

263> 这时我们不禁要问：`total` 和 `trans` 是如何初始化的呢？

我们没有为这些对象提供初始值，因此我们知道它们执行了默认初始化（参见 2.2.1 节，第 40 页）。类通过一个特殊的构造函数来控制默认初始化过程，这个函数叫做**默认构造函数**（**default constructor**）。默认构造函数无须任何实参。

如我们所见，默认构造函数在很多方面都有其特殊性。其中之一是，如果我们的类没有显式地定义构造函数，那么编译器就会为我们隐式地定义一个默认构造函数。

编译器创建的构造函数又被称为**合成的默认构造函数**（**synthesized default constructor**）。对于大多数类来说，这个合成的默认构造函数将按照如下规则初始化类的数据成员：

- 如果存在类内的初始值（参见 2.6.1 节，第 64 页），用它来初始化成员。
- 否则，默认初始化（参见 2.2.1 节，第 40 页）该成员。

因为 `Sales_data` 为 `units_sold` 和 `revenue` 提供了初始值，所以合成的默认构造函数将使用这些值来初始化对应的成员；同时，它把 `bookNo` 默认初始化成一个空字符串。

某些类不能依赖于合成的默认构造函数

合成的默认构造函数只适合非常简单的类，比如现在定义的这个 `Sales_data` 版本。对于一个普通的类来说，必须定义它自己的默认构造函数，原因有三：第一个原因也是最容易理解的一个原因就是编译器只有在发现类不包含任何构造函数的情况下才会替我们生成一个默认的构造函数。一旦我们定义了一些其他的构造函数，那么除非我们再定义一个默认的构造函数，否则类将没有默认构造函数。这条规则的依据是，如果一个类在某种情况下需要控制对象初始化，那么该类很可能在所有情况下都需要控制。



只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数。

第二个原因是对于某些类来说，合成的默认构造函数可能执行错误的操作。回忆我们之前介绍过的，如果定义在块中的内置类型或复合类型（比如数组和指针）的对象被默认初始化（参见 2.2.1 节，第 40 页），则它们的值将是未定义的。该准则同样适用于默认初始化的内置类型成员。因此，含有内置类型或复合类型成员的类应该在类的内部初始化这些成员，或者定义一个自己的默认构造函数。否则，用户在创建类的对象时就可能得到未定义的值。



如果类包含有内置类型或者复合类型的成员，则只有当这些成员全都被赋予了类内的初始值时，这个类才适合于使用合成的默认构造函数。

264> 第三个原因是有的时候编译器不能为某些类合成默认的构造函数。例如，如果类中包含一个其他类类型的成员且这个成员的类型没有默认构造函数，那么编译器将无法初始化该成员。对于这样的类来说，我们必须自定义默认构造函数，否则该类将没有可用的默认构造函数。在 13.1.6 节（第 449 页）中我们将看到还有其他一些情况也会导致编译器无法生成一个正确的默认构造函数。

定义 `Sales_data` 的构造函数

对于我们的 `Sales_data` 类来说，我们将使用下面的参数定义 4 个不同的构造函数：

- 一个 `istream&`，从中读取一条交易信息。

- 一个 `const string&`, 表示 ISBN 编号; 一个 `unsigned`, 表示售出的图书数量; 以及一个 `double`, 表示图书的售出价格。
- 一个 `const string&`, 表示 ISBN 编号; 编译器将赋予其他成员默认值。
- 一个空参数列表 (即默认构造函数), 正如刚刚介绍的, 既然我们已经定义了其他构造函数, 那么也必须定义一个默认构造函数。

给类添加了这些成员之后, 将得到

```
struct Sales_data {
    // 新增的构造函数
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(std::istream &); // 之前已有的其他成员
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

= default 的含义

我们从解释默认构造函数的含义开始:

```
Sales_data() = default;
```

首先请明确一点: 因为该构造函数不接受任何实参, 所以它是一个默认构造函数。我们定义这个构造函数的目的仅仅是因为我们既需要其他形式的构造函数, 也需要默认的构造函数。我们希望这个函数的作用完全等同于之前使用的合成默认构造函数。

在 C++11 新标准中, 如果我们需要默认的行为, 那么可以通过在参数列表后面写上 **= default** 来要求编译器生成构造函数。其中, `= default` 既可以和声明一起出现在类的内部, 也可以作为定义出现在类的外部。和其他函数一样, 如果`= default` 在类的内部, 则默认构造函数是内联的; 如果它在类的外部, 则该成员默认情况下不是内联的。



上面的默认构造函数之所以对 `Sales_data` 有效, 是因为我们为内置类型的数据成员提供了初始值。如果你的编译器不支持类内初始值, 那么你的默认构造函数就应该使用构造函数初始值列表 (马上就会介绍) 来初始化类的每个成员。

构造函数初始值列表

接下来我们介绍类中定义的另外两个构造函数:

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

这两个定义中出现了新的部分, 即冒号以及冒号和花括号之间的代码, 其中花括号定义了

(空的) 函数体。我们把新出现的部分称为构造函数初始值列表 (constructor initialize list)，它负责为新创建的对象的一个或几个数据成员赋初值。构造函数初始值是成员名字的一个列表，每个名字后面紧跟括号括起来的（或者在花括号内的）成员初始值。不同成员的初始化通过逗号分隔开来。

含有三个参数的构造函数分别使用它的前两个参数初始化成员 bookNo 和 units_sold，revenue 的初始值则通过将售出图书总数和每本书单价相乘计算得到。

只有一个 string 类型参数的构造函数使用这个 string 对象初始化 bookNo，对于 units_sold 和 revenue 则没有显式地初始化。当某个数据成员被构造函数初始值列表忽略时，它将以与合成默认构造函数相同的方式隐式初始化。在此例中，这样的成员使用类内初始值初始化，因此只接受一个 string 参数的构造函数等价于

```
// 与上面定义的那个构造函数效果相同
Sales_data(const std::string &s):
    bookNo(s), units_sold(0), revenue(0) { }
```

通常情况下，构造函数使用类内初始值不失为一种好的选择，因为只要这样的初始值存在我们就能确保为成员赋予了一个正确的值。不过，如果你的编译器不支持类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员。

Best Practices

构造函数不应该轻易覆盖掉类内的初始值，除非新赋的值与原值不同。如果你不能使用类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员。

266 有一点需要注意，在上面的两个构造函数中函数体都是空的。这是因为这些构造函数的唯一目的就是为数据成员赋初值，一旦没有其他任务需要执行，函数体也就为空了。

在类的外部定义构造函数

与其他几个构造函数不同，以 istream 为参数的构造函数需要执行一些实际的操作。在它的函数体内，调用了 read 函数以给数据成员赋以初值：

```
Sales_data::Sales_data(std::istream &is)
{
    read(is, *this); // read 函数的作用是从 is 中读取一条交易信息然后
                      // 存入 this 对象中
}
```

构造函数没有返回类型，所以上述定义从我们指定的函数名字开始。和其他成员函数一样，当我们在类的外部定义构造函数时，必须指明该构造函数是哪个类的成员。因此，Sales_data::Sales_data 的含义是我们定义 Sales_data 类的成员，它的名字是 Sales_data。又因为该成员的名字和类名相同，所以它是一个构造函数。

这个构造函数没有构造函数初始值列表，或者讲得更准确一点，它的构造函数初始值列表是空的。尽管构造函数初始值列表是空的，但是由于执行了构造函数体，所以对象的成员仍然能被初始化。

没有出现在构造函数初始值列表中的成员将通过相应的类内初始值（如果存在的话）初始化，或者执行默认初始化。对于 Sales_data 来说，这意味着一旦函数开始执行，则 bookNo 将被初始化成空 string 对象，而 units_sold 和 revenue 将是 0。

为了更好地理解调用函数 `read` 的意义，要特别注意 `read` 的第二个参数是一个 `Sales_data` 对象的引用。在 7.1.2 节（第 232 页）中曾经提到过，使用 `this` 来把对象当成一个整体访问，而非直接访问对象的某个成员。因此在此例中，我们使用`*this` 将“`this`”对象作为实参传递给 `read` 函数。

7.1.4 节练习

练习 7.11：在你的 `Sales_data` 类中添加构造函数，然后编写一段程序令其用到每个构造函数。

练习 7.12：把只接受一个 `istream` 作为参数的构造函数定义移到类的内部。

练习 7.13：使用 `istream` 构造函数重写第 229 页的程序。

练习 7.14：编写一个构造函数，令其用我们提供的类内初始值显式地初始化成员。

练习 7.15：为你的 `Person` 类添加正确的构造函数。

7.1.5 拷贝、赋值和析构



<267

除了定义类的对象如何初始化之外，类还需要控制拷贝、赋值和销毁对象时发生的行为。对象在几种情况下会被拷贝，如我们初始化变量以及以值的方式传递或返回一个对象等（参见 6.2.1 节，第 187 页和 6.3.2 节，第 200 页）。当我们使用了赋值运算符（参见 4.4 节，第 129 页）时会发生对象的赋值操作。当对象不再存在时执行销毁的操作，比如一个局部对象会在创建它的块结束时被销毁（参见 6.1.1 节，第 184 页），当 `vector` 对象（或者数组）销毁时存储在其中的对象也会被销毁。

如果我们不主动定义这些操作，则编译器将替我们合成它们。一般来说，编译器生成的版本将对对象的每个成员执行拷贝、赋值和销毁操作。例如在 7.1.1 节（第 229 页）的书店程序中，当编译器执行如下赋值语句时，

