有此要求；其中可以包含多个键的容器包含 multi，以无序方式存储的容器以 unordered 开头。
无序容器使用散列函数（hash function）来组织元素。

Elements Ordered by Key

372. `map` 关联数组；保存键值对；
373. `set` 集合，只包含键；
374. `multimap` 键可以出现多次的 map；
375. `multiset` 键可以出现多次的 set；

Unordered Collections

376. `unordered_map` 由散列函数组织的关联数组；
377. `unordered_set` 由散列函数组织的集合；
378. `unordered_multimap` 无序且键可以出现多次的关联数组；
379. `unordered_multiset` 无序且键可以出现多次的集合；

11.1 使用关联容器

map 类型经常被称为是关联数组（associative array），关联数组类似于常规数组，除了其可以用非整数（integer）作为下标。map 中的值使用键进行查找而不是位置。
set 就是键的集合。set 最常用于查询一个值是否在集合中。

使用 map

如以下计数代码：

```
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout << w.first << " occurs " << w.second
        << ((w.second > 1) ? " times" : " time") << endl;
```

此处需要注意的是以单词作为下标对 word_count 取值时，如果值不存在就会创建一个新的将值设置为 0，并返回值的引用。

我们可以用范围 for (range for) 来遍历 map，其每个结果是一个 pair 对象，pair 类本身也是一个模板类并且有两个公有的数据成员 first 和 second。此处的 first 就是键，second 就是关联的值。

需要了解的是迭代遍历的结果是按照映射的键的顺序进行输出的。

使用 set

```
set<string> exclude = {"The", "But", "And", "Or", "An", "A"};
```

与别的容器一样，set 是模板。想要定义一个 set ，需要指定元素的类型，在上面的例子就是 string。与顺序容器一样，可以用列表初始化关联容器的元素。

为了检查键是否存在，可以调用 set 的 find 函数：

```
if (exclude.find(word) == exclude.end())
```

find 函数返回一个迭代器，如果键在 set 中，那么将返回指向那个键的迭代器。如果元素没有找到，find 将返回尾后迭代器。

11.2 关联容器简介

有序和无序的关联容器都支持通用的容器操作，这些在 §9.2 中介绍过。关联容器不支持顺序容器特有的与位置相关的操作，如 push_front 或 back 。由于关联容器中的元素是按照键进行存储的，这些操作对于关联容器来说就没有意义了。同样，关联容器不支持以一个元素值和个数的构造函数和插入操作。

除了支持与顺序容器一样的通用操作之外，关联容器还提供另外一些操作和类型别名，并且无序关联容器还提供了对 hash 性能进行调优的操作。

关联容器的迭代器是双向迭代器的（bidirectional）。

11.2.1 定义关联容器

当定义 map 时，需要指定键和值的类型；当定义 set 时需要指定键的类型。每个关联容器都定义了一个默认构造函数，其将创建一个指定类型的空容器。可以将关联容器初始化为另外一个相同类型容器的副本，或者以一系列值进行初始化，只要这些值可以转为容器的类型。在新标准下可以用列表初始化元素：

```
map<string, size_t> word_count;
set<string> exclude = {"the", "but", "and", "or", "an"};
map<string, string> authors = { {"Joyce", "James"}, {"Austen", "Jane"} };
```

与往常一样，初始值必须可以转为容器中的元素类型。对于 set ，其元素类型就是键的类型。当初始化 map 时，必须同时提供 key 和 value。我们将键-值对放在大括弧中如：{key, value} 用于表示这两个值共同构成 map 中的一个元素。

初始化 multimap 和 multiset

map 和 key 中的键必须是唯一的；对于一个给定的 key 只能有唯一元素与之对应。multimap 和 multiset 容器没有类似的限制；相同的键可以有多个元素与之对应。

11.2.2 对于 key 类型的要求

关联容器的键是有限制的。我们将在 11.4 中介绍无序容器的键的要求。对于有序容器 —— map，multimap，set 和 multiset —— 键类型必须定义操作对元素进行比较。默认情况下，标准库使用 < 操作符用于建之间的比较。在 set 类型中，键就是元素类型；在 map 类中，键是元素的 first 的类型。

注意：传递给排序算法的可调用对象必须满足与关联容器中的键相同的要求。

有序容器的 key 类型

类似于给算法提供我们自己的比较操作，我们也可以给键提供我们的比较操作用于替换 < 操作符。这个操作必须符合严格弱顺序（strict weak ordering）。可以认为严格弱顺序与“小于”等同，尽管我们函数可能实现的比较复杂。定义的比较函数必须有以下性质：

380. 两个 key 不能互相“小于”对方；如果 k1 小于 k2，那么 k2 必须不小于 k1；
381. 如果 k1 小于 k2，并且 k2 小于 k3，那么 k1 必须小于 k3；
382. 如果两个键，两个 key 都不小于对方，那么可以认为这两个键是相等的，如果 k1 == k2，并且 k2 == k3 那么，k1 必须与 k3 相等；

如果两个键是相等的，容器认为这两者是一样。当用作 map 的键时，那么将只要一个元素与这两个键匹配，而其中任何一个键都可以作为 map 中的键。

注意：在实践中，作为键的类型需要其 < 操作的行为与我们通常遇到的行为是一致的；

使用比较函数来比较 key 类型

容器用于组织元素的操作也是容器类型的一部分。为了定义我们自己的操作，必须在定义关联容器类型时提供操作的类型。操作类型被放置在元素类型的后面，这两个类型都放在尖括号中。提供给关联容器的比较函数必须与关联容器的类型相符合。如：

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs) {
    return lhs.isbn() < rhs.isbn();
}

multiset<Sales_data, decltype(compareIsbn)*> bookstore(compareIsbn);
```

当用 decltype 来表示一个函数指针时，必须提供 * 来表示我们确实需要的函数的指针，这是由于 decltype 推断出来的是函数的真实类型。当将 compareIsbn 指针作为构造函数实参时，可以直接使用其名字，而不必使用 &compareIsbn 的形式，这是由于用于参数的函数名字会隐式转为函数指针。

11.2.3 pair 类型

pair 类型定义在 utility 头文件中。pair 中有两个数据成员。与容器一样，pair 是模板。在创建 pair 时必须提供两个类型名字用于表示数据成员的类型。对于两个类型是否相同没有要求，如：

```
pair<string, string> anon;
pair<string, size_t> word_count;
pair<string, vector<int>> line;
```

pair 的默认构造函数对数据成员进行值初始化。我们依然可以给每个成员提供初始值：

```
pair<string, string> author{"James", "Joyce"};
```

pair 的数据成员是 public 的。这两个成员的名字是 first 和 second，通过成员访问符就可以访问这两个成员。

pair 只有很受限的一小部分操作：

383. `pair<T1, T2> p;` p 是一个 pair，其中类型 T1 和 T2 所表示的成是值初始化的；
384. `pair<T1, T2> p(v1, v2);` p 是一个 pair，其中类型 T1 和 T2 所表示的成员分别从 v1 和 v2 进行初始化；
385. `pair<T1, T2> p = {v1, v2};` 与 `p(v1, v2)` 是相同的；
386. `make_pair(v1, v2)` 从 v1 和 v2 中初始化一个 pair 对象，pair 的类型是通过 v1 和 v2 的类型进行推断的；
387. `p.first` 返回 p 的 public 数据成员 first；
388. `p.second` 返回 p 的 public 数据成员 second；
389. `p1 relop p2` 关系操作符 relop（< > <= >=），关系操作符被定义为字典序：`p1 < p2` 的含义与 `p1.first < p2.first` 或者 `!(p2.first < p1.first) && p1.second < p2.second` 是一样的；
390. `p1 == p2 p1 != p2` 如果两个 pair 的 first 和 second 数据成员分别相等，那么可以认为这两个 pair 是相等的；

创建 pair 的函数

如果一个函数需要返回 pair 对象，在新标准下可以列表初始化这个返回值，如：

```
pair<string, int>
process(vector<string> &v)
{
```

```
    if (!v.empty())
        return {v.back(), v.back().size()};
    else
        return pair<string, int>();
}
```

同样还可以使用 make_pair 来创建新的 pair 对象。

11.3 关联容器的操作

关联容器还定义了额外的一些类型，这些类型表示容器的 key 和 value 类型：

391. key_type 容器的 key 的类型；
392. mapped_type 每个 key 对应的值的类型；只有 map 定义了此类型；
393. value_type 对于 set 来说与 key_type 一样，对于 map 来说就是 pair<const key_type, mapped_type>；

对于 set 类型来说，key_type 和 value_type 是一样的；set 中保存的值就是键。对于 map 来说，其元素是键值对，意味着每个元素是一个 pair 对象，pair 对象中包含了键和与其对应的值。由于我们不能改变元素的键，所以 pair 中的键是 const 的：

```
set<string>::value_type v1; //string
set<string>::key_type v2; //string
map<string, int>::value_type v3; //pair<const string, int>
map<string, int>::key_type v4; //string
map<string, int>::mapped_type v5; //int
```

与顺序容器一样，我们使用作用域操作符 (::) 来获取类型成员如：map<string, int>::key_type，只有 map 类型（unordered_map, unordered_multimap, multimap 和 map）才有 mapped_type 。

11.3.1 关联容器迭代器

当解引用一个迭代器时，我们得到容器的 value_type 类型的值的引用。对于 map 来说其 value_type 是一个 pair 其中 first 保存着一个 const 的键，而 second 保存着其对应的值，如：

```
auto map_it = word_count.begin();
cout << map_it->first;
cout << " " << map_it->second;
map_it->first = "new key"; //error: key is const
++map_it->second;
```

笔记：需要记住的是 map 的 value_type 是一个 pair 对象，可以改变其 second 数据成员，而不能改变其 first 成员。

set 的迭代器是 const 的

尽管 set 类型同时定义了 iterator 和 const_iterator 类型，两种迭代器类型都是只读的。正如我们不能改变 map 元素的键，set 中的键也是 const 的。我们可以使用 set 的迭代器进行读而不能写其元素的值：

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42; //error: set 中的键是只读的
    cout << *set_it << endl;
}
```

遍历关联容器

map 和 set 类型都提供了 begin 和 end 操作。使用这两个函数来获取迭代器来遍历容器，如：

```
auto map_it = word_count.cbegin();
while (map_it != word_count.cend()) {
    cout << map_it->first << " occurs "
         << map_it->second << " times" << endl;
    ++map_it;
}
```

当我们用迭代器去遍历 map，multimap，set 以及 multiset 时，其迭代器产生的元素是按照键的正序排列的。

关联容器和算法

通常我们是不将通用算法用于关联容器的。关联容器的 key 是 const 的意味着我们不能将关联容器的迭代器传递给那些需要修改元素或者对容器元素进行重排序的算法。这些算法需要对元素进行写操作。而 set 类型中的元素是 const 的，map 的元素是 pair 类型，其 first 成员是 const 的。

关联容器可以用于读元素的算法。然而，这些算法中的大部分都适用于搜索顺序容器。由于关联容器中的元素可以通过其键快速被查找到，使用通用搜索算法几乎总是错误的想法。每个关联容器都定义了名为 find 的成员函数，可以用于直接从关联容器中查找与给定 key 相关联的元素。我们也可以使用通用 find 算法来查找元素，但是算法执行的是顺序搜索。使用 find 成员函数明显是更加快的方式。

在实践中，如果我们真的将关联容器用于通用算法，要么将其用于 source sequence 要么用于 destination。如，调用通用 copy 算法从一个关联容器拷贝元素到另一个序列中。类似的，我们可以调用 inserter 来绑定一个插入迭代器到关联容器上。使用 inserter，我们可以将关联容器作为另外一个算法的目的地（destination）。

11.3.2 添加元素

insert 成员添加一个元素或一系列元素。由于 map 和 set（以及与之对应的无序类型）包含的 key 都是唯一的，如果插入的元素是已经存在的将没有任何效果，如：

```
vector<int> ivec = {2,4,6,8,2,4,6,8};
set<int> set2;
set2.insert(ivec.cbegin(), ivec.cend()); //set2 有 4 个元素，没有重复的元素
set2.insert({1,3,5,7,1,3,5,7}); //set2 有 8 个元素
```

关联容器的 insert 操作： - `c.insert(v)` `c.emplace(args)` v `value_type` 对象；args 用于构建一个元素。对于 map 和 set，只有与给定 key 关联的元素不在 c 中才会被插入。返回值是 pair 类型，其中 first 是给定 key 关联的元素的迭代器，其 second 成员是一个 bool 值表示元素是否被插入成功；对于 multimap 和 multiset，插入给定元素，并返回新元素的迭代器； - `c.insert(b, e)` `c.insert(il)` b 和 e 是表示类型为 `c::value_type` 的元素范围；il 则是括弧中的一系列值（初始值列表），返回 void；对于 map 和 set，将插入不存在于容器中的 key 关联的元素；对于 multimap 和 multiset 则将插入范围内的每个元素； - `c.insert(p, v)` `c.emplace(p, args)` 类似于 `insert(v)` 和 `emplace(args)`，但是使用迭代器 p 作为开始搜索插入元素的位置的索引，返回与给定 key 关联的元素的迭代器；

以一对迭代器或初始值列表作为参数的 insert 版本的运作模式与对应的构造函数是类似的，只有与给定 key 关联的第一个元素会被插入。

添加元素到 map 中

当给 map 插入元素时，需要记住元素的类型是 pair。通常当我们插入时，我们并没有现成的 pair 对象。相反，我们会在 insert 的参数中创建一个新的 pair 对象：

```
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

正如我们所见，在新标准下创建 pair 最简单的方式是使用括弧初始化（brace initialization）于参数列表。作为另外一种选择，我们可以调用 `make_pair` 或者显式构建 pair。请注意最后一个 insert 中的参数：`map<string, size_t>::value_type(s, 1)` 构建一个合适的 pair 类型的新对象用于插入到 map 中去。

检查 insert 的返回值

insert 或者 emplace 的返回值将根据容器类型和参数的不同而不同。对于 key 是唯一的容器，insert 和 emplace 的添加一个元素的版本将返回一个 pair 对象用于判断插入是否发生。pair 的 first 成员是一个与给定 key 关联的元素迭代器；second 成员是一个 bool 值用于指示元素是否被插入，或者是否已经存在。如果键已经存在于容器中，那么 insert 将不做任何事，并且返回值的 bool 部分是 false，如果键不存在，那么元素将被插入并且 bool 是 true。

添加元素到 multiset 和 multimap 中

有时我们希望一个 key 可以关联多个值，比如我们希望映射作者到其所写的书的名字上。在这种情况下，一个作者可能有个条目，所以我们使用 multimap 而不是 map。用于 multi 容器中的 key 不需要是唯一的，在这些键上 insert 总是插入元素：

```
multimap<string, string> authors;
authors.insert({"Brath, John", "Sot-Weed Factor"});
authors.insert({"Brath, John", "Lost in the Funhouse"});
```

对于允许多个 key 的容器，只插入一个元素的 insert 操作将返回新元素的迭代器。没有必要返回 bool，因为 insert 总是添加新的元素。

11.3.3 移除元素

关联容器定义了三个版本的 erase，与顺序容器一样，可以用 erase 擦除一个元素（通过传递一个迭代器）或者传递迭代器范围来擦除一个范围内的元素。这些版本的 erase 类似于对应的顺序容器的操作：指定的元素被移除并返回 void。

394. `c.erase(k)` 从 c 中移除任何与键 k 相对应的元素，返回 `size_type` 的值用于表示移除的元素的个数；
395. `c.erase(p)` 从 c 中移除由迭代器 p 表示的元素，p 必须确实表示 c 中的一个元素；它不能等于 `c.end()`，返回指向 p 的下一个元素的迭代器或者当 p 为最后一个元素时返回 `c.end()`；
396. `c.erase(b, e)` 从 c 中移除由 b 和 e 所表示的范围中的元素，返回 e；

关联容器还支持一个额外的 erase 操作，这个版本的 erase 有一个 `key_type` 参数，这个版本的 erase 将移除所有与给定 key 相关的元素，并返回一个计数器告知移除了多少个元素。对于键是唯一的容器来说，erase 的返回值要么是 0 要么是 1，如果返回 0 则想要移除的元素不在容器中。对于 multi 容器来说，被移除的元素的数目可能会大于一。

11.3.4 map 的下标操作

map 和 `unordered_map` 容器提供了下标操作符和一个对应的 at 函数。set 类型不支持下标，原因是 set 中没有“值”与键相对应。元素自己就是键，所以“获取与 key 相关的值”是没有意义的。我们不能对 `multimap` 和 `unordered_multimap` 进行下标操作，原因是可能有超过一个值与给定的键对应。

397. `c[k]` 返回与键 k 关联的元素；如果 k 不在 c 则添加一个新的值初始化的元素与键 k 进行关联；
398. `c.at(k)` 检查并访问与键 k 相关的元素；如果 k 不在 c 中则抛出 `out_of_range` 异常；

与别的下标操作符一样，map 的下标取一个索引（键 key）然后提取出与这个键关联的值。然而，与别的下标操作符不一样的是，如果键不存在，一个新的元素将被创建出来与这个 key 关联并且插入到 map 中去。关联的值是值初始化的（value initialized）。

由于下标操作符可能会插入一个元素，我们只能将下标操作用于非 const 的 map 之上。

注意：map 的下标运算与数组和 vector 的下标操作有非常大的不同：使用不存在于容器中的 key，将导致与之关联的元素添加到 map 中。

使用下标操作返回的值

map 的下标操作与别的下标操作符的不同之处在于其返回值。通常，对迭代器进行解引用得到的类型与下标操作符得到的类型是一样的。但对于 map 来说是不成立的：当我们对 map 进行下标运算时，我们得到一个 `mapped_type` 类型的对象；当我们解引用 map 的迭代器时，我们得到一个 `value_type` 类型的对象。

与别的下标运算一样的是，map 的下标运算返回一个左值。由于返回的值是左值，我们对元素进行读或写：

```
cout << word_count["Anna"];
++word_count["Anna"];
cout << word_count["Anna"];
```

注意：与 vector 和 string 不一样的是，map 的下标操作符的返回值类型与通过解引用 map 的迭代器得到的类型是不一样的。

有时我们仅仅想知道一个元素是否在容器中，而不想在元素不存在时添加一个。在这种情况下，我们一定不能使用下标操作符。

11.3.5 访问元素

关联容器提供了许多中方式来查找一个给定的元素。使用哪个操作取决于我们尝试解决什么样的问题。如果我们只关心一个特定的元素是否在容器中，最好是使用 find。对于键是唯一的容器来说，使用 find 或

者 count 都是一样的。然而，对于具有 multi 键的容器来说，count 所做的工作更多：如果一个元素是存在的，它依然必须对有多少个元素具有相同的键进行计数。如果我们不需要计数，那么最好是使用 find：

lower_bound 和 upper_bound 对于无序容器来说是非法的，下标和 at 操作只对非 const 的 map 和 unordered_map 有效

399. c.find(k) 返回与键 k 关联的第一个元素的迭代器，当 k 不在容器中时返回尾后迭代器；
400. c.count(k) 返回与键 k 关联的元素的个数。对于键是唯一的容器来说，结果总是零或一；
401. c.lower_bound(k) 返回键不小于 k 的第一个元素的迭代器；
402. c.upper_bound(k) 返回键大于 k 的第一个元素的迭代器；
403. c.equal_range(k) 返回与键 k 相关的所有元素的一个迭代器范围的 pair，如果 k 不在容器中，两个成员都是 c.end()；

使用 find 替换 map 的下标

对于 map 和 unordered_map 类型，下标操作符是最简单的获取一个值的方式。然而，正如我们所看到的，使用下标有一个重要的副作用：如果 key 当时不在 map 中，那么下标操作会插入一个与这个 key 相关联的元素。这个行为是否正确取决于我们的期望。

有时我们仅仅只是想知道一个给定的 key 是否在 map 中而不想改变这个关联容器。我们就不能用下标操作符来确认元素是否包含在容器中，这是由于下标操作符会在键不在容器中时插入一个新的与之相关的元素。在这种情况下，我们应该使用 find 函数：

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

在 multimap 和 multiset 中查找元素

在键是唯一的关联容器中查找一个元素是十分简单的——元素要么在要么不在容器中。对于允许多个相同的键存在的容器，这个过程就更为复杂了：与给定 key 关联的元素可能多个。当 multimap 或者 multiset 有多个与给定 key 关联的元素时，这些元素在容器中是相邻存储的。如上面提到的作者和标题之间的 multimap，我们想要打印跟特定作者关联的所有书名，至少有三种方式来解决这个问题，最简单而明显的方式使用 find 和 count：

```
string search_item("Alain de Botton");
auto entries = authors.count(search_item);
auto iter = authors.find(search_item);
while (entries) {
    cout << iter->second << endl;
    ++iter; //推进到下一个标题
    --entries; //记录已经打印的个数
}
```

find 返回与给定 key 关联的第一个元素的迭代器。标准库保证迭代 multimap 或 multiset 将顺序返回与给定键关联的所有元素。

迭代器方式的解决方案

另一种选择是使用 lower_bound 和 upper_bound 来解决问题。这两个操作都以 key 为参数并返回一个迭代器。如果 key 在容器中，由 lower_bound 返回的迭代器将会指向与 key 关联的第一个元素，而由 upper_bound 返回的迭代器则指向与 key 关联的最后一个元素的后一个位置。如果元素不在 multimap 钟，那么 lower_bound 和 upper_bound 将返回相等的迭代器；两个都将返回一个迭代器指向 key 可以合适插入的位置（插入后依然保持有序）。因此，调用 lower_bound 和 upper_bound 于相同的 key 上时，将产生与此 key 关联的所有元素的迭代器范围。

当然，这两个操作返回的迭代器可能是容器的尾后迭代器（off-the-end iterator）。如果查找的 key 在容器中是最大的，那么 upper_bound 将返回尾后迭代器。如果 key 不在容器中且比容器中任何键都大，那么 lower_bound 将返回尾后迭代器。

注意：lower_bound 返回的迭代器可能不会指向指定 key 所关联的元素。如果 key 不在容器中，那么 lower_bound 将返回一个表示这个 key 可以插入的位置的迭代器，当插入之后元素的顺序将保持不变。

使用这两个操作，我们可以重写程序为：

```
for (auto beg = authors.lower_bound(search_item),
          end = authors.upper_bound(search_item);
```

```
     beg != end; ++beg) {
        cout << beg->second << endl;
}
```

这两个操作不会告知 key 是否在容器中，其重要之处在于返回值的行为与迭代器范围(iterator range)一样。如果这个 key 没有关联的元素，那么 lower_bound 和 upper_bound 的返回值相同。两个结果都指向 key 插入的位置。

假如有元素与这个 key 关联，beg 将指向第一个元素。通过递增 beg 可以遍历与这个 key 关联的元素。迭代器 end 表示我们已经遍历了所有的元素，当 beg 和 end 相等时，就表示我们已经遍历了全部的元素。

注意：当 lower_bound 和 upper_bound 返回相同的迭代器，表示给定的 key 不在容器中。

equal_range 函数

剩下的解决方案是最直接的：相较于调用 upper_bound 和 lower_bound，我们可以调用 equal_range。这个函数以一个 key 为参数并返回一个 pair 类型以包含 iterator，如果 key 存在于容器中，那么 first 表示与键关联的第一个元素，second 表示与 key 关联的最后一个元素的下一个位置。如果没有匹配的元素，两个迭代器都指向 key 插入的位置。我们可以用 equal_range 再次修改程序：

```
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first) {
        cout << pos.first->second << endl;
     }
```

equal_range 的返回值 pair 中的 first 成员包含的迭代器与 lower_bound 的返回值一样，second 成员与 upper_bound 函数的返回值一样。因此，在这个程序中，pos.first 与 beg 是一样的，pos.second 与 end 是一样的。

11.4 无序容器

在新标准中定义四个无序关联容器（unordered associative containers）。相较于前面的关联容器使用比较操作（comparison operation）来组织元素，这些容器使用的哈希函数（hash function）以及 key 类型的相等比较操作符。无序容器最常用于 key 类型没有明显的顺序关系的情形。这些容器也用于那些维护元素的顺序是很昂贵的应用。

尽管哈希在原则上可以达到更好的平均性能，想要在实践中达到好的结果通常需要一些性能上的测试和调整。因而，通常使用有序容器更加容器且通常拥有更好的性能。

提示：仅仅在 key 类型天生是不可排序的，或者性能测试显示哈希可以解决问题时才使用无序容器。

使用无序容器

除了管理哈希的操作之外，无序容器提供与有序容器一样的操作（find，insert 和别的操作）。意味着用于 map 和 set 的操作同样可以用于 unordered_map 和 unordered_set 容器。对于 multi 的无序容器也是一样的，有序容器的所有操作都可以用于无序容器。

因而，我们通常使用无序容器来替换相应的有序容器，或者执行相反的替换。然而，由于元素不是有序存储的，使用无序容器的输出与使用有序容器的相同程序的输出通常是不一样的。

管理 bucket

无序容器被组织为一系列桶(bucket)，每个桶中装有零个或多个元素。这些容器使用哈希函数(hash function)来将元素映射到桶上。为了访问元素，容器首先计算元素的哈希值（hash code），用以得到需要搜索的桶。容器将所有哈希值相同的元素都放在一个相同的桶中。如果容器允许一个给定的 key 有多个元素与之相关联，那么这些关联的元素都在同一个桶中。因而，无序容器的性能取决于哈希函数的质量以及桶的数量（number）和大小（size）。

其中哈希函数必须对于相同的参数总是产生相同的结果。理想情况下，哈希函数将每个特定的值映射到独一无二的桶上。然而，一个哈希函数允许将多个不同的 key 的元素映射到相同的桶上。当一个桶中有多个元素时，这些元素将被顺序查找以发现我们想要的那一个。通常情况下，计算元素的哈希值和查找其 bucket 是很快的操作。然而，如果桶中有很多元素，将需要非常多的比较操作来找到那个特定的元素。

无序容器提供了一系列函数用于管理 bucket。这些成员函数让我们可以查询容器的状态以及强制容器在需要时重新组织。

Table 11.8 无序容器管理操作

Bucket Interface

404. `c.bucket_count()` 查询正在使用的 bucket 的个数；
405. `c.max_bucket_count()` 这个容器可以容纳的 bucket 的最大数目；
406. `c.bucket_size(n)` n 号桶容纳的元素的个数；
407. `c.bucket(k)` 返回键 k 可能被找到的桶，类型是 `size_type`

Bucket 迭代器

408. `local_iterator` 可以访问 bucket 中的元素的迭代器类型；
409. `const_local_iterator` bucket iterator 的 const 版本；
410. `c.begin(n)` `c.end(n)` bucket n 中的首元素迭代器以及尾后元素迭代器；
411. `c.cbegin(n)` `c.cend(n)` 上一条目中的 const 版本迭代器；

hash 策略

412. `c.load_factor()` 每个 bucket 中的平均元素个数，返回值是 float 类型；
413. `c.max_load_factor()` c 试图维护的桶的平均大小。c 会增加桶的数量以维护 `load_factor <= max_load_factor`，返回值是 float 类型；
414. `c.rehash(n)` 重新调整存储从而 `bucket_count >= n` 并且 `bucket_count > size/max_load_factor`，如果 n 大于容器的当前桶的数目（`bucket_count`），rehash 将强制执行，新的桶数目将大于或等于 n，如果 n 小于容器的当前桶的数目那么这个函数可能没有任何作用；
415. `c.reserve(n)` 重新调整存储从而 c 可以包含 n 个元素而不需要 rehash；

无序容器的 key 类型要求

默认情况下，无序容器使用相等操作符于键类型上来比较元素。它们使用 `hash<key_type>` 类型的对象来产生每个元素的哈希值。标准库提供了内置类型（包括指针）的 hash 模板来产生哈希值。标准库同时还定义了一些库类型的 hash 模板，包括 string 类型和智能指针类型。因而，我们可以直接定义内置类型或者 string 或智能指针的无序容器。

然而，我们不能直接定义我们自己的类类型作为键的无序容器。与容器不一样，我们不能直接使用 hash 模板。相反，我们必须提供自己的 hash 模板版本，在 16.5 中将介绍使用类模板特例化（class template specializations）来提供自己的 hash 模板版本。

除了使用默认的 hash 模板之外，我们还可以提供自己的相等比较函数和计算哈希值的函数。这个技术在 11.2.2 中使用过。如：

```
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
}

using SD_multiset = unordered_multiset<Sales_data, decltype(hasher)*, decltype(eqOp)*>;

SD_multiset bookstore(42, hasher, eqOp);
```

上面的代码同时指定了 hash 函数和相等性比较函数，如果我们的类有自己的 == 操作符，可以仅仅只覆盖哈希函数：

```
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

新版本的 C++ 最重要的更新之一就是提供了更为强大的智能指针（smart pointer），智能指针是模拟指针的抽象数据结构，提供了额外的功能包括内存管理（memory management）或者界限检查（bounds checking）。这些特性在保留性能的情况下，减少了因为指针滥用导致的难以查找的 bug。智能指针常用于跟踪其指向的内存，亦可用于管理其它资源，比如：网络连接和文件句柄。

智能指针能够自动回收内存，对象自动析构。但拥有对象的最后一个智能指针销毁时（本地变量离开其作用域），其会自动析构对象并回收内存。其中 shared_ptr 通过引用计数来实现，而 unique_ptr 则完全拥有其指向的对象。

有自动垃圾回收机制的语言不需要智能指针用于内存管理，但依然可以用于缓存管理和其它资源管理（如：文件句柄和网络）。
前面章节介绍的对象使用都没有动态内存。全局对象在程序启动时分配，并在程序结束时销毁。本地自动对象在指令流经过其定义位置时创建，在离开其创建块时销毁。本地静态对象则在第一次使用时创建，在程序结束时销毁。
动态分配对象（dynamically allocated object）的生命周期独立于其创建的位置，除非是显式销毁，否则将持续到程序结束。如何正确释放动态对象非常容易产生 bug。为了安全使用动态对象，C++ 提供了两种智能指针来管理动态分配对象。智能指针保证会在没有任何地方引用了此对象时，释放其内存。
之前的程序只使用了静态（static）和堆栈（stack）内存。静态内存用于本地静态变量、类静态成员和定义在任何函数之外的变量。堆栈则用于函数内定义的自动对象。在静态和堆栈内存中分配的对象，其创建和销毁都由编译器管理。除此之外，每个程序还有一个内存池，这种内存被称为自由内存（free store）或堆（heap）。程序使用堆来分配给动态对象，即在运行时分配内存给对象。这种对象的生命周期由程序进行管理，代码中必须显式销毁不再使用的对象。
除非是必要的情况下，不应该直接管理动态内存，因为，这是非常容易出错的。

12.1 动态内存和智能指针

在 C++ 中使用 new 操作来分配和初始化动态对象，并返回一个指向对象的指针。delete 操作符则以此指针为操作数，销毁其指向的对象，并释放其内存。
动态内存容易出错的原因在于：很难在正确的时机释放内存。我们可能忘记释放并造成内存泄漏（memory leak）或者在指针依然在使用时释放其内存，这种时候指针指向的内存是无效的。
为了使得动态内存易于使用并且更加安全，新标准中提供了两个智能指针来管理动态对象。`shared_ptr` 允许多个指针指向同一个对象，`unique_ptr` 则拥有其指向的对象，因而是排外的。标准库还定义了 `weak_ptr` 表示对 `shared_ptr` 管理的对象的弱引用。所有这三个类都定义在 `memory` 头文件中。

12.1.1 shared_ptr 类

指针指针是模板类，创建智能指针需要提供指向的对象类型作为模板参数。如：

```
shared_ptr<string> p1;
shared_ptr<list<int>> p2;
```

默认初始化的智能指针表示空指针。
使用智能指针的方式于常规指针是一样的。解引用智能指针返回其指向的对象引用。当将智能指针用于条件语句中，效果相当于测试其是否是空指针。
以下是 `shared_ptr` 和 `unique_ptr` 共有的操作：

416. `shared_ptr<T> sp unique_ptr<T> up` 指向 T 类型的对象的空指针；
417. `p` 将 p 用于条件中，如果其指向一个对象将返回 true；
418. `*p` 解引用 p 从而得到其指向的对象，如果没有 p 是空的，结果未定义；
419. `p->mem` 等同于 `(*p).mem`；
420. `p.get()` 返回 p 中保存的对象指针。使用时需要当心：返回的指针所指向的对象可能被智能指针删除；
421. `swap(p, q) p.swap(q)` 交换 p 和 q 中的指针；

以下是 `shared_ptr` 特有的操作：

422. `make_shared<T>(args)` 返回一个类型为 T 的动态对象的智能指针，使用 args 进行初始化对象；
423. `shared_ptr<T>p(q)` p 是 `shared_ptr` q 的拷贝，将增加 q 的引用计数，q 中指针必须可以转为 `T*`；
424. `p = q` p 和 q 是指向可转换指针的智能指针 `shared_ptr`。减少 p 的引用计数，并增加 q 的引用计数，当 p 的引用计数为 0 时，删除其所指向的动态对象的内存；
425. `p.unique()` 当 p 的引用计数是 1 时，返回 true，否则返回 false；
426. `p.use_count()` 返回 p 所指向对象的引用计数，可能是一个很慢的操作，主要用于调试目的；

make_shared 函数

最安全的分配和使用动态内存的方式就是调用库函数 `make_shared`。这个函数分配并初始化动态对象，然后返回一个指向它的 `shared_ptr` 智能指针。`make_shared` 被定义在 memory 头文件中，它是一个模板函数，调用时需要提供需要创建的对象类型。如：

```
shared_ptr<int> p3 = make_shared<int>(42);
shared_ptr<string> p4 = make_shared<string>(10, '9');
shared_ptr<int> p5 = make_shared<int>();  //此时 p5 指向的对象将被值初始化
auto p6 = make_shared<vector<string>>(); //分配一个空的 vector<string> 对象
```

make_shared 使用其参数构建一个给定类型的对象，创建类对象时传的参数必须匹配其任一构造函数的原型，创建内置类型对象则直接传递其值。如果没有传递任何参数，则对象是值初始化的。

拷贝和赋值 shared_ptr

当拷贝或赋值 shared_ptr 时，会相应更新各自对动态对象的引用计数。当拷贝 shared_ptr 时，计数增加，例如：当用于初始化另一个 shared_ptr 或者在赋值表达式中处于等号右边，或传递给函数、从函数中返回都会增加其引用计数。而当给 shared_ptr 赋予新值时，或者 shared_ptr 对象本身被销毁时，引用计数就会减少。

一旦引用计数变为 0 之后，shared_ptr 就会自动释放其指向的对象的内存。

```
auto r = make_shared<int>(42);
r = q;
```

赋值给 r，将增加 q 所指向的对象的引用，而减少 r 原本所指向对象的引用，最终结果将导致那个对象被销毁。

实现上是否使用计数器或者别的数据结构来跟踪到底有多少个计数器指向同一个对象。关键点在于 shared_ptr 类本身去跟踪有多少个智能指针指向同一个对象并且在合适的时机自动释放。

shared_ptr 自动释放其指向的对象…

当最后一个指向对象的 shared_ptr 被销毁时，其指向的对象将自动销毁。销毁对象使用的成员函数是析构函数（destructor）。类似于构造函数，每个类都有析构函数。构造函数用于控制初始化，析构函数控制当对象销毁时发生什么。

shared_ptr 的析构函数递减其引用计数，当引用计数变为 0 时，shared_ptr 的析构函数将销毁其指向的对象，并释放其内存。

并自动释放与其相关动态对象的内存

shared_ptr 可以自动释放动态对象使得使用动态内存相当简单。

```
shared_ptr<Foo> factory(T arg)
{
    return make_shared<Foo>(arg);
}
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
}
```

当 p 被销毁时，其引用计数将递减。由于 p 是指向 factory 分配的动态内存的唯一指针，当 p 被销毁时，它会自动销毁其指向的对象，并且内存将被释放。而如果有任何其它 shared_ptr 指向这个对象，那么它就不会被释放内存。

由于内存只有到了最后一个 shared_ptr 销毁后才会释放，所以重要的是确保当不再需要动态对象时，shared_ptr 对象不会一直存在。一种可能保存不必要的 shared_ptr 是在将其放在容器中，之后又调整了容器使得不再需要所有元素，应当将不需要的元素擦除。

类与具有动态生命周期的资源

程序在以下三种情况下会使用动态内存：

(47) 不知道需要多少对象；

(48) 不知道需要的对象的精确类型；

(49) 在多个对象之间共享数据；

定义 StrBlob 类

代码见：StrBlob.cc

最好的实现新集合类型的方式是使用容器库来管理元素，这样可以让容器库来管理元素的内存。然而，在 StrBlob 中不能直接存储 vector，原因在于 vector 需要在与 StrBlobPtr 共享。为了共享元素可以在其中一个销毁时依然存在，需要将 vector 放在动态内存中，并由 shared_ptr 进行管理。

值得注意的是 StrBlob 有一个以 initializer_list<string> 为参数的构造函数，这是新标准中给列初始化专门设计的。

12.1.2 直接管理内存

参考代码：new_delete.cpp

语言本身定义了两个操作符来分配和释放内存。new 用于分配，delete 用于释放。使用这两个操作符进行内存管理比之智能指针是更为易错的方式。并且，自己管理内存的类不能依赖于默认定义的拷贝、赋值和析构函数。因而，使用智能指针的程序将更加容易书写和调试。

只有当学会了如何定义拷贝、赋值和析构函数时，才能够直接管理内存，此刻就只用智能指针进行管理。

使用 new 来动态分配和初始化对象

在堆上分配的对象是不具名的（unnamed），new 没有任何方式可以给其分配的对象取名。相反，new 返回一个指向其分配的对象的指针。如：

```
//pi 指向动态分配的不具名的，未初始化的 int 值
int *pi = new int;
//ps 指向空字符串（调用默认构造函数得到）
string *ps = new string;
```

默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或符合类型的对象是未定义值。类类型对象将执行默认构造函数进行初始化。

初始化动态分配对象可以使用直接初始化形式，可以使用 C++11 之前的括号形式，亦可以使用新标准下的列初始化形式（括弧形式）。如：

```
int *pi = new int(1024);
string *ps = new string(10, '9');
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

如果在动态分配的对象后跟随一对空括号或空括弧，对象将是值初始化的。对于内置类型则为 0，对于类类型对象则调用其默认构造函数。如：

```
string *ps1 = new string; //默认初始化为空字符串
string *ps = new string(); //值初始化为空字符串
int *pi1 = new int; //*pi1 的值是未初始化的
int *pi2 = new int(); //*pi2 的值是 0
```

总是初始化动态分配的对象是一个好的习惯。

有时候可以在等号的右边使用 auto，当使用 new 操作符时，而且括号中的只有一个参数时，编译器使用括号中的值类型去推断生成的动态对象的类型。此时将使用传入的参数去初始化动态对象，它们具有一样的值和类型。如：

```
//p 指向与 obj 同类型的对象，并且以 obj 为初始值
auto p1 = new auto(obj);
//以下写法是错误的!!!
auto p2 = new auto{a,b,c};
```

动态分配 const 对象

```
const int *pci = new const int(1024);
const int *pcs = new const string;
```

所有的常量，包括动态分配的常量都必须进行初始化。对于定义了默认构造函数的类类型的动态对象可以调用其默认构造函数进行隐式初始化，所以可以不用提供初始值。而任何其它类型，特别是内置类型以及没有默认构造函数的类类型都必须进行显式初始化。由于分配的对象是 const 的，所以返回的指针亦是指向 const 的对象。

内存耗尽

当内存耗尽时，new 操作符会抛出 bad_alloc 异常。可以通过使用另一种形式的 new 来禁止 new 抛出异常，这种新形式的 new 称为定位 new（placement new），这种新的表达式可以传递额外的参数给 new。如：