```
total = trans; // 处理下一本书的信息
```

它的行为与下面的代码相同

```
// Sales_data 的默认赋值操作等价于：  
total.bookNo = trans.bookNo;  
total.units_sold = trans.units_sold;  
total.revenue = trans.revenue;
```

我们将在第 13 章中介绍如何自定义上述操作。



某些类不能依赖于合成的版本

尽管编译器能替我们合成拷贝、赋值和销毁的操作，但是必须要清楚的一点是，对于某些类来说合成的版本无法正常工作。特别是，当类需要分配类对象之外的资源时，合成的版本常常会失效。举个例子，第 12 章将介绍 C++ 程序是如何分配和管理动态内存的。而在 13.1.4 节（第 447 页）我们将会看到，管理动态内存的类通常不能依赖于上述操作的合成版本。

不过值得注意的是，很多需要动态内存的类能（而且应该）使用 `vector` 对象或者 `string` 对象管理必要的存储空间。使用 `vector` 或者 `string` 的类能避免分配和释放内存带来的复杂性。

进一步讲，如果类包含 `vector` 或者 `string` 成员，则其拷贝、赋值和销毁的合成版本能够正常工作。当我们对含有 `vector` 成员的对象执行拷贝或者赋值操作时，`vector` 类会设法拷贝或者赋值成员中的元素。当这样的对象被销毁时，将销毁 `vector` 对象，也就是依次销毁 `vector` 中的每一个元素。这一点与 `string` 是非常类似的。



在学习第 13 章关于如何自定义操作的知识之前，类中所有分配的资源都应该直接以类的数据成员的形式存储。



7.2 访问控制与封装

268

到目前为止，我们已经为类定义了接口，但并没有任何机制强制用户使用这些接口。我们的类还没有封装，也就是说，用户可以直达 `Sales_data` 对象的内部并且控制它的具体实现细节。在 C++ 语言中，我们使用访问说明符（access specifiers）加强类的封装性：

- 定义在 `public` 说明符之后的成员在整个程序内可被访问，`public` 成员定义类的接口。
- 定义在 `private` 说明符之后的成员可以被类的成员函数访问，但是不能被使用该类的代码访问，`private` 部分封装了（即隐藏了）类的实现细节。

再一次定义 `Sales_data` 类，其新形式如下所示：

```
class Sales_data {
public:           // 添加了访问说明符
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&); 
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:          // 添加了访问说明符
    double avg_price() const
    { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

作为接口的一部分，构造函数和部分成员函数（即 `isbn` 和 `combine`）紧跟在 `public` 说明符之后；而数据成员和作为实现部分的函数则跟在 `private` 说明符后面。

一个类可以包含 0 个或多个访问说明符，而且对于某个访问说明符能出现多少次也没有严格限定。每个访问说明符指定了接下来的成员的访问级别，其有效范围直到出现下一个访问说明符或者到达类的结尾处为止。

使用 `class` 或 `struct` 关键字

在上面的定义中我们还做了一个微妙的变化，我们使用了 `class` 关键字而非 `struct` 开始类的定义。这种变化仅仅是形式上有所不同，实际上我们可以使用这两个关键字中的任何一个定义类。唯一的一点区别是，`struct` 和 `class` 的默认访问权限不太一样。

类可以在它的第一个访问说明符之前定义成员，对这种成员的访问权限依赖于类定义

的方式。如果我们使用 `struct` 关键字，则定义在第一个访问说明符之前的成员是 `public` 的；相反，如果我们使用 `class` 关键字，则这些成员是 `private` 的。

出于统一编程风格的考虑，当我们希望定义的类的所有成员是 `public` 的时，使用 `<269> struct`；反之，如果希望成员是 `private` 的，使用 `class`。



使用 `class` 和 `struct` 定义类唯一的区别就是默认的访问权限。

7.2 节练习

练习 7.16：在类的定义中对于访问说明符出现的位置和次数有限定吗？如果有，是什么？什么样的成员应该定义在 `public` 说明符之后？什么样的成员应该定义在 `private` 说明符之后？

练习 7.17：使用 `class` 和 `struct` 时有区别吗？如果有，是什么？

练习 7.18：封装是何含义？它有什么用处？

练习 7.19：在你的 `Person` 类中，你将把哪些成员声明成 `public` 的？哪些声明成 `private` 的？解释你这样做的原因。

7.2.1 友元



既然 `Sales_data` 的数据成员是 `private` 的，我们的 `read`、`print` 和 `add` 函数也就无法正常编译了，这是因为尽管这几个函数是类的接口的一部分，但它们不是类的成员。

类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数成为它的友元（friend）。如果类想把一个函数作为它的友元，只需要增加一条以 `friend` 关键字开始的函数声明语句即可：

```
class Sales_data {  
    // 为 Sales_data 的非成员函数所做的友元声明  
    friend Sales_data add(const Sales_data&, const Sales_data&);  
    friend std::istream &read(std::istream&, Sales_data&);  
    friend std::ostream &print(std::ostream&, const Sales_data&);  
    // 其他成员及访问说明符与之前一致  
  
public:  
    Sales_data() = default;  
    Sales_data(const std::string &s, unsigned n, double p):  
        bookNo(s), units_sold(n), revenue(p*n) {}  
    Sales_data(const std::string &s): bookNo(s) {}  
    Sales_data(std::istream&);  
    std::string isbn() const { return bookNo; }  
    Sales_data &combine(const Sales_data&);  
  
private:  
    std::string bookNo;  
    unsigned units_sold = 0;  
    double revenue = 0.0;  
};  
// Sales_data 接口的非成员组成部分的声明  
Sales_data add(const Sales_data&, const Sales_data&);
```

```
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
```

友元声明只能出现在类定义的内部，但是在类内出现的具体位置不限。友元不是类的成员也不受它所在区域访问控制级别的约束。我们将在 7.3.4 节（第 250 页）介绍更多关于友元的知识。



一般来说，最好在类定义开始或结束前的位置集中声明友元。

关键概念：封装的益处

封装有两个重要的优点：

- 确保用户代码不会无意间破坏封装对象的状态。
- 被封装的类的具体实现细节可以随时改变，而无须调整用户级别的代码。

一旦把数据成员定义成 `private` 的，类的作者就可以比较自由地修改数据了。当实现部分改变时，我们只需要检查类的代码本身以确认这次改变有什么影响；换句话说，只要类的接口不变，用户代码就无须改变。如果数据是 `public` 的，则所有使用了原来数据成员的代码都可能失效，这时我们必须定位并重写所有依赖于老版本实现的代码，之后才能重新使用该程序。

把数据成员的访问权限设成 `private` 还有另外一个好处，这么做能防止由于用户的原因造成数据被破坏。如果我们发现有程序缺陷破坏了对象的状态，则可以在有限的范围内定位缺陷：因为只有实现部分的代码可能产生这样的错误。因此，将查错限制在有限范围内将能极大地降低维护代码及修正程序错误的难度。



尽管当类的定义发生改变时无须更改用户代码，但是使用了该类的源文件必须重新编译。



友元的声明

友元的声明仅仅指定了访问的权限，而非一个通常意义上的函数声明。如果我们希望类的用户能够调用某个友元函数，那么我们就必须在友元声明之外再专门对函数进行一次声明。

为了使友元对类的用户可见，我们通常把友元的声明与类本身放置在同一个头文件中（类的外部）。因此，我们的 `Sales_data` 头文件应该为 `read`、`print` 和 `add` 提供独立的声明（除了类内部的友元声明之外）。



许多编译器并未强制限定友元函数必须在使用之前在类的外部声明。

271

一些编译器允许在尚无友元函数的初始声明的情况下就调用它。不过即使你的编译器支持这种行为，最好还是提供一个独立的函数声明。这样即使你更换了一个有这种强制要求的编译器，也不必改变代码。

7.2.1 节练习

练习 7.20: 友元在什么时候有用? 请分别列举出使用友元的利弊。

练习 7.21: 修改你的 Sales_data 类使其隐藏实现的细节。你之前编写的关于 Sales_data 操作的程序应该继续使用, 借助类的新定义重新编译该程序, 确保其工作正常。

练习 7.22: 修改你的 Person 类使其隐藏实现的细节。

7.3 类的其他特性

虽然 Sales_data 类非常简单, 但是通过它我们已经了解 C++语言中关于类的许多语法要点。在本节中, 我们将继续介绍 Sales_data 没有体现出来的一些类的特性。这些特性包括: 类型成员、类的成员的类内初始值、可变数据成员、内联成员函数、从成员函数返回 `*this`、关于如何定义并使用类类型及友元类的更多知识。

7.3.1 类成员再探

为了展示这些新的特性, 我们需要定义一对相互关联的类, 它们分别是 Screen 和 Window_mgr。

定义一个类型成员

Screen 表示显示器中的一个窗口。每个 Screen 包含一个用于保存 Screen 内容的 string 成员和三个 `string::size_type` 类型的成员, 它们分别表示光标的位置以及屏幕的高和宽。

除了定义数据和函数成员之外, 类还可以自定义某种类型在类中的别名。由类定义的类型名字和其他成员一样存在访问限制, 可以是 `public` 或者 `private` 中的一种:

```
class Screen {
public:
    typedef std::string::size_type pos;
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

我们在 Screen 的 `public` 部分定义了 `pos`, 这样用户就可以使用这个名字。Screen 的用户不应该知道 Screen 使用了一个 `string` 对象来存放它的数据, 因此通过把 `pos` 定义成 `public` 成员可以隐藏 Screen 实现的细节。272

关于 `pos` 的声明有两点需要注意。首先, 我们使用了 `typedef` (参见 2.5.1 节, 第 60 页), 也可以等价地使用类型别名 (参见 2.5.1 节, 第 60 页):

```
class Screen {
public:
    // 使用类型别名等价地声明一个类型名字
    using pos = std::string::size_type;
    // 其他成员与之前的版本一致
};
```

其次, 用来定义类型的成员必须先定义后使用, 这一点与普通成员有所区别, 具体原因将

在 7.4.1 节（第 254 页）解释。因此，类型成员通常出现在类开始的地方。

Screen 类的成员函数

要使我们的类更加实用，还需要添加一个构造函数令用户能够定义屏幕的尺寸和内容，以及其他两个成员，分别负责移动光标和读取给定位置的字符：

```
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // 因为 Screen 有另一个构造函数,
                        // 所以本函数是必需的
    // cursor 被其类内初始值初始化为 0
    Screen(pos ht, pos wd, char c): height(ht), width(wd),
        contents(ht * wd, c) { }
    char get() const                                // 读取光标处的字符
    { return contents[cursor]; }                   // 隐式内联
    inline char get(pos ht, pos wd) const;          // 显式内联
    Screen &move(pos r, pos c);                   // 能在之后被设为内联
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

因为我们已经提供了一个构造函数，所以编译器将不会自动生成默认的构造函数。如果我们的类需要默认构造函数，必须显式地把它声明出来。在此例中，我们使用`=default`告诉编译器为我们合成默认的构造函数（参见 7.1.4 节，第 237 页）。

需要指出的是，第二个构造函数（接受三个参数）为 `cursor` 成员隐式地使用了类内初始值（参见 7.1.4 节，第 238 页）。如果类中不存在 `cursor` 的类内初始值，我们就需要像其他成员一样显式地初始化 `cursor` 了。

273> 令成员作为内联函数

在类中，常有一些规模较小的函数适合于被声明成内联函数。如我们之前所见的，定义在类内部的成员函数是自动 `inline` 的（参见 6.5.2 节，第 213 页）。因此，`Screen` 的构造函数和返回光标所指字符的 `get` 函数默认是 `inline` 函数。

我们可以在类的内部把 `inline` 作为声明的一部分显式地声明成员函数，同样的，也能在类的外部用 `inline` 关键字修饰函数的定义：

```
inline                                         // 可以在函数的定义处指定 inline
Screen &Screen::move(pos r, pos c)
{
    pos row = r * width;                      // 计算行的位置
    cursor = row + c;                         // 在行内将光标移动到指定的列
    return *this;                             // 以左值的形式返回对象
}
char Screen::get(pos r, pos c) const // 在类的内部声明成 inline
{
    pos row = r * width;                      // 计算行的位置
    return contents[row + c];                 // 返回给定列的字符
}
```

虽然我们无须在声明和定义的地方同时说明 `inline`, 但这么做其实是合法的。不过, 最好只在类外部定义的地方说明 `inline`, 这样可以使类更容易理解。



和我们在头文件中定义 `inline` 函数的原因一样 (参见 6.5.2 节, 第 214 页), `inline` 成员函数也应该与相应的类定义在同一个头文件中。

重载成员函数

和非成员函数一样, 成员函数也可以被重载 (参见 6.4 节, 第 206 页), 只要函数之间在参数的数量和/或类型上有所区别就行。成员函数的函数匹配过程 (参见 6.4 节, 第 208 页) 同样与非成员函数非常类似。

举个例子, 我们的 `Screen` 类定义了两个版本的 `get` 函数。一个版本返回光标当前位置的字符; 另一个版本返回由行号和列号确定的位置的字符。编译器根据实参的数量来决定运行哪个版本的函数:

```
Screen myscreen;
char ch = myscreen.get();      // 调用 Screen::get()
ch = myscreen.get(0,0);       // 调用 Screen::get(pos, pos)
```

可变数据成员

有时 (但并不频繁) 会发生这样一种情况, 我们希望能修改类的某个数据成员, 即使是在一个 `const` 成员函数内。可以通过在变量的声明中加入 `mutable` 关键字做到这一点。

一个可变数据成员 (`mutable data member`) 永远不会是 `const`, 即使它是 `const` 对象的成员。因此, 一个 `const` 成员函数可以改变一个可变成员的值。举个例子, 我们将给 `Screen` 添加一个名为 `access_ctr` 的可变成员, 通过它我们可以追踪每个 `Screen` 的成员函数被调用了多少次: ◀ 274

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr; // 即使在一个 const 对象内也能被修改
    // 其他成员与之前的版本一致
};
void Screen::some_member() const
{
    ++access_ctr;           // 保存一个计数值, 用于记录成员函数被调用的次数
    // 该成员需要完成的其他工作
}
```

尽管 `some_member` 是一个 `const` 成员函数, 它仍然能够改变 `access_ctr` 的值。该成员是个可变成员, 因此任何成员函数, 包括 `const` 函数在内都能改变它的值。

类数据成员的初始值

在定义好 `Screen` 类之后, 我们将继续定义一个窗口管理类并用它表示显示器上的一组 `Screen`。这个类将包含一个 `Screen` 类型的 `vector`, 每个元素表示一个特定的 `Screen`。默认情况下, 我们希望 `Window_mgr` 类开始时总是拥有一个默认初始化的

C++ 11 Screen。在 C++11 新标准中，最好的方式就是把这个默认值声明成一个类内初始值（参见 2.6.1 节，第 64 页）：

```
class Window_mngr {
private:
    // 这个 Window_mngr 追踪的 Screen
    // 默认情况下，一个 Window_mngr 包含一个标准尺寸的空白 Screen
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};
```

当我们初始化类类型的成员时，需要为构造函数传递一个符合成员类型的实参。在此例中，我们使用一个单独的元素值对 `vector` 成员执行了列表初始化（参见 3.3.1 节，第 87 页），这个 `Screen` 的值被传递给 `vector<Screen>` 的构造函数，从而创建了一个单元素的 `vector` 对象。具体地说，`Screen` 的构造函数接受两个尺寸参数和一个字符值，创建了一个给定大小的空白屏幕对象。

如我们之前所知的，类内初始值必须使用=的初始化形式（初始化 `Screen` 的数据成员时所用的）或者花括号括起来的直接初始化形式（初始化 `screens` 所用的）。



当我们提供一个类内初始值时，必须以符号=或者花括号表示。

275

7.3.1 节练习

练习 7.23: 编写你自己的 `Screen` 类。

练习 7.24: 给你的 `Screen` 类添加三个构造函数：一个默认构造函数；另一个构造函数接受宽和高的值，然后将 `contents` 初始化成给定数量的空白；第三个构造函数接受宽和高的值以及一个字符，该字符作为初始化之后屏幕的内容。

练习 7.25: `Screen` 能安全地依赖于拷贝和赋值操作的默认版本吗？如果能，为什么？如果不能，为什么？

练习 7.26: 将 `Sales_data::avg_price` 定义成内联函数。



7.3.2 返回*this 的成员函数

接下来我们继续添加一些函数，它们负责设置光标所在位置的字符或者其他任一给定位置的字符：

```
class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
    // 其他成员和之前的版本一致
};

inline Screen &Screen::set(char c)
{
    contents[cursor] = c;           // 设置当前光标所在位置的新值
    return *this;                  // 将 this 对象作为左值返回
}
```

```

    }
    inline Screen &Screen::set(pos r, pos col, char ch)
    {
        contents[r*width + col] = ch;      // 设置给定位置的新值
        return *this;                      // 将 this 对象作为左值返回
    }
}

```

和 move 操作一样，我们的 set 成员的返回值是调用 set 的对象的引用（参见 7.1.2 节，第 232 页）。返回引用的函数是左值的（参见 6.3.2 节，第 202 页），意味着这些函数返回的是对对象本身而非对象的副本。如果我们把一系列这样的操作连接在一条表达式中的话：

```
// 把光标移动到一个指定的位置，然后设置该位置的字符值
myScreen.move(4,0).set('#');
```

这些操作将在同一个对象上执行。在上面的表达式中，我们首先移动 myScreen 内的光标，然后设置 myScreen 的 contents 成员。也就是说，上述语句等价于

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

如果我们令 move 和 set 返回 Screen 而非 Screen&，则上述语句的行为将大不相同。在此例中等价于：

```
// 如果 move 返回 Screen 而非 Screen&
Screen temp = myScreen.move(4,0);           // 对返回值进行拷贝
temp.set('#');                                // 不会改变 myScreen 的 contents
```

< 276

假如当初我们定义的返回类型不是引用，则 move 的返回值将是 *this 的副本（参见 6.3.2 节，第 201 页），因此调用 set 只能改变临时副本，而不能改变 myScreen 的值。

从 const 成员函数返回*this

接下来，我们继续添加一个名为 display 的操作，它负责打印 Screen 的内容。我们希望这个函数能和 move 以及 set 出现在同一序列中，因此类似于 move 和 set，display 函数也应该返回执行它的对象的引用。

从逻辑上来说，显示一个 Screen 并不需要改变它的内容，因此我们令 display 为一个 const 成员，此时，this 将是一个指向 const 的指针而 *this 是 const 对象。由此推断，display 的返回类型应该是 const Sales_data&。然而，如果真的令 display 返回一个 const 的引用，则我们将不能把 display 嵌入到一组动作的序列中去：