```
int *p1 = new int; //失败时抛出 std::bad_alloc
int *p2 = new (nothrow) int; //失败时返回空指针
```

这里传递给 new 一个标准库中的对象 nothrow，这个对象定义在 new 头文件中。nothrow 对象告诉 new 一定不要抛出任何异常，如果无法分配内存时就返回空指针。

释放动态内存

C++ 使用 delete 操作符来释放不再使用的动态内存，其操作数是一个指向需要释放的动态对象的指针。如：

```
delete p;
```

delete 表达式做了两件事：析构指针所指向的对象，释放其内存。

指针和 delete

传递给 delete 的指针必须指向动态分配内存或者是空指针。删除一个不是由 new 分配的内存指针，或者删除一个指针两次，结果将是未定义的。

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete pi1; //未定义行为，pi1 指向本地变量
delete pd;  //ok
delete pd2; //未定义行为，删除了两次
delete pdi2; //ok，删除空指针是可以的
```

通常编译并不知道一个指针指向的是静态的还是动态分配的对象。编译器也无法知道指针指向的对象是否已经被释放过了。几乎所有编译器都认为删除指针是合法的，即便它们本身的确是错误的。

尽管动态分配的 const 对象本身不能改变，对象却是可以被销毁的。通过在 const 动态对象的指针上调用 delete 将回收其内存。

动态分配的对象只能通过 delete 进行释放

通过 shared_ptr 分配的动态对象会在最后一个 shared_ptr 销毁时自动回收。但是直接通过 new 分配的动态内存是不会自动回收的。由内置指针管理的动态对象只有显式删除才会回收内存。直接返回动态内存指针的函数将释放的内存的责任交给了调用者，调用者必须记得回收这一部分内存。但是经常会有调用者忘记这个事。

与类类型不同的事，当内置类型对象销毁时不会发生任何事情。特别是，当一个指针离开作用域，不会在其指向的对象上发生任何事情。如果指针指向了动态内存，这块内存不会自动释放。

管理动态内存是易错的

用 new 和 delete 来管理内存有三个易错的地方：

(50) 忘记 delete 内存，从而造成内存泄漏；测试内存泄漏是十分困难的，几乎只有在运行足够长并且内存被耗尽时才能发现；

(51) 在动态对象被删除之后依然还在使用；这种错误可以通过在释放内存后将指针设为空来解决；

(52) 删除同一块内存两次；这在同时有两个指针指向同一块动态内存时可能会发生。如果其中一个指针已经被删除了，然而其它的指针并不知道这块内存已经被删除了。这种错误通常很容易发生，但是很难定位；

通过在任何情况下都使用智能指针可以避免以上的错误，智能指针只有在没有其它智能指针仍然在指向这块动态内存时，才会销毁这块内存。

在删除指针所指向的对象后，重置指针

当删除一个指针时，指针将变得无效。尽管指针已经无效了，在很多机器上指针依然保有那个已经被释放的内存的地址。此时指针已经变成了悬挂指针（dangling pointer），即一个指向不存在的对象的指针。悬挂指针的问题与未初始化的指针是一样的。通过删除内存后 nullptr 赋值给指针，可以保证当删除后指针不指向任何对象。

尽管这样做了依然不能彻底解决问题，原因在于还有别的指针可能会指向相同的内存。仅仅只重置那个删除内存的指针，并不会对其它指针造成任何影响，这样其它指针依然可能会被错误使用，从而访问已经被删除的对象。在真实系统中想要找出所有的指向同一个内存的指针是非常困难的。

12.1.3 将 shared_ptr 运用于 new

当不初始化智能指针，其将被初始化为空指针。如果将智能指针初始化为一个从 new 返回的指针，那么此智能指针将接管这块动态内存。如：

```
shared_ptr<double> p1;
shared_ptr<int> p2(new int(42));
```

其它的定义和改变 shared_ptr 的方式：

427. shared_ptr<T> p(q); p 将管理由内置指针 q 所指向的对象，q 必须指向由 new 分配的的内存，并且可以转为 T*；
428. shared_ptr<T> p(u); p 接管 unique_ptr u 的对象所有权，并使得 u 为空指针；
429. shared_ptr<T> p(q, d); p 接管指针 q 所指向的对象所有权，p 将使用可调用对象 d 替换 delete 来释放 q 所指向的对象；
430. shared_ptr<T> p(p2, d); p 是 shared_ptr p2 的拷贝，增加引用计数，但是 p 用可调用对象 d 代替 delete 来释放内存；
431. p.reset() p.reset(q) p.reset(q, d) 如果 p 是指向对象的唯一指针，reset 将会释放 p 所指向的对象。如果提供了额外的内置指针 q ，将在之后使得 p 指向 q，否则将使 p 变为空指针。如果提供了可调用对象 d，将调用 d 而不是 delete 来释放内存；

以上以内置指针作为参数的构造函数是 explicit 的，因而，不能隐式将内置指针转为智能指针。我们必须使用直接初始化的形式初始化智能指针。

默认情况下用于初始化智能指针的内置指针必须指向动态内存，因为，默认情况下智能指针会使用 delete 来释放相关的对象。我们可以将智能指针指向其它类型的资源，然而，这样做必须提供我们操作来替换 delete 。

不要混合使用内置指针和智能指针

shared_ptr 只能与其它拷贝自己的 shared_ptr 配合使用。当在创建动态对象时就将其它与一个 shared_ptr 绑定，将没有机会将这个对象与另外一个独立的 shared_ptr 进行绑定。如果混合使用内置指针，将导致在智能指针已经释放掉了内存，而指针并不知道这种情况，结果将导致指针变为悬挂指针。一旦将 shared_ptr 与内置指针绑定，这个智能指针将获取内存的所有权，从而，不应该再继续使用内置指针来访问那块内存了。如：

```
void process(shared_ptr<int> ptr)
{
    //ptr 将销毁其指向的对象
}
int *x = new int(1024);
process(x);
int j = *x; //未定义的，x 是悬挂指针
```

使用内置指针访问智能指针所拥有的对象是很危险的，因为我们不知道对象将在何时被销毁。

不要使用 shared_ptr.get() 得到的指针用于初始化或者赋值给另外一个智能指针。这个函数的目的在于某些以前的函数并不接受智能指针作为参数，get 返回的指针一定不能在外部被删除。尽管编译器不检查，但使用此返回的指针绑定到另外一个智能指针是一个错误。

12.1.4 智能指针和异常

之前提到过使用异常的程序需要保证，当异常发生时资源可以被回收，其中一个简单地方法就是使用智能指针。智能指针可以保证即使是在函数异常退出地情况下依然会正确释放不再使用地内存。而内置指针则不做任何事情，由于在函数外部根本无法访问这块内存，从而就造成了内存泄漏。

```
void f()
{
    shared_ptr<int> sp(new int(42)); //即使发生异常亦能正确释放
}
void f2()
{
    int *ip = new int(42);
    delete ip; //发生异常将无法回收其内存
}
```

有一些类被设计用于 C 和 C++ 的胶水层时，通常需要用用户自己手动释放内存。可以使用用于管理动态内存的技术来管理这些资源。即将资源交给 shared_ptr 进行管理。首先需要定义一个函数来替代

delete 操作符，可以调用这个删除器（deleter）并以存储在 shared_ptr 中的指针作为参数来进行实际的清理工作。如：deleter_func.cc 展示了用法。

智能指针只有被恰当的使用才能发出作用，以下是一些约定：

432. 不要使用相同的内置指针去初始化超过一个智能指针；
433. 不要使用删除 get 函数返回的指针；
434. 不要用 get 函数返回指针去初始化或 reset 别的智能指针；
435. 当使用 get 函数返回的指针时，应当记住当最后一个智能指针销毁时，这个指针会变得无效；
436. 使用智能指针管理资源而不是内存时，记得传递一个删除器（deleter）过去；

12.1.5 unique_ptr

unique_ptr 具有其指向的对象的所有权。不似 shared_ptr ，只有一个 unique_ptr 指向一个对象，其将独占对象。对象将在 unique_ptr 销毁时被释放。

以下是 unique_ptr 特有的操作：

437. unique_ptr<T> u1 unique_ptr<T, D> u2 定义两个 unique_ptr 空指针，它们可以指向类型为 T 的对象。u1 使用 delete 来释放指针，u2 使用类型为 D 的可调用对象进行释放；
438. unique_ptr<T, D> u(d) 定义 unique_ptr 空指针，使用类型为 D 的可调用对象 d 进行对象释放；
439. u = nullptr 删除 u 所指向的对象，并使其称为空指针（只接收 nullptr 类型）；
440. u.release() 交出 u 所指向对象的控制权，返回 p 所指向对象的内置指针，并使得 u 为空指针；
441. u.reset() 删除 u 所指向的对象；
442. u.reset(q) u.reset(nullptr) 删除 u 所指向的对象，并使得 u 指向内置指针所指向的对象，否则使得 u 为空指针；

unique_ptr 没有类似于 make_shared 的函数，相反，我们通常直接将 unique_ptr 直接与 new 返回的内置指针绑定。与 shared_ptr 一样，只能使用直接初始化对其进行初始化，而不能直接用内置指针对智能指针进行等号初始化。因为 unique_ptr 拥有其指向的对象，所以，unique_ptr 不支持拷贝和赋值。

调用 release 会切断 unique_ptr 和对象之间的关系，返回的指针通常用于初始化或赋值另外一个智能指针。这样对象所有权就从一个智能指针转移到了另外一个智能指针，然而，如果我们不使用另外一个智能指针来接收这个指针，将由程序员来管理这个资源。

传递和返回 unique_ptr

不能拷贝 unique_ptr 的原则有一个例外就是可以拷贝或赋值一个即将销毁的 unique_ptr，在新标准中这叫移动（move），将在第十三章进行讨论。如：

```
unique_ptr<int> clone(int p) {
    return unique_ptr<int>(new int(p));
}
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int(p));
    return ret;
}
```

新的程序应该放弃使用 auto_ptr 的冲动，auto_ptr 是一个不好的设计，不能用于容器中，不能从函数中返回。

12.1.6 weak_ptr

weak_ptr 是一种不控制其指向的对象的生命周期的智能指针，相反它指向 shared_ptr 管理的对象。将 weak_ptr 绑定到 shared_ptr 并不会改变其引用计数。当最后一个 shared_ptr 被销毁时，其管理的对象依然会被释放，即使 weak_ptr 依然指向这个对象。因此，称之为弱指针。

以下是 weak_ptr 的常用操作：

443. weak_ptr<T> w 创建 weak_ptr 的空指针，其指向 T 类型对象；
444. weak_ptr<T> w(sp) 创建指向与 shared_ptr sp 所指向相同对象的 weak_ptr ，T 必须与 sp 所指向对象的类型可以相互转换；
445. w = p p 可以是 shared_ptr 或者 weak_ptr ，赋值后 w 指向与 p 一样的对象；

446. `w.reset()` 使得 w 为空指针；
447. `w.use_count()` 返回指向同一个对象的 `shared_ptr` 的个数；
448. `w.expired()` 如果没有 `shared_ptr` 指向对象时返回 true，否则返回 false；
449. `w.lock()` 如果已经过期，则返回一个空的 `shared_ptr` ，否则返回指向该对象的 `shared_ptr`；

使用 `weak_ptr` ，可以在不影响其指向的对象的生命周期的情况下，安全的访问该对象。

12.2 动态数组

有时候希望在程序中可以一次性分配一个数组的内存。C++ 提供了两种方式来这样做：通过新的 new ，或者通过模板类 allocator 来分离分配和初始化过程。一般来说 allocator 将得到更好的性能以及更灵活的内存管理。

然而，除非是希望直接管理动态数组，使用标准库中容器通常是更好的方式。使用容器将更加简单，更少的 bug 而且具有更好的性能。而且，可以使用默认定义的拷贝、赋值和析构函数。而动态分配数组的类则需要自己定义这些函数来进行相应的内存管理。

除非你直到如何定义拷贝、赋值和析构函数，不要在类中使用这些函数。

12.2.1 new 和数组

通过在类型后给出数组的长度，new 即为我们创建出一个动态数组，并返回指向首元素的指针。

```
int *pia = new int[get_size()];
```

方括号中的值必须是一个整数，但不必是常量。

可以用一个类型别名来动态分配数组，虽然没有用到方括号，但依然是数组形式的 `new[]`。如：

```
typedef int arrT[42];
int *p = new arrT;
delete [] p;  //即使使用了别名，其释放时依然需要方括号
```

当用 new 分配数组时，返回的并不是数组类型，而是指向首元素的指针。即使使用类型别名，返回的数据类型依然是元素的指针类型。这种情况下分配数组的事实表面上是不可见的，因为没有 `[num]`，即使这样 new 返回的依然是元素的指针。

由于分配的内存不是数组类型，所以根本不能调用 `begin` 和 `end` 标准函数。这些函数需要知道数组的维度才能返回指向首元素和尾后元素的指针。由于同样的原因，不能对动态数组使用范围 for。

初始化动态分配的数组

通常由 new 分配的对象，不论是单个对象还是数组，都是默认初始化的。可以在分配数组后加上括号使其进行值初始化，如：

```
int *pia = new int[10]; //全部是未定义值
int *pia2 = new int[10](); //值初始化为 0
```

在新标准下可以使用括弧中的值对动态分配的数组进行列初始化。如：

```
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
```

如果给的值比数组的长度小，其余的元素将被值初始化。如果多于数组长度，则无法编译通过。值得注意的是不能在括号中填入初始值来初始化元素。

动态创建空数组是合法的，当用 0 作为数组长度而执行 new 操作时，返回的合法的非 0 指针。这个指针被保证不与任何其它 new 返回的指针值一样。这个指针类似于尾后指针，其可以与其它指针进行比较、可以加减 0、可以减去自己产生 0，但是不能用此指针进行解引用。

释放动态数组

与释放单个对象不一样的是，释放动态数组必须加上 `[]` 来指示当前释放的是数组。如：

```
delete p; //p 必须指向一个动态分配的对象或者是空指针
delete [] pa; //pa 必须指向一个动态分配的数组或者是空指针
```

数组中元素以相反的顺序进行析构，即第一个元素最后析构。当所有元素都析构之后，整个内存被回收。当删除数组时，空方括号是必不可少的。它告诉编译器后面的指针 pa 指向的是一个数组。如果在删除数组时没有提供方括号，或者在删除对象时提供了方括号，那么行为将是未定义的。

即便是对于使用了类型别名的数组，其被释放时依然需要使用方括号。原因在于指针指向的首元素，而不是类型别名表面上的类型对象。

编译器并不会在我们释放数组内存时忘记给出方括号而提示我们，或者当释放单个对象时却添加了方括号。然而结果确实未定义的。

智能指针和动态数组

标准库提供了数组版本的 unique_ptr 来管理动态数组的内存。为了使用这个类，需要在模板参数中的对象类型中加上方括号。如：

```
unique_ptr<int[]> up(new int[10]);
up.release();
```

当 unique_ptr 指向数组时，不能调用箭头或点号进行成员访问。毕竟，它指向的是一个数组而不是单个对象。另一方面，可以使用此指针进行下标操作来访问数组中的元素。如：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i;
```

以下是指向数组的 unique_ptr 特有的操作，除了不支持成员访问外，其它的操作与指向单个对象的 unique_ptr 是一样的：

450. unique_ptr<T[]>u u 可以指向一个动态分配的数组，元素类型为 T；
451. unique_ptr<T[]> u(p) u 指向动态分配的数组，此数组由内置指针 p 指向；
452. u[i] 返回位置 i 处的元素， u 必须是指向数组；

与 unique_ptr 不同，shared_ptr 没有提供直接管理动态数组的支持，如果想要使用 shared_ptr 就必须得自己提供删除器。如：

```
shared_ptr<int> sp(new int[10], [](int *p){ delete[] p; });
sp.reset();
```

12.2.2 allocator 类

限制 new 使用的原因是 new 既分配内存又要构建对象。相同的，delete 既要析构又要释放内存。当我们动态创建对象时，这种方式是合适的。但当我们需要分配一大块内存时，通常会在之后需要的时候进行构建对象，在这种情况下最好是将分配和构建分离开来。以下演示 new 的局限性：

```
string *const p = new string[n];
string s;
string *q = p;
while (cin >> s && q != p + n)
    *q++ = s;
```

这里有个问题就是可能根本不需要那么多元素，但是那些没用到的元素依然需要进行初始化。另外一个原因是那些用到的元素，马上又被新值给覆盖掉了。所以，这是一种浪费。更重要的某些类根本就没有默认构造函数，根本就不可能以这种方式分配动态数组。

allocator 类

在 memory 头文件中定义了 allocator 类，其可以将分配和构建分开。它提供类型识别的分配未构建的内存。以下是 allocator 类支持的操作：

453. allocator<T> a; 定义一个可以分配 T 类型内存的 allocator 对象 a；
454. a.allocate(n) 分配足够容纳 n 个未初始化的 T 类型对象的内存；
455. a.deallocate(p, n) 释放 p 所指向的内存，其中 p 的指针类型必须是 T*，并且必须是之前由 allocate 分配的内存，n 必须是当时调用时传递的尺寸。所有这些已经构建过的对象都必须在调用此函数之前先被调用 destroy 函数进行析构；
456. a.construct(p, args) p 必须是指向类型 T 的裸内存的指针，args 则被传递给 T 类型的构造函数，args 必须符合其中一个构造函数的原型，这个构造函数将被用于构建 T 类型对象；
457. a.destroy(p) 在 p 指向的对象上进行析构，其中 p 必须是 T* 类型的；

allocator 是模板类，所以在定义 allocator 时需要提供对象类型作为模板参数。如：

```
allocator<string> alloc;
auto const p = alloc.allocate(n); //分配 n 个未构建的字符串
```

allocator 分配的内存是未构建的。新标准中允许调用 construct 成员函数来在指定位置构建对象。

```
auto q = p;
alloc.construct(q++); //构建空字符串
alloc.construct(q++, 10, 'c'); //q 是 cccccccccc
alloc.construct(q++, "hi"); //q 是 hi
```

使用没有构建的对象是一种错误。必须在构建之后才能使用对象。当使用完毕后必须调用 destroy 进行析构，destroy 函数以指向对象的指针为参数，调用其析构函数。如：

```
while (q != p)
    alloc.destroy(--q);
```

只能对已经构建的对象进行析构，如果对未构建过的对象进行析构结果将是未定义的。已经被析构的对象占用的内存，可以被用于别的对象，或者将其返回给系统。通过 deallocate 来释放整个内存，如：

```
alloc.deallocate(p, n);
```

传递给 deallocate 的指针一定不能是空指针，且必须指向由 allocate 分配内存所返回的指针，并且 n 必须与传递给 allocate 进行分配时一致。

复制和填充未初始化的内存

allocator 类定义了两个可以构建对象的算法，以下这些函数将在目的地构建元素，而不是给它们赋值：

458. `uninitialized_copy(b, e, b2)` 从由迭代器 b 和 e 指示的元素范围拷贝到由 b2 迭代器所指示裸内存。b2 必须是由 allocate 分配的，并且足够容纳拷贝进来的数据；
459. `uninitialized_copy_n(b, n, b2)` 从迭代器 b 开始拷贝 n 个值到迭代器 b2 所指示的裸内存中。限制与上面一致；
460. `uninitialized_fill(b, e, t)` 在有迭代器 b 和 e 指示的范围内，填充 t 的拷贝；
461. `uninitialized_full_n(b, n, t)` 在从迭代器 b 开始 n 个元素的裸内存上填充 t 的拷贝；

与 copy 不同的是以上 `uninitialized_copy` 是在目的地进行构建而非赋值，与 copy 一样，它也返回递增后的目的地迭代器。

关键术语

462. 分配器（allocator）标准库中的类，用于分配未构建的内存；
463. 悬挂指针（dangling pointer）：一个指向已经被回收的内存的指针，继续使用此指针是未定义行为，并且错误非常难定位；
464. 删除器（deleter）：传递给智能指针用于替换 delete 来销毁指针绑定的对象；
465. 定位 new（placement new）：new 的一种形式，具有额外的参数传递给 new 后的括号，如：`new (nothrow) int` 告诉 new 不能在无法分配内存时抛出异常；

C++ 的核心概念就是类。C++ 类定义构造函数来控制当类对象初始化时应该做什么。类同样可以定义函数来控制如何进行拷贝、赋值、移动和销毁。在这些方面 C++ 有别于其它语言，很多其它语言并不提供控制这些方面的基础设施。本章将介绍拷贝控制方面的知识，并且将引入新标准的两个概念：右值引用（rvalue reference）和移动操作（move operation）。

十四章则主要讲类类型如何进行操作符重载（operator overloading），使得类可以像内置类型一样使用内置操作符。其中值得注意的是调用操作符（function call operator），这样就可以像调用函数一样，对对象进行调用。并且将介绍新的库设施来简化不同类型的可调用对象，使其以统一的方式书写。最后还会讲特殊的成员函数——转换操作符（conversion operator），将定义从类对象到别的类型的转换。

写自己的面向对象或泛型需要对 C++ 有一个很好的理解，接下来几章的内容将是很高级的主题。

本章将学习类是如何控制当此类对象进行拷贝、赋值、移动和销毁时所做的事。类控制这些动作的成员函数分别是：拷贝构造函数（copy constructor）、移动构造函数（move constructor）、拷贝赋值操作符（copy-assignment operator）、移动赋值操作符（move-assignment operator）和析构函数（destructor）。

当我们定义类时，我们必须定义（不管是显式还是编译器生成）当对象被拷贝、移动、赋值或者销毁时发生的动作。其中拷贝、移动构造函数定义当对象由另外一个相同类型的对象进行初始化时发生的事。拷贝、移动赋值操作符定义当将类对象赋值给同类其它对象时发生的事。析构函数定义当对象消失时发生的事。以上这些操作被称为拷贝控制（copy control）。

如果类没有定义拷贝控制成员函数，编译器会自动定义这些函数。所以很多类可以忽略拷贝控制成员函数。然而，对于某些类来说，依赖于默认定义的拷贝控制会出问题。通常，实现拷贝控制操作最难的部分就是意识到什么时候需要自己定义这些函数。

拷贝控制是定义 C++ 类的非常重要的一部分。C++ 初学者通常对必须定义当发生拷贝、赋值、移动或销毁时的动作很疑惑。这个问题将变得更加复杂，因为，假如不定义的话，编译器会隐式定义这些函数，而且很有可能与我们的期望不符合。

13.1 拷贝、赋值和销毁

先从最基本的拷贝构造函数、拷贝赋值操作符和析构函数开始。

13.1.1 拷贝构造函数

只要是第一个参数是此类对象的引用并且所有其它参数都是默认值就是拷贝构造函数。其第一个参数必须是引用类型，并且几乎总是 const 引用类型，尽管也可以使用非 const 引用。拷贝构造函数将隐式用于不少场景下，因此，拷贝构造函数不应该是 explicit 的。

合成构造函数

当我们不定义拷贝构造函数时，编译器将自动合成一个。与默认构造函数不同的是，拷贝构造函数即使是在定义了其它构造函数时依然会合成。除非拷贝构造函数被定义为 delete 的（此时合成拷贝构造函数（synthesized copy constructor）将阻止拷贝该类的对象，尝试去调用将编译不通过），否则，合成构造函数将执行逐个成员拷贝（memberwiese copies）其参数的所有成员到被创建的对象中去，拷贝的顺序是按照在类中定义的顺序。

成员的类型决定了成员是如何进行拷贝的：类成员将调用其拷贝构造函数进行拷贝；内置类型成员则直接拷贝。尽管程序代码不能直接拷贝数组，编译器合成的拷贝构造函数通过拷贝其每一个元素来拷贝整个数组。类类型元素调用元素的拷贝构造函数进行拷贝。

拷贝初始化

当使用直接初始化时，编译器将使用常规的函数匹配来选择合适的构造函数以匹配程序员提供的参数。当使用拷贝初始化（copy initialization）时，编译器将右边操作数拷贝到正在创建的对象中，当需要时会对操作数进行转换。

拷贝初始化通常使用拷贝构造函数。如果一个函数有移动构造函数（move constructor），拷贝初始化在特定条件下将使用移动构造函数。

拷贝初始化不仅仅发生在定义变量时使用 =，也会出现在一下情形中：

466. 以非引用方式传递对象给函数；
467. 从函数中以非引用方式返回值；
468. 括弧初始化数组中的元素或者聚合类；

有些类会使用拷贝初始化来构建其分配的动态对象，如容器初始化时提供的元素将被拷贝初始化，用 push 和 insert 方式插入的元素亦是拷贝初始化的。相反，用 emplace 的则是直接初始化的。

参数和返回值

函数调用中以非引用形式传递的参数是拷贝初始化的。同样，函数返回非引用类型的值，返回值将被用于初始化调用位置处的结果临时量。拷贝构造函数将被用于初始化非引用类型的参数解释了为何拷贝构造函数本身的参数必须是引用。

编译器可以绕过拷贝构造函数

在拷贝初始化中，编译器允许跳过拷贝/移动构造函数，而直接创建对象，这被称为拷贝消除（copy elision），也叫做返回值优化（Return Value Optimization（RVO）），调用函数直接在栈上给返回值分配内存，然后将其地址传递给被调用者，被调用者直接在这个空间上构建对象，从而消除了从里边往外边的拷贝需求。但是即便是编译器可以消除拷贝/移动构造函数的调用，这两个函数本身必须存在，并且时可访问的。 ###

13.1.2 拷贝-赋值操作符

跟拷贝构造函数一样，如果没有定义拷贝赋值操作符，编译器会自动隐式合成一个。

重载赋值操作符介绍

重载操作符是一个函数，函数名字是 operator 后跟操作符的符号，重载赋值操作符就是 `operator=` ，操作符函数同样有返回值和参数列表。

重载操作符的参数表示操作符的操作数。其中一些操作符必须被定义为成员函数。当一个操作是成员函数，其做操作数被隐式绑定为 this 指针，而右操作数则作为参数显式传入。拷贝赋值操作符以相同类型的对象做为参数。

为了与内置类型的赋值操作符一致，重载赋值操作符通常返回其左操作数的引用。标准库要求存储在容器中的类对象有一个返回其左操作数的赋值操作符。

合成拷贝赋值操作符

如果类没有重载赋值操作符，编译器会合成拷贝赋值操作符（synthesized copy-assignment operator），与拷贝构造函数一样，一些类的合成拷贝赋值操作符会被定义为 delete ，从而禁用其赋值。否则，将使用每

个所有非静态成员的拷贝赋值操作符将其从右边操作数赋值到左边操作数。数组成员以赋值每个成员的方式来赋值。合成的拷贝操作符返回其左操作数的引用。

13.1.3 析构函数

析构函数的作用域构造函数恰好相反：构造函数负责初始化所有非静态数据成员，并做一些别的工作；析构函数负责释放对象使用的资源，并且销毁其所有非静态数据成员。

析构函数的名字是 ~ 后接类名，其没有返回值，没有参数。由于其没有参数，所以不能被重载，一个类只有一个析构函数。

析构函数所做的事

析构函数分为两部份：函数体和析构部分。在构造函数中，成员先于函数体被初始化。在析构函数中，函数体先执行，然后成员按照在类定义中出现的相反顺序进行析构。

析构函数体可以执行类设计者需要对象在生命的最后阶段需要做的事，通常，析构函数体释放一个对象分配的资源。析构函数中没有对应于构造函数的初始化列表的部分来控制成员怎样被销毁。析构部分是隐式的，析构时发生什么将由成员的类型决定，类成员执行其自己的析构函数，内置类型成员则不执行任何析构操作。

特别是，动态分配的对象所返回的内置指针作为成员，在被析构时不会自动 delete 掉其指向的对象。与之不同的是，智能指针是类类型并且有析构函数。所以，智能指针将自动销毁其指向的动态对象。

何时调用析构函数

析构函数将在对象被销毁时自动调用，以下情况下将销毁对象：

469. 变量离开作用域时将被销毁；
470. 对象成员将在对象被销毁时跟着被销毁；
471. 容器或数组中的元素将在其容器被销毁时跟着被销毁；
472. 动态分配的对象将在 delete 时被销毁；
473. 临时量将在其所在的整个表达式结束时被销毁；

由于析构函数是自动运行的，所以，程序可以分配资源并且不必考虑何时应该释放资源。析构函数不会在引用或者指向对象的指针离开作用域时自动运行。

合成析构函数

编译器为所有没有定义析构函数的类合成析构函数。与拷贝构造函数和拷贝赋值操作符一样，有些类的合成析构函数被定义为不允许类对象被析构。否则，合成析构函数的函数体是空的。

所有的成员都在函数体执行后自动销毁。值得指出的是析构函数体是不直接销毁其成员本身的。成员销毁是在析构函数体后的隐式析构部分进行销毁的。析构函数体与逐个成员析构一起构成销毁对象的过程。

13.1.4 三/五法则

以上介绍的是三个基本的拷贝控制的操作：拷贝构造函数、拷贝赋值操作符和析构函数。在新标准下，类还可以定义移动构造函数和移动赋值操作符。并不需要总是定义所有这些操作，可以定义其中一两个。但是通常这些操作需要被当作一个整体，仅仅只需要其中一个的情况很少发生。

需要析构函数的类通常需要拷贝和赋值函数

首要原则是如果一个类需要析构函数，那么几乎可以肯定需要拷贝构造函数和拷贝赋值操作符。通常，考察一个类是否需要析构函数是一个容易的事。通常一个类申请了资源，它就会需要一个析构函数。

需要拷贝构造函数的通常意味着需要拷贝赋值操作符，反之一样

一些类可以只定义拷贝和赋值对象所需要的操作函数，比如：每个对象都有自己的唯一 id 。所以，第二原则就是：如果一个类需要拷贝构造函数，它一定需要拷贝赋值操作符，并且相反是一样的。然而，需要这两者并不一定意味着需要析构函数。

13.1.5 使用 =default

当显式要求编译器为我们合成拷贝控制成员时，将它们定义为 =default 形式。当将 =default 放在类体中的声明处时，合成的函数隐式是内联的。如果不希望合成的成员是内联的，可以将 =default 放在成员定义处。只能将 =default 放在有合成版本的成员函数后。如：

```
class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
```

```
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
};
Sales_data& Sales_data::operator=(const Sales_data&) =default;
```

13.1.6 禁用拷贝

某些类并不需要拷贝控制函数，如：iostream 类。这些类必须定义成禁用拷贝控制函数。不定义这些函数是不行的，因为，编译器会隐式合成这些函数。

将函数定义为 delete

在新标准下可以通过将函数定义为被删除的函数（deleted functions），来禁用拷贝。被删除的函数是被声明但是不能被使用的函数。通过在函数后放置 = delete 来定义被删除的函数。被删除的函数不是未定义的函数，被删除的函数依然出现在函数匹配的候选函数中。但是，当其被选为最优函数时，将产生编译错误。

=delete 只能放在类定义内的成员函数声明处，不能放在定义处。原因在于，调用成员函数通常需要知道成员函数的声明。而类外的定义处则是生成函数代码的地方。

所有成员函数都可以被定义为被删除的函数。

析构函数不应该被定义为被删除的函数

如果将析构函数定义为被删除的，那么将毫无机会来销毁对象了。编译器将不允许程序定义这种类的变量或者创建临时量。并且，不能定义其成员类型有被删除的析构函数的类的变量或者临时量。尽管不能定义变量或者临时量，但是可以动态分配这种对象，除了不能删除这种动态对象。如：

```
struct NoDtor {
    NoDtor() = default;
    ~NoDtor() = delete;
};
NoDtor *p = new NoDtor();
delete p; //错误!!不能删除这种对象
```

合成的拷贝控制成员可能是被删除的

对于某些类，编译器合成的拷贝控制函数是被删除的函数：

474. 合成的析构函数是被删除的，如果类有一个成员，其析构函数是被删除的或者不可访问（private）；
475. 合成的拷贝构造函数是被删除的，如果其成员自己的拷贝构造函数是被删除的或者不可访问。或者其成员的析构函数是被删除的或者不可访问；
476. 合成的拷贝赋值操作符是被删除的，如果其成员的拷贝赋值操作符是被删除的或者不可访问，或者类有一个 const 或引用成员；
477. 合成的默认构造函数是被删除的，如果其成员的析构函数是被删除的或者不可访问，或者有一个引用成员并且没有类内初始值，或者有一个 const 成员其类没有定义默认构造函数，并且没有类内初始值；

一句话概括，如果一个类的成员没有默认构造函数、拷贝、赋值、析构函数，那么对应的成员将是被删除的函数。

也许成员的析构函数被删除或不可访问将导致合成的默认和拷贝构造函数被定义为被删除的是令人诧异的，原因是，如果对象不能被销毁，那么它就不能被创建。

尽管可以给引用赋值，但是这样做将改变引用绑定的对象的值，如果为这样的类合成拷贝赋值操作符，那么左操作数将继续绑定相同的对象，它不会绑定到右操作数所绑定的对象。这种行为通常不是所希望的那样，所以，这种合成的拷贝赋值操作符将被定义为被删除的。

换一句话说，当一个类的成员不能被拷贝、赋值或销毁时，其对应的拷贝控制成员将被合成为被删除的。

private 拷贝控制

在新标准之前，类通过将拷贝构造函数和拷贝赋值操作符定义为 private 来阻止类被拷贝。由于拷贝构造函数和拷贝赋值操作符是 private 的，用户代码不能拷贝此对象。然而，友元和成员函数依然可以进行拷贝。为了阻止友元和成员对其进行拷贝，可以将拷贝控制成员声明为 private 的，并且不提供定义。只声明但是不定义一个成员函数是合法的，尝试去使用这个未定义的成员将导致链接错误。新标准中应当尽量使用 =delete 来阻止拷贝，而不是将成员声明为 private 的。

13.2 拷贝控制和资源管理

通常，如果类有自己管理的资源（动态分配的内存、网络、文件句柄等）肯定需要定义拷贝控制成员。这些类需要定义析构函数来释放资源。一旦其需要析构函数，那就意味着肯定需要拷贝构造函数和拷贝赋值操作符。

拷贝类对象有两个设计决定：以值的方式拷贝，以指针的方式拷贝。行为与值一样的类有其自己的状态，当拷贝这种对象时，拷贝后的对象与原始对象是互相独立的，对任何一方作出改变都不会影响到另外一方。行为与指针类似的类则共享状态，当拷贝这种对象时，原始对象和拷贝后的对象具有相同的底层数据。对任何一样的改变都会影响到另外一方。

通常，类直接拷贝内置类型成员，这些成员就是值，其行为与值是完全一样的。拷贝指针的不同方式将影响对象是值还是指针。

13.2.1 类像值一样

参考代码：use_has_ptr.cc

为了表现的像值一样，每个对象都有自己的一份资源拷贝。在 HasPtr 中，拷贝构造函数将拷贝 string 而不仅仅是字符串；析构函数将释放 string；拷贝赋值操作符将释放掉对象原有的 string 然后从右操作数中拷贝 string 过来。

以值方式进行拷贝赋值操作符

赋值操作符通常结合了析构函数和拷贝构造函数的行为。与析构函数一样，赋值操作将销毁其左边的操作数的资源。与拷贝构造函数一样，赋值将拷贝其右边操作数的数据。最重要的是，即便是对象给自己赋值行为亦必须是正确的。更重要的，应该保证赋值操作即便是在过程中抛出异常，其左操作数的状态是不变的。以先拷贝右边的值，然后再释放左边的数据，然后更新指针指向新分配的字符串，来保证安全进行赋值操作。

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps);
    delete ps;
    ps = newp;
    i = rhs.i;
    return *this;
}
```

在赋值操作符中，先执行构造函数的工作：newp 的初始化与 HasPtr 的拷贝构造函数中的 ps 的初始化是一样的。与析构函数一样，然后 delete 掉 ps 当前指向的内存。剩下的就是将新的指针和 int 值拷贝到当前对象中。

当写赋值操作符的成员函数时，一个好的模式是先复制右操作数的资源到一个临时量中。在拷贝之后，就可以安全的释放掉左操作数中的数据。最后将临时量中的数据拷贝到左操作数中。

13.2.2 定义类像指针一样

参考代码：use_has_ptr2.cc

最好的将类定义为表现地与指针一样的方式是使用 shared_ptr 来管理资源。

偶尔希望直接管理资源时需要自己定义引用计数(reference count)，倾向于将引用计数器放在动态内存中，每个对象保留一个指向这个计数器的指针。

其拷贝赋值操作符同时做了拷贝构造函数和析构函数的工作。赋值操作符将增加右操作数的引用计数（拷贝构造函数的工作）并减少左操作数的引用计数，当引用计数变为 0 的时候删除掉其内存（析构函数的工作）。

13.3 交换

除了定义拷贝控制成员外，需要管理资源的类通常还会定义 swap 函数。定义 swap 函数对于希望改变元素的顺序的算法来说特别重要。这种算法在需要交换元素的顺序时调用 swap 函数。如果一个类定义了自己的 swap 函数，算法将使用类定义的版本。否则，将使用库中定义的版本，这个版本在概念上会执行一次拷贝和两次赋值。这样的定义需要多次拷贝值，如果是程序员自己定义的版本，那么只需要交换其中的指针即可。如：

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
```

```
};
inline void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);
    swap(lhs.i, rhs.i);
}
```

定义 swap 函数并不是必须的，然而定义 swap 是对分配了资源的类的重大优化。

swap 成员函数应该调用 swap，而不是 std::swap

在 swap 函数中应该调用 swap 函数而不是 std::swap ，原因在于，不加修饰的 swap 可能会调用特定于成员类型的版本。如果有一个类型特定的 swap 函数，那个版本将会优于 std 下的 swap 函数，如此，如果有特定版本将会调用这个特定版本，如果没有将会调用标准版本。

在赋值操作符中使用 swap

定义了 swap 函数的类经常将 swap 用于定义赋值操作符，这种定义方式被称为拷贝-交换（copy and swap），将其左操作数与右操作数的一个拷贝进行交换。如：

```
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    swap(*this, rhs);
    return *this;
}
```

需要注意的是 rhs 是按值传递的，因而，rhs 是调用拷贝构造函数构建的。而 swap 函数将这个本地变量与左操作数进行交换，最后这个本地变量被销毁，使得旧数据被释放。

使用拷贝交换的赋值操作是自动异常安全的，并且可以正确处理自我赋值。

13.4 拷贝控制实例

参考代码：message.cc

13.5 管理动态内存的类

一些类需要分配不定数量的内存，这种类最好是用库容器来装载数据。然而，这种策略并不适合于所有类，有些类需要自己分配数据。这些类通常定义自己的拷贝控制成员来管理内存分配。

参考代码：StrVec.cc

当调用 reallocate 时，并不需要将就数据中的字符串拷贝到新的数据中，原因是旧数据马上就要被丢弃了，此时应该移动而不是拷贝这个数据，这样就避免了给字符串重新分配内存。

移动构造函数和 std::move

通过移动构造函数可以将给定对象的资源移动到将要被构建的对象中去。移动构造函数将保证被移动的对象，其内部状态是可以被有效的析构的。如：指针变为空指针。

同时标准库中定义了一个新函数 `std::move` 在 utility 头文件中。使用 move 函数有两点需要注意：1. 如果希望调用移动构造函数需要调用 `std::move` 来告诉编译器使用移动构造函数，否则，将会使用拷贝构造函数。2. 调用 move 必须加上作用域限定符，显式告诉编译器我们调用的是哪个版本的函数；如：

```
alloc.construct(dest++, std::move(*elem++));
```

调用 move 将返回一个结果，这个结果将导致 construct 函数使用 string 的移动构造函数。由于使用了移动构造函数，由这些 string 管理的内存将不会被拷贝。相反，新构建的 string 将获取旧 string 的所有权。在移动元素之后，就可以将旧的元素给 free 掉了，移动构造函数将保证可以安全的将 string 析构掉或者被赋予新的值。

13.6 移动对象

新标准的一个特色就是可以移动对象而不是拷贝对象。当对象一旦拷贝完就销毁时，移动而不是拷贝对象将提供重大的性能提升。有一些类的资源是不可共享的，这种类型的对象可以被移动但不能被拷贝，如：IO 或 `unique_ptr`。

在早期语言版本中，并没有什么方式可以直接移动对象。即便是在不需要拷贝的时候依然会执行拷贝。同样，在之前存储在容器中的对象必须是可拷贝的，在新标准中，可以在容器中使用不可拷贝但是可以移动的对象。

库容器、string 和 `shared_ptr` 支持拷贝和移动，IO 和 `unique_ptr` 则只能移动不能拷贝。

13.6.1 右值引用

为了支持移动操作，新标准引入了一种新的引用称之为右值引用（rvalue reference）。右值引用是必须绑定到右值的引用，右值引用使用 && 符号，相较于左值引用的 & 。右值引用有一个特性就是其只能绑定到即将销毁的对象上，因而，可以自由的移动右值引用对象中的资源。

左值表示对象的身份，而右值表示对象的值。不能将左值引用（lvalue reference）绑定到需要转型的值、字面量或者返回右值的表达式上。右值引用则刚好相反：可以将右值引用绑定到以上的值，但不能直接将右值引用绑定到左值。如：

```
int i = 42;
int &r = i;
int &&rr = i; //错误：不能将右值引用绑定到左值上
int &r2 = i * 42; //错误：不能将左值引用绑定到右值上
const int &r3 = i * 42; //可以将 const 左值引用绑定到任何类型的值上（const/非 const 的左/右值）
int &&rr2 = i * 42; //将右值引用绑定到右值上
```

返回左值引用的函数和赋值、下标操作、解引用和前缀自增/自减操作符都是返回左值的表达式，可将左值引用绑定到这些表达式的结果中。

返回非引用类型的函数与算术、关系、位操作和后缀自增/自减的操作符都是返回右值的表达式，可将 const 左值引用和右值引用绑定到这种表达式上。

左值持久；右值短暂

左值具有持久的状态，而右值要么是字面量，要么就是临时量。由于右值引用只能绑定到临时量上，可知其绑定的对象是即将被销毁的，对象是被独占的，没有其它对象使用它。这些事实告诉我们使用右值引用的代码可以自由的移动右值引用所绑定对象的资源。

右值引用绑定即将被销毁的对象，因而，可以从右值引用所绑定对象中自由的移动资源。

变量是左值

一个变量就是一个表达式，其只有一个操作数而没有操作符。变量表达式是左值。因而，不能将右值引用绑定到一个定义为右值引用的变量上。如：

```
int &&rr1 = 42;
int &&rr2 = rr1; //错误：rr1 是左值，因而不能这样定义
```

一个变量就是一个左值；不能直接将右值引用绑定到一个变量上，即使这个变量被定义为右值引用类型也不可以。

move 库函数

可以显式将左值强转为对应的右值引用类型，也可以通过调用 move 库函数来获取绑定到左值的右值引用，其被定义在 utility 头文件中。如：

```
int &&rr3 = std::move(rr1);
```

调用 move 告知编译器，以右值方式对象一个左值。特别需要了解的是调用 move 将承诺：不会再次使用 rr1 ，除非是赋值或者析构。当调用了 move 之后，不能对这个对象做任何值上的假设。可以析构或赋值给移动后的对象，但在此之前不能使用其值。

使用 move 的代码应该使用 std::move ，而不是 move，这样做可以避免潜在的名字冲突。

13.6.2 移动构造函数和移动赋值

为了让我们自己的类可以执行移动操作，需要定义移动构造函数和移动赋值操作符。这些成员类似于对应的拷贝赋值操作，但是他们将从给定对象中偷取资源而不是复制。

与拷贝构造函数一样，移动构造函数也有一个引用类型的初始参数。不同于拷贝构造函数的是，移动构造函数的是右值引用。与拷贝构造函数一样，其它的参数必须有默认实参。

除了移动资源，移动构造函数需要保证移动后的对象的状态是析构无害的。特别是，一旦资源被移动后，原始对象就不在指向移动了的资源，这些所有权被转移给了新创建的对象。如：

```
StrVec::StrVec(StrVec &&s) noexcept :
    elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    s.elements = s.first_free = s.cap = nullptr;
}
```

与拷贝构造函数不同，移动构造函数并不会分配新资源；其将攫取参数中的内存，在此之后，构造函数体将参数中的指针都设置为 nullptr，当一个对象被移动后，这个对象依然存在。最后移动后的对象将被析构，意味着析构函数将在此对象上运行。析构函数将释放其所拥有的资源，如果没有将指针设置为 nullptr 的，就会将移动了的资源给释放掉。

移动操作，库容器和异常

移动操作通常不必自己分配资源，所以移动操作通常不抛出任何异常。当我们写移动操作时，由于其不会抛出异常，我们应当告知编译器这个事实。除非编译器知道这个事实，它将必须做额外的工作来满足移动构造操作将抛出异常。

通过在函数参数列表后加上 noexcept ，在构造函数时则，noexcept 出现在参数列表后到冒号之间，来告知编译器一个函数不会抛出异常。如：

```
class StrVec {
public:
    StrVec(StrVec &&) noexcept;
};
StrVec::StrVec(StrVec &&s) noexcept : { ... }
```

必须同时在类体内的声明处和定义处同时指定 noexcept。

移动构造函数和移动赋值操作符，如果都不允许抛出异常，那么就应该被指定为 noexcept。

告知移动操作不抛出异常是由于两个不相关的事实：第一，尽管移动操作通常不抛出异常，它们可以这样做。第二，有些库容器在元素是否会在构建时抛出异常有不同的表现，如：vector 只有在知道元素类型的移动构造函数不会抛出异常才使用移动构造函数，否则将必须使用拷贝构造函数；

移动赋值操作符

代码如：

```
StrVec& StrVec::operator=(StrVec &&rhs) noexcept
{
    if (this == &rhs)
        return *this;
    free();
    elements = rhs.elements;
    first_free = rhs.first_free;
    cap = rhs.cap;

    rhs.elements = rhs.first_free = rhs.cap = nullptr;
    return *this;
}
```

移动赋值操作符不抛出异常应当用 noexcept 修饰，与拷贝赋值操作符一样需要警惕自赋值的可能性。移动赋值操作符同时聚合了析构函数和移动构造函数的工作：其将释放左操作数的内存，并且占有右操作数的内存，并将右操作数的指针设为 nullptr。

移动后的对象必须是可以析构的

移动对象并不会析构那个对象，有时在移动操作完成后，被移动的对象将被销毁。因而，当我们写移动操作时，必须保证移动后的对象的状态是可以析构的。StrVec 通过将其指针设置为 nullptr 来满足此要求。除了让对象处于可析构状态，移动操作必须保证对象处于有效状态。通常来说，有效状态就是可以安全的赋予新值或者使用在不依赖当前值的方式下。另一方面，移动操作对于遗留在移动后的对象中的值没有什么特别要求，所以，程序不应该依赖于移动后对象的值。

例如，从库 string 和容器对象中移动资源后，移动后对象的状态将保持有效。可以在移动后对象上调用 empty 或 size 函数，然而，并不保证得到的结果是空的。可以期望一个移动后对象是空的，但是这并不保证。

以上 StrVec 的移动操作将移动后对象留在一个与默认初始化一样的状态。因而，这个 StrVec 的所有操作将与默认初始化的 StrVec 的操作完全一样。其它类，有着更加复杂的内部结构，也许会表现的不一致。在移动后操作，移动后对象必须保证在一个有效状态，并且可以析构，但是用户不能对其值做任何假设。

合成移动操作

编译器会为对象合成移动构造函数和移动赋值操作符。然而，在什么情况下合成移动操作与合成拷贝操作是十分不同的。

与拷贝操作不同的，对于某些类来说，编译器根本不合成任何移动操作。特别是，如果一个类定义自己的拷贝构造函数、拷贝赋值操作符或析构函数，移动构造函数和移动赋值操作符是不会合成的。作为结果，有些类是没有移动构造函数或移动赋值操作符。同样，当一个类没有移动操作时，对应的拷贝操作将通过函数匹配被用于替代移动操作。

编译器只会在类没有定义任何拷贝控制成员并且所有的非 static 数据成员都是可移动的情况下才会合成移动构造函数和移动赋值操作符。编译器可以移动内置类型的成员，亦可以移动具有对应移动操作的类类型成员。

移动操作不会隐式被定义为删除的，而是根本不定义，当没有移动构造函数时，重载将选择拷贝构造函数。

当用 =default 要求编译器生成时，如果编译器无法移动所有成员，将会生成一个删除的移动操作。被删除的函数不是说不能被用于函数重载，而是说当其是重载解析时最合适的候选函数时，将是编译错误。

478. 与拷贝构造函数不同，当类有一个成员定义了自己的拷贝构造函数，但是没有定义移动构造函数时使用拷贝构造函数。当成员没有定义自己的拷贝操作但是编译器无法为其合成移动构造函数时，其移动构造函数被定义为被删除的。对于移动赋值操作符是一样的；
479. 如果类有一个成员其移动构造函数或移动赋值操作符是被删除的或不可访问的，其移动构造函数或移动赋值操作符被定义为被删除的；
480. 与拷贝构造函数一样，如果其析构函数是被删除的或不可访问的，移动构造函数被定义为被删除的；
481. 与拷贝赋值操作符一样，如果其有一个 const 或引用成员，移动赋值操作被定义为删除的；

如果一个类定义自己的移动构造函数或移动赋值操作符，那么合成的拷贝构造函数和拷贝赋值操作符都将被定义为被删除的。

右值移动，左值拷贝

当一个类既有移动构造函数又有拷贝构造函数，编译器使用常规的函数匹配来决定使用哪个构造函数。拷贝构造函数通常使用 const StrVec 引用类型作为参数，因而，可以匹配可以转为 StrVec 类型的对象参数。而移动构造函数则使用 StrVec && 作为参数，因而，只能使用非 const 的右值。如果调用拷贝形式的，需要将参数转为 const 的，而移动形式的却是精确匹配，因而，右值将调用移动形式的。

右值在无法被移动时进行拷贝

如果一个类有拷贝构造函数，但是没有定义移动构造函数，在这种情况下编译不会合成移动构造函数，意味着类只有拷贝构造函数而没有移动构造函数。如果一个类没有移动构造函数，函数匹配保证即便是尝试使用 move 来移动对象时，它们依然会被拷贝。

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); //拷贝构造函数
};
Foo x;
Foo y(x);  //拷贝构造函数；x 是左值
Foo z(std::move(x)); //拷贝构造函数；因为没有移动构造函数
```

调用 move(x) 时返回 Foo&& ，Foo 的拷贝构造函数是可行的，因为可以将 Foo&& 转为 const Foo& ，因而，使用拷贝构造函数来初始化 z 。

使用拷贝构造函数来替换移动构造函数通常是安全的，对于赋值操作符来说是一样的。拷贝构造符合移动构造函数的先决条件：它将拷贝给定的对象，并且不会改变其状态，这样原始对象将保持在有效状态内。

拷贝和交换赋值操作与移动

```
class HasPtr {
public:
    HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) {
        p.ps = 0;
    }
    HasPtr& operator=(HasPtr rhs)
    {
```

```
        swap(*this, rhs);
        return *this;
    }
};
```

赋值操作符的参数是非引用类型的，所以参数是拷贝初始化的。根据参数的类型，拷贝初始化可能使用拷贝构造函数也可能使用移动构造函数。左值将被拷贝，右值将被移动。因而，这个移动操作符既是拷贝赋值操作符又是移动赋值操作符。如：

```
hp = hp2;
hp = std::move(hp2);
```

所有五个拷贝控制成员应该被当做一个整体：通常，如果一个类定义了其中任何一个操作，它通常需要定义所有成员。有些类必须定义拷贝构造函数，拷贝赋值操作符和析构函数才能正确工作。这种类通常有一个资源是拷贝成员必须拷贝的，通常拷贝资源需要做很多额外的工作，定义移动构造函数和移动赋值操作符可以避免在不需要拷贝的情况的额外工作。

移动迭代器

在新标准中，定义了移动迭代器（move iterator）适配器。移动迭代器通过改变迭代器的解引用操作来适配给定的迭代器。通常，迭代器解引用返回元素的左值引用，与其它迭代器不同，解引用移动迭代器返回右值引用。调用函数 `make_move_iterator` 将常规迭代器变成移动迭代器，移动迭代器的操作与原始迭代器操作基本一样，因而可以将移动迭代器传给 `uninitialized_copy` 函数。如：

```
uninitialized_copy(make_move_iterator(begin()), make_move_iterator(end()));
```

值得一提的是标准库没有说哪些算法可以使用移动迭代器，哪些不可以。因为移动对象会破坏原始对象，所以将移动迭代器传给那些不会在移动后访问其值的算法才合适。

慎用移动操作：由于移动后的对象处于中间状态，在对象上调用 std::move 是很危险的。当调用 move 后，必须保证没有别的用户使用移动后对象。

谨慎克制的在类内使用 move 可以提供重大的性能提升，在用户代码中使用 move 则更可能导致难以定位的 bug，相比较得到的性能提升是不值得的。

在类实现代码外使用 std::move，必须是在确实需要移动操作，并且保证移动是安全的。

13.6.3 右值引用和成员函数

除了构造函数和赋值操作符外提供拷贝和移动版本亦会受益。这种可以移动的成员函数中一个使用 const 左值引用，另一个使用非 const 右值引用。如：

```
void push_back(const X&); //拷贝：绑定到任何类型的 X
void push_back(X&&); //移动：绑定到可修改的右值 X
```

可以传递任何可以转换为类型 X 的对象给拷贝版本，这个版本从参数中拷贝数据。只能将非 const 右值传递给移动版本。此版本比拷贝版本更好的匹配非 const 右值（精确匹配），因而，在函数匹配中将是更优的，并且可以自由的从参数中移动资源。

通常上面这种重载方式不会使用 `const X&&` 和 `X&` 类型的参数，原因在于移动数据要求对象是非 const 的，而拷贝数据则应该是 const 的。

以拷贝或移动的方式对函数进行重载，常用的做法是一个版本使用 const T& 为参数，另外一个版本使用 T&& 为参数。

右值与左值引用的成员函数

有些成员函数是只允许左值调用的，右值是不能调用的，如：在新标准前可以给两个字符串拼接的结果赋值：`s1 + s2 = "wow!";`，在新标准中可以强制要求赋值操作符的左操作数是左值，通过在参数列表后放置引用修饰符（reference qualifier）可以指示 this 的左值/右值特性。如：

```
class Foo {
public:
    Foo& operator=(const Foo&) &;
};
Foo& Foo::operator=(const Foo& rhs) &
{ return *this; }
```

引用修饰符可以是 & 或者 && 用于表示 this 指向左值或右值。与 const 修饰符一样，引用修饰符必须出现在非 static 成员函数的声明和定义处。被 & 修饰的函数只能被左值调用，被 && 修饰的函数只能被右值调用。

一个函数既可以有 const 也可以有引用修饰符，在这种情况下，引用修饰符在 const 修复符的后面。

如：

```
class Foo {
public:
    Foo someMem() const &;
};
```

重载带引用修饰符的成员函数

可以通过函数的引用修饰符进行重载，这与常规的函数重载是一样的，&& 可以在可修改的右值上调用，const & 可以在任何类型的对象上调用。如：

```
class Foo {
public:
    Foo sorted() &&; //可以在可修改的右值上调用
    Foo sorted() const &; //可以在任何类型的 Foo 上调用
};
```

当定义具有相同名字和相同参数列表的成员函数时，必须同时提供引用修饰符或者都不提供引用修饰符，如果只在其中一些提供，而另外一些不提供就是编译错误。如：

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; //错误：必须提供引用修饰符

    //全不提供引用修饰符是合法的
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);
    Foo sorted(Comp*) const;
};
```

关键术语

482. 拷贝-交换(copy and swap)：一种书写赋值操作符的技术，先将右操作数拷贝到参数中，然后调用 swap 将其与左操作数进行交换；
483. 拷贝赋值操作符（copy-assignment operator）：拷贝赋值操作符与本类的 const 引用对象作为参数，返回对象的引用。如果类不定义拷贝赋值操作符，编译器将合成一个；
484. 拷贝构造函数（copy constructor）：将新对象初始化为本类的另一个对象的副本的构造函数。拷贝构造函数将在以非引用方式传递参数或从函数中返回时默认调用。如果类不定义的话，编译器将合成一个；
485. 拷贝控制（copy control）：用于控制对象被拷贝、移动、赋值和销毁时应当做什么的成员函数。如果类不定这些函数，编译器将在合适的时候合成它们；
486. 拷贝初始化（copy initialization）：使用 = 形式的初始化，或者当传递参数、按值形式返回值，或者初始化数组或聚合类时，将进行拷贝初始化。拷贝初始化将根据初始值是左值还是右值，使用拷贝构造函数或者移动构造函数；
487. 被删除的函数（deleted function）：不被使用的函数，通过 =delete 来删除函数。使用被删除的函数是告知编译器在进行函数匹配时，如果匹配到被删除的函数就报编译器错误；
488. 析构函数（destructor）：当对象离开作用域时调用的特殊成员函数来清理对象。编译器自动销毁每个数据成员，类成员通过调用其析构函数进行销毁，内置类型或符合类型将不做任何析构操作，特别是指向动态对象的指针不会被自动 delete；
489. 逐个成员拷贝/赋值（memberwise copy/assign）：合成的拷贝/移动构造函数和拷贝/移动赋值操作符的运作方式。依次对所有的数据成员，拷贝/移动构造函数通过从参数中拷贝/移动对应的成员进行初始化；拷贝/移动赋值操作符则依次对右操作数的各个成员进行拷贝/移动赋值；内置类型的成员是直接进行初始化或赋值的。类类型成员则调用对应的拷贝/移动构造函数或拷贝/移动赋值操作符；

490. move 函数（move function）：用于将左值绑定到右值引用的库函数。调用 move 将隐式保证不会使用移动后的对象值，唯一的操作是析构或者赋予新值；
491. 移动赋值操作符（move-assignment operator）：参数是右值引用的赋值操作符版本。通常移动赋值操作符将其右操作数的数据移动到左操作数。在赋值后，必须保证可以安全的析构掉右操作数；
492. 移动构造函数（move constructor）：以右值引用为参数的构造函数。移动构造函数将参数中的数据移动到新创建的对象中。在移动后，必须保证可以安全地析构掉右操作数；
493. 移动迭代器（move iterator）：迭代器适配器，包装一个迭代器，当其解引用时返回右值引用；
494. 右值引用（rvalue reference）：对即将被销毁的对象的引用；

在第 4 章中，我们了解到 C++ 定义了大量操作符，并且为内置类型之间的转换定义了自动转换。这些使得程序员可以很方便的写出混合类型的表达式。
同时 C++ 允许我们定义当将操作符运用于类类型对象时的含义。它还允许我们定义类类行之间的转换。类类型转换的使用类似于内置类型转换在需要时将一个类型的对象隐式转为另外一个类型对象。
操作符重载（operator overloading）允许我们定义运用于类类型的操作符的含义。谨慎地使用操作符重载可以使得程序更加容易读和写。如果之前地 `Sales_item` 类定义了重载的输入、输出和加操作符，那么可以如下操作：

```
cout << item1 + item2;
```

如果没有定义这些重载操作符的话，操作就会没有那么简洁：

```
print(cout, add(data1, data2));
```

14.1 基本概念

重载操作符是具有特殊名字的函数：关键字 `operator` 后跟被定义的操作符的符号。与任何别的函数一样，重载操作符有返回值类型、参数列表和函数体。
重载的操作符函数具有与操作符的操作数一样的参数个数。一元操作符只有一个参数；二元操作符有两个参数。在二元操作符中，左边的操作数传递给第一个参数，右边的操作数传递给第二个参数。除了重载函数调用操作符（function-call operator）`operator()` 之外，重载的操作符没有默认的参数。
如果操作符函数是一个成员函数，第一个（左边）操作数将绑定到隐式的 this 指针。由于第一个操作数隐式绑定到 this，成员操作符函数的显式参数将比操作符的操作数少一个。
注意：当重载操作符是成员函数，this 将绑定到左手边的操作数。成员操作符函数将少一个显式的参数。
一个操作符函数必须要么是类的成员要么至少有一个参数是此类类型：

```
//错误：不能重新定义内置类型的操作符
int operator+(int, int);
```

这个限制意味着我们不能改变内置类型的操作符的含义。
我们可以重载大部分但不是全部的操作符。下表将说明哪些是可以重载的，哪些是不能重载的操作符。
`new` 和 `delete` 将在 19.1.1 节说明。

可以重载的操作符

```
+ - * / % ^
& | ~ ! , =
< > <= >= ++ --
<< >> == != && ||
+= -= /= %= ^= &=
|= *= <<= >>= [] ()
-> ->* new new[] delete delete[]
```

不能重载的操作符

```
:: .* . ?:
```

我们只能重载已经存在的操作符，而不能发明新的操作符符号（symbol）。对于符号如（`+ - * 和 &`）同时可以作为一元和二元操作符的。两者之一或者两种性的操作符都可以被重载。参数的个数决定哪个操作符被重载。
重载的操作符与内置类型的操作符具有相同的优先级（precedence）和结合性（associativity）。而不管操作数的类型。

直接调用重载操作符函数

通常我们通过使用操作符于合适类型的参数上间接调用重载的操作符函数。然而，我们可以通过调用常规函数的方式直接调用重载操作符函数。我们直接输入函数的名字（operator op）并传递合适类型的合适数目的参数：

```
data1 + data2; //常规的表达式
operator+(data1+data2); //相同的函数调用方式
```

这些调用是相同的：两者都调用非成员函数 operator+，并传递 data1 作为第一个参数以及 data2 作为第二个参数。

我们调用一个成员操作函数与调用任何别的成员函数的方式是一样的。我们指定对象（或指针）的名字，然后使用点号（或箭头）操作符来获取想要调用的函数：

```
data1 += data2; //表达式方式的调用
data1.operator+=(data2); //相同的成员函数调用方式
```

上面两个语句都调用成员函数 operator+=，将 this 绑定到 data1 的地址上，将 data2 作为参数传递。

有些操作符不应该被重载

回想一个有些操作符保证操作数的求值顺序是固定的。由于重载操作符就是函数调用，这些保证就不能运用于重载的操作符。尤其是，逻辑与（&&）和逻辑或（||）以及逗号操作符的操作数求值顺序就不会保留。特别是，重载版本的 && 和 || 操作符不能保留内置操作符的短路求值（short-circuit evaluation）特性。两个操作数将总是被求值。

由于这些操作符的重载版本不会保留求值顺序和/或短路求值，重载它们并不是一个好主意。如果用户习惯的方式被改变了是令人惊讶的。

不要重载逗号操作符的另外一个原因是（同样使用于取地址 & 操作符）是语言定义了当逗号和取地址操作符用于类类型对象时的含义。由于这些操作符有内置的含义，它们通常不应该被重载。

最佳实践：通常，逗号、取地址和逻辑与以及逻辑或操作符不应该被重载。

使用与内置操作符含义一致的定义

当我们设计类时，我们应该总是第一想到这个类应该提供什么操作。只有当你了解需要哪些操作时，你才能决定是将这个操作定义为常规函数或者是重载的操作符。那些在逻辑上匹配操作符的操作是定义重载操作符的好的候选对象：

495. 如果类有 IO 操作，将移位操作符定义地与内置类型的 IO 含义一致；
496. 如果类有一个操作可以比较相等性，定义 operator==，如果类有 operator=，通常需要定义 operator!=；
497. 如果一个类具有单一的自然顺序（natural ordering）操作，定义 operator<，如果类有 operator<，它通常需要所有的关系操作符；
498. 重载操作符的返回值类型通常需要与内置版本的操作符的返回值类型：逻辑和关系操作符应该返回 bool 值，算术操作符应该返回本类类型的值，赋值和复合赋值操作符应该返回左手操作数的引用；

赋值和复合赋值操作符

赋值操作符应该表现地类似于编译器合成的操作符：在赋值之后，左边和右边的操作数应该具有相同的值，并且操作符应该返回左边操作数的引用。重载的赋值操作符应该是内置类型的赋值操作的泛化（generalize）而不是绕过它。

注意：谨慎使用操作符重载

每个操作符在用于内置类型时都有一个固定含义。比如：二元 + 的含义就是表示加法。将二元 + 映射到类类型的类似操作上将提供方便的简化符号。如，标准库 string 类型遵从许多编程语言中共通的约定，用 + 表示拼接，将一个 string 添加到另外一个。

操作符重载在内置操作符能够在逻辑上映射到我们的类型上的操作时时最有用的。使用重载的操作符而不是命名的操作将使得我们程序更加自然和直观。滥用操作符重载则使得类难以理解。

明显的滥用操作符重载在现实中是十分少见的。更加常见的是扭曲一个操作符的“正常”含义来强制适用于一个给定的类型。只有在操作对于用户来说是明确（unambiguous）的时候才应该使用操作符。如果一个操作符看起来好像有多于一个解释，那么这个操作符就是模糊的。

如果一个类具有算术（arithmetic）或者按位（bitwise）操作符，那么同时提供对应的复合赋值操作符是一个好的想法。当然这些重载的操作符应该在行为上与内置操作符的含义一致。

选择作为成员或者非成员实现

当我们定义重载操作符时，我们必须决定使得操作符作为一个类成员还是一个普通的非成员函数。在某些情况下，这是没得选的，一些操作符必须是成员；在另外一些情况，我们又不能让它成为成员函数。

下面的指导方针可以帮助我们决定是否让一个操作符成为成员或者一个普通的非成员函数：

499. 赋值（=）、下标（[]）、调用（()）和成员访问箭头（->）必须被定义为成员函数；
500. 复合赋值操作符通常应该（ought）是成员，然而，不像赋值操作符，这不是必须的；
501. 改变对象状态的操作符或者与这个类类型关系十分密切（closely tied）的操作符应该（should）被定义为成员，如：自增、自减和解引用操作符；
502. 对称操作符——它们可以转换任何一个操作数，比如算术运算、相等性比较、关系比较和按位操作符——通常应该（should）被定义为常规的非成员函数；

程序员会期望将对称操作符用于混合类型（mixed types）的表达式中。比如我们可以将 int 和 double 类型值进行相加。加法是对称的，因为我们可以让左边或者右边操作数的类型作为重载操作符的类型。如果我们想要提供类似的混合类型表达式于类对象上，那么操作符就必须被定义为非成员函数。

当我们将一个操作符定义为成员函数时，左边操作数将必须是操作符作为成员的类的对象。如：

```
string s = "world";
string t = s + "!";
string u = "hi" + s; //如果 + 是 string 的成员，此句将是错误
```

如果 operator+ 是 string 类的一个成员，那么第一个加法将等价于 s.operator+("!")，同样，"hi" + s 将等价于 "hi".operator+(s)，然而类型 "hi" 是 const char*，那个类型是内置类型；它根本没有成员函数。

由于 string 将 + 定义为普通的非成员函数，"hi" + s 等价于 operator+("hi", s)，与任何函数调用一样，其中任意一个实参都可以转为合适的形参类型。它唯一的要求就是至少有一个操作数是 string 类型，且两个操作数都可以明确地转换为 string。

14.2 输入输出操作符

正如我们所见，IO 标准库使用 >> 和 << 来表达输入和输出。IO 标准库自身定义如何读写内置类型的这些操作符的版本。如果类需要支持 IO，那么同样需要定义自己的这些操作符的版本。

14.2.1 重载输出操作符 <<

通常输出操作符的第一个形参是一个非 const ostream 对象的引用。第二个参数应该是一个我们想要打印的类类型的 const 对象引用。为了以其它的输出操作符一致，operator<< 通常应该返回其 ostream 参数。如：

```
ostream & operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

输出操作符通常只做最少的格式化

内置类型的输出操作符只做最少的格式化，特别是不打印任何换行符。用户对于类输出操作符也期待类似的行为。输出操作符只做最少的格式化将让用户控制输出的细节。

IO 操作符必须是非成员函数

符合 iostream 标准库中的约定的输入输出操作符应该被定义为常规的非成员函数。这些操作符不能我们自己的类的成员。如果是的话，那么左边的操作数将不得不是我们自己的类类型对象：data << cout;

如果他们必须是哪个类的成员的话，他们最好是 istream 或者 ostream 的成员。然而，这些对象是标准库的一部分，我们是不能添加成员到这些类的。

因而，如果我们想要为我们的类型定义 IO 操作符的话，我们必须将其定义为非成员函数。当然，IO 操作符通常需要读或写非 public 数据成员。因而，IO 操作符通常被声明为友元。

14.2.2 重载输入操作符 >>

通常输入操作符的第一个参数是输入流对象的引用，第二个参数是写入的对象的非 const 引用。操作符通常返回给定流对象的引用。第二个参数必须是非 const 的，是由于输入操作符的目的就在于将输入写入到此对象中。如：

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price;
    is >> item.bookNo >> item.units_sold >> price;
    if (is)
      item.revenue = item.units_sold * price;
    else
      item = Sales_data();
    return is;
}
```

if 检查读取是否成功，如果发生了 IO 错误，操作符将给定的对象重置为空的 Sales_data 对象。这样将保证对象处于一致的状态。（Effective C++ 要求的基本异常安全就是让对象不论任何时候都处于一致的状态，而“强烈保证”则是不论发生任何异常，对象处于不变的状态）。

注意：输入操作符必须处理可能出现的输入错误；输出操作符通常没有这样的烦恼；

在输入时发生的错误

在输入操作符中可能发生如下种类的错误：

503. 读操作可能会应为流包含了不正确的类型的数据。比如，如果想要读取两个数字类型的数据，但是输入流中包含的不是数字类型的，那么读取和接下来的使用将会失败；
504. 任何读操作都可能会遇到到达文件尾部（end-of-file）或者一些别的错误；

相较于每次读都进行检查，我们在读取所有数据之后并在使用这些数据之前进行一次检查。将对象置于有效的状态是非常重要的，因为对象可能会在错误发生前被部分地改变。

将对象置于一种有效的状态，将保护那些忽略了输入错误可能性的用户。对象将依然处于可用的状态，类似的，对象不会导致误导的结果，这是因为数据是内在一致的。

最佳实践输入操作符应该决定在错误发生时采取生么措施进行错误恢复。

指示发生的错误

一些输入操作符需要做一些额外的数据校验。如需要对数据的合法范围进行校验，或者数据是合法的格式。在这种情况下输入操作符需要设置流的条件状态(condition state)来表示错误，即便从技术上来说实际上 IO 是成功的。通常输入操作符只能设置 failbit。设置 eofbit 将暗含文件被耗尽，设置 badbit 将表示流损坏。这些错误最好是留给 IO 库自己去设置。

14.3 算术和关系操作符

通常，我们讲算术和关系运算符定义为非成员函数，这样可以让左边或者右边的操作数可以进行合适的转换。这些操作符不应该改变操作数的状态，所以参数通常是 const 引用类型。

一个算术操作符通常会产生一个新的值，这个值是计算两个操作数所得到的。这个值区别于任何一个参数，并且是在本地变量中计算的。这个操作返回这个本地变量的拷贝作为结果。定义算术操作符的类通常会定义对应的复合赋值操作符。当一个类同时具有这两个操作符时，通常讲算术操作符定义为使用复合赋值操作符时更加高效的，如：

```
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
  Sales_data sum = lhs;
  sum += rhs;
  return sum;
}
```

提示：同时定义了算术运算符和相应的复合赋值操作符的类应该将算术运算符实现为复合赋值操作。

14.3.1 相等操作符

通常，C++ 的类定义相等操作符来测试两个对象是否相等。它们通常会比较每一个数据成员，只有在对应的所有成员都相等时才会认为是相等。如：

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
  return lhs.isbn() == rhs.isbn() &&
         lhs.units_sold == rhs.units_sold &&
         lhs.revenue == rhs.revenue;
```

```
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
  return !(lhs == rhs);
}
```

这些函数的定义是十分简单的，更为重要的它们所涉及到的设计原则：

505. 如果一个类有操作来决定两个对象是否相等，它应该将函数定义为 operator== 而不是具名函数；用户会期待使用 == 来进行对象的比较；提供 == 意味着它们不需要学习和记住操作的新的名字；并且如果类定义了 == 操作符将更加容易使用标准库容器和算法；
506. 如果一个类定义了 operator==，那么这个操作符通常应该决定给定对象是否具有相等的数据；
507. 通常，相等操作符应该是可传递的，意味着如果 a == b 并且 b ==c，那么 a == c 应该同样为真；
508. 如果一个类定义了 operator==，那么它通常应该定义 operator!=，两者是相互依存的；
509. 相等或不等操作符应该将其工作交给另外一个去完成。意味着，其中一个操作符将做真正的比较对象的操作，而另外一个应该调用这个来完成其工作；

最佳实践 如果一个类具有逻辑上的相等比较操作通常应该定义 operator=，类定义 == 将使得其容易与通用算法一起使用。

14.3.2 关系操作符

定义相等操作符的类同样也会定义关系操作符。特别是由于关联容器和一些算法使用小于操作符，那么定义 operator< 将十分有用。

通常关系运算符应该：

(53) 定义与作为关联容器中的键的要求一致的顺序关系；并且

(54) 如果类同时定义了 =，那么应该定义与 == 一致的顺序关系。特别是，如果两个对象有 != 的性质，那么一个对象应该 < 另外一个。

对于像 Sales_data 这种没有逻辑上的 < 概念的类型，最好是不要定义关系操作符。

最佳实践如果存在 < 操作的单一逻辑上的定义，那么通常我们应该定义 < 操作符。然而，如果类同时有 ==，只有在 < 和 == 操作符产生一致的结果时才重载 < 操作符。

14.4 赋值操作符

除了可以将相同类型的对象拷贝赋值或移动赋值给另外一个对象之外，一个类还可以定义额外的赋值操作符用于将其它类型的对象作为右边的操作数。

比如，vector 类定义了第三个赋值操作符，其参数是括号包围的元素（a braced list of elements），我们可以按如下方式使用操作符：

```
vector<string> v;
v = {"a", "an", "the"};
```

我们可以将这个操作符添加到我们自己的 StrVec 类中：

```
class StrVec {
public:
  StrVec &operator=(std::initializer_list<std::string>);
};
```

为了与内置类型的赋值操作（并且与已经定义的拷贝赋值和移动赋值操作符一致），我们的新的赋值操作符将返回左操作数的引用。

```
StrVec &StrVec::operator=(std::initializer_list<string> il)
{
  auto data = alloc_n_copy(il.begin(), il.end());
  free();
  elements = data.first;
  first_free = cap = data.second;
  return *this;
}
```

与拷贝赋值和移动赋值操作符一样，其它重载的赋值操作符应该释放掉已经存在有元素并且创建新的元素。不同于拷贝赋值和移动赋值操作符，这个操作符不需要检查自赋值（self-assignment）。参数的类型是 initializer_list<string> 意味着 il 不可能与 this 所表示的对象相同。

提示：赋值操作符可以被多次重载。赋值操作符不管参数类型是什么都必须定义为成员函数。

复合赋值操作符

复合赋值操作符并不需要必须是成员。然而，我们倾向于将所有的赋值操作包括复合赋值操作定义在类中。为了与内置复合赋值操作符保持一致，这些操作符将返回左操作数的引用。比如：

```
Sales_data &Sales_data::operator+=(const Sales_data &rhs)
{
  units_sold += rhs.units_sold;
  revenue += rhs.revenue;
  return *this;
}
```

最佳实践赋值操作符必须是成员，复合赋值操作符应该是成员。这些操作符应该返回左边操作数的引用。

14.5 下标操作符

表示容器的类型通常会定义下标操作符 operator[] 来通过位置获取元素。重载下标操作符必须是成员函数。

为了兼容常规的下标操作符的含义，重载下表操作符通常返回获取的元素的引用。通过返回引用，下标操作可以用于赋值操作的任何一边。因而，同时定义 const 和非 const 版本的操作符是一个好的主意。当运用于 const 对象时，下标操作应该应该返回一个 const 引用，那么就不能对返回的对象进行赋值。

最佳实践当一个类有下标操作符时，它通常应该定义两个版本：一个返回非 const 引用，另一个是 const 成员并返回 const 引用。如：

```
class StrVec {
public:
  std::string &operator[](std::size_t n) { return elements[n]; }
  const std::string &operator[](std::size_t n) const
  { return elements[n]; }
private:
  std::string *elements;
};
```

我们可以按照类似于对 vector 或数组进行下标操作的方式使用这些曹祖福。由于重载下标操作符返回的是一个元素的引用，如果 StrVec 是非 const 的，我们就可以对元素进行赋值；如果对 const 对象进行下标操作，我们便不能这样做：

14.6 自增和自减操作符

自增（++）和自减（--）操作符最常被迭代器类实现。这个操作符让类在序列上的元素之间移动。语言并不要求这些操作符必须是类的成员。然而，由于这些操作符改变了它们操作的对象的状态，我们倾向于让它们成为成员。

对于内置类型，同时存在前置和后置版本的自增和自减操作符。我们同样也能同时为类类型定义前置和后置的版本。

最佳实践定义自增和自减操作符的类应该同时定义前置和后置版本。这些操作符通常应该被定义为成员。

定义前置自增/自减操作符

为了演示，我们给 StrBlobPtr 类定义前置的自增和自减操作符：

```
class StrBlobPtr {
public:
  StrBlobPtr &operator++();
  StrBlobPtr &operator--();
};
```

最佳实践为了与内置类型操作符保持一致，前置操作符应该返回自增后或者自减后的对象的引用。

区别前置和后置操作符

当同时定义前置和后置版本时会遇到一个问题：正常的重载不能区分这两个操作符。前置和后置版本使用相同的符号，意味着重载版本具有相同的名字。它们同时具有相同的数目和类型的操作数。

为了解决这个问题，后置版本有一个额外的（不使用的）int 类型参数。当我们使用后置版本的操作符时，编译器给这个形参提供 0 作为实参。尽管后置版本的函数可以使用这个额外的参数，通常是不应该使用。

这个参数本身就是后置操作符正常工作所不需要的。它存在的唯一目的就是让前置版本与后置版本进行区分。如：

```
class StrBlobPtr {
public:
  StrBlobPtr operator++(int);
  StrBlobPtr operator--(int);
};
```

最佳实践为了与内置类型操作符保持一致，后置操作符应该返回旧的（未自增或者未自减）的值。这个值将作为值返回而不是引用。

注意 int 参数没有被使用，所以我们没有给其一个名字。

显式调用后置操作符

我们可以显式调用重载的操作符作为另外一种在表达式中使用操作符的方式。如果我们想要用函数调用方式调用后置版本，我们就必须自己提供这个整数参数：

```
StrBlobPtr p(a1);
p.operator++(0); //调用后置版本的 operator++
p.operator++(); //调用前置版本的 operator++
```

传递过去的值通常是被忽略的，但是依然需要传递这是为了告知编译器使用的是后置版本。

14.7 成员访问操作符

解引用（*）和箭头（->）操作符通常用于表示迭代器的类中，以及智能指针类。

```
class StrBlobPtr {
public:
  std::string &operator*() const
  { return (*p)[curr]; }
  std::string *operator->() const
  { return & this->operator*(); }
};
```