```
Screen myScreen;
// 如果 display 返回常量引用，则调用 set 将引发错误
myScreen.display(cout).set('*');
```

即使 myScreen 是个非常量对象，对 set 的调用也无法通过编译。问题在于 display 的 const 版本返回的是常量引用，而我们显然无权 set 一个常量对象。



一个 const 成员函数如果以引用的形式返回 *this，那么它的返回类型将是常量引用。

基于 const 的重载

通过区分成员函数是否是 const 的，我们可以对其进行重载，其原因与我们之前根据指针参数是否指向 const（参见 6.4 节，第 208 页）而重载函数的原因差不多。具体说

来，因为非常量版本的函数对于常量对象是不可用的，所以我们只能在一个常量对象上调用 `const` 成员函数。另一方面，虽然可以在非常量对象上调用常量版本或非常量版本，但显然此时非常量版本是一个更好的匹配。

在下面的这个例子中，我们将定义一个名为 `do_display` 的私有成员，由它负责打印 `Screen` 的实际工作。所有的 `display` 操作都将调用这个函数，然后返回执行操作的对象：

```
277> class Screen {
public:
    // 根据对象是否是 const 重载了 display 函数
    Screen &display(std::ostream &os)
        { do_display(os); return *this; }
    const Screen &display(std::ostream &os) const
        { do_display(os); return *this; }

private:
    // 该函数负责显示 Screen 的内容
    void do_display(std::ostream &os) const { os << contents; }
    // 其他成员与之前的版本一致
};
```

和我们之前所学的一样，当一个成员调用另外一个成员时，`this` 指针在其中隐式地传递。因此，当 `display` 调用 `do_display` 时，它的 `this` 指针隐式地传递给 `do_display`。而当 `display` 的非常量版本调用 `do_display` 时，它的 `this` 指针将隐式地从指向非常量的指针转换成指向常量的指针（参见 4.11.2 节，第 144 页）。

当 `do_display` 完成后，`display` 函数各自返回解引用 `this` 所得的对象。在非常量版本中，`this` 指向一个非常量对象，因此 `display` 返回一个普通的（非常量）引用；而 `const` 成员则返回一个常量引用。

当我们在某个对象上调用 `display` 时，该对象是否是 `const` 决定了应该调用 `display` 的哪个版本：

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

建议：对于公共代码使用私有功能函数

有些读者可能会奇怪为什么我们要费力定义一个单独的 `do_display` 函数。毕竟，对 `do_display` 的调用并不比 `do_display` 函数内部所做的操作简单多少。为什么还要这么做呢？实际上我们是出于以下原因的：

- 一个基本的愿望是避免在多处使用同样的代码。
- 我们预期随着类的规模发展，`display` 函数有可能变得更加复杂，此时，把相应的操作写在一处而非两处的作用就比较明显了。
- 我们很可能在开发过程中给 `do_display` 函数添加某些调试信息，而这些信息将在代码的最终产品版本中去掉。显然，只在 `do_display` 一处添加或删除这些信息要更容易一些。

- 这个额外的函数调用不会增加任何开销。因为我们在类内部定义了 `do_display`，所以它隐式地被声明成内联函数。这样的话，调用 `do_display` 就不会带来任何额外的运行时开销。

在实践中，设计良好的 C++ 代码常常包含大量类似于 `do_display` 的小函数，通过调用这些函数，可以完成一组其他函数的“实际”工作。

7.3.2 节练习

练习 7.27：给你自己的 `Screen` 类添加 `move`、`set` 和 `display` 函数，通过执行下面的代码检验你的类是否正确。

```
Screen myScreen(5, 5, 'X');
myScreen.move(4,0).set('#').display(cout);
cout << "\n";
myScreen.display(cout);
cout << "\n";
```

练习 7.28：如果 `move`、`set` 和 `display` 函数的返回类型不是 `Screen&` 而是 `Screen`，则在上一个练习中将会发生什么情况？

练习 7.29：修改你的 `Screen` 类，令 `move`、`set` 和 `display` 函数返回 `Screen` 并检查程序的运行结果，在上一个练习中你的推测正确吗？

练习 7.30：通过 `this` 指针使用成员的做法虽然合法，但是有点多余。讨论显式地使用指针访问成员的优缺点。

7.3.3 类类型

每个类定义了唯一的类型。对于两个类来说，即使它们的成员完全一样，这两个类也是两个不同的类型。例如：

```
struct First {
    int memi;
    int getMem();
};

struct Second {
    int memi;
    int getMem();
};

First obj1;
Second obj2 = obj1;           // 错误：obj1 和 obj2 的类型不同
```

< 278

 Note

即使两个类的成员列表完全一致，它们也是不同的类型。对于一个类来说，它的成员和其他任何类（或者任何其他作用域）的成员都不是一回事儿。

我们可以把类名作为类型的名字使用，从而直接指向类类型。或者，我们也可以把类名跟在关键字 `class` 或 `struct` 后面：

```
Sales_data item1;           // 默认初始化 Sales_data 类型的对象
class Sales_data item1;      // 一条等价的声明
```

上面这两种使用类类型的方式是等价的，其中第二种方式从 C 语言继承而来，并且在 C++ 语言中也是合法的。

类的声明

就像可以把函数的声明和定义分离开来一样（参见 6.1.2 节，第 186 页），我们也能仅声明类而暂时不定义它：

```
class Screen; // Screen 类的声明
```

279 这种声明有时被称作 **前向声明**（forward declaration），它向程序中引入了名字 Screen 并且指明 Screen 是一种类类型。对于类型 Screen 来说，在它声明之后定义之前是一个**不完全类型**（incomplete type），也就是说，此时我们已知 Screen 是一个类类型，但是不清楚它到底包含哪些成员。

不完全类型只能在非常有限的情景下使用：可以定义指向这种类型的指针或引用，也可以声明（但是不能定义）以不完全类型作为参数或者返回类型的函数。

对于一个类来说，在我们创建它的对象之前该类必须被定义过，而不能仅仅被声明。否则，编译器就无法了解这样的对象需要多少存储空间。类似的，类也必须首先被定义，然后才能用引用或者指针访问其成员。毕竟，如果类尚未定义，编译器也就不清楚该类到底有哪些成员。

在 7.6 节（第 268 页）中我们将描述一种例外的情况：直到类被定义之后数据成员才能被声明成这种类类型。换句话说，我们必须首先完成类的定义，然后编译器才能知道存储该数据成员需要多少空间。因为只有当类全部完成后类才算被定义，所以一个类的成员类型不能是该类自己。然而，一旦一个类的名字出现后，它就被认为是声明过了（但尚未定义），因此类允许包含指向它自身类型的引用或指针：

```
class Link_screen {
    Screen window;
    Link_screen *next;
    Link_screen *prev;
};
```

7.3.3 节练习

练习 7.31： 定义一对类 X 和 Y，其中 X 包含一个指向 Y 的指针，而 Y 包含一个类型为 X 的对象。

7.3.4 友元再探

我们的 Sales_data 类把三个普通的非成员函数定义成了友元（参见 7.2.1 节，第 241 页）。类还可以把其他的类定义成友元，也可以把其他类（之前已定义过的）的成员函数定义成友元。此外，友元函数能定义在类的内部，这样的函数是隐式内联的。

类之间的友元关系

举个友元类的例子，我们的 Window_mgr 类（参见 7.3.1 节，第 245 页）的某些成员可能需要访问它管理的 Screen 类的内部数据。例如，假设我们需要为 Window_mgr 添加一个名为 clear 的成员，它负责把一个指定的 Screen 的内容都设为空白。为了完成这一任务，clear 需要访问 Screen 的私有成员；而要想令这种访问合法，Screen 需要

把 Window_mgr 指定成它的友元:

```
class Screen {
    // Window_mgr 的成员可以访问 Screen 类的私有部分
    friend class Window_mgr;
    // Screen 类的剩余部分
};
```

如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员。通过上面的声明，Window_mgr 被指定为 Screen 的友元，因此我们可以将 Window_mgr 的 clear 成员写成如下的形式：

```
class Window_mgr {
public:
    // 窗口中每个屏幕的编号
    using ScreenIndex = std::vector<Screen>::size_type;
    // 按照编号将指定的 Screen 重置为空白
    void clear(ScreenIndex);
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};

void Window_mgr::clear(ScreenIndex i)
{
    // s 是一个 Screen 的引用，指向我们想清空的那个屏幕
    Screen &s = screens[i];
    // 将那个选定的 Screen 重置为空白
    s.contents = string(s.height * s.width, ' ');
}
```

一开始，首先把 s 定义成 screens vector 中第 i 个位置上的 Screen 的引用，随后利用 Screen 的 height 和 width 成员计算出一个新的 string 对象，并令其含有若干个空白字符，最后我们把这个含有很多空白的字符串赋给 contents 成员。

如果 clear 不是 Screen 的友元，上面的代码将无法通过编译，因为此时 clear 将不能访问 Screen 的 height、width 和 contents 成员。而当 Screen 将 Window_mgr 指定为其友元之后，Screen 的所有成员对于 Window_mgr 就都变成可见的了。

必须要注意的一点是，友元关系不存在传递性。也就是说，如果 Window_mgr 有它自己的友元，则这些友元并不能理所当然地具有访问 Screen 的特权。



每个类负责控制自己的友元类或友元函数。

令成员函数作为友元

除了令整个 Window_mgr 作为友元之外，Screen 还可以只为 clear 提供访问权限。当把一个成员函数声明成友元时，我们必须明确指出该成员函数属于哪个类：

```
class Screen {
    // Window_mgr::clear 必须在 Screen 类之前被声明
    friend void Window_mgr::clear(ScreenIndex);
    // Screen 类的剩余部分
};
```

要想令某个成员函数作为友元，我们必须仔细组织程序的结构以满足声明和定义的彼此依赖关系。在这个例子中，我们必须按照如下方式设计程序：

- 首先定义 `Window_mgr` 类，其中声明 `clear` 函数，但是不能定义它。在 `clear` 使用 `Screen` 的成员之前必须先声明 `Screen`。
- 接下来定义 `Screen`，包括对于 `clear` 的友元声明。
- 最后定义 `clear`，此时它才可以使用 `Screen` 的成员。

函数重载和友元

尽管重载函数的名字相同，但它们仍然是不同的函数。因此，如果一个类想把一组重载函数声明成它的友元，它需要对这组函数中的每一个分别声明：

```
// 重载的 storeOn 函数
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);

class Screen {
    // storeOn 的 ostream 版本能访问 Screen 对象的私有部分
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};
```

`Screen` 类把接受 `ostream&` 的 `storeOn` 函数声明成它的友元，但是接受 `BitMap&` 作为参数的版本仍然不能访问 `Screen`。



友元声明和作用域

类和非成员函数的声明不是必须在它们的友元声明之前。当一个名字第一次出现在一个友元声明中时，我们隐式地假定该名字在当前作用域中是可见的。然而，友元本身不一定真的声明在当前作用域中（参见 7.2.1 节，第 241 页）。

甚至就算在类的内部定义该函数，我们也必须在类的外部提供相应的声明从而使得函数可见。换句话说，即使我们仅仅是用声明友元的类的成员调用该友元函数，它也必须是被声明过的：

```
struct X {
    friend void f() { /* 友元函数可以定义在类的内部 */ }
    X() { f(); }                                // 错误：f 还没有被声明
    void g();
    void h();
};

void X::g() { return f(); }                    // 错误：f 还没有被声明
void f();                                     // 声明那个定义在 X 中的函数
void X::h() { return f(); }                    // 正确：现在 f 的声明在作用域中了
```

282 >

关于这段代码最重要的是理解友元声明的作用是影响访问权限，它本身并非普通意义上的声明。



请注意，有的编译器并不强制执行上述关于友元的限定规则（参见 7.2.1 节，第 241 页）。

7.3.4 节练习

练习 7.32: 定义你自己的 Screen 和 Window_mgr，其中 clear 是 Window_mgr 的成员，是 Screen 的友元。

7.4 类的作用域

每个类都会定义它自己的作用域。在类的作用域之外，普通的数据和函数成员只能由对象、引用或者指针使用成员访问运算符（参见 4.6 节，第 133 页）来访问。对于类类型成员则使用作用域运算符访问。不论哪种情况，跟在运算符之后的名字都必须是对应类的成员：

```
Screen::pos ht = 24, wd = 80;           // 使用 Screen 定义的 pos 类型
Screen scr(ht, wd, ' ');
Screen *p = &scr;
char c = scr.get();                   // 访问 scr 对象的 get 成员
c = p->get();                      // 访问 p 所指对象的 get 成员
```

作用域和定义在类外部的成员

一个类就是一个作用域的事实能够很好地解释为什么当我们在类的外部定义成员函数时必须同时提供类名和函数名（参见 7.1.2 节，第 230 页）。在类的外部，成员的名字被隐藏起来了。

一旦遇到了类名，定义的剩余部分就在类的作用域之内了，这里的剩余部分包括参数列表和函数体。结果就是，我们可以直接使用类的其他成员而无须再次授权了。

例如，我们回顾一下 Window_mgr 类的 clear 成员（参见 7.3.4 节，第 251 页），该函数的参数用到了 Window_mgr 类定义的一种类型：

```
void Window_mgr::clear(ScreenIndex i)
{
    Screen &s = screens[i];
    s.contents = string(s.height * s.width, ' ');
}
```

因为编译器在处理参数列表之前已经明确了我们当前正位于 Window_mgr 类的作用域中，所以不必再专门说明 ScreenIndex 是 Window_mgr 类定义的。出于同样的原因，编译器也能知道函数体中用到的 screens 也是在 Window_mgr 类中定义的。 ◀ 283

另一方面，函数的返回类型通常出现在函数名之前。因此当成员函数定义在类的外部时，返回类型中使用的名字都位于类的作用域之外。这时，返回类型必须指明它是哪个类的成员。例如，我们可能向 Window_mgr 类添加一个新的名为 addScreen 的函数，它负责向显示器添加一个新的屏幕。这个成员的返回类型将是 ScreenIndex，用户可以通过它定位到指定的 Screen：

```
class Window_mgr {
public:
    // 向窗口添加一个 Screen，返回它的编号
    ScreenIndex addScreen(const Screen&);
    // 其他成员与之前的版本一致
};

// 首先处理返回类型，之后我们才进入 Window_mgr 的作用域
```

```
Window_mgr::ScreenIndex
Window_mgr::addScreen(const Screen &s)
{
    screens.push_back(s);
    return screens.size() - 1;
}
```

因为返回类型出现在类名之前，所以事实上它是位于 `Window_mgr` 类的作用域之外的。在这种情况下，要想使用 `ScreenIndex` 作为返回类型，我们必须明确指定哪个类定义了它。

7.4 节练习

练习 7.33: 如果我们给 `Screen` 添加一个如下所示的 `size` 成员将发生什么情况？如果出现了问题，请尝试修改它。

```
pos Screen::size() const
{
    return height * width;
}
```



7.4.1 名字查找与类的作用域

在目前为止，我们编写的程序中，**名字查找**（name lookup）（寻找与所用名字最匹配的声明的过程）的过程比较直截了当：

- 首先，在名字所在的块中寻找其声明语句，只考虑在名字的使用之前出现的声明。
- 如果没找到，继续查找外层作用域。
- 如果最终没有找到匹配的声明，则程序报错。

284

对于定义在类内部的成员函数来说，解析其中名字的方式与上述的查找规则有所区别，不过在当前的这个例子中体现得不太明显。类的定义分两步处理：

- 首先，编译成员的声明。
- 直到类全部可见后才编译函数体。

Note

编译器处理完类中的全部声明后才会处理成员函数的定义。

按照这种两阶段的方式处理类可以简化类代码的组织方式。因为成员函数体直到整个类可见后才会被处理，所以它能使用类中定义的任何名字。相反，如果函数的定义和成员的声明被同时处理，那么我们将不得不在成员函数中只使用那些已经出现的名字。

用于类成员声明的名字查找

这种两阶段的处理方式只适用于成员函数中使用的名字。声明中使用的名字，包括返回类型或者参数列表中使用的名字，都必须在使用前确保可见。如果某个成员的声明使用了类中尚未出现的名字，则编译器将会在定义该类的作用域中继续查找。例如：

```
typedef double Money;
string bal;
class Account {
public:
```

```

    Money balance() { return bal; }
private:
    Money bal;
    // ...
};

```

当编译器看到 `balance` 函数的声明语句时，它将在 `Account` 类的范围内寻找对 `Money` 的声明。编译器只考虑 `Account` 中在使用 `Money` 前出现的声明，因为没找到匹配的成员，所以编译器会接着到 `Account` 的外层作用域中查找。在这个例子中，编译器会找到 `Money` 的 `typedef` 语句，该类型被用作 `balance` 函数的返回类型以及数据成员 `bal` 的类型。另一方面，`balance` 函数体在整个类可见后才被处理，因此，该函数的 `return` 语句返回名为 `bal` 的成员，而非外层作用域的 `string` 对象。

类型名要特殊处理

一般来说，内层作用域可以重新定义外层作用域中的名字，即使该名字已经在内层作用域中使用过。然而在类中，如果成员使用了外层作用域中的某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字：

```

typedef double Money;
class Account {
public:
    Money balance() { return bal; } // 使用外层作用域的 Money
private:
    typedef double Money;           // 错误：不能重新定义 Money
    Money bal;
    // ...
};

```

需要特别注意的是，即使 `Account` 中定义的 `Money` 类型与外层作用域一致，上述代码仍然是错误的。

尽管重新定义类型名字是一种错误的行为，但是编译器并不为此负责。一些编译器仍将顺利通过这样的代码，而忽略代码有错的事实。



类型的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类名的定义之后。

成员定义中的普通块作用域的名字查找

成员函数中使用的名字按照如下方式解析：

- 首先，在成员函数内查找该名字的声明。和前面一样，只有在函数使用之前出现的声明才被考虑。
- 如果在成员函数内没有找到，则在类内继续查找，这时类的所有成员都可以被考虑。
- 如果类内也没找到该名字的声明，在成员函数定义之前的作用域内继续查找。

一般来说，不建议使用其他成员的名字作为某个成员函数的参数。不过为了更好地解释名字的解析过程，我们不妨在 `dummy_fcn` 函数中暂时违反一下这个约定：

```

// 注意：这段代码仅为了说明而用，不是一段很好的代码
// 通常情况下不建议为参数和成员使用同样的名字
int height;                      // 定义了一个名字，稍后将在 Screen 中使用

```