箭头操作符通过调用解引用操作符并返回那个操作符的返回元素的地址来避免做任何实际的工作。

注意箭头操作符必须是成员。解引用操作符就没有要求必须是成员，但通常应该被定义为成员。

这里值得注意的是我们将这些操作符定义为 const 成员。不像自增和自减操作符，获取成员不会改变 StrBlobPtr 自身的状态。同样需要注意的是这些操作符返回一个非 const string 对象的引用或指针。它们这样做的原因在于我们知道 StrBlobPtr 只能绑定到非 const StrBlob 对象上。以下是使用过程：

```
StrBlob a1 = { "hi", "bye", "now" };
StrBlobPtr p(a1);
*p = "okay";
cout << p->size() << endl;
cout << (*p).size() << endl;
```

箭头操作符的返回值的限制

与绝大多数其它操作符一样，我们可以定义 operator* 做任何我们喜欢的操作，如返回固定值 42 或者打印对象的内容。当箭头操作符的重载不能这么做，箭头操作符不能丢失其成员访问的基本含义。我们不能改变箭头操作符获取成员的事实。

当我们书写 point->mem 时，point 必须要么是类类型对象的指针要么是一个重载了 operator-> 的类对象。根据 point 的类型，书写 point->mem 等价于：

```
(*point).mem; // point 是内置指针类型
point.operator->()->mem; // point 是类类型对象
```

除此之外的任何含义都是错误。意味着 point->mem 执行以下逻辑：

(55) 如果 point 是指针，那么内置箭头操作符将被运用，意味着这个表达式等价于 (*point).mem，指针被解引用并且指定的成员从结果对象中取出。如果 point 指向的类型没有名字为 mem 的成员，那么代码将发生错误；

(56) 如果 point 是一个定义了 operator-> 的类对象，那么 point.operator->() 的结果将被用于获取 mem。如果结果是一个指针，那么从在这个指针上执行步骤 1。如果结果是一个自身重载了

operator->() 对象，那么步骤 2 将在那个对象上重复。这个过程一直持续到要么得到一个对象（这个对象有指定的成员）的指针，要么返回一个其它的值，这第二种情况下代码是错误的。

注意重载的箭头操作符必须要么返回一个类类型的指针要么是一个定义了自己的箭头操作符的类类型对象。

14.8 函数调用操作符

重载了调用操作符的类允许这个类型的对象就好像是函数一样使用。由于此类还存储了状态，它们将比常规函数更加的灵活。

作为一个简单的例子，下面的 absInt 就有一个调用操作符返回其参数的绝对值：

```
struct absInt {
  int operator()(int val) const {
    return val < 0 ? -val : val;
  }
};
```

这个类定义了单一操作：函数调用操作符。这个操作符以 int 类型作为实参，并返回实参的绝对值。我们通过类似函数调用的方式将参数列表运用于 absInt 对象来调用这个 () 操作符。如

```
int i = -42;
absInt absObj;
int ui = absObj(i); //将 i 传递给 absObj.operator()
```

尽管 absObj 是一个对象不是函数，我们可以“调用”这个对象。调用一个对象将运行其重载的调用操作符。在这种情况下，这个操作符取一个 int 值，并返回其绝对值。

注意函数调用操作符必须是成员函数。一个类型可以定义多个调用操作符版本，其中每一个必须在参数的个数或类型不一样。

定义了调用操作符的类对象被称为函数对象（function objects），这种对象“在行为上类似于函数”，因为我们可以调用它们。

具有状态的函数对象类（Function-Object Classes with State）

与别的类一样，函数对象类除了 operator() 外，可以有额外的成员。函数对象类经常包含数据成员用于定制调用操作符。如我们可以在调用函数时提供不同的分割符，我们可以如下定义类：

```
class PrintString {
public:
  PrintString(ostream &o = cout, char c = ' '): os(o), sep(c) { }
  void operator() (const string &s) const {
    os << s << sep;
  }
  private:
    ostream &os;
    char sep;
};
```

这个类的构造函数以输出流的引用和一个字符作为分割符。它使用 cout 和空格作为默认的实参。调用操作符的函数体则使用这些成员来定义给定的 string 对象。

当我们定义 PrintString 对象时，我们可以使用默认的或者提供我们自己的分割符或者输出流：

```
PrintString printer;
printer(s);
PrintString errors(cerr, '\n');
errors(s);
```

函数对象最长用于通用算法的实参。我们可以将 PrintString 的对象传递给 for_each 算法来打印容器中的内容：

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

14.8.1 Lambdas 是函数对象

上面的小节中，我们使用了 PrintString 对象作为实参来调用 for_each，这个用法类似于我们在§10.3.2 中使用的 lambda 表达式。当我们写 lambda 时，编译器将其翻译成一个匿名类的匿名对象（unnamed object of an unnamed class）。这个类从 lambda 中产生并包含一个函数调用操作符。如：

```
stable_sort(words.begin(), words.end(),
  [](const string &a, const string &b) {
    return a.size() < b.size();
  }
);
```

将表现得类似于下面的类的匿名对象：

```
class ShorterString {
public:
  bool operator()(const string &s1, const string &s2) const
  { return s1.size() < s2.size(); }
};
```

这个生成的类只有一个成员 —— 函数调用操作符，它以两个 string 为参数并比较它们的长度。如我们在 §10.3.3 中所见，默认情况下 lambda 不会改变其捕获变量。因而，默认情况下由 lambda 生成的类的函数调用操作符是一个 const 成员函数。如果 lambda 被声明为 mutable，那么调用操作符将不是 const 的。

表示具有捕获值的 lambda 的类

正如我们所见，当一个 lambda 按照引用捕获一个变量时，将有程序保证被引用的变量在 lambda 执行时依然存在。这样编译器就允许直接使用引用而不需要将引用作为数据成员存储在生成的类中。

作为比较，如果变量是按照值捕获的则被拷贝到 lambda 中。因而，从 lambda 中生成的类将有数据成员与每个值捕获的变量对应。这些类还有一个构造函数来初始化这些数据成员，其值来自于捕获的变量。如：

```
auto wc = find_if(words.begin(), words.end(), [sz](const string &a) { return a.size() >
sz; });
```

将产生如下类的代码：

```
class SizeComp {
public:
  SizeComp(size_t n): sz(n) { }
  bool operator()(const string &s) const {
    return s.size() >= sz;
  }
private:
  size_t sz;
};
```

不像我们的 ShorterString 类，这个类有一个数据成员以及一个构造函数来初始化这个成员。这个合成的类没有默认构造函数；为了使用这个类，我们必须传递参数：

```
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

从 lambda 表达式中生成的类有一个被删除的默认构造函数、被删除的赋值操作符以及默认析构函数。类是否有默认或删除的拷贝/移动构造函数取决于捕获的数据成员的类型，这与普通类的规则是一样的。

14.8.2 标准库中的函数对象

标准库定义一系列类来表示算术、关系和逻辑操作符。每个类定义了一个调用操作符以运用其类名所表示的操作。比如，plus 类有一个调用操作符来运用 + 于一对操作数；modulus 类定义了一个调用操作符以运用 % 操作符；equal_to 类运用 ==；等等。

这些类都是需要提供一个类型的类模板。这些类型指定了调用操作符的参数的类型。比如，plus<string> 运用 string 的加操作符于 string 对象；plus<int> 的操作数是 int；plus<Sales_data> 将 + 运用于 Sales_data 对象；等等；

```
plus<int> intAdd; //可以对两个 int 值做加法的函数对象
negate<int> intNegate; //可以对一个 int 取反的函数对象
int sum = intAdd(10, 20); // == 30
sum = intNegate(intAdd(10, 20)); // == -30
sum = intAdd(10, intNegate(10)); // == 0
```

以下是定义于 functional 头文件中的类型：

算术运算

```
plus<Type> minus<Type> multiplies<Type> divides<Type> modulus<Type> negate<Type>
```

关系比较

`equal_to<Type> not_equal_to<Type> greater<Type> greater_equal<Type> less<Type> less_equal<Type>`

逻辑运算

`logical_and<Type> logical_or<Type> logical_not<Type>`

使用标准库函数对象于通用算法

表示操作符的函数对象类经常被用于重载算法所使用的默认操作符。如我们所知，默认情况下，排序算法使用 `operator<` 来将序列排序为升序序列。为了按照降序排列，我们可以传递一个 greater 类型的对象。这个类将生成一个调用操作符以调用底层元素类型的 `operator>` ，如：

```
sort(svec.begin(), svec.end(), greater<string>());
```

这里第三个参数是一个类型为 `greater<string>` 的匿名对象。当 sort 比较元素时，不是使用 `operator<` 而调用给定 greater 函数对象。那个对象将运用 string 元素的 `>` 操作符。

这些库中的函数对象的一个重要方面就是库保证它们可以工作于指针上。回想以下比较两个不相关的指针是未定义（undefined）的。然而，我们也许想基于在内存中的地址对一个指针 vector 进行 sort，标准库函数对象就可以做到：

```
vector<string *> nameTable;
//错误：nameTable 中的指针是不相关的，所以 < 是未定义的
sort(nameTable.begin(), nameTable.end(),
  [](string *a, string *b) { return a < b; });
//ok：库保证 less 在指针类型上工作良好
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

值得说明的是关联容器使用 `less<key_type>` 来排序元素。因而，我们可以定义指针的 set 或者使用指针作为 map 中的键而不用直接指定 less 对象。

14.8.3 可调用对象和 std::function

C++ 有多种可调用对象：函数和函数指针，lambdas，由 bind 创建的对象，重载函数调用操作符的类。

与任何别的对象一样，可调用对象是有类型的。如，每个 lambda 有一个自己的唯一的匿名类类型。函数和函数指针类型根据它们的返回值类型和参数类型的不同而有所不同，等等。

然而，两个不同类型的可调用对象也许会有相同的调用签名（call signature）。这个调用签名说明了调用此对象时返回的类型以及必须传递的参数类型。下面是函数类型的调用签名：`int(int, int)` 表示一个以两个 int 为参数返回一个 int 的函数类型。

不同的类型可以有相同的调用签名

有时我们希望将有着同一个调用前面的几个可调用对象看做是同一个类型。如考察下面的不同类型的可调用对象：

```
//函数
int add(int i, int j) { return i + j; }
//lambda
auto mod = [](int i, int j) { return i % j; };
//函数对象类
struct div {
  int operator()(int denominator, int divisor) {
    return denominator / divisor;
  }
};
```

尽管它们的类型不一样，它们的调用签名是一样的：`int(int, int)`。我们也许想用这些可调用对象来创建一个简单的计算器。为了达到目的，我们需要定义一个函数表（function table）来存储这些可调用对象的“指针”。如果我们将函数表定义为如下：

```
map<string, int(*)(int, int)> binops;
```

我们可以将 add 以 `binops.insert({"+", add});` 添加进去，但是我们不能添加 `mod` ，因为 mod 是 lambda，然而每个 lambda 都有自己的类类型。这与 binops 中的值的类型是不一致。

标准库 std::function 类型

我们通过一个定义在 functional 头文件中的新的标准库类 `std::function` 来解决此问题；下表列举了定义在 function 中的操作：

510. `function<T> f;` f 是一个空的 function 对象，其可以存储 T 所表示的调用签名的可调用对象（T 是形如 `retType(args)` 的格式）；
511. `function<T> f(nullptr);` 显式构建一个空的 function；
512. `function<T> f(obj);` 存储可调用对象 obj 的一份拷贝到 f 中；
513. `f` 将 f 作为条件使用；如果 f 中持有一个可调用对象返回 true，否则返回 false；
514. `f(args)` 传递 args 去调用 f 中的对象；

定义为 function<T>的成员类型

515. `result_type` 这个 function 类型的可调用对象的返回值类型；
516. `argument_type first_argument_type second_argument_type` 当 T 只有一个或两个参数时的参数类型。如果 T 只有一个参数，`argument_type` 就是那个类型。如果 T 有两个参数，`first_argument_type` 和 `second_argument_type` 分别是哪些参数类型。

function 是模板。与其它模板一样，当我们创建 function 类型对象时我们必须提供额外的信息，在这里是调用签名，如：

```
function<int(int, int)>
```

我们可以用上面的 function 类型来表示可调用对象：接收两个 int 参数并返回一个 int 结果。如：

```
function<int(int, int)> f1 = add;
function<int(int, int)> f2 = div();
function<int(int, int)> f3 = [](int i, int j) { return i*j};
cout << f1(4, 2) << endl;
cout << f2(4, 2) << endl;
cout << f3(4, 2) << endl;
```

我们可以按照这个方式重新定义我们的 map：

```
map<string, function<int(int, int)>> binops = {
  {"+", add}, //函数指针
  {"-", std::minus<int>()}, //库函数对象
  {"/", div()}, //用户定义函数对象
  {"*", [](int i, int j) { return i * j; }}, //匿名 Lambda
  {"%", mod} //具名 Lambda
};
```

我们的 map 有五个元素，尽管其底层类型都不一样，我们可以将其存储到同一个调用签名的 function 类型下。

重载的函数和 function

我们不能直接将一个重载的函数的名字存储到 function 类型的对象中：

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data &, const Sales_data &);
map<string, function<int(int, int)>> binops;
binops.insert({"+", add}); //错误：哪一个 add？
```

一种解决这种二义性的方式是存储函数指针而不是函数的名字：

```
int(*fp)(int, int) = add;
binops.insert({"+", fp});
```

或者使用 lambda 包装一下：

```
binops.insert({"+", [](int a, int b){return add(a, b);}});
```

注意：新标准库中 function 类与之前版本中的 `unary_function` 和 `binary_function` 是不相关的。这些类已经被更加通用的 `bind` 函数给取代了。

14.9 重载、转换和操作符

在 §7.5.4 中我们看到非 explicit 的具有一个参数的构造函数定义了隐式转换（implicit conversion）。这种构造函数将参数类型的对象转换到类类型对象。我们还可以定义从类类型到别的类型的转换。我们通过定义一个转换操作符来定义从类类型的转换。转换构造函数（converting constructor）和转换操作符

（conversion operators）定义类类型转换（class-type conversions）。这些转换也被称为是用户定义的转换（user-defined conversions）。

14.9.1 转换操作符

转换操作符（conversion operator）是一种特殊的成员函数，可以将一个类类型的值转为一个其它类型的值。转换函数通常由这样一种通用的形式：`operator type() const;` 其中 `type` 表示一个类型。转换操作符可以为任何可以被函数返回的类型（除了 void）定义。转换成数组或者函数类型是被禁止的。转换成指针类型（数据和函数指针）以及引用类型是允许的。

转换操作符没有显式说明返回值类型并且没有参数，它们必须被定义为成员函数。转换操作符通常不应该改变发生转换的对象的状态。因而，转换操作符通常被定义为 const 成员。

注意转换函数必须是成员函数，并且不指定返回值类型，其参数列表必须是空的。这个函数通常应该是 const 的。

定义一个具有转换操作符的类

如下代码：code/conversion_operator.cc，其中构造函数将算术类型转为 SmallInt，转换操作符将 SmallInt 转为 int：

```
SmallInt si;
si = 4; //隐式将 4 转为 SmallInt，然后调用 SmallInt::operator=
si + 3; //隐式将 si 转为 int，然后执行整数加法
```

尽管编译器一次只会执行一个用户定义的转换 §4.11.2，隐式用户定义的转换在内置标准转换之前或之后发生。因而，我们可以传递任何算术类型给 SmallInt 的构造函数。通常，我们可以使用转换操作符将 SmallInt 转为 int 然后将结果 int 值转为别的算术类型：

```
SmallInt si = 3.14;
si + 3.14;
```

由于转换操作符是隐式运用的，没有任何途径去传递参数给这些函数。因而，转换操作符不能被定义为接收参数。尽管转换函数没有指定其返回值类型，每个转换函数都必须返回一个与其名字所表示的类型一样的值。

注意：避免滥用转换函数

与使用重载操作符一样，谨慎地使用转换操作符可以极大地简化类设计者的工作，并且使得类更加容易使用。然而，一些转换是容易被误导的。当没有明显的在类类型与要转换的类型之间的单一映射，转换操作符将是误导的。

比如，考虑表示日期的 `Date` 类，我们可能会认为在 `Date` 和 int 之间提供转换是一个好主意。然而，这个转换函数应该返回什么呢？这个函数也许会返回一个年月日的值，也可以返回从 1970 年 1 月 1 日起走过的秒数。这两种方式都可以有效表示，当问题是没有单一的一对一的映射，所以最好是不要定义转换操作符。相反，类应该定义一个或多个常规的成员函数来提取这些信息。

转换操作符可能会产生令人惊讶的结果

在实践中，类极少提供转换操作符。用户经常会惊讶于一个转换自动就发生了，而不是显式进行转换。然而，有一个重要的例外是：定义从类类型到 bool 值的转换并不少见。

在标准的早期版本中，类想要定义到 bool 的转换会面临一个问题：由于 bool 是算术运算符，类类型对象如果被转为 bool ，那么它可以用于任何算术类型被期待的上下文中。这种转换可能会发生在令人吃惊的地方。特别是，比如 istream 有一个转为 bool 的转换符，下面的代码将是合法的：

```
int i = 42;
cin << i; //cin 隐式转为 bool，之后将发生移位操作
```

显式转换操作符

为了阻止上面的问题，新标准中引入了 explicit 转换操作符（explicit conversion operators）：

```
class SmallInt {
public:
  explicit operator int() const { return val; }
};
```

与 explicit 构造函数一样，编译器不会自动运用 explicit 转换操作符进行隐式转换：

```
SmallInt si = 3;
si + 3; //错误：用到了隐式转换，但是重载的操作符是显式的
static_cast<int>(si) + 3; //ok：显式调用转换
```

如果转换操作符是 explicit 的，我们依然可以做转换。然而，除了一个例外之外，我们必须使用 cast 进行显式转换。

这个例外是编译器会将 explicit 转换用在条件中，如下：

517. 如果条件是一个 if, while 或者 do 语句；
518. 如果条件表达式在 for 语句的头部；
519. 如果作为逻辑非（!）、或（|）或者与（&&）操作符的操作数；
520. 条件操作符（?:）的条件表达式部分中；

转换为 bool 值

在早期版本的标准中，IO 类型定义了 void* 类型的转换。这样就避免了上面的问题。在新标准下，IO 库定义了 explicit 转换为 bool 的方法。

最佳实践 将类转为 bool 通常是用于条件，因而，operator bool 通常应该被定义为 explicit 的。

14.9.2 避免转换二义性

如果一个类有一个或多个转换，重要是确保从类类型到目标类型只有一条路径。如果有超过一条路径来执行转换，那么想要写出无二义性的代码将会很难。

有两种方式会导致多中转换路径。第一种是两个类都提供了相互的转化。如，相互转换存在于当类 A 定义了转换构造函数其接收类 B 的对象作为参数，并且 B 自身定义了一个转换操作符用于转为类型 A。

第二中方式是定义了多个从或到相互之间可以转换的类型的转换。最明显的例子就是内置算术类型。一个给定的类应该最多定义一个从或到算术类型的转型。

警告通常，给类之间定义相互转换，或者定义从或到两个以上的算术类型的转换是个坏主意。

参数匹配和相互转换

下面的例子将定义两种方式用于从 A 到 B 的转换：通过 B 的转换操作符或者通过 A 的构造函数其参数是 B：

```
struct B;
struct A {
  A() = default;
  A(const B&);
};
struct B {
  operator A() const;
};
A f(const A&);
B b;
A a = f(b); //错误的二义性：f(B::operator A()) 或 f(A::A(const B&))
```

由于有两种方式从 B 得到一个 A，编译其不知道该运行哪个转换；调用 f 便是模糊的。这个调用可以使用 A 的构造函数并接收 B 作为参数，或者可以使用 B 的转换操作符将 B 转为 A。由于两个函数是一样好的，这个调用便是错误的。

如果我们想要使用这个调用，我们不得不显式调用转换操作符或者构造函数：

```
A a1 = f(b.operator A());
A a2 = f(A(b));
```

注意我们不能用 cast 来解决二义性，这是由于 cast 本身就有相同的二义性问题。

二义性和多重转换到内置类型

二义性还会发生在类定义了多个转换到（或者从）相互之间相关的类型。最简单的方式是定义转换到（或从）算术类型。

如下面的类型定义两个从不类型的算术类型的转换构造函数，已经两个到不同类型的算术类型的转换操作符：

```
struct A {
  A(int = 0);
  A(double);
```

```
  operator int() const;
  operator double() const;
};
void f2(long double);
A a;
f2(a); //二义性错误，有两个一样可行的转换函数 f(A::operator int()) 和 f(A::operator double())
long lg;
A a2(lg); //二义性错误：A::A(int) 和 A::A(double);
```

在 f2 的调用中，两个转化都不是精确匹配 long double 类型。然而，两个转化都可以使用，并在之后跟者一个标准转换从而得到一个 long double。因此，没有一个转换是由于另外一个的；这个调用便是模糊的。

当我们尝试用 long 初始化 a2 时遇到了通常的问题。没有一个构造函数是精确匹配 long 的。每个都需要参数在被构造函数前进行转换，要么是从 long 到 double 的转换，要么是从 long 到 int 的转换。这两个转换序列都是可行的，因而调用是模糊的。

调用 f2 以及初始化 a2 是模糊的，这是由使用到的标准转换具有相同的优先级。当用户定义的转换被使用时，标准转换的优先级将被用于选择最佳的匹配，如：

```
short s = 42;
A a3(s); //将 short 提升为 int 要优于将 short 转为 double，因而使用的 A::A(int)
```

注意当两个用户定义的转换被使用时，标准转换的优先级如果有的话（在转换函数之前或之后运用），将被用于选择最佳匹配。

重载函数和转换构造函数

在多个转换中选择一个在我们调用重载函数时将更为复杂。如果两个或多个转换提供可行匹配，那么这些转换被认为是一样好。例如，当我们调用重载函数其参数是不同的类类型，但是定义了相同的转换构造函数时会发生二义性错误：

```
struct C {
  C(int);
};
struct D {
  D(int);
};
void manip(const C&);
void manip(const D&);
manip(10);  //二义性错误：manip(C(10)) 或 manip(D(10))
```

C 和 D 都有构造函数接收 int 类型的参数。任何一个都可以用于匹配 manip 的一个版本。因此，调用是模糊的。可以通过显式构建正确的类型对象来消除二义性。如：`manip(C(10));`

在调用重载函数的过程中需要使用构造函数或者 cast 来转换参数通常意味着差的设计。

警告：转换和操作符

正确设计重载操作符，转换构造函数以及转换函数需要小心。特别是，如果类同时定义了转换操作符和重载操作符时容易产生二义性。下面这些规则将会有所帮助：

521. 不要定义相互转换类，如果类 Foo 有一个构造函数以类 Bar 的对象为参数，不要在 Bar 中定义转换操作符到类型 Foo；
522. 避免转换内置算术类型，特别是如果你已经定义了一个到算术类型的转换，那么：
    1. 不要定义以算术类型为参数的重载操作符。如果用户需要使用这些操作符，转换操作会将你的类型的对象到内置类型，然后使用内置类型的操作符进行计算；
    2. 不要定义转换到超过一个算术类型。让标准转换提供到其它算术类型的转换；

最简单的规则是：除了定义 explicit 转换到 bool，避免定义转换函数，并且限制非 explicit 构造函数。

重载函数和用户定义的转换

如果重载函数的一个调用，有两个或更多的用户定义的转换提供可行匹配，那么转换被认为是一样好的。任何标准转换的优先级都不会被考虑。内置转换是否需要只有重载集合只有在调用匹配时使用相同的转换

函数（the same conversion function），意思是只有在都使用相同的用户定义的转换之后才会考虑接下来的标准转换。

以下，manip 即便在其中一个类的构造函数的参数需要标准转换也是二义性调用：

```
struct E {
  E(double);
};
void manip2(const C&);
void manip2(const E&);
manip2(10); //二义性错误：两个不同的用户定义的转换都可以使用：manip2(C(10)) 或 manip2(E(double(10)))
```

在这个例子中，C 有一个从 int 而来的转换，E 有一个从 double 而来的转换。调用 `manip2(10)`，两个 manip2 函数都是可行的：

523. `manip2(const C&)` 是可行的，原因是 C 有一个接收 int 的转换构造函数，这个构造函数的参数是精确匹配的；
524. `manip2(const E&)` 是可行的，原因是 E 有一个接收 double 的转换构造函数，我们可以使用标准转换来转换 int ；

由于重载函数需要不同的用户定义的转换，这个调用就是模糊的。在这里即便其中一个需要标准转换，另外一个是精确匹配，编译器依然认为这个调用是错误的。

注意 在重载函数的调用中，只有在可行函数（viable functions）需要相同的用户定义转换时，其优先级才会被考虑。如果需要不同的用户定义转换，那么这个调用就是模糊的。

14.9.3 函数匹配和重载操作符

重载的操作符是重载的函数。普通的函数匹配用于决定使用哪个操作符（内置的还是重载）运用于给定的表达式。然而，当一个操作符函数被用于表达式中，候选的函数集合要多于常规的函数调用，如果 a 是一个类类型，表达式 `a sym b` 可能是：

```
a.operator sym(b); //a 有一个重载操作符 sym 作为成员函数
operator sym(a, b); //重载操作符 sym 是常规函数
```

不同于常规的函数调用，我们不能使用调用的形式来区分我们是使用成员函数还是非成员函数。当我们将一个重载的操作符用于类类型的操作数时，候选的函数包括常规的非成员版本的操作符，以及内置版本的操作符。如果左边的操作数是类类型，并且如果它定义了成员版本的重载操作符也会被包含进来。当我们调用一个命名函数时，相同名字的成员和非成员函数并不会彼此重载。这是由于调用成员函数和非成员函数的语法形式是决然不同的。当一个调用是通过类类型对象（或者引用或者指针）时，那么只有那个类的成员函数会被考虑。当我们将重载操作符用于表达式中时，没有任何东西来指示我们是使用成员还是非成员函数。因此，成员和非成员版本都必须被考虑。

注意 在表达式中使用的操作符的候选函数集合同时包括成员和非成员函数。

如：

```
class SmallInt {
  friend SmallInt operator+(const SmallInt &, const SmallInt &);
public:
  SmallInt(int = 0);
  operator int() const { return val; }
private:
  std::size_t val;
};
```

我们可以把两个 SmallInt 对象做加法，但是当我们想进行混合加法时会陷入二义性问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2; //使用重载 operator+
int i = s3 + 0; //错误：二义性
```

第二个加法之所以是模糊的，原因在于我们可以将 0 转为 SmallInt 然后使用 SmallInt 的 operator+ 做运算，或者将 s3 转为 int 然后使用内置加法；

警告 为同一个类同时提供转换到算术类型的转换函数和重载操作符可能会在重载操作符和内置操作符之间导致二义性。

面向对象变量建立三大概念上：数据抽象（data abstraction）、继承（inheritance）和动态绑定（dynamic binding）。继承和动态绑定在两个方面影响如何写程序：使得定义类似但不相同的类更加容易，使得用户代码可以相同的方式调用它们而忽略其中的差异。
很多应用包含相关但是有略微不同的概念。面向对象编程（OOP）刚好非常适合这种应用。

15.1 面向对象：介绍

面向对象编程（object-oriented programming）的关键思想在于数据抽象、继承和动态绑定。使用数据抽象，可以将类的接口和实现进行分离。通过继承，可以定义概念上相互关联且类型相似的类。通过动态绑定，可以在使用这些对象时忽略它们的细节上的不同。

继承

通过继承关联起来的类组成了层级。通常层级的顶端是一个基类（base class），其它类直接或间接的继承之，这些继承的类称之为派生类（derived classes）。基类定义层级中所类型都共通的成员，每个派生类定义特定于派生类自己的成员。如 Quote.cc 中的 isbn() 函数在基类中定义，因为，这是在整个层级都共通的成员，而派生类定义自己的 `net_price(size_t)` 函数，因为每个类有其自己的不同策略，需要 Quote 和 Bulk_qute 类定义自己的版本。
在 C++ 中，基类区分了每个类具有不同实现的函数与希望派生类只继承而不能做出改变的成员函数。基类将希望派生类定义自己的版本的函数为 `virtual` 的。
派生类需要指定其继承的类，指定方式使用类继承列表（class derivation list），即冒号后跟着一列由逗号分隔的基类，每个基类有一个可选的访问说明符（access specifier）。如：

```
class Bulk_quote : public Quote {
public:
    double net_price(std::size_t) const override;
};
```

由于 Bulk_quote 在派生列表中使用 public，就可以像使用 Quote 一样使用 Bulk_quote，派生类必须在其类体中声明所有其想定义自己版本的基类虚函数。派生类可以在这些函数上包含 virtual 关键字，但不是必需的。新标准允许派生类通过在参数列表后包含 override 关键字，来显式说明成员函数是覆盖其继承的一个虚函数。

动态绑定

通过动态绑定，可以使用相同的代码来平滑处理基类和派生类对象，在这里是 Quote 和 Bulk_quote。示例代码：

```
double
print_total(std::ostream &os, const Quote &item, std::size_t n)
{
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn()
       << " # sold: " << n << " total due: " << ret << std::endl;
    return ret;
}
```

由于 item 是 Quote 的引用，调用函数时既可以传递 Quote 对象也可以传递 Bulk_quote 对象，并且由于 `net_price` 是虚函数，由于调用 `net_price` 函数是通过引用，具体调用哪个版本的函数，将依据传入对象的类型决定。如果传入 Bulk_quote 的对象则调用 Bulk_quote 的版本，如果传入 Quote 类的对象，则调用 Quote 的版本。
由于调用哪个版本是由实参的类型决定的，而实参类型只有在调用时才能知道。因而，动态绑定有时也被称为运行时绑定（run-time binding）。
在 C++ 中，动态绑定发生在虚函数通过基类的引用或指针调用时。

15.2 定义基类和子类

在绝大多数情况下，定义基类和派生类的方式与其它类是差不多的。

15.2.1 定义基类

作为继承层次的根类总是定义虚析构函数。基类通常应该定义虚析构函数，即便不做任何工作，虚析构函数依然是需要的。

成员函数和继承

派生类从其基类中继承成员，然而，派生类需要为特定于类型的操作提供自己的函数定义。派生类需要覆盖（override）掉其从基类继承来的定义，并提供自己的定义。
C++ 的基类必须明确区分希望派生类覆盖的函数和希望派生类继承而不改变的函数。基类将希望派生类覆盖的函数定义为 virtual 的。通过指针或引用调用虚函数，这个调用将是动态绑定的。根据引用或指针绑定的不同对象类型，基类或者其中之一的派生类的虚函数版本将被调用。
基类通过在声明前加上关键字 virtual 来指明成员函数是动态绑定的。所有的非静态成员函数（除了构造函数）都可以是 virtual 的。virtual 关键只出现在类体内的函数声明处，而不会被用于类体外的函数定义处。在基类中被定义为 virtual 的函数，其在派生类中隐式也是 virtual 的。
没有被定义为 virtual 的成员函数将在编译时确定下来，而不是运行时。如：`isbn` 函数只有一份定义，不论是以引用、指针还是对象值进行调用，都可以在编译时确定调用哪个函数，即 Quote 中的版本。

访问控制和继承

派生类继承基类中的所有成员，但这并不意味着派生类可以访问基类中的所有成员。与其它使用基类的代码一样，派生类可以使用基类的 public 成员，但是不能访问基类的 private 成员。基类将只允许派生类访问，而不允许其它用户代码访问的成员定义为 protected。

15.2.2 定义子类

派生类必须指定从哪个类继承，这是通过类继承列表（class derivation list），即冒号后的一列由逗号分隔的类名字，类必须是在之前定义过的（可以是未完成类型 incomplete type）。每个基类名字前可以放置可选的访问说明符，必须是 public 、 protected 或 private 中的一个。
派生类必须将所有要覆盖的继承来的成员函数进行类内声明。因而，Bulk_quote 类必须包含 `net_price` 的成员函数声明。
继承列表中的访问说明符将决定派生类的用户代码是否可以知道派生类从哪个基类继承而来。当继承是 public 的，基类的 public 成员变成了派生类的接口的一部分。并且，可以将公共派生的类型对象绑定到基类的指针或引用。
绝大多数类只会直接从一个基类继承，这种形式的继承称之为单继承（single inheritance），在 18 章将描述继承列表中包含多于一个基类的继承。

派生类中的虚函数

派生类经常但不总是覆盖其继承的虚函数。如果派生类不覆盖其基类的虚函数，那么与别的成员一样，派生类将继承基类中定义的版本。
派生类将在其覆盖的函数上包含 virtual 关键字，但是不是必须这么做。新标准允许派生类显式告知它将覆盖一个继承自基类的虚函数。它是通过在参数列表后指定 override 关键字，或者如果成员函数是 const 的或者有引用修饰符，那么就放在 const 或引用修饰符后。

派生类对象和派生类到基类的转换

一个派生类对象包含多个部分：包含派生类自己定义的成员的子对象，加上每一个基类的子对象。如：`Bulk_quote` 类的对象包含两个部分：自己定义的成员组成的子对象，与基类 `Quote` 子对象。由于派生类对象包含每个基类对应的子对象，可以把派生类对象当作基类对象一样使用，特别是，可以将基类对象的引用或指针绑定到派生对象的基类部分。如：

```
Quote item;
Bulk_quote bulk;
Quote *p = &item;
p = &bulk;
Quote &r = bulk;
```

这种转换称为派生类到基类的转换（derived-to-base conversion），这种转换是由编译器隐式执行的。由于这种转换是隐式的，可以将派生类对象或派生对象的引用用于需要基类对象引用的地方。同样的，可以将派生类对象的指针用于需要基类指针的地方。
派生类对象包含其基类的子对象是理解继承如何工作的关键。

派生类构造函数

尽管派生对象包含从基类继承来的成员，它不能直接初始化这些成员。与任何别的创建基类对象的代码一样，派生类必须使用基类构造函数来初始化基类部分。
每个类的构造函数控制其成员如何进行初始化。

对象的基类部分与派生类的数据成员一起在构造函数的初始化阶段进行初始化。与初始化成员一样，派生类构造函数使用构造初始值列表来传递参数给基类构造函数。如：

```
Bulk_quote(const std::string &book, double p, std::size_t qty, double disc):
    Quote(book, p), min_qty(qty), discount(disc) { }
```

当基类函数的整个函数体执行完之后，将执行数据成员的初始化，然后执行派生构造函数体。

与数据成员一样，除非是另外指定，派生类的基类部分是默认初始化的。为了调用不同的基类构造函数，可以在构造初始值中使用基类名字跟上一个参数列表，这些参数被用于选择使用哪个基类构造函数用于初始化派生对象的基类部分。

基类对象总是先初始化，然后是派生类的数据成员根据在类体中声明的顺序进行初始化。

在派生类中使用基类成员

派生类可以访问基类的 public 和 protected 成员。派生类的作用域被嵌套在基类的作用域中，那么在派生类成员函数中使用派生类自己定义的成员和使用基类中定义的成员没有区别。

关键概念：尊重基类的接口

理解每个类定义自己的接口是很重要的，与类对象进行交互时应该使用那个类的接口，即便那个对象是派生对象的基类部分。因而，派生类构造函数不会直接初始化基类的成员。派生构造函数的函数体可以给 public 和 protected 基类成员进行赋值。尽管它可以这样做，通常不应该这样做。与任何别的使用基类的用户代码一样，派生类应该尊重其基类的接口，所以应该使用基类的构造函数来初始化其继承的成员。

继承和静态成员

如果在基类中定义了静态成员，那么整个继承层级中只有此成员的唯一定义。不管从一个基类中派生了多少类，每个静态成员只存在一份实例。如：

```
class Base {
public:
    static void statmem();
};
class Derived : public Base {
    void f(const Derived &);
};
```

静态成员遵循常规的访问控制。如果一个成员在基类中是 private 的，那么派生类将无法访问它。如果成员是可访问的，则可以基类或派生类中使用此 static 成员。如：

```
void Derived::f(const Derived &derived_obj)
{
    //可以通过基类访问
    Base::statmem();
    //可以通过派生类访问
    Derived::statmem();
    //可以通过派生对象访问基类中的静态成员
    derived_obj.statmem();
    //可以通过当前对象访问
    statmem();
}
```

声明派生类

派生类的声明与常规类的声明是一样的。声明中包含类的名字，但不包含派生列表。如：

```
class Bulk_quote : public Quote; //错误：声明类中不能出现派生列表
class Bulk_quote; //正确的声明派生类的方式
```

声明的目的在于让程序知晓某个名字的存在以及它表示什么类型的实体，如：类、函数或变量。派生列表和所有其他的定义细节必须一起出现在类体中。

被用作基类的类

一个类在被用作基类之前必须定义而不能仅仅只声明。原因在于，每个派生类都包含并且可能使用其从基类继承来的成员。为了使用这些成员，派生类必须知道它们具体是什么。这也隐式说明一个类不能派生它本身。

一个类是基类，同时它是一个派生类。如：

```
class Base {};
class D1 : public Base {};
//D1 既是基类也是派生类
class D2 : public D1 {};
```

在这个层级中，Base 是 D1 的直接基类（direct base），是 D2 的间接基类（indirect base）。直接基类被放在派生列表中，间接基类是派生类通过其直接基类继承来的。

每个类都继承其直接基类的所有成员。最具体的派生类将继承其直接基类的所有成员，直接基类中的成员包含它自己从它的基类中继承来的成员，以此类推到整个继承链的顶端。所以，最具体的派生对象将包含其直接基类的子对象以及每个间接基类的子对象。

阻止继承

有时我们不希望一个类被继承，在新标准中可以通过在类名后加上 final 来阻止类被当作基类。如：

```
class NoDerived final {};
class Base {};
class Last final : public Base {}
```

15.2.3 转换和继承

理解派生类到基类的转换时理解 C++ 中面向对象编程的关键。通常，只能将引用和指针绑定到有相同类型的对象上，或者绑定到可以进行 const 转换的对象上。存在继承关系的类时一个例外：可以将基类的指针或引用绑定到这个类的派生对象上。

这个事实有很重要的暗示：当使用基类的引用或指针时，不知道绑定的对象的真实类型是什么，这个对象可能是基类对象，也可能是派生类对象。而且与内置指针一样，智能指针也支持派生类到基类的转换，可以将指向派生对象的指针存储到基类的智能指针。

静态类型和动态类型

当使用存在继承关系的类时，应该区分变量或表达式的静态类型（static type）以及其背后的动态类型（dynamic type）。表达式的静态类型在编译时就是已知的，它是变量声明时的类型或者表达式的结果类型。动态类型是变量或表达式所表示的在内存中的真正对象的类型，这个类型必须到运行时才能知道。如：

```
double ret = item.net_price(n);
```

item 的静态类型是 Quote&，动态类型则依据绑定到 item 的参数类型，这个类型直到运行时才能知道。如果传递 Bulk_quote 那么 item 的静态类型与动态类型将不一样，此时，item 的静态类型是 Quote&，而其动态类型是 Bulk_quote 。

既不是引用也不是指针的表达式的静态类型和动态类型是一样的。理解基类的指针或引用的静态类型和动态类型不一样是至关重要的。

没有从基类到派生类的隐式转换

从派生类到基类的转换是因为每个派生对象都包含基类部分，这个部分可以被基类的指针或引用绑定。基类对象可以独立存在，也可以作为派生对象的一部分。一个非派生类一部分的基类对象只包含基类定义的成员，并不包含派生类定义的成员。所以，并不存在从基类到派生类的自动转换。如：

```
Quote base;
Bulk_quote *bulkP = &base; //错误：不能将基类转为派生类
Bulk_quote &bulkRef = base; //错误：不能将基类转为派生类
```

如果以上赋值是合法的，那么将在 bulkP 或 bulkRef 中使用 base 中不存在的成员。有一点令人惊奇的是，即便基类指针或引用绑定到派生对象，其依然不能转为派生类。如：

```
Bulk_quote bulk;
Quote *itemP = &bulk;
Bulk_quote *bulkP = itemP; //错误：不能将基类转为派生类
```

编译器没有在编译时以任何方式知道从基类转为派生类是否是安全的。编译器只能依据指针或引用的静态类型来判断转换是否是合法的。如果基类有一个或多个虚函数，可以使用 dynamic_cast 来请求带运行时检查的转换。同样，在确实知道从基类到派生类的转换是安全的，可以使用 static_cast 类覆盖掉编译器的规则。

对象间不存在转换

派生类到基类的自动转换只发生于引用或指针类型。从派生对象到基类对象是不存在转换的。然而，从表现上看经常可以将派生对象转为基类，只是这和引用、指针的那种派生类到基类的转换是不一样的。

当初始化一个类对象时，将调用构造函数。当赋值时将调用赋值操作符。这些成员函数通常具有一个该类对象的 const 引用的参数。由于其参数接收引用，派生类到基类的转换允许我们传递一个派生对象给基类的拷贝/移动操作成员函数。这些操作并不是 virtual 的。当传递派生对象给基类构造函数时，基类中定义的构造函数将被执行。那个构造函数只能识别基类自己定义的成员，类似的，如果将一个派生对象赋值给基类对象，基类中定义的赋值操作符将被执行。那个操作符只能识别基类中定义的成员。由于派生类部分被忽略了，所以派生类部分被裁剪（sliced down）掉了。
当用派生对象去初始化或赋值基类对象时，只有基类部分被拷贝、移动或赋值，派生类部分将被忽略。

15.3 虚函数

C++ 的动态绑定发生在虚成员函数通过基类的引用或指针调用时发生。由于直到运行时才知道哪个函数版本被调用，虚函数必须总是被定义。通常，不使用的函数时不需要提供定义的。然而，必须给每个虚函数提供定义而不管它有没有被使用，因为编译器无法知道一个虚函数是否被使用。

关键概念：存在继承关系的类之间的转换

理解存在继承关系的类之间的转换需要理解三点：

525. 派生类到基类之间的转换仅被运用于指针或者引用类型；
526. 没有隐式的从基类到派生类之间的转换；
527. private 或 protected 继承的派生类有时不能执行派生类到基类的转换；

由于自动转换只发生于指针和引用，绝大多数继承层级中的类隐式或显式地定义拷贝控制成员，因而，可以将派生对象拷贝、移动或赋值给基类对象。然而，这种拷贝、移动或赋值仅仅只处理派生对象中的基类部分，称之为裁剪（sliced down）。

调用虚函数将在运行时解析

当通过引用或指针调用虚函数时，编译器将生成代码可以在运行时决定具体调用哪个函数。被调用的函数是与指针或引用绑定的对象的动态类型一致的版本。如：

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(cout, derived, 10);
```

第一个调用中 item 绑定到 Quote 对象上，从而 print_total 中的 **net_price** 调用 Quote 中的版本。第二调用 item 绑定到 Bulk_quote 对象，在这个调用中将调用 Bulk_quote 中定义的 **net_price**。
理解动态绑定只发生在虚函数通过指针或引用被调用的过程中是至关重要的。当虚函数在 plain 对象上（非引用非指针）调用时，调用的解析发生在编译期。如：

```
Quote base = derived;
base.net_price(20);
```

上面的 base 类型就是 Quote，在编译期就可以决定调用 Quote 中定义的 **net_price**。

关键概念：C++ 中的多态

OOP 的关键思想是多态。指针或引用的静态类型和动态类型可以不一样的事实是 C++ 支持多态的基石。当通过基类的引用或指针调用函数时，无法知道到底调用的对象的类型是什么。对象可以是基类对象也可以是派生类对象。如果调用的函数是 virtual 的，那么决定调用哪个函数将推迟到运行时。调用的虚函数版本是指针或引用绑定的对象的类所定义的。
另一方面，调用非 virtual 函数将在编译期进行解析。同样，在 plain 对象上调用任何函数（virtual or not）都是在编译期进行解析的。对象的类型是固定不可变的，不能做任何事从而使得其动态类型和静态类型不一样。因而，在 plain 对象上进行调用时在编译期决定调用对象的类所定义的版本。
虚函数在运行时进行解析只发生于当通过引用或指针进行调用时。只有在这种情况下，对象的动态类型才可能与其静态类型不一样。

派生类中的虚函数

当派生类覆盖一个虚函数时，可以但不是必须提供 virtual 关键字，一旦一个函数被声明为 virtual，它将在所有派生类中保持 virtual。覆盖继承的虚函数的派生类函数必须在派生类中进行参数列表完全一致的声明。派生类中的覆盖的虚函数可以返回一个基类中的返回类型的子类型的指针或引用，如果不是则必须完全匹配。如：D 从 B 派生而来，B 中的虚函数返回 B*，那么 D 中的覆盖的虚函数可以返回 D*，这种返回类型需要可以执行派生类到基类的转型。如果 D 从 B private 派生而来，那么这种转换将不可见。

final 和 override 说明符

在派生类中可以定义与基类中的虚函数同名的函数，但其参数列表不一样。编译器将这个函数认为与基类中的函数是独立的。在这种情况下派生类中的版本并不覆盖基类中的版本。在实践中，这种声明通常是错误的，类作者希望覆盖一个基类中的虚函数，但是写错误了参数列表。

查找这种 bug 是十分困难的，在新标准中可以在派生类的虚函数声明中指定 override 来明确表示是覆盖基类的虚函。这样就要求编译器帮助我们检查是否是真的覆盖了一个基类的虚函数，如果不是，编译器将拒绝编译。如：

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};
struct D1 : B {
    void f1(int) const override; //ok
    void f2(int) override; //错误：B 中没有 f2(int) 函数
    void f3() override; //错误：f3 不是虚函数
    void f4() override; //错误：B 中没有名为 f4 的函数
};
```

D1 中的 f2 是一个新的函数，只是恰巧其名字与 B 中的函数同名。由于显式给出了 override 关键字，但是没有覆盖一个基类虚函数，编译将出错。

由于只有虚函数可以被覆盖，编译器将拒绝 D1 中的 f3，这是由于 B 中的 f3 不是虚函数。

除此之外，可以将函数指定为 final 的，任何尝试覆盖一个被声明为 final 的函数将被认为是编译错误。如：

```
struct D2 : B {
    void f1(int) const override final ;
};
struct D3 : D2 {
    void f2();  //从非直接基类中覆盖 f2 函数
    void f1(int) const; //错误，直接基类将其声明为 final 的
};
```

final 和 override 说明符需要放在参数列表（包括 const 和引用限定符）和后置返回类型之后。

虚函数和默认参数

与常规成员函数一样，虚函数可以有默认参数。当一个调用使用默认实参时，使用的默认值是调用此函数的对象的静态类型中所定义的默认值。也就是说，当通过基类的指针或引用进行调用时，使用基类中定义的默认实参。即使是绑定到派生类对象并且派生类的覆盖虚函数被调用，这个基类默认实参依然会被使用。派生类中的覆盖的虚函数被传递基类中定义的默认参数，如果派生函数依赖于不同参数，那么程序的行为可能会不一致。

如果虚函数使用了默认实参，通常应该总是在派生类和基类中使用相同的实参。

绕过虚函数机制

在一些情况下，我们希望阻止虚函数调用的动态绑定。通过使用作用域操作符可以调用虚函数的特定版本，这个函数调用解析发生在编译期。如：

```
//调用基类的版本，忽略 baseP 的动态类型
double undiscounted = baseP->Quote::net_price(42);
```

通常，只有成员函数或友元函数中的代码应当使用作用域操作符来绕过虚函数机制。

需要这种绕过的最常见的场景是：当一个派生类虚函数调用基类的版本，基类的版本做了继承层级中所有类型都需要做的通用工作。派生类中定义的版本只需要做特定于该派生类的额外工作。如果派生虚函数在调用其基类版本时忽略了作用域操作符，这个调用将被解析为调用其本身，从而将是无限递归。

15.4 抽象基类

参考代码：Disc_Quote.cc 中的 Quote 、Disc_quote 、Bulk_quote 类。

纯虚函数

在上面的代码中 Disc_quote 类是通用的打折书的概念，而不是具体的策略。所以，应当阻止用户实例化此类的对象。通过将 net_price 定义为纯虚函数（pure virtual function）来实现这个设计目标，并且明确告知 net_price 函数没有任何含义。与通常的虚函数不同，纯虚函数是可以不定义的。通过在函数参数列表后，分号前写上 = 0 来表明一个虚函数是纯虚函数。= 0 只能出现在类体内的虚函数声明处。如：

```
double net_price(std::size_t) const = 0;
```

尽管 Disc_quote 不能被直接创建对象，其派生类的构造函数依然需要使用 Disc_quote 的构造函数来构建 Disc_quote 类部分。值得一提的是，可以给纯虚函数提供一个定义，然而，函数体必须在类外进行定义，不能类内给一个纯虚函数提供定义。

有纯虚函数的类是抽象基类

包含或者继承但没有覆盖纯虚函数的类是一个抽象基类（abstract base class）。抽象基类可以定义接口给派生类去覆盖。不能直接创建抽象基类的对象。这里由于 net_price 是纯虚函数，所以，不能直接定义 Disc_quote 的对象。

从抽象基类继承的派生类必须定义所有纯虚函数，否则派生类也是抽象的。

派生类构造函数只能初始化其直接基类

此版本中的 Bulk_quote 有一个直接基类 Disc_quote，和一个非直接基类 Quote，每个 Bulk_quote 对象有三个子对象：Bulk_quote 部分，Disc_quote 部分和 Quote 类的子对象。

由于每个类控制本类对象的初始化过程，因而，即便 Bulk_quote 类没有任何数据成员，它依然需要 4 个构造函数参数。构造函数先调用直接基类 Disc_quote 的构造函数进行初始化，这个构造函数先调用它自己的直接基类 Quote 的构造函数进行初始化，但 Quote 的构造函数执行完返回时，Disc_quote 构造函数将继续执行，并最终返回到 Bulk_quote 构造函数，这个构造函数没有任何额外的事需要做，直接返回。

关键概念：重构

重构（refactoring）需要重新设计类继承体系，并将操作或数据从一个类移动到另一个类。重构在面向对象编程中十分常见。值得一提的是即便改变了继承层次，使用 Bulk_quote 和 Quote 类的代码将保持不变，但需要重新编译整个代码。

15.5 访问控制与继承

正如每个控制其自动成员的初始化。每个类也会控制其成员是否对派生类可见。

protected 成员

类使用 protected 修饰那些对派生类可见，但是对客户代码不可见的成员。protected 可以认为是 private 和 public 的混合：

528. 与 private 一样，protected 成员对类的用户是不可见的；

529. 与 public 一样，protected 成员对派生类的成员和友元是可见的；

还有一条很重要的关于 protected 特性是：派生类的成员和友元只能通过派生对象访问基类的 protected 成员。派生类不能访问独立的基类对象的 protected 成员。如：

```
class Base {
protected:
    int prot_mem;
};
class Sneaky : public Base {
    friend void clobber(Sneaky &); //可以访问 Sneaky::prot_mem
    friend void clobber(Base &); //不能访问 Base::prot_mem
    int j;
};
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
//错误：clobber 不能访问 Base 中的 protected 成员
void clobber(Base &b) { s.prot_mem = 0; }
```

第二个函数不是 Base 的友元，因而，它不能访问 Base 对象的受保护成员。为了防止第二种用法，派生类的成员或友元只能访问嵌套在派生对象中的基类子对象中的受保护成员。对于独立的基类对象并不具有特殊的访问权限。

public, private 和 protected 继承

访问一个类继承来的成员是由基类中的访问说明符和派生列表中的访问说明符共同决定的。参考代码：protected_access.cc。

派生访问说明符并不影响派生类的成员和友元对其直接基类的成员访问权限。访问直接基类的成员是由基类自身的访问说明符决定的。public 继承和 private 继承的派生类都可以访问基类的 protected 成员，而都不能访问基类的 private 成员。

派生访问说明符的作用在于控制派生类的用户对从基类继承来的成员的访问权限，这些用户包括从这个派生继承的其它派生类。如：

```
Pub_Derv d1; //从基类中继承来的成员是 public 的
Priv_Derv d2; //从基类中继承来的成员是 private 的
d1.pub_mem(); //pub_mem 在 d1 中是可见的
d2.pub_mem(); //错误：pub_mem 在 d2 中是不可见的
```

Pub_Derv 和 Priv_Derv 都从基类中继承 pub_mem 函数。当继承是 public 的，成员将保持其可见性，所以 d1 可以调用 pub_mem。而 Priv_Derv 中基类的成员是 private 的，此类的用户不能调用 pub_mem 。

派生类的派生访问说明符还会控制从此派生类进行继承的类对其基类的访问权限。如：

```
struct Derived_from_Public : public Pub_Derv {
    //Base::prot_mem 在 Pub_Derv 中保持 protected
    int use_base() { return prot_mem; }
};
struct Derived_from_Private : public Priv_Derv {
    //错误：Base::prot_mem 在 Priv_Derv 中是 private 的
    int use_base() { return prot_mem; }
};
```

对于 Priv_Derv 的派生类来说，Priv_Derv 从基类中继承来的所有成员都是私有的。

如果使用 protected 继承，Base 中的 public 成员在派生类中将变成 protected 的，其用户将不能访问继承来的成员，但是其子类可以访问这些成员。

派生类到基类的转换的可见性

派生类到基类的转换是否可见取决于哪些代码尝试使用这些转换以及派生列表的访问说明符。假设 D 继承自 B：

530. 仅当 D 是公有继承 B 时，用户代码可以使用派生类到基类的转换，若以 protected 或 private 继承则不可以；
531. D 的成员函数或友元可以使用派生类到基类的转换，而忽略 D 是如何继承 B 的，派生类到直接基类的转换对于派生类的成员和友元来说总是可访问的；
532. D 的派生类的成员或友元，当 D 是 public 或 protected 继承 B 时，可以使用派生类到基类的转换。如果 D 是私有继承 B 则这种转换不可使用；

在给定代码的任何位置，如果基类的 public 成员是可见的，那么派生类到此基类的转换就是可见的。

关键概念：类设计和 protecte 成员

在没有继承时，类只有两种用户：普通用户和实现者。普通用户书写使用类对象的代码；这种代码只能访问类的 public 成员（接口）。实现者书写类的成员和友元中的代码，类的成员和友元可以访问类的 public 和 private 部分（实现）。

在继承的情况下，有第三种用户：派生类。基类使得派生类可访问的实现部分为 protected 。protected 成员对于普通用户来说不可见，private 成员对于派生类和派生类的友元来说不可见。

与别的类一样，基类使其接口成员为 public 的。作为基类的类会将其实现分为可被派生类访问和保留给基类自身和其友元访问的部分。如果提供给派生类用于其实现的操作或数据应该设为 protected 的，否则，实现成员应该设为 private 的。

友元关系和继承

正如友元不具有传递性质（一个类是另一个类的友元并不意味着这个类自己的友元可以访问那个类），友元关系不会被继承。基类的友元对于派生类成员没有特殊的访问权限，派生类的友元对于基类成员没有特殊访问权限。如：

```
class Base {
    friend class Pal; //Pal 对 Base 的派生类没有特殊访问权限
};
class Pal {
public:
    int f(Base b) { return b.prot_mem; }
    //错误：Pal 不是 Sneaky 的友元，不能访问私有成员
    int f2(Sneaky s) { return s.j; }
    //对基类的访问有基类自己控制，即便基类内嵌在派生对象中
    //即便要访问的是 private 成员
    int f3(Sneaky s) { return s.pri_mem; }
};
```

f3 是合法的确有点奇怪，但是这与每个类控制自己的成员的访问权限并不冲突。Pal 是 Base 的友元，所以 Pal 可以访问 Base 对象的私有成员，这种访问权限即便是当基类对象是其派生类对象的一部分时依然成立。

如果一个类是另一个类的友元，那么仅仅是那个类的友元，那个类的基类或派生类都不是这个类的友元。如：

```
//D2 不能访问 Base 类的 protected 和 private 成员
class D2 : public Pal {
public:
    int mem(Base b)
    { return b.prot_mem; }
};
```

友元关系不能继承；每个类控制自己的成员的访问级别。

改变单个成员的访问权限

有时我们需要改变特定的派生类继承来的名字的访问级别。可以通过使用 using 声明来指定。如：

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base {
public:
    using Base::size;
protected:
    using Base::n;
};
```

这里值得注意的是派生类不能改变基类的私有成员的访问级别，原因在于它们对派生类根本不可见。由于 Derived 使用了 private 继承，所以，size 和 n 默认是 Derived 的私有成员。using 声明调整了这些成员的可见性。Derived 的用户代码可以访问其 size 成员，Derived 的派生类可以访问 n 成员。类内的 using 声明可以将其直接或间接基类的可访问名字指定为任何访问级别。如果 using 出现在 private 部分，则只能被类的成员或友元访问。如果出现在 public 部分，则可以被所有用户代码访问。如果出现在 protected 部分，则名字可以被成员、友元和派生类访问。

派生类只能给它可以访问的基类名字使用 using 声明。

默认继承的派生访问说明符

C++ 中可以使用 struct 和 class 关键字定义类，并且具有不同的默认访问说明符。类似的，默认派生说明符依赖于使用哪个关键字定义派生类。用 class 定义的派生类默认是 private 继承；用 sturct 定义的派生类默认是 public 继承。如：

```
class Base {};
struct D1 : Base {}; //默认共有继承
class D2 : Base {}; //默认私有继承
```

任何用关键字 struct 或 class 关键字定义的类之间有什么更深层次的不同是一个普遍的错误。它们之间唯一的区别就是成员的默认访问说明符和默认的派生访问说明符之间的不同。除此之外更无别的区别了。

最佳实践

私有派生的类应该显式给出 private 而不是依赖于默认访问说明符。通过显式指出让其它程序员可以知道私有继承是有意为之，而不是由于错误导致的。

15.6 继承下的类作用域

每个类定义其自己的作用域，在其中它的成员被定义。在继承下，派生类的作用域被嵌套在基类的作用域中。如果一个名字在派生类中无法被解析，那么将继续查找基类作用域。由于继承嵌套了类作用域，所以派生类的成员可以使用基类的成员就像是其自己的成员一样。

名称查找发生在编译期间

对象、引用、指针的静态类型决定了哪些对象的成员是可见的。即便静态类型和动态类型不一样，也是静态类型决定哪些成员是可以使用的。假设 Disc_quote 有一个名为 discount_policy 的成员，如下代码示例：

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk;
Quote *itemP = &bulk;
bulkP->discount_policy();
//错误：itemP 的类型是 Quote* ， 它没有 discount_policy 成员
itemP->discount_policy();
```

名称冲突和继承

与其它内嵌作用域一样，派生类可以复用其直接或间接基类的名字。与往常一样，定义在内部作用域的名字将隐藏定义在外部作用域的名字。如：

```
struct Base {
    Base() : mem(0) {}
protected:
    int mem;
};
struct Derived : Base {
    Derived(int i) : mem(i) {}
    //Base::mem 将被隐藏，返回的是本类自己定义的 mem
    int get_mem() { return mem; }
protectd:
    int mem;
};
```

在派生类中定义与基类同名的成员，将隐藏基类成员的直接使用。

使用作用域操作符来访问被隐藏的成员

通过使用作用域操作符可以访问被隐藏的基类成员，如：

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
};
```

作用域操作符覆盖了常规的名称查找规则，编译器将从 Base 类的作用域开始查找名字 mem 。

最佳实践

除了覆盖继承的虚函数外，派生类通常不应该复用基类中的名字。

关键概念：名称查找和继承

理解函数调用在 C++ 中是如何解析的对于理解 C++ 的继承非常重要。例如：`p->mem()` 的调用过程将发生以下步骤：

533. 首先查看 p 的静态类型，调用成员函数的对象必须是类类型；
534. 在静态类型的类中查找 mem 成员，如果没有找到，继续从其直接基类向上查找，直到找到或者最后一个基类被查找过，如果依然没有找到则产生编译错误；
535. 一旦 mem 被找到了，执行常规的类型检查，查看函数原型与调用是否匹配，以及重载函数解析；
536. 如果 mem 是虚函数，并且调用是通过引用或指针发生的，那么编译器将产生代码，其将执行在运行时根据被绑定的对象的动态类型调用不同的 mem 版本；

537. 否则，如果 mem 是非虚函数，或者调用是通过对象实体（非引用或指针）发生的，编译器产生常规的函数调用代码；

通常，名称查找发生在类型检查前

声明在内部作用域的函数将不会重载定义在外部作用域的函数。因而，定义在派生类中的函数不会重载基类中的成员函数。如果派生类中的成员与基类成员同名，那么派生成员将隐藏基类成员。即便是函数拥有不同的形参列表基类成员也会被隐藏。即便在派生类中同名的成员是数据成员，也会隐藏基类中的同名函数。如：

```
struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int);
};
Derived d;
Base b;
b.memfcn();
d.memfcn(10);
d.memfcn(); //错误：Base::memfcn() 被隐藏
d.Base::memfcn(); //调用 Base::memfcn()
```

Derived 中声明的 memfcn 隐藏了 Base 中声明的 memfcn。第三个调用之所以是无法完成的，原因在于，为了解析这个调用，编译器将在 Derived 中查找名字 memfcn，并且找到了。此时将不再继续往上查找。而 Derived 中的 memfcn 是一个接收一个 int 参数的函数，此调用与之函数不匹配，所以无法调用完成。

虚函数和作用域

如果基类和派生类的同名成员的参数不一样，那么将无法通过基类的指针或引用调用派生类的版本。如：

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    int fcn(int);
    virtual void f2();
};
class D2 : public D1 {
public:
    int fcn(int);
    int fcn();
    void f2();
};
```

以上 D1 中的 fcn 并没有覆盖 Base 中的虚函数 fcn，因为他们具有不同的参数列表。相反却隐藏了基类中的 fcn。D1 有两个名为 fcn 的函数：基类的虚函数 fcn，和自己定义的非虚函数 fcn(int)，但基类的虚函数被隐藏了，因而，无法直接调用。

通过基类调用隐藏的虚函数

以下有几个复杂的调用过程，如果理解了这几个对于理解 C++ 的隐藏特性有帮助：

```
Base bobj;
D1 d1obj;
D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); // virtual call, will call Base::fcn at run time
bp2->fcn(); // virtual call, will call Base::fcn at run time
bp3->fcn(); // virtual call, will call D2::fcn at run time
D1 *d1p = &d1obj; D2 *d2p = &d2obj;
```

```
bp2->f2(); // error: Base has no member named f2
d1p->f2(); // virtual call, will call D1::f2() at run time
d2p->f2(); // virtual call, will call D2::f2() at run time
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42); // error: Base has no version of fcn that takes an int
p2->fcn(42); // statically bound, calls D1::fcn(int)
p3->fcn(42); // statically bound, calls D2::fcn(int)
```

上面 p1, p2, p3 恰巧都指向 D2 类型的对象。然后，由于 fcn(int) 并不是虚函数，所以调用时解析是由指针所指对象的静态类型决定的。

覆盖重载函数

与其它函数一样，不管成员函数是否为虚函数都可以进行重载。派生类可以覆盖其继承的零个或多个重载函数。如果派生类想让所有继承来的重载函数都能够通过派生类访问，就需要覆盖基类中的所有重载函数，或者一个都不覆盖。

有时派生类仅想覆盖一部分但不是所有基类的重载函数。如果为了这个目标而必须覆盖所有的基类函数就不划算了。除了覆盖所有基类的重载函数，派生类可以使用 using 声明来使得所有基类重载函数可以被派生类的用户代码访问。在引入了派生类的所有的重载函数之后，派生类只需要定义特定于本类的行为的函数，并且可以使用基类定义的其它函数。

之前谈过的 using 声明的规则依然适用于现在的重载函数；只有派生类可以访问的基类成员才能被 using 声明导入；派生类的用户代码是否可以访问这些继承过来的名字取决于 using 声明所在的位置。

15.7 构造函数与拷贝控制

与其它类一样，继承层次中的类控制本类对象如何进行创建、拷贝、移动和赋值或析构。与任何别的类一样，如果基类或派生类本身没有定义自己的拷贝控制操作，编译器会合成这些操作。同样，在某些情况下，合成的版本可能是被删除的函数。

15.7.1 虚析构函数

继承对于基类的拷贝控制的最大最直接的影响就是基类必须定义虚析构函数。析构函数是 virtual 的，将允许继承层级中的对象可以被动态析构。由于使用 delete 对动态对象的指针进行删除会调用其析构函数，指针指向对象的静态类型可能与动态类型不一样。如 `Quote*` 指针可能指向 `Bulk_quote` 对象，如果要让编译器成功调用 `Bulk_quote` 的析构函数，就必须让 Quote 的析构函数是虚函数。如：

```
class Quote {
public:
    //如果基类指针指向派生对象，
    //当被删除时，需要析构函数是虚函数
    virtual ~Quote() = default;
};
```

析构函数的 virtual 属性是可以继承的。因而，Quote 的派生类的析构函数也是虚函数，而不管析构函数是合成的还是用户提供的。只要基类的析构函数是虚函数，那么 delete 基类的指针，将会调用正确的析构函数。

如果在基类析构函数不是虚函数的情况下调用实际上指向派生对象的基类指针，行为将是未定义的。

基类的析构函数是第十三章中的规则“如果一个需要析构函数那么它同样需要拷贝和赋值操作”的例外。基类几乎总是需要析构函数，这样才能使得析构函数是 virtual 的。如果基类的空析构函数仅仅是为了使其为虚函数，那么类具有析构函数并不意味着其需要赋值操作或拷贝构造函数。

虚析构函数将关闭合成的移动操作

基类需要虚析构函数对基类和派生类的定义有一个重大的间接影响：如果一个类定义了析构函数，即便使用的是 `= default` 来使用合成版本的，编译器也不会为这个合成任何移动操作。

15.7.2 合成拷贝控制和继承

在基类和派生类中合成的拷贝控制成员与任何别的合成的构造函数、赋值操作符和析构函数是一样的：它们将逐成员初始化、赋值、销毁类的成员。另外，合成的成员将使用基类的对应操作初始化、赋值或销毁其直接基类子对象。如：合成的 Bulk_quote 默认构造函数将调用 Disc_quote 的默认构造函数，而 Disc_quote 则继续调用 Quote 的默认构造函数；

同样的，合成的 Bulk_quote 拷贝构造函数使用合成的 Disc_quote 拷贝函数，而 Disc_quote 拷贝构造函数则继续调用 Quote 的拷贝构造函数。基类的成员是合成的还是用户定义的都是不要紧的，关键在于基类的对应成员是可访问的，并且不是被删除的函数。

析构函数除了析构其自己的成员，在析构阶段还会销毁其直接基类，这个过程是通过调用直接基类自己的析构函数完成的。然后一直向上调用直到继承层次的根。

正如所见，Quote 没有合成的移动操作，这是由于它定义了析构函数。当任何时候需要用到 Quote 的移动操作时，就会用拷贝操作来替换它。 Quote 没有移动操作意味着它的派生类也没有移动操作。

基类和派生类中被删除的拷贝控制成员

除了在第十三章中说明的将导致合成的拷贝控制成员为被删除的函数的原因外，这里将说明额外的原因：基类的定义方式将导致派生类的拷贝控制成员是被删除的函数。

538. 如果基类的默认构造函数、拷贝构造函数或拷贝赋值操作符或析构函数是被删除的或者不可访问的，那么派生类的对应成员也被定义为被删除的函数；
539. 如果基类有一个被删除的或不可访问的析构函数，那么派生类合成的默认和拷贝构造函数将是被删除的函数；
540. 与往常一样，编译器不会合成被删除的移动操作。当使用 `= default` 来请求移动操作时，如果基类的对应操作是被删除的或者不可访问的，或者基类的析构函数是被删除的或不可访问的；

    参考代码：delete_derived_copy_control.cc

在实践中，如果基类没有默认或拷贝或移动构造函数，派生类也不应该有对应的成员。

移动操作和继承

由于基类缺少移动操作将抑制派生类合成移动操作，那么在确实需要移动操作时应该在基类中定义移动操作。即便是使用合成版本，也是需要显式定义的。一旦显式定义了移动操作，也就必须显式定义拷贝操作，因为，当定义了移动构造函数或移动赋值操作符时，其合成拷贝构造函数和拷贝赋值操作符将被定义为被删除的。

15.7.3 子类拷贝控制成员

当派生类定义拷贝、移动操作，这些操作需要拷贝、移动整个对象，包括基类成员。

定义派生拷贝、移动构造函数

当定义派生类的拷贝、移动构造函数，通常需要调用基类对应的构造函数来初始化对象的基类部分。如果不调用基类的构造函数，那么编译器将隐式调用基类的默认构造函数，但这肯定是不正确的。如果想要拷贝、移动基类部分，需要在构造函数初始值列表中显式调用基类对象的拷贝、移动构造函数。

定义派生赋值操作符

派生类的赋值操作符必须显式对基类部分进行赋值。如：

```
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs);
    return *this;
}
```

此操作从显式调用基类的赋值操作符来对派生对象的基类部分进行赋值开始，基类操作符可以正确处理基类对象的赋值，如：自赋值，并在合适的时机释放左操作数中的资源，并在将 rhs 的值赋值给左操作数。一旦基类操作符完成后，将继续执行派生类自己的赋值操作。

派生类的析构函数

派生类的成员和基类部分都会在析构函数后的析构阶段隐式销毁。因而，与构造函数和赋值操作符不一样，派生析构函数只需要销毁派生类自己分配的资源。对象的销毁顺序与构造顺序刚好相反：派生析构函数先执行，然后基类构造函数被调用，沿着继承链一直往上执行析构。

在构造函数和析构函数中调用虚函数

派生对象中的基类部分先构建，当基类构造函数执行时，其对象的派生部分还没有初始化。而，派生对象的析构则是反方向的，所以，当基类的析构函数执行时，其派生部分已经被销毁了。所以，当基类的这两个成员执行时，对象是不完全的。

为了兼容这种不完全，当对象正在构造时，其类型被认为与构造函数所在类是一样的；调用虚函数会被解析为构造函数所在类的那个版本。这对于析构函数来说也是一样的。调用虚函数可以是直接调用，也可以是构造或析构函数调用的函数间接调用了这个虚函数。
如果在基类的构造函数中调用了派生类的虚函数版本，此时，派生部分的成员还没有被初始化，如果允许这样做将导致程序无法正确运行。
当构造函数或析构函数调用一个虚函数，所调用的版本是与构造函数或析构函数在同一个类中的版本。

15.7.4 继承的构造函数

在新标准中，派生类可以复用直接基类的构造函数。尽管，这不是常规意义上的继承。一个只能继承来自直接基类的构造函数，并且派生类不会继承它的默认、拷贝和移动构造函数，原因是编译器会其合成这些构造函数。
通过 using 声明可以让派生类继承基类的构造函数。如：

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote;
    double net_price(std::size_t) const;
};
```

常规的 using 声明只是让名字可见而已。当其运用到构造函数时，using 声明将导致编译器生成代码。编译器将生成与基类一一对应的构造函数，这些编译器生成的构造函数有如下形式：`derived(params) : base(params) {}`，如果派生类有自己的成员，要么执行类内初始化，要么就是默认初始化的。

继承的构造函数的特质

using 声明的构造函数不会随着 using 所在的位置改变继承来的构造函数的访问级别。不管 using 身处何处，基类中的 private 构造函数依然是 private 的；protected 和 public 构造函数也是一样。
另外 using 声明不能指定 explicit 或 constexpr，如果基类构造函数是 explicit 的，派生构造函数具有一样的性质。如果基类构造函数具有默认参数，这些参数不会被继承，相反，派生类将有多个继承而来的构造函数，每一个会将连续省略一个默认实参。
在继承基类的构造函数的同时，派生类可以定义自己的构造函数版本。如果定义的参数列表与基类中的一样，那么基类中的版本将不继承。继承的构造函数不会被认为是用户提供的构造函数，所以，如果一个类只有继承而来的构造函数，那么它还会有合成的默认构造函数。

15.8 容器和继承

当使用容器存储来自继承层次的对象时，通常得使用间接的方式存储对象。原因，不能在容器中持有不同类型的元素。由于对象被赋值给基类对象时是裁剪（sliced down）的，容器与有继承关系的类型不能很好的混合使用。
在容器中放入指针或智指针。当使用容器来存储有继承关系的类型时，通常将容器定义为存储基类的指针或智能指针。而且，存储智能指针是一种更加推崇的方案。

关键术语

541. 抽象基类（abstract base class）：带有一个或多个纯虚函数的类，不能创建一个抽象基类的对象；
542. 可访问的（accessible）：可以通过派生对象访问的基类成员。可见性取决于派生列表中的访问说明符以及基类的成员的访问级别；
543. 类派生列表（class derivation list）：基类的列表，每个可以带一个可选的访问说明符；
544. 动态绑定（dynamic binding）：推迟到运行时决定选择执行哪个函数；在 C++ 中，动态绑定指的是基于引用或指针所绑定的对象的类型，从而，在运行时决定选择运行哪个虚函数；
545. 多态（polymorphism）：运用于面向对象编程，表示根据引用或指针所绑定对象的动态类型来获取类型特定的行为；
546. 动态类型（dynamic type）：运行时中的对象类型。指针或引用的对象的动态类型可能与其静态类型不一样。基类的指针或引用可以绑定到派生类对象上，在这种情况下静态类型是基类，但动态类型是派生类；
547. 静态类型（static type）：定义变量时提供的类型或表达式求值时得到结果类型，静态类型可以在编译期知道；

548. 直接基类（direct base class）：派生类直接继承的基类，直接基类出现在派生类的派生列表中，直接基类自身可以是一个派生类；
549. 间接基类（indirect base class）：不出现在派生类的派生列表中的基类。此类由直接基类（直接或间接）继承，是派生类的间接基类；
550. 虚函数（virtual function）：定义类型特定行为的成员函数。通过引用或指针调用虚函数将在运行时进行解析，这是基于引用或指针实际上所绑定的对象的类型；
551. 裁剪（sliced down）：当派生类对象被用于初始化或赋值给一个基类对象时发生的事。对象的派生部分将被裁剪掉，只留下基类部分给基类对象；
552. 运行时绑定（run-time binding）：与动态绑定一样；
553. 纯虚函数（pure virtual）：虚函数在函数体内使用 `=0` 进行声明。纯虚函数不需要定义（可以进行定义）。有纯虚函数的类时抽象类，如果一个派生类不定义自己的继承来的纯虚函数版本，那么派生类也是抽象的；

面向对象（OOP）和泛型编程（generic programming）都是处理在书写程序时未知的类型，所不同的是 OOP 处理直到运行时才知道的类型，而泛型编程则处理知道编译时才知道的类型。
当书写泛型程序时，写出来的代码与特定类型是独立的。当使用泛型程序时则需要提供类型或者值（作为泛型实参）给泛型代码。
模板（Template）是泛型编程的基石，使用模板并不需要太多关于模板的知识。在本章将学习如何定义自己的模板。
模板是 C++ 中的泛型编程的基石。模板是编译器生成类和函数的蓝本（blueprint）或公式（formula），当使用泛型类型（generic type）或者泛型函数（generic function）时，必须提供必要的信息从而将蓝本转换为特定的类或函数，转换过程发生在编译期间。

16.1 定义模板

为每个类型都定义相同的操作是十分繁琐的，而且需要知道所有操作的类型，更合理的方式是定义函数模板，然后在使用时提供类型即可。参考代码：template.cc。

16.1.1 函数模板

相比于为每个类型定义一个新的函数，可以定义函数模板（function template）。函数模板是合成特定类型版本的蓝本。如：

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (std::less<T>()(v1, v2)) return -1;
    if (std::less<T>()(v2, v1)) return 1;
    return 0;
}
```

模板定义从关键字 template 开始，后面跟着模板参数列表（template parameter list），这是放在尖括号中的一个逗号分隔的一个或多个模板参数（template parameters）。
在模板定义中，模板参数列表不能是空的。
模板参数列表与函数参数列表类似，模板参数表示在类或函数定义中使用到类型或值。当使用模板时，通过隐式或显式的方式将模板实参（template argument）绑定到模板参数。
以上的 compare 函数声明了一个类型参数 T，在 compare 内部，使用 T 来表示一个类型，T 真正表示什么类型将在编译期间由其是如何使用决定的。

实例化函数模板

当调用函数模板时，编译器使用调用实参来推断模板实参。编译器使用实参的类型来决定绑定到模板参数 T 上的类型。通常函数模板是通过推断得到的，不需要显式提供，而且可能推断出来的类型与调用实参的类型并不一样。如：

```
cout << compare(1, 0) << endl;
```

这里实参的类型是 int，编译器将 int 推断为模板实参，并且将此实参绑定到模板参数 T 上。编译器使用推断的模板参数来实例化（instantiate）一个特定版本的函数。当编译器实例化一个模板时，乃是使用模板实参替换对应的模板参数来创建一个模板的新“实例”。如还可以做一下调用：

```
vector<int> vec1{1,2,3}, vec2{4,5,6};
cout << compare(vec1, vec2) << endl;
```

将实例化一个版本，其中 T 被 vector<int> 替换。这些编译器生成的函数被统称为模板的实例（instantiation）。

模板类型参数

compare 函数具有一个模板类型参数（template type parameter），通常，可以将类型参数当作一个类型说明符（type specifier）来使用，这与使用内置类型或类类型是一样的。特别是，类型参数可以被使用于返回类型或函数参数类型，或者变量声明以及强转时的类型。如：

```
template <typename T> T foo(T* p)
{
    T tmp = *p;
    return tmp;
}
```

每个类型参数都可以被放在关键字 class 或 typename 之后。class 与 typename 在这种情况下是完全一样的，且可以互换。而使用 typname 关键字是一个更加直观的方式，毕竟可以使用内置类型作为模板类型实参，而且 typename 更加清晰地告知后面跟随的名字是类型名字。然而，当 typename 被添加到 C++ 语言时，模板已经被广泛运用了；一些程序员依然在继续使用 class 关键字。

非类型模板参数

除了可以定义类型参数，还是可以定义带有非类型参数（nontype parameters）的模板。非类型参数表示一个值而不是类型。非类型参数通过使用特定类型名字而不是 class 或 typename 来指定。当模板被实例化时，非类型参数将被一个用户提供的值或者编译器推断的值替代。这些值必须是常量表达式（constant expressions），只有这样编译器才能在编译期间实例化模板。如：

```
template <unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    cout << N << std::endl;
    cout << M << std::endl;
    return strcmp(p1, p2);
}
compare("hi", "mom");
```

编译器将会使用字面量的长度替换 N 和 M 非类型参数来实例一个模板版本。上面将被实例化为：

```
int compare(const char (&p1)[3], const char (&p2)[4]);
```

一个非类型参数可能是整数类型、函数或对象的指针或左值引用。绑定到整型的实参必须是常量表达式；绑定到指针或引用的实参必须具有静态生命周期（static lifetime），不能使用常规本地对象或动态对象的地址或引用作为实参，指针参数还可以被实例化为 nullptr 或零值常量表达式；

模板的非类型参数在模板定义中是一个常量值。一个非类型参数可以在需要常量表达式的时候使用，如：作为数组的长度。

提供给非类型模板参数的模板实参必须是常量表达式。

内联（inline）和 constexpr 函数模板

函数模板可以与常规函数一样被声明为 inline 或 constexpr 的，inline 或 constexpr 关键字放在模板参数列表后，在返回类型前：

```
template <typename T> inline T min(const T &, const T &);
```

书写类型无关（Type-Independent）的代码

如 compare 函数中使用 const T & 作为参数，那么可以传递任何类型的参数进去，扩大了函数的使用范围。而内部都使用 less<T> 可调用对象，less 对象可以处理好指针不是指向同一个数组的情况，若用 < 得到的结果将是未定义的（undefined）。