```

class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height) {
        cursor = width * height;      // 哪个 height? 是那个参数
    }
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};

```

286

当编译器处理 `dummy_fcn` 中的乘法表达式时，它首先在函数作用域内查找表达式中用到的名字。函数的参数位于函数作用域内，因此 `dummy_fcn` 函数体内用到的名字 `height` 指的是参数声明。

在此例中，`height` 参数隐藏了同名的成员。如果想绕开上面的查找规则，应该将代码变为：

```

// 不建议的写法：成员函数中的名字不应该隐藏同名的成员
void Screen::dummy_fcn(pos height) {
    cursor = width * this->height;           // 成员 height
    // 另外一种表示该成员的方式
    cursor = width * Screen::height;          // 成员 height
}

```



尽管类的成员被隐藏了，但我们仍然可以通过加上类的名字或显式地使用 `this` 指针来强制访问成员。

其实最好的确保我们使用 `height` 成员的方法是给参数起个其他名字：

```

// 建议的写法：不要把成员名字作为参数或其他局部变量使用
void Screen::dummy_fcn(pos ht) {
    cursor = width * height;                 // 成员 height
}

```

在此例中，当编译器查找名字 `height` 时，显然在 `dummy_fcn` 函数内部是找不到的。编译器接着会在 `Screen` 内查找匹配的声明，即使 `height` 的声明出现在 `dummy_fcn` 使用它之后，编译器也能正确地解析函数使用的是名为 `height` 的成员。

类作用域之后，在外围的作用域中查找

如果编译器在函数和类的作用域中都没有找到名字，它将接着在外围的作用域中查找。在我们的例子中，名字 `height` 定义在外层作用域中，且位于 `Screen` 的定义之前。然而，外层作用域中的对象被名为 `height` 的成员隐藏掉了。因此，如果我们需要的是外层作用域中的名字，可以显式地通过作用域运算符来进行请求：

```

// 不建议的写法：不要隐藏外层作用域中可能被用到的名字
void Screen::dummy_fcn(pos height) {
    cursor = width * ::height;                // 哪个 height? 是那个全局的
}

```



尽管外层的对象被隐藏掉了，但我们仍然可以用作用域运算符访问它。

在文件中名字的出现处对其进行解析

<287

当成员定义在类的外部时，名字查找的第三步不仅要考虑类定义之前的全局作用域中的声明，还需要考虑在成员函数定义之前的全局作用域中的声明。例如：

```
int height;                                // 定义了一个名字，稍后将在 Screen 中使用
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0;                      // 隐藏了外层作用域中的 height
};
Screen::pos verify(Screen::pos);
void Screen::setHeight(pos var) {
    // var: 参数
    // height: 类的成员
    // verify: 全局函数
    height = verify(var);
}
```

请注意，全局函数 `verify` 的声明在 `Screen` 类的定义之前是不可见的。然而，名字查找的第三步包括了成员函数出现之前的全局作用域。在此例中，`verify` 的声明位于 `setHeight` 的定义之前，因此可以被正常使用。

7.4.1 节练习

练习 7.34：如果我们把第 256 页 `Screen` 类的 `pos` 的 `typedef` 放在类的最后一行会有什么情况？

练习 7.35：解释下面代码的含义，说明其中的 `Type` 和 `initVal` 分别使用了哪个定义。如果代码存在错误，尝试修改它。

```
typedef string Type;
Type initVal();
class Exercise {
public:
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}
```

7.5 构造函数再探

<288

对于任何 C++ 的类来说，构造函数都是其中重要的组成部分。我们已经在 7.1.4 节（第 235 页）中介绍了构造函数的基础知识，本节将继续介绍构造函数的一些其他功能，并对

之前已经介绍的内容进行一些更深入的讨论。

7.5.1 构造函数初始值列表

当我们定义变量时习惯于立即对其进行初始化，而非先定义、再赋值：

```
string foo = "Hello World!";           // 定义并初始化
string bar;                           // 默认初始化成空 string 对象
bar = "Hello World!";                // 为 bar 赋一个新值
```

就对象的数据成员而言，初始化和赋值也有类似的区别。如果没有在构造函数的初始值列表中显式地初始化成员，则该成员将在构造函数体之前执行默认初始化。例如：

```
// Sales_data 构造函数的一种写法，虽然合法但比较草率：没有使用构造函数初始值
Sales_data::Sales_data(const string &s,
                      unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

这段代码和我们在 237 页的原始定义效果是相同的：当构造函数完成后，数据成员的值相同。区别是原来的版本初始化了它的数据成员，而这个版本是对数据成员执行了赋值操作。这一区别到底会有什么深层次的影响完全依赖于数据成员的类型。

构造函数的初始值有时必不可少

有时我们可以忽略数据成员初始化和赋值之间的差异，但并非总能这样。如果成员是 `const` 或者是引用的话，必须将其初始化。类似的，当成员属于某种类类型且该类没有定义默认构造函数时，也必须将这个成员初始化。例如：

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

[289] 和其他常量对象或者引用一样，成员 `ci` 和 `ri` 都必须被初始化。因此，如果我们没有为它们提供构造函数初始值的话将引发错误：

```
// 错误：ci 和 ri 必须被初始化
ConstRef::ConstRef(int ii)
{ // 赋值：
    i = ii;                     // 正确
    ci = ii;                    // 错误：不能给 const 赋值
    ri = i;                     // 错误：ri 没被初始化
}
```

随着构造函数体一开始执行，初始化就完成了。我们初始化 `const` 或者引用类型的数据成员的唯一机会就是通过构造函数初始值，因此该构造函数的正确形式应该是：

```
// 正确：显式地初始化引用和 const 成员  
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
```



如果成员是 `const`、引用，或者属于某种未提供默认构造函数的类类型，我们必须通过构造函数初始值列表为这些成员提供初值。

建议：使用构造函数初始值

在很多类中，初始化和赋值的区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。

除了效率问题外更重要的是，一些数据成员必须被初始化。建议读者养成使用构造函数初始值的习惯，这样能避免某些意想不到的编译错误，特别是遇到有的类含有需要构造函数初始值的成员时。

成员初始化的顺序

显然，在构造函数初始值中每个成员只能出现一次。否则，给同一个成员赋两个不同的初始值有什么意义呢？

不过让人稍感意外的是，构造函数初始值列表只说明用于初始化成员的值，而不限定初始化的具体执行顺序。

成员的初始化顺序与它们在类定义中的出现顺序一致：第一个成员先被初始化，然后第二个，以此类推。构造函数初始值列表中初始值的前后位置关系不会影响实际的初始化顺序。

一般来说，初始化的顺序没什么特别要求。不过如果一个成员是用另一个成员来初始化的，那么这两个成员的初始化顺序就很关键了。

举个例子，考虑下面这个类：

```
class X {  
    int i;  
    int j;  
public:  
    // 未定义的：i 在 j 之前被初始化  
    X(int val): j(val), i(j) { }  
};
```

< 290

在此例中，从构造函数初始值的形式上来看仿佛是先用 `val` 初始化了 `j`，然后再用 `j` 初始化 `i`。实际上，`i` 先被初始化，因此这个初始值的效果是试图使用未定义的值 `j` 初始化 `i`！

有的编译器具备一项比较友好的功能，即当构造函数初始值列表中的数据成员顺序与这些成员声明的顺序不符时会生成一条警告信息。



最好令构造函数初始值的顺序与成员声明的顺序保持一致。而且如果可能的话，尽量避免使用某些成员初始化其他成员。

如果可能的话，最好用构造函数的参数作为成员的初始值，而尽量避免使用同一个对

象的其他成员。这样的好处是我们可以不必考虑成员的初始化顺序。例如，`X` 的构造函数如果写成如下的形式效果会更好：

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

在这个版本中，`i` 和 `j` 初始化的顺序就没什么影响了。

默认实参和构造函数

`Sales_data` 默认构造函数的行为与只接受一个 `string` 实参的构造函数差不多。唯一的区别是接受 `string` 实参的构造函数使用这个实参初始化 `bookNo`，而默认构造函数（隐式地）使用 `string` 的默认构造函数初始化 `bookNo`。我们可以把它们重写成一个使用默认实参（参见 6.5.1 节，第 211 页）的构造函数：

```
class Sales_data {
public:
    // 定义默认构造函数，令其与只接受一个 string 实参的构造函数功能相同
    Sales_data(std::string s = "") : bookNo(s) { }
    // 其他构造函数与之前一致
    Sales_data(std::string s, unsigned cnt, double rev):
        bookNo(s), units_sold(cnt), revenue(rev*cnt) { }
    Sales_data(std::istream &is) { read(is, *this); }
    // 其他成员与之前的版本一致
};
```

在上面这段程序中，类的接口与第 237 页的代码是一样的。当没有给定实参，或者给定了一个 `string` 实参时，两个版本的类创建了相同的对象。因为我们不提供实参也能调用上述的构造函数，所以该构造函数实际上为我们的类提供了默认构造函数。

291



如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数。

值得注意的是，我们不应该为 `Sales_data` 接受三个实参的构造函数提供默认值。因为如果用户为售出书籍的数量提供了一个非零的值，则我们就会期望用户同时提供这些书籍的售出价格。

7.5.1 节练习

练习 7.36: 下面的初始值是错误的，请找出问题所在并尝试修改它。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

练习 7.37: 使用本节提供的 `Sales_data` 类，确定初始化下面的变量时分别使用了哪个构造函数，然后罗列出每个对象所有数据成员的值。

```
Sales_data first_item(cin);
int main() {
    Sales_data next;
    Sales_data last("9-999-99999-9");
}
```