模板程序应该尽可能的减少对实参类型的要求。

模板编译

编译器并不是遇到模板定义时生成代码，而是在实例化特定版本的模板时生成代码。当使用模板时才生成代码将影响到如何组织源代码和错误何时被发现。

通常，调用函数时编译器只要看到函数的声明即可。使用类对象时，编译器需要知道类的定义（class definition）和成员函数的声明，而不需要成员函数的定义，因而，将类定义和函数声明放在头文件中，而函数定义或类成员函数定义则放在源文件中。
不过模板却有不同：为了生成实例，编译器需要知道函数模板的定义以及类模板成员函数（class template member function）的定义。因而，与非模板代码不同，模板代码的头文件中通常包含声明和定义。
函数模板和类模板的成员函数定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种类型的名字：与模板参数无关的、与模板参数有关的。
模板提供者需要保证所有与模板参数无关的名字在模板被使用时是可见的。另外，模板提供者必须保证模板的定义（包括类模板成员定义）在模板被实例化时是可见的；
保证声明所有与实例化模板的类型参数相关的函数、类型和操作符是可见的，如：模板类型参数的成员函数、操作符重载以及内部定义的类型。
模板提供者应该将所有模板定义和定义模板时用到的名字声明都放在头文件中。模板的用户使用时必须包含模板头文件以及用于实例化模板的类型的头文件。

编译错误大部分在实例化期间发现

代码直到模板被实例化时才生成影响了何时才能知道模板代码中的编译错误。通常，有三个阶段编译器可能会发现错误。
第一，当编译器模板本身时，可以发现一些语法错误；第二，当编译器发现模板被使用时，会检查是否参数数目与模板定义一致；第三，与第二个阶段几乎是在一起发生的，在实例化期间，可以发现类型相关的错误（不存在的操作），取决于编译器如何管理实例，这些错误可能在链接期间被发现。
当书写模板时，不应该让类型过度具体（限制过多），但模板通常都会对其使用的类型做出一些假设。
将由调用者保证传给模板的实参符合模板对类型的要求，即所有的操作都是存在的，并且表现的是符合要求的。

16.1.2 类模板

类模板（class template）是合成类的蓝本。与函数模板不同的是编译器不能推断出类模板的模板参数的类型。相反，使用类模板时必须提供额外的信息，这些信息将放在类模板名后的尖括号中。这些额外信息是模板实参列表（template arguments list），用于替换模板参数列表（template parameters list）。

定义类模板

参考代码：template_Blob.cc
与函数模板一样，类模板以关键字 template 跟随模板参数列表。在类模板的定义中（包括成员定义），可以如使用常规类型或值一样使用模板参数，这些模板参数将在使用时被提供实参。
如在 Blob 类模板中，返回值类型是 T&&，当实例化 Blob 时，T 将被替换为指定的模板实参类型。

实例化类模板

当使用类模板时，必须提供额外的信息，这个额外信息就是显式模板实参（explicit template arguments），它们将被绑定到模板的参数上。编译器使用这些模板实参来实例化从模板中实例化一个特定的类。如：

```
Blob<int> ia;
Blob<int> ia2 = {0,1,2,3,4};
```

编译器为每一个不同的元素类型生成不同的类，如：

```
Blob<string> names;
Blob<double> prices;
```

以上将实例化两个完全不同的类：以 string 为元素的 Blob 和以 double 为元素的 Blob。类模板的每个实例都构成完全独立的类。类型 `Blob<string>` 和其它的 Blob 类型之间没有任何关系，亦没有任何特殊的访问权限。

在模板作用域中实例化一个模板类型

为了便于阅读模板的类代码，谨记类模板名字不是类型的名字。一个类模板被用于实例化类型，而实例化的类型总是包含模板的实参。可能会令人疑惑的是，在类模板中通常不使用真实存在的类型或值作为模板的实参，而是使用模板自己的参数作为模板实参。比如：Blob 中的 data 被定义为 `std::shared_ptr<std::vector<T>> data;` 便是这个道理。当实例化 `Blob<int>` 时，data 将被实例化为 `shared_ptr<vector<int>>`。

类模板的成员函数

与常规函数一样，可以给类模板在类体内或类体外定义成员函数。同样，定义在类体内的成员函数是隐式内联的。类模板的成员函数本身是常规函数。然而，类模板的每个实例都有自己的成员版本。因而，每个类模板的成员函数具有与类模板一样的模板参数。因而，在类模板外定义的成员函数以关键字 template 跟随类模板参数列表开始。并且，在类外定义成员函数必须说明其属于哪个类。而且这个类名由于是从模板实例化而来的，需要包含模板实参。当定义成员函数时，模板的实参与模板的参数一样。因而整个形式如下：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

具体例子：

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

与其它定义在类模板外的成员函数一样，构造函数也是先声明类模板的模板参数。如：

```
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>()) { }
```

此处亦是使用类模板自己的类型参数作为 vecotr 的模板实参。

实例化类模板成员函数

默认情况下，类模板的成员函数只有在程序使用模板函数时才会实例化。如果一个成员函数没有用到，那么就不会被实例化。成员只有在使用到时才会实例化的事实，使得我们可以实例化一个类，其使用到的类型实参只符合模板操作的部分要求。通常，一个类模板实例化类的成员只有在使用时才会被实例化。

简化在类代码中使用模板类名（Template Class Name）

在类模板自身的作用域中，可以使用模板名而不需要实参。当在类模板的作用域内，编译器认为使用模板自己的地方就像是指定了模板实参为模板自己的参数。如：

```
template <typename T> class BlobPtr {
public:
    BlobPtr& operator++();
};
```

与以下代码是一样的：`BlobPtr<T>& operator++();`

在类模板体外使用类模板名字

当在类模板体外定义成员时，必须记住直到看到类名时才处于类地作用域中。如：

```
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    BlobPtr ret = *this;
    ++*this;
    return ret;
}
```

由于返回类型出现在类作用域的外面，所有必须告知返回类型是 BlobPtr 以其类型参数为实参的实例。在函数体内部，我们处于类的作用域中，所以在定义 ret 时不再需要重复模板实参，当不再提供模板实参，编译器认为我们使用与成员实例一样的类型实参。

类型模板和友元

当类定义中包含友元声明时，类和友元可以相互不影响的时模板或者不是模板。类模板可以有非模板的友元，授权友元访问其所有的模板实例。如果友元自身是模板，授权友元的类控制访问权限是授给模板的所有实例还是给特定的实例。

一对一友元

最常见的友元形式就是一个类模板与另一个模板（类或函数）的对应实例之间建立友元关系。如：Blob 类模板和 BlobPtr 类模板之间的友元关系，以及相等性判断（==）操作符之间的关系。代码如下：

```
template <typename> class BlobPtr;
template <typename> class Blob;
template <typename T>
bool operator==(const Blob<T>&, const Blob<T>&);
template <typename T> class Blob {
friend class BlobPtr<T>;
friend bool operator==<T>(const Blob<T>&, const Blob<T>&);
};
```

为了指定模板（类或函数）的特定实例，我们必须首先声明模板本身。模板的声明包括模板的模板参数列表。

友元声明使用 Blob 的模板参数作为它们的模板实参。因而，这种友元被严格限定在具有相同类型的模板实参的 BlobPtr 和相等操作符的实例之间。如：

```
Blob<char> ca; //BlobPtr<char> and operator==<char> are friends
Blob<int> ia; //BlobPtr<int> and operator==<int> are friends
```

BlobPtr<char> 的成员可以访问 ca 的非共有部分，但 ca 与 ia 之间没有任何特殊的访问权限。

通用和特例（Specific）的模板友元

一个类可以让另一个模板的所有实例都是其友元，或者将友元限定在某一个特定的实例。如：

```
template <typename T> class Pal;
class C {
friend class Pal<C>; //Pal instantiated with class C is a friend to C
//all instances of Pal2 are friends to C;
//no forward declaration required when we befriend all instantiations
template <typename T> friend class Pal2;
};
template <typename T> class C2 {
//each instantiation of C2 has the same instance of Pal as a friend
friend class Pal<T>; //a template declaration for Pal must be in scope

//all instances of Pal2 are friends of each instance of C2,
//prior declaration is not needed
template <typename X> friend class Pal2;

//Pal3 is a nontemplate class that is a friend of every instance of C2
//prior declaration for Pal3 is not needed
friend class Pal3;
};
```

为了让所有的实例都是友元，友元的声明中所使用的模板参数必须与类模板所使用的不一样，就像这里的 C2 中的 T 和 Pal2 中的 X 一样。

与模板本身的类型参数成为友元

在新标准中，可以使得模板的类型参数成为友元：

```
template <typename Type> class Bar {
friend Type;
};
```

这里指明所有用于实例化 Bar 的任何类型都是 Bar 的友元。这里需要指出的是尽管一个友元通常是一个类或函数，但是 Bar 也可以用内置类型进行实例化。这种友元关系是允许的，这样才可以将 Bar 用内置类型进行实例化。

模板的类型别名

类模板的一个实例就是一个类类型，与任何别的类类型一样，可以定义一个 typedef 来作为实例化类的别名。如：

```
typedef Blob<string> StrBlob;
```

由于模板不是类型，所以不能定义 typedef 作为模板的别名。也就是说不能定义 typedef 来指向 Blob<T>。

然而，在新标准下可以用 using 声明来指定类模板的别名。如：

```
template <typename T> using twin = pair<T, T>;
twin<string> authors; //authors is a pair<string, string>
```

模板类型别名是一族类的别名。当定义模板类型别名时，可以固定一个或多个模板参数，如：

```
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> books;
partNo<Vehicle> cars;
partNo<Student> kids;
```

这里将 partNo 定义为一族 pair 类，其中第二个成员是 unsigned，partNo 的使用者只能指定 pair 的第一个成员，却不能对第二个成员做出选择。

类模板的静态成员

与其它类一样，类模板可以声明静态成员（static members）。如：

```
template <typename T> class Foo {
public:
    static std::size_t count() { return ctr; }
private:
    static std::size_t ctr;
};
```

Foo 的每个类实例都有自己的静态成员实例。也就是说对于每个给定类型 X，都有一个 Foo<X>::ctr 和一个 Foo<X>::count 成员，而 Foo<X> 的所有对象都共享相同的 ctr 对象和 count 函数。

与任何别的 static 数据成员一样，每个类实例必须只有一个 static 数据成员的定义。然而，每个类模板的实例有一个完全不一样的对象，因而，在定义静态数据成员时与在类外定义成员函数类似。如：

```
template <typename T>
size_t Foo<T>::ctr = 0;
```

与类模板的任何成员类似，定义的开始部分是模板参数列表，后跟随成员的的类型，然后是成员名字。成员名字中包含成员的类名，而此时的类名是从一个模板中生成而来的，所以包含模板的实参。因而，当 Foo 为每个特定义的模板实参进行实例化时，一个独立的 ctr 将为此类类型进行实例化并初始化为 0。

与非模板类的静态成员类似，可以通过类的对象或者使用作用域操作符对静态成员进行直接访问。当然，为了通过类使用一个静态成员，必须是一个特定的类实例才行。如：

```
Foo<int> fi;  //实例化类 Foo<int> 和静态数据成员 ctr
auto ct = Foo<int>::count(); //实例化 Foo<int>::count 函数
ct = fi.count(); //使用 Foo<int>::count
ct = Foo::count(); //error: which template instantiation?
```

与任何别的成员函数类似，静态成员函数仅在被使用时实例化。

16.1.3 模板参数

与函数参数名字类似，模板参数名字没有本质的含义。通常将类型参数记作 T，但可以使用任何名字：

```
template <typename Foo>
Foo calc(const Foo &a, const Foo &b)
{
    Foo temp = a;
    return temp;
}
```

模板参数和作用域

模板参数遵循常规的作用域规则。模板参数的名字可以在其声明之后使用，直到模板的声明或定义的尾部。与任何别的名字类似，模板参数隐藏任何外部作用域的相同名字的声明。与绝大多数上下文不一致的是，作为模板参数的名字不能在模板中复用。如：

```
typedef double A;
template <typename A, typename B>
void f(A a, B b)
{
    A tmp = a; //tmp has same type as the template parameter A, not double
    double B; //error: redeclares template parameter B
}
```

常规的名称隐藏规则导致 A 的类型别名被类型参数 A 所隐藏。由于不能复用模板参数的名字，将 B 声明为变量名是一个错误。

由于模板参数名字不能被复用，模板参数名字只能在给定模板参数列表中出现一次。如：

```
//error: illegal reuse of template parameter name V
template <typename V, typename V> //...
```

模板声明

模板声明必须包含模板的参数列表，如：

```
//declares but does not define compare and Blob
template <typename T> int compare(const T &, const T &);
template <typename T> class Blob;
```

与函数参数一样，模板参数的名字不需要在声明和定义之间完全一样，如：

```
//all three uses of calc refer to the same function template
template <typename T> T calc(const T &, const T &);
template <typename U> U calc(const U &, const U &);
template <typename Type>
Type calc(const Type &a, const Type &b) { /* ... */ }
```

以上三个用法都是表示同一个函数模板。当然，给定模板的所有声明和定义都必须具有相同数目和种类（类型或非类型）的参数。

最佳实践：被某个文件所需要的所有模板声明都应该放在文件的头部，它们最好放在一起，并且是在所有使用这些名字之前就声明。

使用类的类型成员

前面的章节描述过使用作用域操作符（::）来访问静态成员和类型成员（type member），在常规代码中，编译器是知道一个名字是类型成员或者是静态成员。然而，当遇到模板时，编译器很可能就无法知道这个信息了，如，给定 T 模板类型参数，当编译器看到 T::mem 时，它将直到实例化时才能直到 mem 是类型成员还是静态成员。然而，有时必须得让编译器知道一个名字表示类型才能正确编译。例如以下表达式：

```
T::size_type * p;
```

编译器必须知道 size_type 是类型，这是在定义一个名字 p 的变量，不然，就不会被处理为静态数据成员 size_type 与变量 p 相乘。

默认情形下，语言认为通过作用域操作符访问的名字不是类型。如果要使用一个模板类型参数的类型成员，必须显式告知编译器这个名字是类型。那就得用 typename 这个关键字了。如：

```
template <typename T>
typename T::value_type top(const T &c)
{
    if (!c.empty())
        return c.back();
    else
        return typename T::value_type();
}
```

以上函数期待一个容器作为其实参，使用 typename 类指定其返回类型，并且在没有元素的情况下生成一个值初始化的元素用于返回。

当想要告知编译器一个名字表示类型时，必须使用关键字 typename 而不是 class。

默认模板实参

与可以给函数参数提供默认实参一样，可以提供默认模板实参（default template arguments），在新标准下可以给函数和类模板提供默认实参。早期的语言版本只允许给类模板提供默认实参。如：

```
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

这里同时提供了模板模板实参和模板函数实参。默认模板实参使用 less 函数对象的 T 类型实例，而默认函数参数告知 f 是 F 类型的默认初始化对象。使用 compare 时可以提供自己的比较操作，但不是必须这么做。如：

```
bool i = compare(0, 42);
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

当 compare 以三个实参进行调用时，第三个参数的必须是可调用对象，并且返回值可以转为 bool，其参数类型必须与前两个参数的类型可以转换。与函数默认参数一样，模板参数的默认实参只有在其右侧的所有参数都具有默认实参时才是合法的。

模板默认实参和类模板

无论何时使用类模板，都必须在模板名之后跟随尖括号。尖括号表示类是从一个模板实例化而来的。特别是，如果一个类模板给所有模板参数都提供了默认实参，并且我们也希望使用这些默认值，还是必须得在模板名字后提供一个空的尖括号对。如：

```
template <class T = int>
class Numbers {
public:
    Numbers(T v = 0):val(v) { }
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; //empty <> says we want the default type
```

16.1.4 成员模板

无论是常规的类还是类模板都可以有一个本身就是模板的成员函数，这种成员被称为成员模板（member templates），成员模板一定不能是虚函数。

常规类的成员模板

常规类中的成员模板与模板函数的写法完全一样。如：

```
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr):os(s) { }
    template <typename T>
    void operator()(T *p) const
    {
        os << "delete unique_ptr" << std::endl;
        delete p;
    }
private:
    std::ostream &os;
};
```

成员模板与别的模板一样，都是从自己的模板参数列表开始的。每个 DebugDelete 对象有自己的 ostream 成员，并且有一个成员函数本身是模板。用法如下：

```
double *p = new double;
DebugDelete d;
d(p);
int *ip = new int;
DebugDelete()(ip);
```

也可以被用于构建 unique_ptr 对象。如：

```
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
unique_ptr<string, DebugDelete> sp(new string, DebugDelete());
```

以上当 unique_ptr 的析构函数被调用时，DebugDelete 的调用操作符将会被调用。因而，无论何时 unique_ptr 的析构函数被实例化，DebugDelete 的调用操作符将被实例化。

类模板的成员模板

可以给类模板定义成员模板，在这种情况下，类和成员的模板参数是各自独立的。如：

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
};
```

此构造函数有其自己的模板类型参数 It 。不同于类模板的常规成员函数，成员模板是函数模板。当在类模板外部定义成员模板时，必须提供同时为类模板和函数模板提供模板参数列表。先提供类模板的参数列表，紧跟着成员模板自己的模板参数列表。如：

```
template <typename T>
template <typename It>
Blob<T>::Blob(It b, It e):data(std::make_shared<std::vector<T>>(b, e)) { }
```

初始化和成员模板

为了实例化类模板的成员模板，必须同时给类和函数模板同时提供模板参数。与往常一样，类模板的实参必须显式提供，而成员模板的实参则从函数调用中推断出来。如：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
Blob<int> a1(begin(ia), end(ia));
Blob<int> a2(vi.begin(), vi,end());
Blob<string> a3(w.begin(), w.end());
```

当定义 a1 时，显式告知编译器去实例化模板参数绑定到 int 的 Blob 版本。而其构造函数自己的类型参数则从 begin(ia) 中推断出来，在此例是 int*。因此，a1 的实例定义是：

```
Blob<int>::Blob(int*, int*);
```

16.1.5 控制实例化

模板只有在使用时才会生成实例意味着同一个实例可能会出现在多个 obj 文件中。当两个或多个分离编译的源文件使用同一个模板且模板实参是一样的，那么在每个文件中都由一个此模板的相同实例。

在大的系统中，在多个文件中过度实例化同一个模板造成的影响将是很大的。在新标准中，可以通过显式实例化（explicit instantiation）来避免这个消耗。显式实例化的形式如下：

```
extern template declaration; //instantiation declaration
template declaration; //instantiation definition
```

显式实例化的声明或定义之前，必须要能够看到模板体的代码。其中 declaration 是将所有的模板参数替换为模板实参，如：

```
extern template class Blob<string>; //declaration
template int compare(const int &, const int &); //definition
```

当编译器看到 extern 模板声明时，它将不会生成在当前文件中生成实例代码。extern 声明意味着在程序的某个地方存在着一个非 extern 的实例，程序中可以有多个 extern 声明，但是只能有一个定义。

由于当使用模板时会自动实例化，所以 extern 声明必须出现在所有使用此实例的代码之前。

函数模板必须被显式实例化，而类模板不一定需要，编译器会隐式实例化类模板。这个特性可能是编译器自己的特性，所以不应依赖于此。最好是对于每个实例声明都对应一个显式地实例定义。

实例定义实例化所有成员

类模板的实例定义实例化模板的所有成员，包括内联成员函数。当编译器看到一个实例定义时，它无法直到到底哪个成员函数将会程序使用，因此，于常规的类模板实例化不同的是，编译器将实例化类的所有成员。即便不使用某个成员，其也必须实例化。结果就是，我们只能显式实例化所有成员都可以使用的模板实例。

16.1.6 效率和灵活性

智能指针类型提供了描述模板设计者的设计选择很好的材料。shared_ptr 可以在创建或 reset 指针时传递一个删除器（deleter）来轻松覆盖之前的。而 unique_ptr 的删除器却是类型的一部分，我们必须在定义 unique_ptr 就显式提供一个类型作为模板实参，因而，给 unique_ptr 定制删除器会更加复杂。删除器是如何被处理的与类的功能似乎并无本质上的关系，但这种实现策略却对性能有重大的影响。

在运行时绑定删除器

shared_ptr 的删除器是间接存储的，意味着可能作为指针或者一个包含指针的类，这是由于其删除器直到运行时才能被知道是何种类型，而且在其生命周期中还可以不断改变。一般来说，一个类的成员类型不会在运行时改变，所以此删除器必须是间接存储的。调用方式如：

```
//value of del known only at runtime; call through a pointer
//del(p) requires runtime jump to del's location
del ? del(p) : delete p;
```

在编译期绑定删除器

由于删除器的类型是作为 unique_ptr 的类型参数指定的，意味着删除器的类型可以在编译期就知道，因而，此删除器可以被直接存储。调用方式如：

```
//del bound at compile time;
//direct call to the deleter is instantiated
del(p); // no runtime overhead
```

这个方式的好处在于不论代码执行哪个类型的删除器，其都是就地执行，意味着不需要做任何跳转，甚至对于简单的函数可以内联到调用处。这是编译期绑定的功劳。

通过在编译期绑定删除器，unique_ptr 避免了调用删除器的运行时消耗；通过在运行时绑定删除器，shared_ptr 带来了灵活性，使其更容易定制新的删除器；

16.2 模板实参推断

默认情况下编译器使用调用实参来决定函数模板的模板参数。这个过程称为模板实参推断（template argument deduction），在推断过程中编译器使用调用的实参类型来选择哪个生成的函数版本是最合适的。

16.2.1 转换和模板类型参数

与常规函数一样，传递给函数的模板的实参被用于初始化函数的参数。类型是模板类型参数的函数参数有特殊的初始化规则。只有非常有限的几个转换是自动运用于这种实参的。相比于转换实参，编译器会生成一个新的实例。

与之前的描述一样，不论是参数还是实参中的顶层 const 都会被忽略。在函数模板中会执行的有限转换分别是：

554. const 转换：如果一个函数参数是 const 的引用或指针，可以传递一个非 const 对象的引用或指针；
555. 数组或函数至指针的转换：如果函数参数不是引用类型，那么指针转换将被运用于数组或函数类型。数组实参将被转为指向其首元素的指针，同样，函数实参将被自动转为指向函数类型的指针；

其它任何类型的转换（算术转换、子类到基类的转换、用户定义转换）都不会执行。如：

```
template <typename T> T fobj(T, T);
template <typename T> T fref(const T &, const T &);
int a[10], b[42];
fobj(a, b);
fref(a, b);
```

在 fobj 调用中，数组的类型不一样是无所谓的。两个数组都转为了指针，fobj 中的模板参数类型是 int*，而调用 fref 却是非法的，当参数是引用时，数组不能自动转为指针，此时 a 和 b 的类型是不匹配的，因而调用是错误。

const 转换和数组或函数至指针的转换是模板类型中的实参到形参的唯一自动转换。

使用相同模板参数类型的函数参数

一个模板类型参数可以被用于多于一个的函数参数类型。由于只存在十分有限的转换，传递给这种参数的实参必须具有完全一样的类型，如果推断出来的类型不匹配，那么调用将会出错。如 :compare(lng, 1024); 的调用将会出错，原因在于第一个参数被推断出来是 long，而第二个是 int，这并不匹配，所以模板实参推断就失败了。为了允许这种实参的常规转换，必须定义两个不同类型的模板参数。如：

```
template <typename A, typename B>
int flexibleCompare(const A &v1, const B &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

此时用户可以提供两个不同类型的实参了。

Normal Conversion Apply for Ordinary Arguments

如果函数模板的参数的类型不是模板类型参数，即其是具体的类型，那么实参可以执行之前描述过的各种转换。如：

```
template <typename T> ostream &print(ostream &os, const T &obj)
{
    return os << obj;
}
```

其中第一个参数是具体的类型 ostream&，所以，可以执行正常的类型转换。

16.2.2 函数模板显式实参

在一些情形下根本不可能让编译器推断出模板的实参。在另外一些情形下，则是我们自己想控制模板的实例化。两者绝大多数时候发生在函数的返回类型与任何参数列表中的类型都不一样时。

指定显示模板实参

通过定义额外的模板参数来表示返回值的类型，如：

```
template <typename T1, typename T2, typename T3>
T1 sum(T2, T3);
```

在这种情况下编译器没有任何办法来推断 T1 的类型，调用必须在每次调用 sum 时为此模板参数提供显示模板实参（explicit template argument）。给函数模板提供显式模板实参与定义类模板实例是一样的，显式模板实参将被放在函数名之后的尖括号中，并且在实参列表之前。如：

```
// T1 is explicitly specified; T2 and T3 are inferred from the argument types
auto val3 = sum<long, long>(i, lng);
```

此调用中显式指定了 T1 的类型，编译器将从 i 和 lng 的类型中推断出 T2 和 T3 的类型。显式模板实参是从模板参数列表中的左边向右边依次匹配的。显式模板实参只能省略尾部的参数，并且是在可以从函数实参中推断出来的那些参数。意味着如下函数必须每次都提供所有的模板实参：

```
template <typename T1, typename T2, typename T3>
T3 alternative_sum(T2, T1);
```

正常的转换可以运用于显式模板实参上

与正常的转换可以运用于具体类型函数参数一样，正常的转换也可以运用于被显式指定模板实参的函数参数上：

```
long lng;
compare(lng, 1024); // error: template parameters don't match
compare<long>(lng, 1024); // ok: instantiates compare(long, long)
compare<int>(lng, 1024); // ok: instantiates compare<int, int>
```

16.2.3 Trailing Return Types and Type Transformation

使用显式模板参数来表示模板函数的返回值，当确实是想自己指定返回类型时是很好的。但有时强制显式模板实参却给自己增加了不必要的负担。比如想返回一个迭代器的解引用得到的元素的引用，在新的标准中可以使用尾后返回类型从而让编译器推断出来，而不需要显式指定模板类型参数。如：

```
// a trailing return lets us declare the return type after the parameter list is seen
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    return *beg;
}
```

这里告知编译器 fcn 的返回类型与解引用 beg 参数是一样的，解引用操作符返回左值，所以 decltype 推断出来的类型是一个元素的引用。如此，如果 fcn 在 string 序列上调用返回类型将是 **string&**，如果序列是 int 的，则返回值是 **int&** 的。

The Type Transformation Library Template Classes

有时我们不能直接返回我们需要的类型，如：想让 fcn 返回元素的值类型而不是引用。从迭代器中根本不可能得到一个值类型。为了获取元素的类型，我们可以使用类型转移模板（type gransformation template）。这些模板被定义在 **type_traits** 头文件中，通常定义在其中的类型被称为模板元编程（template metaprogramming）。

For Mod<T>, where Mod is	If T is	Then Mod<T>::type is
remove_reference	X& or X&&otherwise	XT

add_const	X&, const X, or functionotherwise	Tconst T
add_lvalue_reference	X&X&&otherwise	TX&T&
add_rvalue_reference	X& or X&&otherwise	TT&&
remove_pointer	X*otherwise	XT
add_pointer	X& or X&&otherwise	X*T*
make_signed	unsigned Xotherwise	XT
make_unsigned	signed typeotherwise	unsigned TT
remove_extent	X[n]otherwise	XT
remove_all_extents	X[n1][n2]…otherwise	XT

以上表格中的 remove_reference 用于获取元素类型，如：remove_reference<int&>::type 的结果是 int 类型。上面的难题的解决方案就是用 remove_reference<decltype(*beg)>::type 表示 beg 所指向的元素的值类型。如：

```
template <typename It>
auto fcn2(It beg, It end) -> typename remove_reference<decltype(*beg)>::type
{
    return *beg;
}
```

type 是类模板的一个类型成员，这个类型由模板参数决定，因而，必须将 typename 关键字放在尾后返回类型的前面，用于告知编译器 type 表示一个类型。

上面表格中所有的类型转义模板都以相同的方式进行工作，每个模板都有一个公共成员 type 来表示类型。这个类型将是与模板参数相关的，且这种关系由模板的名字表达。如果不能进行对应的转换，type 成员将返回模板参数类型本身。如 T 本身就是不是指针，那么 remove_pointer<T>::type 就是 T 本身。

16.2.4 函数指针和实参推断

当使用函数模板来初始化或赋值函数指针时，编译器使用指针的类型来推断模板的实参。如：

```
template <typename T>
int compare(const T &, const T &);
int (*pf1)(const int &, const int &) = compare;
```

pf1 的参数类型决定了 T 的模板实参类型，此处模板实参类型是 int，此时 pf1 指向 compare 的以 int 实例化的 compare 函数。如果模板实参不能从函数指针中推断出来，如：

```
void func(int(*)(const string &, const string &));
void func(int(*)(const int &, const int &));
func(compare); // error: which instantiation of compare?
```

问题在于从 func 无法推断出 compare 的模板参数类型，可以通过显示指定模板实参，如：

```
func(compare<int>); // passing compare(const int &, const int &)
```

当取函数模板实例的地址，上下文必须要让其能够让所有的模板参数（类型的或值的）可以被唯一推断。

16.2.5 模板实参推断和引用

如果函数的参数是模板类型的引用，需要记住的是：常见的引用绑定规则依然有效（左值只能绑定到左值，右值只能绑定到右值）；并且此时 const 是底层 const 而不是顶层 const 。

左值引用函数参数的类型推断

当一个函数参数是模板类型参数的左值引用如 T&，绑定规则告诉我们只能传递左值过去，实参可以有 const 修饰，如果实参是 const 的，那么 T 将被推断为 const 类型。如：

```
template <typename T> void f1(T&);
f1(i); // i is an int; T is int
f1(ci); // ci is a const int; T is const int
f1(5); // error: argument to a & parameter must be an lvalue
```

如果一个函数参数是 const T& 的，那么绑定规则告诉我们可以传递任何类型（const 或非 const 对象、临时量或字面量）的实参过去。由于函数参数本身是 const 的，T 推断出来的类型将不在是 const 的了，因为，const 已经是函数参数类型的一部分；因而，它将不必在是模板参数的一部分。如：

```
template <typename T> void f2(const T&); // can take an rvalue
// parameter in f2 is const &; const in the argument is irrelevant
```

```
// in each of these three calls, f2's function parameter is inferred as const int&
// and T is int
f2(i);
f2(ci);
f2(5);
```

右值引用函数参数的类型推断

当函数参数是一个右值引用时，形如：T&&，绑定规则告诉我们可以传递右值给这个参数。当这样做时，类型推断表现的类似于给左值引用函数参数一样，给类型参数 T 推断出来的类型就是这个右值类型。如：

```
template <typename T> void f3(T&&);
f3(42); // argument is an rvalue of type int; template parameter T is int;
```

Reference Collapsing（引用折叠） and Rvalue Reference Parameters

虽然语言不允许将右值绑定到左值引用上，但是语言允许将左值绑定到右值引用上。语言通过两点实现了这个特性，其一将影响如何从右值引用参数中推断出 T 的类型，当传递左值（i）给函数参数是模板类型参数的右值引用（T&&）时，编译器会将 T 推断为左值引用，所以 f3(i) 中，编译器将 T 推断为 int& 而不是 int 。将 T 推断为 int& 似乎意味着 f3 的函数参数是对类型 int& 的右值引用。然而，语言不允许直接定义引用的引用，却允许间接通过类型别名或模板类型参数来达到此目标。

其二就是：如果间接创建了引用的引用，那么这些引用将被折叠（collapse），除了一个之外所有的折叠结果都是左值引用类型，新标准中将折叠规则扩展到了右值引用。只有右值引用的右值引用才会被折叠为右值引用。规则如下：

556. X& &, X& && 和 X&& & 被折叠为类型 X&；
557. X&& && 被折叠为 X&&；

引用折叠只会发生于间接创建的引用的引用，比如在类型别名或模板参数时。

结合引用折叠规则和右值引用参数的特殊类型推断规则，将使得 f3 可以对左值进行调用。如：

```
f3(i); // template parameter T is int&, result in f3<int&>(int&)
f3(ci); // template parameter T is const int&, result in f3<const int&>(const int&)
```

这导致了两个重要的结果：

558. 函数参数是模板类型参数的右值引用（T&&）可以被绑定到左值；
559. 如果其实参是左值，那么推断的模板实参类型将是左值引用类型，且函数参数将被实例化为一个左值引用参数；

可以传递任何类型的实参给类型为 T&& 的函数参数，当传递左值时，函数参数将被实例化为左值引用 T& 。

书写函数参数是右值引用的模板函数

模板参数可以被推断为引用类型对模板内的代码具有惊人的影响。如：

```
template <typename T> void f3(T&& val)
{
    T t = val;  // copy or binding a reference?
    t = fcn(t); // does the assignment change only t or val and t?
    if (val == t) { /* ... */ }  // always true if T is a reference type
}
```

根据传递个实参是左值还是右值，会产生行为非常不同的代码。要保证这种代码正确工作是非常困难的。在现实中，右值引用参数只会被运用于以下两种情况：函数模板将其参数进行转发（forwarding）或者函数模板是重载的，这个将在后面介绍。这里可以先提前声明的是这种函数模板与参数是右值引用的常规函数一样可以进行重载。如：

```
template <typename T> void f(T&&); // binds to nonconst rvalues (1)
template <typename T> void f(const T&); // lvalues and const rvalues (2)
```

上面遵循十三章介绍过的右值引用函数参数的重载规则，所有的左值将优先选择函数 (2)，可修改的右值将优先选择函数 (1)。

16.2.6 理解 std::move

库函数 move 就是使用右值引用很好的例子。标准库定义 move 函数如下：

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
```

```
{
    return static_cast<remove_reference<T>::type&&>(t);
}
```

代码简短而微妙，由于 move 的参数是模板类型参数的右值引用 T&&，通过引用折叠这个函数参数可以匹配任何类型的实参，特别是既可以传左值或者右值给 move，如：

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // ok, moving from an rvalue
s2 = std::move(s1); // ok, but after the assignment s1 has indeterminate value
```

需要注意的是，当传递左值过去时，`static_cast<string&&>(t)` 代码会将一个左值引用（string&）强转为右值引用（string&&）。

static_cast 将左值引用转为右值引用是允许的

虽然不能隐式将左值转为右值引用，但是可以使用 static_cast 将左值强制转为右值引用。将右值引用绑定到左值使得操作右值引用的代码可以攫取左值的内容。通过让程序员只能使用强制转换，语言允许这样的用法。而通过强制（forcing）这样做，语言避免了程序员无意中这样做。最后，虽然可以直接写强转表达式，更为简单的方式是使用 move 函数。而且，使用 move 函数将更容易查找这些想要攫取左值内容的代码。

16.2.7 Forwarding

一些函数需要其一个或多个实参以类型保持完全不变的方式转发（forwarding）给另外一个函数。在这种情况下，我们需要保存转发实参的所有信息，包括实参是否为 const 或者实参是左值还是右值。如以下函数：

```
// template that takes a callable and two parameters
// and calls the given callable with the parameters "flipped"
// flip1 is an incomplete implementation: top-level const and references are lost
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
```

这个函数将在传给 flip1 一个其参数是引用类型的函数 f 之前工作完全正常。如以下将无法正常工作：

```
void f(int v1, int &v2) // note v2 is a reference
{
    cout << v1 << " " << ++v2 << endl;
}
```

如果以此函数去调用 flip1 函数，那么 f 将不能改变原始参数，改变的将是被复制的参数。为了达到转发的目的非得将 flip1 的参数改成右值引用形式，只有这样才能保持参数的类型信息。通过将其参数定义为模板类型参数的右值引用（T&&）来保持实参的整个类型信息（引用以及 const 性质）。通过将参数定义为引用可以保持参数的 const 性质，这是因为 const 在引用类型中是底层的。通过引用折叠，如果将函数参数定义为 T1&& 或 T2 &&，将可以保留 flip1 函数实参的左值/右值属性。修改代码如下：

```
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
```

这个版本的 flip2 只解决了问题的一般，flip2 函数可以调用以左值引用为参数的函数，但是对以右值引用为参数的函数就无能为力了。如：

```
void g(int &&i, int &j)
{
    cout << i << " " << j << endl;
}
flip2(g, i, 42); // error: can't initialize int&& from an lvalue
```

原因在于任何函数参数与变量都是左值，即便其初始值是右值。因而，在 flip2 中调用 g 将传递左值给 g 的右值引用参数。解决的办法自然是使用 std::forward 标准函数来保留原始实参的全部类型信息，

与 move 一样，forward 定义在 utility 头文件中。forward 在调用时必须传递一个显式的模板实参，forward 将返回显式实参类型的右值引用。意味着 forward<T> 的返回类型是 T&& 。forward 的实现如下：

```
// Sample implementation of std::forward
// For lvalues (T is T&), take/return lvalue refs.
// For rvalues (T is T), take/return rvalue refs.
template <typename T>
T&& forward(T &&param)
{
    return static_cast<T&&>(param);
}
```

通常 forward 被用于给传递一个定义为模板类型参数的右值引用的函数参数。通过对其返回类型进行引用折叠，forward 保留了其给定实参的左值/右值属性，如：

```
template <typename Type> void intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
}
```

此处使用它 Type 作为 forward 的显式模板类型参数，由于 arg 是模板类型参数的右值引用，Type 将表示传递给 arg 的实参的所有类型信息。如果 arg 的实参是一个右值，那么 Type 将是非引用类型，则 forward<Type> 将返回 Type&& ；如果 arg 的实参是左值，那么通过引用折叠规则，Type 本身就是左值引用类型，则 forward<Type> 的返回类型将是左值引用的右值引用，通过引用折叠最终的返回类型将是左值引用。

通过在类型为模板类型参数的右值引用（T&&）的函数参数调用 forward 标准函数，将保持函数实参的所有类型细节。

使用 forward 来重写 flip 函数：

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

16.3 重载和模板

函数模板可以被别的模板或非模板函数重载。与往常一样，同名的函数必须在参数的数目或类型上有所差异。由于函数模板的出现导致的函数匹配的差异将表现在以下几个方面：

560. 一个调用的候选函数（candidate function）包括所有模板实参推断成功的函数模板实例；
561. 候选函数模板实例讲总是可行函数（viable function），这是由于模板实参推断会排除所有不可行的模板；
562. 与往常一样，可行函数事按照需要进行的转型进行排序的，对于函数模板来说这种转型是非常有限的；
563. 与往常一样，如果一个函数比其它函数提供了更优的匹配，此函数将被选中。然而，如果有好几个函数提供了一样好的匹配，那么：
    1. 如果其中有一个非模板函数，那么这个非模板函数将被调用；
    2. 如果没有非模板函数，但是多个函数模板中其中一个更加特化（specialized），那么这个更特化的函数模板将被调用；
    3. 否则，调用是模糊的（ambiguous）；

为了正确定义重载的函数模板需要对类型之间的关系以及模板函数的有限转型有一个良好的理解；

书写重载模板

如下两个函数是重载的模板，如：

```
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret;
    ret << t;
    return ret.str();
}
```

```
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p;
    if (p)
        ret << " " << debug_rep(*p);
    else
        ret << " null pointer";
}
```

如果调用：string s("hi"); cout << debug_rep(s) << endl; 将只调用第一个函数 debug_rep 函数。如果以 cout << debug_rep(&s) << endl; 进行调用则两个模板函数都是可行的实例。第一个是 debug_rep(const string* &) 其中 T 被推断为 string*；第二个是 debug_rep(string*); 其 T 被推断为 string ；
然而第二个是精确匹配这个调用，第一个却需要将指针转为 const 的，函数匹配将更加青睐于第二个模板。

多个可行模板

考虑以下调用：

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

这里两个模板都是可行的而且是精确匹配的。第一个将被实例化为 debug_rep(const string* &)，其 T 将绑定到 string*；第二个将被实例化为 debug_rep(const string*) 其 T 将绑定到 const string 上。此时常规的函数匹配将无法区别哪一个调用是更优的。然而，由于新加的关于模板的规则，将调用 debug_rep(T*) 函数，这个函数是更加特化的模板。原因在于，如果没有这个规则，将没有任何方法在这种情况下以 const 指针调用指针版本的 debug_rep。问题在于，debug_rep(const T&) 可以在任何类型上调用，包括指针类型。而 debug_rep(T*) 则只能在指针左值上调用，显然是更为特化的版本。
当多个重载版本的模板提供一样优秀的函数调用匹配，最特化的版本将胜出。

非模板和模板之间的重载

考虑非模板版本的 debug_rep 函数：