练习 7.38: 有些情况下我们希望提供 `cin` 作为接受 `istream&` 参数的构造函数的默认实参，请声明这样的构造函数。

练习 7.39: 如果接受 `string` 的构造函数和接受 `istream&` 的构造函数都使用默认实参，这种行为合法吗？如果不，为什么？

练习 7.40: 从下面的抽象概念中选择一个（或者你自己指定一个），思考这样的类需要哪些数据成员，提供一组合理的构造函数并阐明这样做的原因。

- | | | |
|-------------|------------|--------------|
| (a) Book | (b) Date | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree |

7.5.2 委托构造函数

C++11 新标准扩展了构造函数初始值的功能，使得我们可以定义所谓的委托构造函数（delegating constructor）。一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些（或者全部）职责委托给了其他构造函数。

和其他构造函数一样，一个委托构造函数也有一个成员初始值的列表和一个函数体。在委托构造函数内，成员初始值列表只有一个唯一的入口，就是类名本身。和其他成员初始值一样，类名后面紧跟圆括号括起来的参数列表，参数列表必须与类中另外一个构造函数匹配。

举个例子，我们使用委托构造函数重写 `Sales_data` 类，重写后的形式如下所示：

```
class Sales_data {
public:
    // 非委托构造函数使用对应的实参初始化成员
    Sales_data(std::string s, unsigned cnt, double price):
        bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    // 其余构造函数全都委托给另一个构造函数
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0, 0) {}
    Sales_data(std::istream &is): Sales_data()
        { read(is, *this); }
    // 其他成员与之前的版本一致
};
```

在这个 `Sales_data` 类中，除了一个构造函数外其他的都委托了它们的工作。第一个构造函数接受三个实参，使用这些实参初始化数据成员，然后结束工作。我们定义默认构造函数令其使用三参数的构造函数完成初始化过程，它也无须执行其他任务，这一点从空的构造函数体能看得出来。接受一个 `string` 的构造函数同样委托给了三参数的版本。

接受 `istream&` 的构造函数也是委托构造函数，它委托给了默认构造函数，默认构造函数又接着委托给三参数构造函数。当这些受委托的构造函数执行完后，接着执行 `istream&` 构造函数体的内容。它的构造函数体调用 `read` 函数读取给定的 `istream`。

当一个构造函数委托给另一个构造函数时，受委托的构造函数的初始值列表和函数体被依次执行。在 `Sales_data` 类中，受委托的构造函数体恰好是空的。假如函数体包含有代码的话，将先执行这些代码，然后控制权才会交还给委托者的函数体。

7.5.2 节练习

练习 7.41: 使用委托构造函数重新编写你的 Sales_data 类，给每个构造函数体添加一条语句，令其一旦执行就打印一条信息。用各种可能的方式分别创建 Sales_data 对象，认真研究每次输出的信息直到你确实理解了委托构造函数的执行顺序。

练习 7.42: 对于你在练习 7.40（参见 7.5.1 节，第 261 页）中编写的类，确定哪些构造函数可以使用委托。如果可以的话，编写委托构造函数。如果不可以，从抽象概念列表中重新选择一个你认为可以使用委托构造函数的，为挑选出的这个概念编写类定义。



7.5.3 默认构造函数的作用

293> 当对象被默认初始化或值初始化时自动执行默认构造函数。默认初始化在以下情况下发生：

- 当我们在块作用域内不使用任何初始值定义一个非静态变量（参见 2.2.1 节，第 39 页）或者数组时（参见 3.5.1 节，第 101 页）。
- 当一个类本身含有类类型的成员且使用合成的默认构造函数时（参见 7.1.4 节，第 235 页）。
- 当类类型的成员没有在构造函数初始值列表中显式地初始化时（参见 7.1.4 节，第 237 页）。

值初始化在以下情况下发生：

- 在数组初始化的过程中如果我们提供的初始值数量少于数组的大小时（参见 3.5.1 节，第 101 页）。
- 当我们不使用初始值定义一个局部静态变量时（参见 6.1.1 节，第 185 页）。
- 当我们通过书写形如 `T()` 的表达式显式地请求值初始化时，其中 `T` 是类型名（`vector` 的一个构造函数只接受一个实参用于说明 `vector` 大小（参见 3.3.1 节，第 88 页），它就是使用一个这种形式的实参来对它的元素初始化器进行值初始化）。

类必须包含一个默认构造函数以便在上述情况下使用，其中的大多数情况非常容易判断。

不那么明显的一种情况是类的某些数据成员缺少默认构造函数：

```
class NoDefault {
public:
    NoDefault(const std::string&); // 还有其他成员，但是没有其他构造函数了
};

struct A { // 默认情况下 my_mem 是 public 的（参见 7.2 节，第 240 页）
    NoDefault my_mem;
};

A a; // 错误：不能为 A 合成构造函数

struct B {
    B() {} // 错误：b_member 没有初始值
    NoDefault b_member;
};
```

Best Practices

在实际中，如果定义了其他构造函数，那么最好也提供一个默认构造函数。

使用默认构造函数

< 294

下面的 obj 的声明可以正常编译通过：

```
Sales_data obj(); // 正确：定义了一个函数而非对象
if (obj.isbn() == Primer_5th_ed.isbn()) // 错误：obj 是一个函数
```

但当我们试图使用 obj 时，编译器将报错，提示我们不能对函数使用成员访问运算符。问题在于，尽管我们想声明一个默认初始化的对象，obj 实际的含义却是一个不接受任何参数的函数并且其返回值是 Sales_data 类型的对象。

如果想定义一个使用默认构造函数进行初始化的对象，正确的方法是去掉对象名之后的空的括号对：

```
// 正确：obj 是个默认初始化的对象
Sales_data obj;
```



WARNING

对于 C++ 的新手程序员来说有一种常犯的错误，它们试图以如下的形式声明一个用默认构造函数初始化的对象：

Sales_data obj();	// 错误：声明了一个函数而非对象
Sales_data obj2;	// 正确：obj2 是一个对象而非函数

7.5.3 节练习

练习 7.43：假定有一个名为 NoDefault 的类，它有一个接受 int 的构造函数，但是没有默认构造函数。定义类 C，C 有一个 NoDefault 类型的成员，定义 C 的默认构造函数。

练习 7.44：下面这条声明合法吗？如果不，为什么？

```
vector<NoDefault> vec(10);
```

练习 7.45：如果在上一个练习中定义的 vector 的元素类型是 C，则声明合法吗？为什么？

练习 7.46：下面哪些论断是不正确的？为什么？

- (a) 一个类必须至少提供一个构造函数。
- (b) 默认构造函数是参数列表为空的构造函数。
- (c) 如果对于类来说不存在有意义的默认值，则类不应该提供默认构造函数。
- (d) 如果类没有定义默认构造函数，则编译器将为其生成一个并把每个数据成员初始化成相应类型的默认值。

7.5.4 隐式的类类型转换



4.11 节（第 141 页）曾经介绍过 C++ 语言在内置类型之间定义了几种自动转换规则。同样的，我们也能为类定义隐式转换规则。如果构造函数只接受一个实参，则它实际上定义了转换为此类类型的隐式转换机制，有时我们把这个构造函数称作转换构造函数 (converting constructor)。我们将在 14.9 节（第 514 页）介绍如何定义将一种类类型转换为另一种类类型的转换规则。

< 295



能通过一个实参调用的构造函数定义了一条从构造函数的参数类型向类类型隐式转换的规则。

在 Sales_data 类中，接受 string 的构造函数和接受 istream 的构造函数分别定义了从这两种类型向 Sales_data 隐式转换的规则。也就是说，在需要使用 Sales_data 的地方，我们可以使用 string 或者 istream 作为替代：

```
string null_book = "9-999-99999-9";
// 构造一个临时的 Sales_data 对象
// 该对象的 units_sold 和 revenue 等于 0, bookNo 等于 null_book
item.combine(null_book);
```

在这里我们用一个 string 实参调用了 Sales_data 的 combine 成员。该调用是合法的，编译器用给定的 string 自动创建了一个 Sales_data 对象。新生成的这个（临时）Sales_data 对象被传递给 combine。因为 combine 的参数是一个常量引用，所以我们可以给该参数传递一个临时量。

只允许一步类类型转换

在 4.11.2 节（第 143 页）中我们指出，编译器只会自动地执行一步类型转换。例如，因为下面的代码隐式地使用了两种转换规则，所以它是错误的：

```
// 错误：需要用户定义的两种转换：
// (1) 把 “9-999-99999-9” 转换成 string
// (2) 再把这个（临时的）string 转换成 Sales_data
item.combine("9-999-99999-9");
```

如果我们想完成上述调用，可以显式地把字符串转换成 string 或者 Sales_data 对象：

```
// 正确：显式地转换成 string，隐式地转换成 Sales_data
item.combine(string("9-999-99999-9"));
// 正确：隐式地转换成 string，显式地转换成 Sales_data
item.combine(Sales_data("9-999-99999-9"));
```

类类型转换不是总有效

是否需要从 string 到 Sales_data 的转换依赖于我们对用户使用该转换的看法。在此例中，这种转换可能是对的。null_book 中的 string 可能表示了一个不存在的 ISBN 编号。

另一个是从 istream 到 Sales_data 的转换：

```
// 使用 istream 构造函数创建一个函数传递给 combine
item.combine(cin);
```

这段代码隐式地把 cin 转换成 Sales_data，这个转换执行了接受一个 istream 的 Sales_data 构造函数。该构造函数通过读取标准输入创建了一个（临时的）Sales_data 对象，随后将得到的对象传递给 combine。