```
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

以及调用：

```
string s("hi");
cout << debug_rep(s) << endl;
```

此时将有两个一样优秀的函数匹配：debug_rep<string>(const string&) 其 T 绑定到 string，以及这里定义的非模板函数。此时编译器将选择非模板版本。这是由于同样的原因，即最特化的将会被选择，非模板函数是比模板版本更加特化的。
需要注意的是在使用任何重载的函数时记得将这些函数（包括模板函数与非模板函数）在同一个头文件中进行声明。这样在进行函数匹配时编译器可以进行完全的考虑，而不至于忽略任何一个。

16.4 可变参数模板

可变参数模板（variadic template）是新加的一种函数模板或类模板，这种模板可以接收可变数量的模板参数。这种可变参数被称为参数包（parameter pack）。语言允许两种参数包：模板参数包（template parameter pack）表示 0 或多个模板参数，以及函数参数包（function parameter pack）表示 0 个或多个函数参数。
语言使用省略号（ellipsis）来表示模板或函数参数包。对于模板参数列表，class... 或 typename... 表示接下的参数表示一系列 0 个或多个类型；省略号后的名字表示其参数包的任何类型的名字。在函数参数列表中，如果一个参数的类型是一个模板参数包，那么它就是一个函数参数包。如：

```
// Args is a template parameter pack; rest is a function parameter pack
template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest);
```

将 foo 声明为一个可变函数，其中一个类型参数是 T，以及一个模板参数包是 Args，这个参数包表示 0 个或多个额外的类型参数。函数参数列表则有一个参数，其类型是 const T&，以及一个函数参数包 rest，这个参数包表示 0 个或多个函数参数。
与之前一样，编译器从函数实参中推断模板参数类型。对于可变模板，编译器还会推断参数包中的数量，如：

```
int i = 0; double d = 3.14; string s= "how now brown cow";
foo(i, s, 42, d); // three parameters in the pack
foo(s, 42, "hi"); // two parameters in the pack
foo(d, s); // one parameter in the pack
foo("hi"); // empty pack
```

编译器将会实例化 4 个不同的 foo 实例：

```
void foo(const int&, const string&, const int&, const double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

sizeof… 操作符
当希望知道参数包中有多少个元素时，可以使用 sizeof... 操作符，与 sizeof 一样，其返回一个常量表达式并且不会对其实参进行求值：

```
template <typename... Args> void g(Args... args) {
    cout << sizeof...(Args) << endl;
    cout << sizeof...(args) << endl;
}
```

16.4.1 书写可变参数函数模板

在前面用过 initializer_list 来定义函数可以接收不定数目的实参，但是这种实参必须是相同的类型，或者类型可以转为相同的类型。不定参数函数被用于及不知道实参的数目也不知道实参的类型。
不定参数函数将总是递归的，第一个调用处理包中的第一个实参，并以后面的实参调用其自身。如：

```
template <typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t;
}

// this version of print will be called for all but the last element
// in the pack
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", ";
    return print(os, rest...);
}
```

参数包可以是一个空包。

16.4.2 包扩展（Pack Expansion）

除了获取包的大小，另外一件可以做的事是对参数进行展开。当展开包时，可以提供一个模式（pattern）给它使用在每个展开的元素上。展开一个包将使得所有元素称为连续的逗号分隔的列表，并且将模式运用于每个元素上。使用省略号来进行包扩展。如 print 函数就有两个扩展。
第一个扩展将模板参数包进行扩展并产生了 print 的函数参数列表，第二个扩展发生在函数体内调用 print 处，这次将产生一个调用 print 的函数实参列表。
Args 的扩展运用 const Args& 模式给每个 Args 模板参数包中的元素，扩展的结果是一个逗号分隔的 0 个或多个参数类型，每个的类型都是 const T & 。
理解包扩展
函数参数包扩展可以运用更为复杂的模式，如：

```
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&...rest)
```

```
{
    return print(os, debug_rep(rest)...);
}
```

此调用 print 将运用模式 `debug_rep(rest)`，将对函数参数包 rest 的每个元素调用 `debug_rep` 扩展的结果将是逗号分隔的一列 `debug_rep` 函数调用。

16.4.3 转发参数包（Forwarding Parameter Packs）

在新标准下，可以将 forward 与可变参数模板一起使用从而让函数在不改变其实参的任何类型信息的情况下转发给其它函数。如：

```
template <class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc();
    alloc.construct(first_free++, std::forward<Args>(args)...);
}
```

其中 `std::forwad<Args>(args)...` 将同时扩展模板参数包 Args 和函数参数包 args 。如：`svec.emplace_back(10, 'c')` 将被扩展为 `std::forward<int>(10), std::forward<char>(c)`。实践中绝大多数的可变参数参数都将其参数转发给其它函数，就如 `emplace_back` 函数一样。

16.5 模板特例（Template Sepcializations）

一个模板并不总是能满足可以所有使其实例化的模板实参的要求。在某些情况下，通用的模板定义还可能是错误的：通用的定义可能无法编译或者做错误的事情。在某些情况下，可以利用特定类型的知识来写出更加有针对性且高效的代码（至少比从模板中实例化更加高效）。当不想或者不能使用模板版本时，可以定义一个类或函数模板的特例化版本。如：

```
template <typename T> int compare(const T &, const T &);  // (1)

template <size_t N, size_t M>
int compare(const char(&)[N], const char(&)[M]);  // (2)
```

以上两个函数，(2) 函数只能对数组或字符串字面量进行调用，如果传入了字符指针，那么总是 (1) 函数被调用。这在有时并不是想要的结果。如：

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2); // calls (1) template
compare("hi" "mom"); // calls (2) template
```

为了处理字符指针的清醒，需要给 (1) 模板定义特例（template specialization）。特例是模板的另外一个定义，其中一个或多个目标那参数具有特定的类型。

定义函数模板特例

当定义函数模板特例时，需要给原模板中所有的模板参数提供实参。为了表示我们的确是在特例化一个模板，需要使用关键字 tempalte 后跟随一个空的尖括号 `<>`，空的尖括号表示给原模板中的所有模板参数都提供了实参。如：

```
template <>
int comapre(const char* const &p1, const char* const &p2)  // (3)
{
    return strcmp(p1, p2);
}
```

这里最难理解的部分就是特例中的函数参数类型必须与模板中的对应类型一样，这里对应的是 (1) 函数，与类型别名一样，当模板参数类型与指针、const 混杂在一起时也会变得复杂。此处 T 的类型时 `const char*` 即指向 const 对象的指针。

函数重载 vs 模板特例

当定义函数模板特例时，我们是在抢编译器的工作。就是说我们给原始模板的特定实例提供我们自己的定义。这里特别需要留意的是特例是一个实例；它不是重载；由于特例实例化一个模板；它不重载这个模板，因而，特例不会影响函数匹配过程。

如果将模板特例定义为独立的非模板函数将会影响函数匹配。如 `compare("hi", "mom")` 如果在将 (3) 定义为模板特例时，选择的依然是 (2) 函数，原因是它更加的特化 (specialized) 而 (3) 是不参与函数匹配的，

而如果将 (3) 定义为接受指针的非模板函数，此时 (3) 将参与函数匹配并且将选择 (3) ，原因是非模板函数是最特化的版本。
为了定义个模板的特例，原始模板必须被声明在作用域内。并且，在任何使用模板实例的代码之前必须特例的声明必须在作用域内，如果不这么做的话就很可能会从模板中实例化出一个与要求不符合的函数，这种错误通常是很难定位的。
模板和它的特例应该定义在同一个头文件中，而且同一个名字的所有模板都应该出现在前面，后面跟随这些模板的特例。

类模板特例

参考代码：template_specialization.cc
hash<Sales_data> 以 template<> 开始用于表示我们定义一个完全的模板特例。在特例中只需要定义需要另行定义的成员函数，可以将特例的成员函数定义在类内或者类外。

类模板部分特例

与函数模板不同的是，类模板特例不用必须提供所有的模板参数实参。可以指定其中一些但不是全部的模板参数(或者参数的某个方面，如 :左值引用或右值引用性质)。一个类模板的部分特例(partial specialization)本身是一个模板。用户必须给特例没有固定的模板参数提供实参。
只能部分特化一个类模板，但不能部分特化函数模板。
部分特化的一个很好的例子是 remove_reference 类，实现如下：

```
// original, most general template
template <class T> struct remove_reference {
    typedef T type;
};
// partial specializations that will be used for lvalue and rvalue references
template <class T> struct remove_reference<T&>
{ typedef T type; } // lvalue references

template <class T> struct remove_reference<T&&>
{ typedef T type; } // rvalue references
```

该模板的第一个定义是最通用的一个版本，可以实例化任何一个类型。它使用模板实参作为成员类型 type 的类型。后面的两个类是原始模板的部分特例。由于部分特例也是一个模板，所以需要先定义模板参数，与别的特例一样，部分特例具有与其进行特化的模板一样的名字，特例的模板参数列表包含所有类型没有完全固定的模板参数，在类名之后的尖括号中是特例所指定的模板实参，这些实参与原始模板中的参数在位置上一一对应。
部分特例的模板参数列表是原始模板参数列表的一个子集，在此例中，特例具有与原始模板相同数目的模板参数，然而，特例中的参数类型却与原始模板不一样，特例分别使用了左值引用或右值引用类型。以下是一些使用场景：

```
int i;
remove_reference<decltype(42)>::type a;
remove_reference<decltype(i)>::type b;
remove_reference<decltype(std::move(i))>::type c;
```

仅仅特化成员而不是整个类

相比于特化整个模板，可以只特化特定的成员函数。如：

```
template <typename T> struct Foo {
    Foo(const T &t = T()) : mem(t) { }
    void Bar() { /* ... */}
    T mem;
};
template <>
void Foo<int>::Bar()
{
    // do whatever specialized processing that applies to ints
}
```

此例中仅特化一个 Foo<int> 的一个成员 Bar，Foo<int> 的其他成员将由 Foo 模板提供。如：

```
Foo<string> fs; // instantiates Foo<string>::Foo()
fs.Bar(); // instantiates Foo<string>::Bar()
Foo<int> fi; // instantiates Foo<int>::Foo()
fi.Bar(); // uses our specialization of Foo<int>::Bar()
```

当将 Foo 运用于任何类型而不是 int 时，成员照旧进行实例化，而当将 Foo 运用 int 时，除了 Bar 之外的成员照旧进行实例化，只有 Bar 将会使用特化的版本。

关键术语

564. 类模板（class template）：可以实例化类的模板定义，其定义形式是使用 template 关键字后跟随逗号分割的一个或多个模板参数，这些参数都放在了尖括号中，后面跟随一个类定义；
565. 默认模板实参（default template arguments）：当用户不提供对应的模板实参时，模板将使用的类型或值；
566. 显式实例（explicit instantiation）：为模板参数提供了所有显式实参的声明，用于指导实例化过程。如果一个声明是 extern 的，模板将不会被实例化；否则，模板将使用这个指定的实参进行实例化。对于任何一个 extern 模板定义，必须有另外一个非 extern 的显式模板实例；
567. 显式模板实参（explicit template argument）：当定义一个模板类类型或调用函数时由用户提供的模板实参。显式模板实参将放在模板名字之后的尖括号中；
568. 函数参数包（function parameter pack）：参数包表示 0 个或多个函数参数；
569. 函数模板（function template）：可以从中实例化特定函数的模板定义，函数模板的定义将使用 template 关键字后跟随放置于尖括号中的用逗号分割的一个或多个模板参数，后再跟随函数定义；
570. 实例化（instantiate）：用模板实参产生模板的一个特定实例的编译器过程，其中模板实参将替换模板参数。函数通过调用中的实参自动进行实例化，类模板则通过显式提供模板实参来实例化；
571. 成员模板（member template）：本身是模板的成员函数，成员模板不是虚函数；
572. 非类型参数（nontype parameter）：表示一个值的模板参数，非类型模板参数的模板实参必须是一个常量表达式；
573. 包扩展（pack expansion）：一个参数包被替换为对应的元素列表的过程；
574. 参数包（parameter pack）：模板或函数参数，其表示 0 个或多个参数；
575. 部分特例（partial specialization）：类模板的一个版本，其中一些但不是全部的模板参数被指定（specified）或完全指定（completely specified）；
576. 模式（pattern）：定义了扩展参数包时运用于每个元素的动作；
577. 类型转发（type transformation）：由库定义的类模板，将其模板类型参数转发给相关的类型；
578. 类型参数（type parameter）：在模板参数列表中用于表示一个类型的名字；

17.1 tuple 类型

17.1.1 定义和初始化 tuple

访问 tuple 的成员

关系和相等操作符

17.1.2 使用 tuple 以返回多个值

返回 tuple 的函数

使用函数返回的 tuple

17.2 bitset 类型

17.2.1 定义和初始化 bitset

用 unsigned 初始化 bitset

用 string 初始化 bitset

17.2.2 bitset 上的操作

从 bitset 中获取值

bitset 的 IO 操作符

使用 bitset

17.3 正则表达式

17.3.1 使用正则表达式库

为 regex 对象指定选项

C++ 可以处理的问题的范围十分广泛，从只需要一个程序员在几个小时内就能解决的小问题，到需要涉及到多个系统协作，有着百万行级别的代码量，并且需要几百个程序员在多年时间内参与的大问题（如：操作系统）。本书前面章节介绍的内容同时适合于所有这些跨度的问题。
语言还包括一些针对大的复杂系统而设计的特性。这些特性包括：异常处理（exception handling），名称空间（namespaces）和多重继承（multiple inheritance），这是本章将阐述的内容。
大型规模编程（Large-scale programming）对语言的要求要远高于小团队开发的系统对语言的要求。这些要求包括：
579. 在独立开发的多个子系统间处理错误；
580. 使用独立开发的库而不产生名称冲突；
581. 对更加复杂的应用概念进行建模；
本章将讲述针对以上需求而设计的语言特性：异常、名称空间和多重继承。

18.1 异常处理

异常处理(Exception handling)允许独立开发的程序部分可以在运行时的错误进行通信(communicate)和处理（handle）。异常使得我们可以分离问题的发现部分和问题的处理部分。程序的一部分可以发现问题，然后将解决问题的工作传递给程序的另一部分。检测部分不需要知道处理部分的细节，反之亦然。
有效的使用异常处理需要理解当异常抛出时发生了什么，当捕获时发生了什么，以及用来传递错误的对象的含义。

18.1.1 抛出异常

在 C++ 中，异常是通过抛出（throwing）一个表达式来引发（raised）的。抛出的表达式的类型以及当前的调用链决定了哪个处理器（handler）将处理此异常。选中的处理器是调用链中匹配抛出对象类型的最近的代码。抛出对象的类型和内容允许抛出部分给处理部分提供出错信息。
当 throw 被执行时，throw 之后的语句是不会执行的。相反，控制(control)将从 throw 转移到对应的 catch 处。catch 子句可能在同一个函数中，也可能在直接或间接调用了发生异常的函数的函数中。控制从一个地方转到另一个地方的事实有两个重要的暗示：
582. 调用链上的所有函数调用将永久退出；
583. 当进入一个处理器时，调用链上创建的对象将被销毁；

栈展开（Stack Unwinding）

当异常抛出时，当前函数的执行被中止并开始搜索匹配的 catch 子句。如果 throw 出现在一个 try 块中，那么与之相对应的 catch 子句将首先被检查，如果找到了匹配的 catch 子句，异常就被此 catch 所处理。否则，如果 try 被嵌套在另外一个 try 中，那么将继续搜索外层的 catch 子句。如果没有任何 catch 匹配此异常，那么当前函数将退出，并且继续搜索发起调用的函数。这样一直向上，称为栈展开，直到找到一个匹配异常的 catch 子句，或者在没有找到任何匹配的 catch 子句时 main 函数自己退出。
假如找到了一个匹配的 catch 子句，将执行 catch 中的代码，当其完成后，将执行其后的第一条非 catch 子句代码。如果没有找到任何匹配的 catch 子句，程序将退出。异常是必须处理的，因为异常的目的就是阻止程序继续按常规执行，如果不处理异常则程序会隐式调用 terminate 库函数来终止程序的执行。
没有被捕获的异常将终止程序的执行。

栈展开时对象将自动被销毁

在栈展开时，调用链中的语句块将会永久退出，通常语句块中将创建本地对象，而本地对象则在语句块退出时销毁。栈展开执行相同的逻辑：当一个语句块在栈展开时退出，编译器保证其中创建的对象被适当的销毁。如果本地对象是类类型，对象的析构函数将执行，如果是内置类型，那么将不执行任何操作直接销毁。
异常可能发生在构造函数中，那么对象可能处于部分构建（partially constructed）状态。其中一些成员已经被初始化了，但是另外一些成员在异常发生时还没有初始化。即便处于部分构建状态，编译器将保证已经构建的成员将被销毁。
同样，异常可能发生在数组或容器元素的初始化过程中，编译器将保证在异常发生前构建的元素将被销毁。

析构函数和异常

析构函数总是执行，而函数中释放资源的代码可能会被跳过影响着我们如何组织程序。如果一个代码块分配了资源，但是异常发生在释放资源的代码之前，那么释放资源的代码将不会执行。另一方面，由类类型

对象分配的资源肯定会被析构函数释放。通过使用类来控制资源的分配，我们可以保证资源总是被合理的释放，而不管函数是正常结束还是由异常导致结束。
栈展开时将执行析构函数影响着我们如何写析构函数。在栈展开时，异常已经引发但是还没有被处理。如果栈展开过程中又抛出一个新的异常，而没有在抛出的函数中捕获的话就会调用 terminate 函数。由于析构函数会在栈展开中调用，析构函数不应该抛出任何它自己不处理的异常。也意味着如果析构函数调用了可能抛出异常的函数，需要将其放在 try 块中，并将异常处理掉。
在现实中，由于析构函数只释放资源，它不太可能会抛出异常。所有的标准库中的类型都保证其析构函数不会抛出异常。
注意：在栈展开中，析构函数抛出了异常但是并没有处理，那么程序将退出。
异常对象
编译器使用抛出表达式来拷贝复制一个特殊对象称为异常对象（exception object）。所以抛出的对象必须是完全类型（complete type），如果对象是类类型，那么其必须具有可访问的析构函数和可访问的拷贝或移动构造函数。如果对象是数组或者函数类型，那么其将被转型为对应的指针类型。
异常对象驻留于编译器管理内存空间中，当任何 catch 子句被调用时，这个异常对象就会被访问，这个异常对象将在异常被处理之后被销毁。
抛出本地对象的指针是错误的用法，因为在栈展开时本地对象会被销毁。

18.1.2 捕捉异常

catch 子句中的异常声明（exception declaration）非常类似于只有一个参数的函数参数列表。当 catch 不需要访问抛出的异常对象时，异常声明中的名字可以省略。
异常声明中的类型决定了可以处理的异常，这个类型必须是完全类型，可以是左值引用但不能是右值引用。catch 子句非常类似于函数体。当进入 catch 子句时，异常声明中的参数将被初始化为异常对象，与函数参数一样，如果 catch 参数是非引用类型，那么 catch 参数是异常对象的拷贝；在 catch 中对参数做的任何改变都是针对本地拷贝而与异常对象本身没有任何关系。如果参数是引用类型，那么与任何别的引用参数是一样的，catch 参数是异常对象的另一个名字。对参数做的任何改变都会反映到异常对象上。
与函数参数一样，如果 catch 参数是基类类型，其可以被初始化为子类类型对象。如果 catch 参数是非引用类型，那么异常对象将被裁剪（sliced down），如果参数是基类类型的引用的话，那么参数将被绑定到异常对象上。
同样，异常声明的静态类型决定了 catch 可以执行的操作，如果 catch 参数是基类类型，那么 catch 就不能执行派生类类型的任何操作。
最佳实践：一个 catch 如果是处理通过继承关联起来的类型的异常对象时，应该将其声明为引用。
查找一个匹配的处理器
在查找匹配的 catch 时，找到的 catch 不需要是最匹配异常的那个，而是第一个匹配异常的那个。因而，在一串 catch 子句中，最具体的 catch 子句应该第一个出现。
由于 catch 子句是由其出现的顺序进行匹配的，使用具有继承关系的异常对象的程序必须将其 catch 子句进行排序使得处理派生类的处理器出现在处理基类的 catch 子句前。
异常匹配的规则比之函数参数匹配更加严格，绝大多数时候 catch 声明的异常类型必须与异常对象的类型完全一致，只有在极少数的情况下两者之间可以有差异：
584. 从非 const 到 const 的转换是允许的，意味着抛出一个非 const 对象可以匹配一个声明为捕获 const 引用的 catch 子句；
585. 从派生类到基类的转换是允许的；
586. 数组可以转为其元素类型的指针；函数可以转为函数类型的指针；
其它的任何转型都是不允许的，特别指出的是不允许标准算术转型和类类型定义的转型。
重新抛出（Rethrow）
有时单个 catch 可能不能完全处理一个异常，它可以通过重新抛出将异常传递给上层的 catch 子句处理。重新抛出形如：`throw;` 就是 throw 后不跟随任何对象。空的 throw 只能出现在 catch 子句中，或者由 catch 调用的函数。如果一个空 throw 出现在非 catch 中，terminate 将被调用。
重新跑出异常不会指定对象，其当前的异常对象被传递到调用链的上层。通常 catch 会改变异常对象，这个改变后的异常对象只有被声明为引用时才会被上层 catch 子句观察到。
捕获所有的处理器（The Catch-All Handler）

可以通过 `catch(...)` 的方式来捕获所有的异常，这个称为 catch-all 处理器，这种处理器通常与重新抛出结合在一起，其做完任何可以做的本地工作然后重新抛出异常。如果将 `catch(...)` 与其它 catch 子句使用时应该放在最后的位置，其后的任何 catch 子句都不会被匹配到。

18.1.3 函数级 try 语句块和构造函数

异常可能会出现在处理构造函数的初始化时，在进入构造函数体之前需要先执行构造函数初始化，在构造函数体中的 catch 不能捕获有初始化器抛出的异常，这是由于构造函数体中的 try 块在异常抛出时还没有起作用。

为了捕获构造函数初始化器中的异常，必须书写构造函数为函数级 try 语句块（function try block）。这种语句块允许我们将一系列 catch 与构造函数的初始化阶段（或析构函数的析构阶段）和构造函数体（或析构函数体）关联起来。如：

```
Blob(std::initializer_list<std::string> il) try :
    data(std::make_shared<std::vector<std::string>>(il)) {
        /* ... */
} catch (const std::bad_alloc &e) {
    /* handle_out_of_memory(e); */
}
```

此时 try 关键字出现在开始构造函数初始化列表的冒号之前，这也在开始构造函数体的大括号之前。与之关联的 catch 子句可以捕获成员初始化中或构造函数体中抛出的异常。

值得注意的是出现在构造函数参数本身时发生的异常不会被函数级 try 语句块捕获，只有开始构造函数的初始化列表后的异常才能被捕获。捕获这种异常的职责是调用表达式的，需要有调用者来处理。

注意：书写函数级 try 块是处理构造函数初始化列表中抛出异常的唯一方法。

18.1.4 noexcept 异常说明符

知道函数不会抛出任何异常对于程序员和编译器来说都是有好处的，程序员书写调用代码将更加简单，编译器则可以生成更加优化的机器码。

在新标准中，函数可以通过 noexcept 说明（noexcept specification），在函数参数列表后放置的 noexcept 关键字表示函数不会抛出异常：

```
void recoup(int) noexcept; // won't throw
void alloc(int); // might throw
```

我们说 recoup 做了不抛出异常说明（nothrowing specification）。

noexcept 说明必须出现在所有的声明和定义处或者不出现在任何一个上。并且放在尾后返回类型之前。同样，noexcept 需要放在函数指针的声明或定义处。不过不需要放在类型别名或 typedef 处。在成员函数中，noexcept 说明符放在 const 或引用限定符之后，在 final 、override 或虚函数的 `= 0` 之前。

违反异常说明

需要指出的是编译器并不会在编译时对 noexcept 说明进行检查，甚至编译器都不能拒绝函数体内包含 throw 语句或者调用可能会抛出异常的函数的语句进行 noexcept 声明，不过某些编译器会对此进行警告。

```
void f() noexcept
{
    throw exception();
}
```

如果 noexcept 函数抛出了异常，编译器将调用 terminate 函数来结束程序执行，这就强制保证了 noexcept 函数不会抛出异常。这里没有特别指定是否会进行栈展开。所有 noexcept 使用的场景应该是如下两个场景：1. 对于函数确实不会抛出异常有充分的自信；2. 对于函数抛出异常，不知如何进行处理；指明函数不会抛出异常保证调用者不需要处理异常，要么函数真的不会抛出异常，要么程序就直接结束了。

警告：编译器通常不能也不会在编译期检查异常说明。

向后兼容

早期版本的 C++ 的异常说明更加复杂，允许指定一个函数可能抛出的异常，但是现在几乎是没有什么人使用这种方式了，并且被废弃了。但是有一个方式是经常使用的就是：`throw()` 来表明函数不抛出任何异常；如：

```
void recoup(int) noexcept;  // recoup doesn't throw
void recoup(int) throw(); // equivalent declaration
```

noexcept 说明的实参

noexcept 可以有一个可选的实参，必须是可以转换为布尔值的，如果为 false 的话就表示可能会抛出异常，true 则不会抛出。如：

```
void recoup(int) noexcept(true);  // recoup won't throw
void alloc(int) noexcept(false); // alloc can throw
```

noexcept 操作符

noexcept 说明的实参经常是由 noexcept 操作符求值所得，noexcept 是一元操作符，返回的 bool 型的右值常量表达式，其求值发生在编译期所以不会对表达式求值，而是进行编译推导。这与 sizeof 是一样的。如：

```
void f() noexcept(noexcept(g()));  // f has same exception specifier as g
```

其中 noexcept(e) 在 e 调用的所有函数都不抛出异常，并且 e 本身不抛出异常的情况下，返回 true，否则就是 false。

异常说明以及指针、虚函数、拷贝控制

将指针声明为只能指向不抛出异常的函数，可以赋值的函数必须是不抛出异常的。而如果将指针声明为可能会抛出异常，那么就无所谓了，任何符合的函数都可以赋值给这种指针。如：

```
void (*pf1)(int) noexcept = recoup;
void (*pf2)(int) = recoup;

pf1 = alloc; // error: alloc might throw but pf1 said it wouldn't
pf2 = alloc;
```

如果一个虚函数将自己声明为不会抛出异常，那么子类的覆盖函数必须同样不抛出异常。而基类虚函数可能会抛出异常，子类覆盖函数则可以更加严格的保证不抛出异常。如：

```
class Base {
public:
    virtual double f1(double) noexcept;
    virtual int f2() noexcept(false);
    virtual void f3();
};

class Derived : public Base {
public:
    double f1(double) override; // error: Base::f1 promises not to throw
    int f2() noexcept(false) override;
    void f3() noexcept override;
};
```

当编译器合成拷贝控制成员时，同样会合成异常说明符。如果合成成员的所有成员和基类对象的对应函数保证不会抛出异常，那么合成的成员不会抛出异常。否则其中任何一个函数可能会抛出异常，那么合成的就会抛出异常。并且，如果我们没有自己定义的析构函数提供异常说明，编译期会自动合成一个异常说明，这个合成的异常说明与合成的析构函数的异常说明一样。

18.1.5 异常类层级

exception 类本身只定义了拷贝构造函数、拷贝赋值操作符和虚析构函数以及虚函数成员 what，what 函数返回 C-字符串指针，并且保证不会抛出异常，这个字符串用于说明异常的错误信息。除此之外可以定义自己的异常类型，这些类型可以继承自 exception 也可以不是。甚至抛出的异常可以是内置类型。

18.2 名称空间

大的项目中通常需要用到第三方库，这些库通常都会定义一个名称空间，没有定义名称空间而直接使用的名字会带来名称空间污染（namespace pollution），C 语言通过将库的名字作为函数等全局名字的前缀来避免此问题。C++ 则提供了名称空间，名称空间可以更有效的更好的管理名字。名称空间将全局名称空间进行切分，每个名称空间就一个作用域。

18.2.1 名称空间定义

名称空间中包含一系列的声明和定义，这些声明和定义必须是可以出现在全局作用域中的：类、变量（以及初始值）、函数以及其定义、模板和其它名称空间。

与任何名字一样，名称空间的名字必须在其作用域中是唯一的。名称空间可以定义在全局作用域中或者在别的名称空间中，但不能在函数或者类中。
名称空间不以分号结束。

每个名称空间是一个作用域

每个名称空间中的每个名字都必须指向独一无二的实体，在不同的名称空间中的相同名字是不同的实体。同一个名称空间中的名字可以被直接访问（包括嵌套的作用域）。而在名称空间之外则需要用完全限定名来进行访问。

名称空间不需要连续

名称空间可以分离定义，在不同的文件或者在同一个文件的不同地方。比如：

```
namespace nsp {}
```

要么定义一个新的名称空间 nsp，要么给已经存在的名称空间增加内容。如果 nsp 之前没有定义过，那么就是创建一个新的名称空间。否则就将定义增加到已经存在的名称空间中去。

名称空间的这种用法主要用来适配类定义以及函数的定义。名称空间中定义类和声明函数是接口的一部分，将被放在头文件中。而名称空间的成员实现则被放在源文件中。通过将接口和实现分离，可以保证名称空间中的名字只被定义以此，但是可以被声明多次。 如果一个名称空间中包含了多个不相关的类型，应该使用分离的文件来书写这些名称空间和每个类型。

需要注意的是，`#include` 必须出现在所有的名称空间之前，否则就是将所有被包含的文件中的名字在我们的名称空间中再次定义一次。

定义名称空间成员

在同一个名称空间中的成员之间相互通过非限定名称进行引用，也可以在名称空间外面定义成员，定义需要指定名字是属于哪个名称空间的。

```
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs, const Sales_data& rhs)
{
    Sales_data ret(lhs);
}
```

这个定义的声明必须存在于对应的名称空间中，与定义类的成员函数一样，函数体是在名称空间中的，所以可以不加限定地使用名称空间中的名字。

模板特例

模板特例需要放在与原始模板相同的作用域中，与别的名字一样，模板特例可以在作用域中声明，然后在外面进行定义。

全局名称空间

在全局作用域中定义的名字被放在全局名称空间（global namespace）中，全局名称空间是隐式定义的，并且存在于每一个程序中。每个文件中定义在全局作用域中的名字都被放到了全局名称空间中。

引用全局名称空间中的名字需要使用 `::member_name` 的方式。

嵌套的名称空间

定义在别的名称空间中的名称空间就是嵌套名称空间。嵌套名称空间是一个嵌套作用域，其作用域被嵌套在另外一个名称空间中。规则与嵌套作用域一样，定义在内部名称空间中的名字会隐藏外部名称空间中的名字，外部名称空间想要访问嵌套名称空间中的名字必须通过完全限定名称进行访问。

inline 名称空间

新标准中引入了一种新的嵌套名称空间，inline namespace。与常规的嵌套名称空间不同的是，inline 名称空间中的名字可以被其直接包含的名称空间直接，而不需要加以限定。如：

```
inline namespace FifthEd {}
inline namespace FifthEd {
    class Query_base {};
}
```

inline 名称空间多用于从应用的一个版本迁移到另一个版本。

unnamed 名称空间

匿名名称空间（unnamed namespace）由关键字 namespace 后直接跟名称空间定义，在匿名名称空间中定义的名字是 static 的，当第一次使用时存在，并且在程序结束时销毁。

匿名名称空间不会跨越多个文件，每个文件有自己的匿名名称空间，它们可以定义同名的名字但是不是同一个实体。不要在头文件中定义匿名名称空间。
匿名名称空间中的名字可以直接被使用，它们的作用域在其直接外部名称空间，所以需要其中的名字没有冲突。如：

```
namespace local {
    namespace {
        int i;
    }
}
local::i = 42;
```

C++ 语言试图用匿名名称空间来替换 static 变量声明。

18.2.2 使用名称空间的成员

通过 using 声明（using declarations）、名称空间别名（namespace aliases）和 using 指令（using directives）来简化名称空间的使用。

名称空间别名

名称空间别名将一个较短的名字作为名称空间名字的别名。如：

```
namespace primer = cplusplus_primer;
```

名称空间别名可以表示一个嵌套的名称空间，如：

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query q;
```

using 声明

using 声明在一次引入一个名称空间的成员。由 using 声明引入的名字遵循常规的作用域规则：从引入的地方可见直到作用域的结尾处结束。外部作用域中的相同名字被隐藏。在内部将其嵌套的作用域中可以不加限定的访问该名字，出了作用域就需要使用完全限定名字。
using 声明可以出现在全局、局部、名称空间和类作用域中。如果类作用域中则 using 声明只能针对基类成员。

using 指令

using 指令可以让名称空间中的所有名字都不加限定的进行访问。using 指令的形式是 `using namespace NAMESPACE`, using 指令只能出现在全局、局部和名称空间作用域中，不能出现在类作用域中。

using 指令和作用域

由 using 指令导入的名字的作用域比 using 声明的更加复杂。using 声明将名字放在 using 声明所在的作用域中，就像定义了一个本地变量一样，如果前面有一个相同的名字就会报错。相反，using 指令将所有的名字提升到最近的名称空间中，这个名称空间同时包含 using 指令后的名称空间以及 using 指令本身所在的名称空间。如以下就是错误的：

```
namespace blip {
    int i = 16, j = 15, k = 23;
}
int j = 0;
void mainp()
{
    using namespace blip;
    ++i;
    ++j; // error: 不知道是 global j 还是 blip::j
    ++::j;
    ++blip::j;
    int k = 97;
    ++k;
}
```

以允许 blip 中的名字与全局名称空间中的名字一样，但是如果想要引用这些名字的话就需要明确限定说明使用的是哪个名字，否则就会产生名字二义性。

头文件和 using 声明或 using 指令

头文件中至应该包含接口部分的名字，不应该包含任何实现部分的名字。因而，头文件不应该在函数或者名称空间外使用 using 声明或 using 指令。
应该尽可能少的使用 using 指令，而在需要的时候使用 using 声明。

18.2.3 类、名称空间和作用域

名称空间中的名字查找一样是从内部作用域往外面不停查找，并且只查找外部作用域在前面声明的名字。名称空间中的类的成员函数中的名字先从成员函数中查找，再从类中查找，然后从所在的外围作用域中查找，最后才是从定义所在的地方进行查找。如：

```
namespace A {
    int i;
    int k;

    class C1 {
    public:
        C1() : i(0), j(0) { }
        int f1() { return k; }
        int f2() { return h; } // error: h 还没有定义
        int f3();
    private:
        int i;
        int j;
    };
    int h = i; // 从 A::i 初始化
}
int A::C1::f3() { return h; }
```

这里需要注意的是 A::h 是在 f2 后定义的，所有无法通过编译，而 f3 是在 A::h 后定义的，所以可以访问。

由实参决定的查找（Argument-Dependent Lookup）和类类型参数

当传递一个类类型对象给函数时，编译器将在正常的作用域查找之外从实参的类定义的名称空间中查找。这个规则将会运用于类类型的引用和指针实参。如：`std::string s; std::cin >> s;` 在调用 >> 就不需要指定 std 名称空间，将会自动从 cin 或者 s 的名称空间从查找函数。
这个规则的意义在于不需要为概念上是类的接口，但不是类的成员函数，在使用时不需要单独的 using 声明。

查找 std::move 和 std::forward

由于 `std::move` 和 `std::forward` 的参数是右值引用，所以是可以匹配任何参数的。这样如何应用程序定义了别的 move 的话就会产生名字冲突（name collision）。所以在使用时尽可能地使用 std 进行限定。

友元声明和由实参决定的查找

如果一个未声明的类或函数第一次出现在友元声明中将被认为是定义在最接近的外围名称空间中，这与由实参决定的名称查找会产生意想不到的结果。如：

```
namesapce A {
    class C {
        friend void f2(); // won't be found
        friend void f(const C&); // found by argument-dependent lookup
    };
}
```

通过由实参决定的名称查找可以调用 f，如：

```
int main()
{
    A::C cobj;
    f(cobj);
    f2();
}
```

由于 f 的参数是类类型，而 f 是隐式声明在名称空间 A 中，所以 f 将被找到并被被调用。

18.2.4 重载和名称空间

由实参决定的名称查找和重载

带有类类型实参的函数查找函数名字时同时将在每个实参所在类及其基类的名称空间中查找此函数名字。这个规则同时会影响重载候选集。每个实参定义所在的名称空间都会被查找，所有这里面的同名函数都会被添加到候选集中。即便是这些函数在调用点看不到也会被添加到候选集中。如：

```
namespace NS {
    class Quote {};
    void display(const Quote&) {}
}
class Bulk_item : public NS::Quote {};
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

以上调用能够通过的原因是 display 在 Bulk_item 的基类所在的名称空间中进行查找同名函数。

重载和 using 声明

using 声明导入的是整个名字，而不是特定的函数。using 声明引入的函数可以对当前作用域中的同名函数进重载，而如何函数原型完全一样则会导致编译错误。

重载和 using 指令

using 指令将名称空间中的成员提升到最近的外围作用域中。如果名称空间中的成员与当前作用域中的名字同名，名称空间中的名字被添加到重载集合中。如果 using 指令引入的名称空间中的函数与当前作用域中的函数具有相同的原型，这不是一种错误。只要在调用时指定希望调用名称空间中的，还是当前作用域中的。

在多个 using 指令之间重载

如果一次性出现多个 using 指令，那么每个名称空间中的名字都会变成候选集中的一员。

18.3 多重继承和虚继承

C++ 是可以多重继承的（Multiple inheritance），意味着一个派生类可以多个直接基类。多重派生类（multiply derived class）继承其所有的父类的属性。尽管在概念上很简单，但是在细节上涉及到多个直接基类的时候会在设计层面和实现层面遇到许多问题。

18.3.1 多重继承

多重继承的派生列表会包含多个基类。如：

```
class Panda : public Bear, public Endangered { /* ... */ }
```

每个基类都有一个可选的访问说明符，如果省略的话就提供默认的说明符，对于 class 是 private，对于 struct 是 public。与单一继承一样，派生列表中的类必须是已经定义的，并且不能是 final 的。语言并没有限制具体可以在派生列表中包含多少个类。每个基类只能在派生列表中出现一次。

多重派生类从每个基类中继承状态

在多重继承下，一个派生类对象将包含所有基类的子对象。如：Panda 类中包含了 Bear 和 Endangered 子对象，以及它自身定义的成员。

派生构造函数需要初始化所有基类

构建派生类对象需要构建和初始化其所有的直接基类子对象。如：

```
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }
// 以下意味着 Bear 对象是默认初始化的
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

构造函数的初始化列表负责初始化所有的直接基类，可将参数传递个直接基类的构造函数作为实参。直接基类的初始化顺序它们出现在派生列表中的顺序。

继承构造函数和多重继承

在新标准下，可以用 using 声明的方式从一个或多个基类中继承构造函数，如果从多个基类中继承具有相同的签名的构造函数将导致编译错误。如果发生了这样的情况需要派生类重新定义此构造函数。如：

```
struct Base1 {
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);
};
struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);
};
struct D1 : public Base1, public Base2 {
    using Base1::Base1;
    using Base2::Base2;
    // D1 must define its own constructor
    D1(const std::string &s) : Base1(s), Base2(s) {}
    // needed once D1 defines its own constructor
    D1() = default;
};
```

析构函数和多重继承

多重继承的析构函数与单一继承的析构函数没有什么不同。析构函数本身只需要负责它自己的资源的释放，成员和基类的资源都由他们各自释放。如果没有定义析构函数，编译器会自动合成一个，其函数体依然是空的。析构的顺序与构造的顺序刚好是完全相反的，所以将先调用成员的析构函数（属于最底层的派生类），再依次以派生列表的相反顺序调用基类的析构函数。

多重派生类的拷贝和移动操作

与单继承一样，如果多重继承的子类如果要定义自己的拷贝、移动构造函数以及赋值操作符将必须拷贝、移动或赋值整个对象。只用派生类使用这些成员的合成版本时，基类才会自动进行拷贝、移动或赋值。在派生类的合成版本中会自动隐式使用基类的对应成员。

18.3.2 转换和多基类

在多重继承下任何可访问的基类的指针或引用都可以绑定到派生对象上。事实上，编译器认为所有以上的转换是同样好的，意味着如下代码将是编译错误：

```
void print(const Bear&);
void print(const Endangered&);
Panda ying_yang("ying_yang");
print(ying_yang); // error: 二义性
```

基于指针或引用类型的查找

在多重继承中，对象、指针、引用的静态类型决定了使用哪个成员，即便是指向子类对象，其其它的基类的接口或者子类自己的接口亦是不可用的。

18.3.3 多重继承下的类作用域

在单一继承下，派生类的作用域被嵌套在直接和间接基类中。名字的查找将沿着继承链一直往上，定义在派生类中的名字将屏蔽掉基类中的名字。在多重继承中，名称查找将同时在所有直接基类中查找，如果一个名字在多个基类中找到就被认为是具有二义性。即便是两个名字所代表的函数的函数原型不一样也是错误的，甚至两个名字其中一个不是函数也会产生二义性错误。与往常一样，名称查找发生在类型检查之前。

多重继承中继承相同名字是可以的，但是如果想要引用其中一个名字则需要指定哪个版本。最好的办法是在派生类中为这些可能产生二义性的名字重新定义一个函数。

18.3.4 虚继承

一个类可能继承一个相同的基类多次，原因在于某些基类都继承自同一个基类。这种情况下将导致同一个基类有两个子对象。但是有时我们需要让这个相同的基类只有一个子对象。那我们通过虚继承（virtual

inheritance）来解决此问题，共享的基类子对象成为虚基类（virtual base class），不管这个虚基类在继承链中出现了多少次，只有一个共享子对象。用法如下：

```
class Raccoon : public virtual ZooAnimal { };
class Bear : virtual public ZooAnimal { };
```

此处 ZooAnimal 是虚基类，virtual 告知愿意在接下来的继承中共享同一个基类对象，对于所使用的基类本身并没有什么特别的限制。

```
class Panda : public Bear, public Raccoon, public Endangered {};
```

Panda 对象中就只有一个 ZooAnimal 子对象了。

支持到基类的转换

不过一个基类是不是虚基类都可以用其指针或引用指向派生类对象。

虚基类成员的可见性

如果虚基类中的成员被其中一个路径上的派生子类对象覆盖而不是被所有路径上的派生子类对象覆盖的话，那么引用这个成员将是派生对象上的成员，如果所有路径都覆盖的话，那么就会产生二义性错误。这时最好的做法就是在底层的派生类对象中重新定义此成员。

18.3.5 构造函数和虚继承

在虚继承中，虚基类是由最后面的派生构造函数进行初始化，否则的话虚基类就可能在所有路径中被初始化，从而导致初始化多次。当继承体系中派生类如果可以独立被创建的话，它的构造函数也是最后面的派生构造函数，此时它也需要对虚基类进行初始化。在这种情况下将由最底层的派生类构造函数首先对虚基类子对象进行初始化，后面遇到的初始化虚基类子对象将会被忽略。然后再按照正常的派生列表顺序进行初始化其它基类。如：

```
Panda::Panda(std::string name, bool onExhibit) :
ZooAnimal(name, onExhibit, "Panda"),
Bear(name, onExhibit),
Raccoon(name, onExhibit),
Endangered(Endangered::critical)
{}
```

虚基类总是在非虚基类前被初始化，而不管它们出现在继承层级的哪个位置。

构造和析构顺序

如果一个类有多个虚基类，那么其顺序将按照出现在派生列表中的顺序进行初始化。如：

```
class Character {};
class BookCharacter : public Character {};
class ToyAnimal {};
class TeddyBear : public BookCharacter,
    public Bear, public virtual ToyAnimal {};
```

将按照如下顺序进行初始化：

```
ZooAnimal();  // Bear 的虚基类
ToyAnimal();  // 直接虚基类
Character();
BookCharacter();
Bear();
TeddyBear();
```

对于拷贝和移动构造函数来说其顺序是一样的，合成的赋值操作符则是按照此顺序进行赋值的。而析构函数则以此反方向执行。

关键术语

587. 构建顺序（constructor order）：在非虚继承的情况下，基类按照出现在派生列表中的顺序进行执行。在虚继承的情况下，虚基类将首先被构建，它们之间是按照出现在派生列表中的顺序进行构建的。只有最底层的派生类可以初始化这个虚基类子对象；出现在中间层次的类对其初始化的运算将会被忽略；
588. 异常对象（exception object）：用于 throw 和 catch 之间进行通讯的对象。此对象在 throw 时创建，是抛出表达式的拷贝，异常对象将存在直到匹配的处理器执行完，对象类型是抛出表达式的静态类型。

589. 函数 try 块（function try block）：使用来捕获构造初始化器中的异常。 try 关键字出现在开始初始化列表的冒号之前，并且 catch 子句出现在函数体的后面；
590. 内联名称空间（inline namespace）：在名称空间前加上 inline，成员将被放在外围的名称空间中；
591. 名称空间（namespace）：将库中的名字收集到一个单一作用域的机制，与其它的 C++ 作用域不一样，名称空间可以被定义几个分离的地方；
592. noexcept 操作符：这个操作符返回一个 bool 来指示一个表达式是否会抛出异常，表达式本身不会被求值。结果是一个常量表达式，当为 true 时表示不抛出异常；
593. noexcept 说明：用于告知函数是否为抛出异常的关键字，这个关键字后可以跟一个括号中的布尔值，当省略时或者布尔值为 true 时表示不会抛出异常。
594. 不抛出异常说明（nonthrowing specification）：函数不会抛出异常的说明，如果一个承诺不抛出异常的函数抛出了异常，将会调用 terminate；
595. 匿名名称空间（unnamed namespace）：没有名字的名称空间。匿名名称空间在每个文件中都有唯一的一个，并且可以直接访问而不需要作用域操作符。匿名名称空间中的名字在文件在是不可见的。

19.1 控制内存分配

当对内存分配有特别需求的时候可以重载 new 和 delete 操作符来控制内存分配。

19.1.1 重载 new 和 delete

重载 new 和 delete 操作符的方式与重载其它操作符的方式有非常大的不同。当使用 new 表达式时会依次发生三件事：1. 调用库函数 operator new 或者 operator new[] 来分配足够的裸的没有类型的内存以存储对象或数组。2. 调用构造函数构造对象。3. 返回指向这个新分配构建的对象的指针。

当使用 delete 表达式时将发生两件事。1. 调用析构函数进行析构；2. 编译器调用库函数 operator delete 或者 operator delete[] 来释放内存。

定义自己的 operator new 和 operator delete 函数可以 hook 掉内存分配过程。即便是库中已经包含了这些函数的定义，我们依然可以定义自己的版本，而编译器并不会抱怨说重复定义。反而，编译器会使用我们的版本。

当我们定义全局的 operator new 和 operator delete 函数，程序将在任何时候需要分配内存时都调用我们的函数。当编译器看到 new 或 delete 表达式时就会找到对应的操作符函数进行调用。如果被分配的对象是类类型，那么将首先在类作用域内（包括其基类作用域中）查找这两个函数，如果存在就调用它们。否则将从全局作用域中查找，如果编译器找到一个用户定义的版本，将调用此版本，如果还是找不到将调用库中的版本。

使用作用域操作符来指定使用的 new 或 delete 的版本，如：`::new` 和 `::delete`

operator new 和 operator delete 接口

库中定义了 8 个重载的 operator new 和 delete 函数，前面四个支持抛出 bad_alloc 异常，后面的则支持不抛出异常。如：

```
//抛出异常的版本
void *operator new(size_t);
void *operator new[](size_t);
void *operator delete(void*) noexcept;
void *operator delete[](void*) noexcept;

//不抛出异常的版本
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

`nothrow_t` 是定义在 new 头文件中的 struct，这个类型没有成员。new 头文件中还定义了 nothrow 的常量对象，用户可以将其传递给不抛出异常的 new 。

与析构函数一样，operator delete 是永远不会抛出异常的。

如果应用程序定义以上函数，必须定义在全局作用域或者作为类的成员，如果定义为类的成员，那么这些操作符函数则隐式是静态的。因而，这些函数不能操作类中的数据成员。

当编译器调用 operator new 时，其第一个参数被初始化为对象的大小。调用 operator new[] 则传递存储给定数目元素的大小。
当定义我们自己的 operator new 时，可以传递额外的参数。如果 new 表达式想要调用这种函数的话，就需要使用定位 new 的形式来传递额外的实参。定义的 operator new 一定不会是以下形式，如：

```
void *operator new(size_t, void*);
```

这个特别形式的函数原型是保留的给库使用，是不可以重复定义的。
调用 operator delete 时，编译器会传递待删除的对象的指针给这个函数。当将 operator delete 或 operator delete[] 定义为一个类成员时，函数可以有第一个参数为 size_t 类型，如果有的话，那么它将会被初始化为第一个参数所表示的对象的大小。真正调用的 delete 函数由被删除的对象的动态类型决定。
定义 operator new 和 operator delete 函数可以改变内存分配的方式，但是不能改变 new 和 delete 操作符的基本含义。

malloc 和 free 函数

malloc 和 free 是继承自 C 的库函数，我们的 operator new 和 operator delete 函数可以将底层工作交给这两个函数。其中 malloc 的参数是 size_t 乃是要分配的字节数，返回内存指针或者在失败时返回 0 。而 free 则以 malloc 的返回值作为参数，释放相关的内存，调用 free(0) 是合法的但是没有任何效果。

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept {
    free(mem);
}
```

19.1.2 定位（placement）new 表达式

operator new 和 operator delete 是库中的常规函数，意味着可以直接调用这些函数。在语言的早期版本，应用程序为了分离分配内存和初始化工作，会调用 operator new 和 operator delete，这与现在的 allocator 中的 allocate 和 deallocate 成员函数的效果是一致的，它们只负责分配和回收内存。
与 allocator 不同的，早期版本无法直接调用构造函数对内存进行对象构造，相反，我们必须使用定位 new 的方式进行构造对象。定位 new 将提供额外的地址信息，如：

```
new (place_address) type; (1)
new (place_address) type (initializers); (2)
new (place_address) type [size]; (3)
new (place_address) type [size] { braced initializer list } (4)
```

其中 place_address 就是希望在此处构建的内存地址。当我们使用以上形式时，调用的就是 operator new(size_t, void*) 的函数，将返回我们给出的指针实参，这个函数是编译器不允许我们重载的。定位 new 的工作就是在我们指定的地方进行对象初始化。
传递给 placement new 的指针可以不是之前 operator new 分配的内存的指针，甚至可以不是动态内存的指针。

显式调用析构函数

虽然不能直接调用构造函数，但是可以直接调用析构函数，与调用任何其它的成员函数一样的方式去调用析构函数。如：

```
string *sp = new string("a values");
sp->~string();
```

与调用 allocator.destroy 一样，析构函数将清理对象但是不会释放其内存，我们可以重用此内存。

19.2 运行时类型识别

运行时类型识别（Runtime type identification, RTTI）通过两个操作符提供：1. typeid 操作符返回给定表达式的类型；2. dynamic_cast 操作符将一个基类指针或者引用转为派生类的指针或引用；

在有些时候我们想通过基类指针或引用去调用派生类的函数是不可能的，原因在于我们无法定义这个虚函数。如果不能使用虚函数，我们可以使用 RTTI 操作符。另一方面，使用这些操作符将比使用虚成员函数更加易错：程序员必须知道对象的真实类型，并且必须检查转型是否成功。
RTTI 应该被限制使用在有限的范围内，更好的方式是定义虚函数而不是直接管理这些类型。

19.2.1 dynamic_cast 操作符

动态转型具有如下形式：

```
dynamic_cast<type*>(e) // (1)
dynamic_cast<type&>(e) // (2)
dynamic_cast<type&&>(e) // (3)
```

其中 type 必须是具有虚函数的类类型。(1) 中 e 必须是有效的指针；(2) 中 e 必须是左值；(3) 中 e 必须不是左值；
在以上所有情形中，e 必须是 type 的共有派生子类、共有基类或者与 type 一致的类型。只有这样才能转型成功，否则转型就会失败。如果动态转型为指针类型失败结果将是 0，如果动态转型为引用类型失败，结果将会抛出 bad_cast 异常。

指针类型动态转换

常用的形式如：

```
if (Derived *dp = dynamic_cast<Derived*>(bp))
{
} else {  // bp 指向一个真正的基类对象
}
```

如果 bp 指向一个派生对象，那么 dp 将会初始化为指向 bp 所指向的 Derived 类型的对象的指针，那么使用 Derived 的操作就是正常的。否则，dp 的值将是 0，此时 if 条件将会失败。
可以在空指针上执行 dynamic_cast ，其结果是一个空指针。

引用类型的动态转换

如果对引用类型进行 dynamic_cast 时，其错误处理方式将会不一样，因为这种方式的转换在无法完成时会抛出异常。如：

```
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
    } catch (bad_cast) {}
}
```

19.2.2 typeid 操作符

RTTI 提供第二个操作符是 typeid 操作符，其可以知道对象的类型是什么。typeid 表达式以 typeid(e) 的形式存在，其中 e 是任何表达式或者类型名字，结构是 type_info 库类或者其公有派生类的一个常量对象引用。type_info 类被定义在 typeinfo 头文件中。
typeid 操作符可以被运用于任何表达式，顶层 const 将会被忽略，如果表达式是引用那么 typeid 将返回其绑定的对象类型。当表达式是数组或函数时，其不会被转为指针，结果将是数组类型或函数类型。
如果操作数不是类类型或者是一个没有虚函数的类，那么结果是操作数的静态类型。当操作数是一个至少有一个虚函数的类类型左值，那么类型将在运行时被求值。

使用 typeid 操作符

通常使用 typeid 来比较两个表达式的类型或者将一个表达式的类型与特定的类型进行比较。如：

```
Derived *dp = new Derived;
Base *bp = dp;
if (typeid(*bp) == typeid(*dp)) {
// bp 和 dp 指向相同类型的对象
}
if (typeid(*bp) == typeid(Derived)) {
// bp 实际指向一个 Derived
}
```

注意这里 typeid 的参数不是指针本身，如果是指针的话将在编译期返回指针的静态类型。

仅当类型具有虚函数时，才需要 typeid 在运行时求得其动态类型，而此时将必须对表达式求值。如果类型没有虚函数，那么 typeid 将在编译期返回表达式的静态类型；编译器不需要对表达式求值就可以知道其静态类型。这使得 **typeid(*p)** 如果 p 所指向的类型没有虚函数的话，p 可以不是有效的指针（如空指针）。

19.2.3 使用 RTTI

使用 RTTI 的一个场景是定义 equal 函数，如果使用在基类中将 equal 函数定义为虚函数，那么其参数将不得不是基类类型，导致函数将无法使用派生类中的特有成员。而且 equal 函数应该是类型不同时返回 false。所以我们的 equal 函数定义如下：

```
class Base {
    friend bool operator==(const Base&, const Base&);
protected:
    virtual bool equal(const Base&) const;
};
class Derived : public Base {
protected:
    bool equal(const Base&) const;
};

// 相等操作符
bool operator==(const Base &lhs, const Base &rhs)
{
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
bool Derived::equal(const Base &rhs) const
{
    auto r = dynamic_cast<const Derived&>(rhs);
    // do the work to compare two Derived objects
}
bool Base::equal(const Base &rhs) const
{
    // do whatever is required to compare to Base objects
}
```

19.2.4 type_info 类

type_info 类的精确定义在不同的编译器实现之间有所不同，但是标准保证这个类定义在 typeinfo 头文件中，并且提供如下函数：

596. **t1 == t2** 当 t1 和 t2 是 **type_info** 类型对象且表示相同的类型时返回 true，否则返回 false；
597. **t1 != t2** 与上一条款相反；
598. **t.name()** 返回类型名字的可打印 C 风格字符串，类型名字是与系统相关的；
599. **t1.before(t2)** 当类型 t1 比 t2 出现的早时返回 true，顺序是编译器相关的；

这个类提供了一个虚析构函数，因为其将作为一个基类。当编译期希望提供额外的信息，它通常会在 **type_info** 的派生类中完成。**type_info** 类没有默认构造函数，拷贝和移动构造函数以及赋值操作符都被定义为被删除的。所以只能通过 typeid 来获得 **type_info** 对象，而没有别的方式可以得到。

type_info 的 name 成员是由编译器决定的，可能不会与程序中使用的名字相匹配，只保证每个类型的名字是唯一的。如：RTTI.cc

19.3 枚举

枚举（enumerations）可以将整数常量集合起来管理。与类一样，每个枚举定义一个新的类型。类是字面类型（literal types）。

C++ 支持两个枚举：带作用域的(scoped)和无作用域的(unscoped)。新标准引入了带作用域的枚举(scoped enumerations）。定义带作用域的枚举使用 enum class 或者 enum struct 关键字，后面跟着枚举名字和逗号分隔的一系列枚举值（enumerators）如：

```
enum class open_modes { input, output, append };
```

如果省略掉 class 或 struct 关键字就是无作用域枚举（unscoped enumeration）。在无作用域 enum 中，枚举名字是可选的。如：

```
enum color {red, yellow, green};
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

如果枚举是没有名字的，那么只能在枚举定义处定义其类型的对象，即在结束的括号处添加逗号分隔的声明列表。

枚举值（Enumerators）

在带作用域的枚举中的枚举值的名字需要按照正常的作用域规则进行，并且在枚举作用域之外是不可访问的。无作用域枚举的枚举值放在了与枚举自身相同的作用域中。如：

```
enum color {red, yellow, green};
enum stoplight {red, yellow, green}; // 错误：重复定义的枚举值
enum class peppers {red, yellow, green};

color eyes = green;
peppers p = green; // peppers 的枚举值不在作用域中
color hair = color::red; // 显式指定是可以的
peppers p2 = peppers::red;
```

默认情况下，枚举值从 0 开始，每个枚举值都比之前的那个大 1 ，我们可以给一个或多个枚举值提供初始值，如：

```
enum class intTypes {
    charType = 8, shortType = 16, intType = 16,
    longType = 32, long_longType = 64
};
```

枚举值的值不需要是唯一的，shortType 和 intType 的值就是一样的。如果省略初始值，则其值比之前的枚举值多 1 。枚举值是常量，如果进行初始化，初始值必须是常量表达式。结果就是枚举值自己也是常量表达式，因而可以用于需要常量表达式的场景。如：

```
constexpr intTypes charbits = intTypes::charType;
```

同样可以将枚举用于 switch 语句中，枚举值可以作为 case 标签。可以将枚举类型作为非类型模板参数（nontype template parameter），可以在类定义中初始化枚举类型的静态数据成员。

与类一样，枚举定义新的类型

只要枚举是由名字的，就可以定义和初始化这个类型的对象。枚举对象只能由其中一个枚举值或者相同枚举的另外一个对象初始化。如：

```
open_modes om = 2; // 错误：2 不是 open_modes 类型
om = open_modes::input;
```

无作用域的枚举的枚举值及其对象可以自动转为整型值，所以它们可以用于任何需要整数值的场景。如：

```
int i = color::red;
int j = peppers::red;
```

Specifying the Size of an enum

如果没有指定枚举值的类型，对于有作用域的枚举来说就是 int 类型，对于无作用域的枚举来说就是足够容纳所有的枚举值的。如果指定了枚举值的类型，那么超出范围将会编译失败。这种方式将保证程序的行为在跨系统时都是一致的。如：

```
enum intValues : unsigned long long {
    charType = 255,
    shortType = 65535,
    intType = 65535,
    longType = 4294967295UL,
    long_longType = 18446744073709551615ULL
};
```

枚举的前置声明

在新标准下，可以前置声明枚举。枚举的前置声明必须指定枚举值的类型（显式或隐式）。如：

```
enum intValues : unsigned long long; // 无作用域枚举，必须指定类型
enum class open_modes; // 带作用域的枚举默认是 int 类型
```

所有的枚举定义和声明都必须是完全一致的（枚举值的类型），特别是不能在一个上下文中声明为无作用域的枚举，在另外一个地方声明为带作用域的枚举。

参数匹配和枚举

接收枚举的函数不能使用具有相同的值的整型值。如：

```
enum Tokens {INLINE = 128, VIRTUAL = 129};
void ff(Tokens);
void ff(int);
int main() {
    Tokens curTok = INLINE;
    ff(128); // ff(int)
    ff(INLINE); // ff(Tokens)
    ff(curTok); // ff(Tokens)
    return 0;
}
```

但是可以将枚举对象传递给接收整形值的函数，枚举将提升为 int 或者 long 等类型。如:

```
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL); // newf(int)
newf(uc); // newf(unsigned char)
```

19.4 类成员指针（Pointer to Class Member）

指向成员的指针（pointer to member）是一种可以指向类的非静态成员的指针。通常指针可以指向一个对象，但是指向成员的指针表示一个类的成员。静态成员则可以使用常规的指针来操作。

成员指针的类型同时具有类和类的成员的类型。只能将这种指针初始化为类的特定成员，而不指定这个成员属于哪个对象。当我们使用成员指针时，才提供在之上操作的对象。如：class_member_pointer.cc。

19.4.1 指向数据成员的指针

定义成员指针必须提供类的名字，如：`const string Screen::*pdata;` 定义 pdata 是“Screen 类的成员指针，其类型是 const string”。初始化的过程如：`pdata = &Screen::contents;` 此处将取地址符作用域类 Screen 的成员，而不是一个内存中的对象。

使用数据成员指针

理解当初始化或者赋值成员指针时，其并没有指向任何数据。它表示一个特定的成员，但没有成员所在的对象的信息。当我们解引用成员指针时需要提供对象。成员指针访问符有两个：`.*` 和 `->*` 。如：

```
Screen myScreen, *pScreen = &myScreen;
auto s = myScreen.*pdata;
s = pScreen->*pdata;
```

函数返回数据成员指针

对成员指针一样运用常规的访问控制，private 的成员只能在类成员内部或者友元中使用。由于数据成员绝大部分都是私有的，所以应该定义一个共有成员函数返回数据成员指针。如：

```
class Screen {
public:
    static const std::string Screen::*data()
    {
        return &Screen::contents;
    }
};
```

当调用时返回一个成员指针，如：

```
const string Screen::*pdata = Screen::data();
```

pdata 仅仅只是指向类 Screen 的一个成员而不是真正的数据。为了使用 pdata，必须将一个对象绑定到成员指针上。如：

```
auto s = myScreen.*pdata;
```

19.4.2 指向成员函数的指针

成员函数指针的定义要加上 classname::* 以及正常的函数指针的说明（返回类型与参数列表），如果成员函数是 const 成员或引用成员，必须在成员指针上体现出来。如：

```
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;
```

Screen::*pmf2 外围的括号是必须的，问题出现在优先级上。没有括号，将被编译器认为是无效的函数声明。

与常规的函数指针不同的是，成员函数和成员指针之间没有直接转换。如：pmf = Screen::get; 就是错误的。

使用成员函数指针

与使用数据成员一样，使用 .* 或 ->* 操作符来通过成员函数指针调用成员函数。如：

```
Screen myScreen, *pScreen = &myScreen;
char c1 = (pScreen->*pmf)();
char c2 = (myScreen.*pmf2)(0, 0);
```

这里使用括号的原因在函数调用符的优先级比成员指针访问符的优先级要高。

由于函数调用的优先级较高，所以声明和调用成员函数指针都需要加上括号，(C::*p)(params) 和 (obj.*p)(args)。

成员指针的类型别名

成员指针的类型别名如：

```
using Action =
char (Screen::*)(Screen::pos, Screen::pos) const;

Action get = &Screen::get;
```

此处 Action 是“指向类 Screen 的成员函数指针，此成员函数是接收两个 pos 类型的参数返回 char 类型返回值的 const 成员”的别名。

与别的函数指针一样，可以将成员函数指针类型作为返回类型或者参数类型，并且这种类型的参数可以有默认实参，如：

```
Screen& action(Screen&, Action = &Screen::get);
```

成员函数指针的表（Tables）

参考：function_table.cc

19.4.3 将成员函数用作可调用对象

与常规的函数指针不一样的是，成员指针并不是一个可调用对象，这种指针并不支持函数调用操作符。这样就不能将其传递给算法函数了。如：

```
auto fp = &string::empty;
// 错误：.* 或 ->* 才能调用成员函数指针
find_if(svec.begin(), svec.end(), fp);
```

find_if 期待一个可调用对象，但是 fp 并不是。

使用函数来产生可调用对象

一种从成员函数指针中获取可调用对象的方式是使用 function 模板。如：

```
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);
```

通常，成员函数所在的对象是通过隐式参数 this 传递的，当用 function 来为成员函数产生可调用对象时，this 隐式参数转为显式参数。当 function 对象包含一个成员函数指针时，它会使用成员指针访问符（.* 和 ->*）来对传入的对象进行成员函数指针调用。

当定义 functin 对象时，必须指定函数的签名，这个签名的第一个参数必须是成员函数所在的对象的类型（在之上函数将会执行），并且必须指出所在的对象类型是指针还是引用。如下面就将所在对象定义为了指针：

```
vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
find_if(pvec.begin(), pvec.end(), fp);
```

使用 mem_fn 产生可调用对象

mem_fn 可以在不提供函数签名的情况下生成一个可调用对象，这个函数也定义在 functional 头文件中。如：

```
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
```

由 mem_fn 生成的可调用对象可以在指针或对象上调用，而不必显式指出来，如：

```
auto f = mem_fn(&string::empty);
f(*svec.begin()); // 使用 obj.*f()
f(&svec[0]); // 使用 ptr->*f()
```

可以认为 mem_fn 生成了一个重载了的调用操作符的可调用对象，其中一个以指针为参数，另一个以对象引用为参数。

使用 bind 生成可调用对象

bind 也可以生成一个可调用对象，如：

```
bind(&string::empty, _1)
```

与 mem_fn 一样，不需要指定所在的对象是指针还是引用，但需要显式使用占位符告知所在对象在第一个参数的位置。

19.5 嵌套类

一个类可以定义在另外一个类中，这样的类成为嵌套类(nested class)，或者叫嵌套类型(nested type)。嵌套类最常用于定义实现类。

嵌套类与其外围类是没有关系的，嵌套类型的对象没有外围类中定义的成员，反之亦然。

嵌套类的名字在外围类是可见的，但是外部就不可见了（如果处于外围类的 private 控制下，在 public 的控制下依然是可见的）。

外围类对于嵌套类没有特殊的访问权限，嵌套类对于外围类也没有特殊的访问权限。嵌套类在外围类中定义一个类型成员（type member）。定义在 public 部分中可以被用于任何地方，定义 protected 中则只能被外围类自身、友元和派生类使用，定义在 private 中则只能被外围类自身和友元访问。

在外围类外部定义嵌套类

嵌套类必须在外围类的内部声明，但是定义可以放在外围类的外部。当在外围类的外部定义嵌套类时，必须同时用外围类名和嵌套类名进行限定。如：

```
class TextQuery {
public:
    class QueryResult;
};
class TextQuery::QueryResult {
    friend std::ostream&
        print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string,
        std::shared_ptr<std::set<line_no>>,
        std::shared_ptr<std::vector<std::string>>);
};
TextQuery::QueryResult::QueryResult(string s,
        shared_ptr<set<line_no>> p,
        shared_ptr<vector<string>> f):
    sought(s), lines(p), file(f) { }
```

这里 QueryResult 的构造函数也不是定义在类体内，必须将构造函数用外围类和嵌套类名进行限定。

嵌套类的静态成员定义

在类外定义如下：

```
int TextQuery::QueryResult::static_mem = 1024;
```

嵌套类作用域中的名称查找

正常的名称查找规则运用于嵌套类。当然，嵌套类由一个额外的外围类作用域可供搜索。嵌套类是外围类的一个类型成员，外围类的成员可以不加限制的使用嵌套类的名字。参考代码：TextQuery.cc/TextQuery.h

返回值类型需要加以限定 TextQuery::QueryResult ，在函数体内则可以直接引用 QueryResult。

嵌套和外围类是独立的

尽管嵌套类定义在外围类的内部，必须理解的是嵌套类对象与外围类对象之间没有必然的联系。嵌套类对象只包含它自己定义的成员，外围类对象也只包含它自己定义的成员，它不能直接访问嵌套类中的数据成员。

19.6 union：空间节约型的类

union 是一种特殊类型的类。union 可以由多个数据成员，但是在任何一个时间点，只有其中之一的成员是有值的。当 union 的一个成员被赋予值之后，其它所有成员都将是相同的底层二进制，至于如何对这些二进制进行解释则有其它成员本身的类型决定。union 需要的内存大小由最大的数据成员决定，内存大小将足够容纳这个数据类型。

与类一样，union 定义一种新的类型。

union 中的数据成员不能是引用，在 C++ 的早期版本中，数据成员的类型只能是内置类型，现在在新版本的 C++ 中可以是有构造函数和析构函数的类类型。

union 可以设置访问权限标签 public 、private 和 protected，默认情况下 union 是 public 的，这与 struct 是一样的。

union 可以定义成员函数，包括构造函数和析构函数，但是 union 不能继承别的类，也不能作为基类，union 也不能有虚函数。

定义 union

union 的定义以 union 关键字开始，后跟随可选的名字，以及一系列在括号中的成员声明。如：

```
union Token {
    char cval;
    int ival;
    double dval;
};
```

使用 union 类型

与内置类型一样，默认情况下 union 是不初始化的。用初始化聚合类（aggregate class）一样的方法来初始化 union，如：`Token token = {'a'};` 其中 a 用来初始化第一个成员 cval。union 的成员通过常规的成员访问符进行访问，如：

```
last_token.cval = 'z';
pt->ival = 42;
```

赋值给 union 对象的数据成员将导致别的数据成员的内容是未定义的。

匿名 union

匿名 union 是没有名字也没有定义对象的 union，当定义匿名 union 时，编译器自动创建一个匿名对象。如：

```
// Defines an unnamed object, whose members we can access directly
union {
    char cval;
    int ival;
    double dval;
};
cval = 'c';
ival = 42;
```

匿名 union 的成员可以在定义这个匿名 union 的作用域中直接访问。匿名 union 不能有 private 或者 protected 成员，也不能定义成员函数。

union 中有类类型成员

新标准中允许定义有构造函数和拷贝控制成员的类类型成员，但是使用这种 union 将比只有内置类型成员的 union 要更加复杂。只有内置类型成员的 union 只需要简单的赋值就可以替换其成员的值，对于类类型成员则需要显式地构造和析构了。

当 union 只有内置类型成员时，编译器可以合成默认构造函数或拷贝控制成员，而如果 union 内有类类型成员，并且其中有类类型成员定义了自己的默认构造函数或拷贝控制成员，union 合成的对应的成员就是被删除的。如果一个类的成员 union ，并且这个 union 有拷贝控制成员是被删除的，那么此类对应的拷贝控制成员也是被删除的。

使用类来管理 union 的成员

如果 union 中有类类型的成员，其复杂度会变得很高，所以一般倾向于将其放在另外一个类中。这样就由这个类来管理 union 的状态转换。代码见：union_class.cc

19.7 本地类

类可以定义在函数体内，这种类被称为本地类（local class）。本地类定义了一个只能在其被定义的作用域中可见的类型，与嵌套类不同在于，本地类的成员十分受限。

本地类的所有成员（包括函数）都必须在类体内定义。本地类的函数代码长度通常会很短，太长则会使得代码难以理解。同样本地类不允许定义静态数据成员。

本地类不能使用函数作用域中的本地变量

本地类可访问的外围作用域中的名字是受限的，本地类只能访问外围作用域中的类型名字、静态变量和枚举值，不能访问外围函数中定义的常规本地变量。如：local_class.cc 所示。

本地类运用正常的保护规则

外围函数无权访问本地类的私有成员，当然，本地类可以将外围函数设置为友元，而本地类更常见的做法是将其成员设置为 public 的，可以访问本地类的程序部分是是否受限的，只能在外围函数中访问。本地类本身已经被封装在了函数的作用域中，如果再通过信息隐藏去封装就是多此一举了。

本地类中的名称查找

发生在本地类体中的名称查找与别的类中的名称查找没什么两样。成员声明中使用到的名字必须是之前出现过的，定义成员则无先后关系，这是由类的两步处理导致的结果。如果名字无法在类中找到，那么就会继续查找外围函数和外围函数的外部作用域。

嵌套本地类

可以在本地类中再嵌套类。这个嵌套类可以在本地类体的外部进行定义，但是嵌套类必须与本地类在同一个作用域中定义，意味着必须定义在同一个函数中。如：

```
void foo()
{
    class Bar {
    public:
        class Nested;
    };
    class Bar::Nested {
    };
}
```

本地类的嵌套类也是本地类，遵循本地类的所有规则。

19.8 固有的不可移植特性

为了支持底层编程（low-level programming），C++ 定义了一些固有不可移植的特性（nonportable features）。不可移植的特性是特定于机器的（machine specific），使用了不可移植的特性通常需要在换了平台时重新对这部分进行编程。其中算术类型的长度在不同机器之间不一样就是一个不可移植的特性。下面将描述从 C 继承来的不可移植特性：位域和 volatile 限定符，以及连接指令 `extern "C"`。

19.8.1 位域（bit-fields）

类可以将数据成员定义为位域（bit-field），位域包含特定的位数（number of bits），它们通常用于传递二进制数据给另外一个程序或者给硬件设施。位域的内存布局是特定于机器的。

位域必须是整数类型或者枚举类型。通常使用的类型是 unsigned 类型，这是由于 signed 位域的行为是由实现决定的。通过在成员名后加冒号和常量表达式指定位数。如：

```
typedef unsigned int Bit;
class File {
    Bit mode : 2;
    Bit modified : 1;
    Bit prot_owner : 3;
    Bit prot_group : 3;
    Bit prot_world : 3;
};
```

位域可能被会打包到一个整数值中去以压缩存储，至于是如何实现的标准并没有规定。地址操作符（&）不能用于位域字段，所以位域字段是没有指针的。

使用位域

位域的访问与普通的成员是一样的。多于一个 bit 位的位域通常使用内置位操作符进行操作。

19.8.2 volatile 限定符

volatile 并不是一个有具体语义的关键字，而是由编译器实现决定的。硬件编程通常会有数据成员的值是外部进程决定的，如：变量是系统时钟。当对象的值会被程序外部改变时，应该将其声明为 volatile 的。volatile 指示编译器不要对这种对象进行优化。

使用 volatile 关键字与 const 限定符是一样的。如：

```
volatile int display_register;
volatile Task *curr_task;
volatile int iax[max_size];
```

volatile 与 const 之间没有任何交互，所以可以同时定义 volatile 和 const 而不相互影响。

类可以定义 volatile 成员函数，volatile 成员函数只能在 volatile 对象上调用。volatile 与指针的相互作用和 const 与指针之间的相互作用是一样的，即指针本身是 volatile 或者指向 volatile 对象的对象，或者两者都是 volatile 的。如：

```
volatile int v;
int *volatile vip;
volatile int *ivp;
volatile int *volatile vivp;
```

与 const 一样，只能将 volatile 对象的地址赋值给指向 volatile 的指针，只能将 volatile 对象用于初始化 volatile 引用。

合成拷贝不会被运用于 volatile 对象

const 与 volatile 的一个最大的不同在于合成的拷贝控制成员不能以 volatile 对象为源，即不可以将其作为初始值或者赋值的右边操作数。如果想要这么做，必须手动定义对应的操作函数。如：

```
class Foo {
public:
    Foo(const volatile Foo&);
    Foo& operator=(volatile const Foo&);
    Foo& operator=(volatile const Foo&) volatile;
};
```

至于是否有必要这么做，决定于应用程序需要解决的问题。

19.8.3 链接指令：extern "C"

C++ 有时会调用 C 语言中书写的函数，想要这么做必须首先声明这些函数。由于 C++ 中的函数与 C 的函数在二进制文件中的符号是不一样的。所以得使用链接指令（linkage directives）来告知 C++ 这个函数是用不同的语言写成的。

声明一个非 C++ 函数

链接指令有两种形式：单行和复合形式。链接指令不能出现在类和函数定义中。相同的连接指令必须出现在一个函数的所有声明处。如：

```
extern "C" size_t strlen(const char*);
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

第一种形式就是 extern 关键字后跟一个字符串，后跟一个普通的函数声明。第二种形式则将所有的函数声明放在一个括弧中。

还有一种用法是将 #include 放在链接指令中，如：

```
extern "C" {
    #include <string.h>
}
```

这样 string.h 头文件中所有函数都被认为是 C 语言中的函数。链接指令是可以嵌套的，如果头文件中包含了一个函数有自己的链接指令，那个函数的链接将不会受到影响。

指向 extern “C” 函数的指针

函数的定义语言是函数类型的一部分，所以函数指针也要告知链接指令。如：

```
extern "C" void (*pf)(int);
```

以上 pf 被认为是一个 C 函数的指针。C 函数的指针与 C++ 函数的指针是不同的类型，所以不能将 C++ 函数用于初始化 C 函数指针（反之亦然）。如：

```
void (*pf1)(int);
extern "C" void (*pf2)(int);
pf1 = pf2; // 错误：pf1 和 pf2 是不同的类型
```

链接指令被运用于整个整个声明，如：

```
extern "C" void f1(void(*)(int));
```

如上语句中，f1 和指针都是 C 函数。如果仅仅指向然指针是 C 函数则需要用到类型别名，如：

```
extern "C" typedef void FC(int);
void f2(FC *);
```

将 C++ 函数导出给其它函数

通过在函数定义上运用链接指令，可以将 C++ 函数导出给 C 语言。如：

```
extern "C" double calc(double dparam) { /* ... */ }
```

当编译器翻译代码时，它会将生成 C 语言的代码。

需要注意的是跨语言的参数和返回值类型是受限的，比如不能传递 nontrivial C++ 类给 C 语言的程序，C 语言不知道构造函数、析构函数和其它与类相关的操作。

预处理器的支持

C++ 编译器定义了宏 __cplusplus ，所以可以用如下方式来加入链接指令。如：

```
#ifdef __cplusplus
extern "C"
#endif
int strcmp(const char*, const char*);
```

重载函数和链接指令

C 语言不支持函数重载，所以只能将重载集合中的一个函数暴露给 C 语言。如：

```
// 错误：将两个同名函数暴露给了 C 语言
extern "C" void print(const char*);
extern "C" void print(int);
```

如果重载函数集中有一个是 C 函数，那么其他所有函数必须是 C++ 函数。如：

```
class SmallInt {};
class BigNum {};
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

C 版本的 calc 可以在 C 和 C++ 中调用，而其它的函数只能在 C++ 中调用，声明的顺序是不重要的。