296 Sales_data 对象是个临时量（参见 2.4.1 节，第 54 页），一旦 combine 完成我们就不能再访问它了。实际上，我们构建了一个对象，先将它的值加到 item 中，随后将其丢弃。

抑制构造函数定义的隐式转换

在要求隐式转换的程序上下文中，我们可以通过将构造函数声明为 **explicit** 加以阻止：

```
class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    explicit Sales_data(const std::string &s): bookNo(s) { }
    explicit Sales_data(std::istream&); // 其他成员与之前的版本一致
};
```

此时，没有任何构造函数能用于隐式地创建 `Sales_data` 对象，之前的两种用法都无法通过编译：

```
item.combine(null_book); // 错误：string 构造函数是 explicit 的
item.combine(cin); // 错误：istream 构造函数是 explicit 的
```

关键字 `explicit` 只对一个实参的构造函数有效。需要多个实参的构造函数不能用于执行隐式转换，所以无须将这些构造函数指定为 `explicit` 的。只能在类内声明构造函数时使用 `explicit` 关键字，在类外部定义时不应重复：

```
// 错误：explicit 关键字只允许出现在类内的构造函数声明处
explicit Sales_data::Sales_data(istream& is)
{
    read(is, *this);
}
```

explicit 构造函数只能用于直接初始化

发生隐式转换的一种情况是当我们执行拷贝形式的初始化时（使用`=`）（参见 3.2.1 节，第 76 页）。此时，我们只能使用直接初始化而不能使用 `explicit` 构造函数：

```
Sales_data item1(null_book); // 正确：直接初始化
// 错误：不能将 explicit 构造函数用于拷贝形式的初始化过程
Sales_data item2 = null_book;
```

 当我们用 `explicit` 关键字声明构造函数时，它将只能以直接初始化的形式使用（参见 3.2.1 节，第 76 页）。而且，编译器将不会在自动转换过程中使用该构造函数。

为转换显式地使用构造函数

<297

尽管编译器不会将 `explicit` 的构造函数用于隐式转换过程，但是我们可以使用这样的构造函数显式地强制进行转换：

```
// 正确：实参是一个显式构造的 Sales_data 对象
item.combine(Sales_data(null_book));
// 正确：static_cast 可以使用 explicit 的构造函数
item.combine(static_cast<Sales_data>(cin));
```

在第一个调用中，我们直接使用 `Sales_data` 的构造函数，该调用通过接受 `string` 的

构造函数创建了一个临时的 `Sales_data` 对象。在第二个调用中，我们使用 `static_cast`（参见 4.11.3 节，第 145 页）执行了显式的而非隐式的转换。其中，`static_cast` 使用 `istream` 构造函数创建了一个临时的 `Sales_data` 对象。

标准库中含有显式构造函数的类

我们用过的一些标准库中的类含有单参数的构造函数：

- 接受一个单参数的 `const char*` 的 `string` 构造函数（参见 3.2.1 节，第 76 页）不是 `explicit` 的。
- 接受一个容量参数的 `vector` 构造函数（参见 3.3.1 节，第 87 页）是 `explicit` 的。

7.5.4 节练习

练习 7.47: 说明接受一个 `string` 参数的 `Sales_data` 构造函数是否应该是 `explicit` 的，并解释这样做的优缺点。

练习 7.48: 假定 `Sales_data` 的构造函数不是 `explicit` 的，则下述定义将执行什么样的操作？

```
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
```

如果 `Sales_data` 的构造函数是 `explicit` 的，又会发生什么呢？

练习 7.49: 对于 `combine` 函数的三种不同声明，当我们调用 `i.combine(s)` 时分别发生什么情况？其中 `i` 是一个 `Sales_data`，而 `s` 是一个 `string` 对象。

- (a) `Sales_data &combine(Sales_data);`
- (b) `Sales_data &combine(Sales_data&);`
- (c) `Sales_data &combine(const Sales_data&) const;`

练习 7.50: 确定在你的 `Person` 类中是否有一些构造函数应该是 `explicit` 的。

练习 7.51: `vector` 将其单参数的构造函数定义成 `explicit` 的，而 `string` 则不是，你觉得原因何在？



7.5.5 聚合类

298 聚合类（aggregate class）使得用户可以直接访问其成员，并且具有特殊的初始化语法形式。当一个类满足如下条件时，我们说它是聚合的：

- 所有成员都是 `public` 的。
- 没有定义任何构造函数。
- 没有类内初始值（参见 2.6.1 节，第 64 页）。
- 没有基类，也没有 `virtual` 函数，关于这部分知识我们将在第 15 章详细介绍。

例如，下面的类是一个聚合类：

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

我们可以提供一个花括号括起来的成员初始值列表，并用它初始化聚合类的数据成员：

```
// val1.ival = 0; val1.s = string("Anna")
Data val1 = { 0, "Anna" };
```

初始值的顺序必须与声明的顺序一致，也就是说，第一个成员的初始值要放在第一个，然后是第二个，以此类推。下面的例子是错误的：

```
// 错误：不能使用"Anna"初始化 ival，也不能使用 1024 初始化 s
Data val2 = { "Anna" , 1024 };
```

与初始化数组元素的规则（参见 3.5.1 节，第 101 页）一样，如果初始值列表中的元素个数少于类的成员数量，则靠后的成员被值初始化（参见 3.5.1 节，第 101 页）。初始值列表的元素个数绝对不能超过类的成员数量。

值得注意的是，显式地初始化类的对象存在三个明显的缺点：

- 要求类的所有成员都是 `public` 的。
- 将正确初始化每个对象的每个成员的重任交给了类的用户（而非类的作者）。因为用户很容易忘掉某个初始值，或者提供一个不恰当的初始值，所以这样的初始化过程冗长乏味且容易出错。
- 添加或删除一个成员之后，所有的初始化语句都需要更新。

7.5.5 节练习

◀ 299

练习 7.52： 使用 2.6.1 节（第 64 页）的 `Sales_data` 类，解释下面的初始化过程。如果存在问题，尝试修改它。

```
Sales_data item = {"978-0590353403", 25, 15.99};
```

7.5.6 字面值常量类



在 6.5.2 节（第 214 页）中我们提到过 `constexpr` 函数的参数和返回值必须是字面值类型。除了算术类型、引用和指针外，某些类也是字面值类型。和其他类不同，字面值类型的类可能含有 `constexpr` 函数成员。这样的成员必须符合 `constexpr` 函数的所有要求，它们是隐式 `const` 的（参见 7.1.2 节，第 231 页）。

数据成员都是字面值类型的聚合类（参见 7.5.5 节，第 266 页）是字面值常量类。如果一个类不是聚合类，但它符合下述要求，则它也是一个字面值常量类：

- 数据成员都必须是字面值类型。
- 类必须至少含有一个 `constexpr` 构造函数。
- 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式（参见 2.4.4 节，第 58 页）；或者如果成员属于某种类类型，则初始值必须使用成员自己的 `constexpr` 构造函数。
- 类必须使用析构函数的默认定义，该成员负责销毁类的对象（参见 7.1.5 节，第 239 页）。

`constexpr` 构造函数

尽管构造函数不能是 `const` 的（参见 7.1.4 节，第 235 页），但是字面值常量类的构造函数可以是 `constexpr`（参见 6.5.2 节，第 213 页）函数。事实上，一个字面值常量类必须至少提供一个 `constexpr` 构造函数。

C++ 11 constexpr 构造函数可以声明成 = default (参见 7.1.4 节, 第 237 页) 的形式 (或者是删除函数的形式, 我们将在 13.1.6 节 (第 449 页) 介绍相关知识)。否则, constexpr 构造函数就必须既符合构造函数的要求 (意味着不能包含返回语句), 又符合 constexpr 函数的要求 (意味着它能拥有的唯一可执行语句就是返回语句 (参见 6.5.2 节, 第 214 页))。综合这两点可知, constexpr 构造函数体一般来说应该是空的。我们通过前置关键字 constexpr 就可以声明一个 constexpr 构造函数了:

```
300> class Debug {
public:
    constexpr Debug(bool b = true): hw(b), io(b), other(b) { }
    constexpr Debug(bool h, bool i, bool o):
        hw(h), io(i), other(o) { }
    constexpr bool any() { return hw || io || other; }
    void set_io(bool b) { io = b; }
    void set_hw(bool b) { hw = b; }
    void set_other(bool b) { other = b; }
private:
    bool hw;                                // 硬件错误, 而非 IO 错误
    bool io;                                 // IO 错误
    bool other;                             // 其他错误
};
```

constexpr 构造函数必须初始化所有数据成员, 初始值或者使用 constexpr 构造函数, 或者是一条常量表达式。

constexpr 构造函数用于生成 constexpr 对象以及 constexpr 函数的参数或返回类型:

```
constexpr Debug io_sub(false, true, false);           // 调试 IO
if (io_sub.any())                                     // 等价于 if(true)
    cerr << "print appropriate error messages" << endl;
constexpr Debug prod(false);                          // 无调试
if (prod.any())                                      // 等价于 if(false)
    cerr << "print an error message" << endl;
```

7.5.6 节练习

练习 7.53: 定义你自己的 Debug。

练习 7.54: Debug 中以 set_ 开头的成员应该被声明成 constexpr 吗? 如果不, 为什么?

练习 7.55: 7.5.5 节 (第 266 页) 的 Data 类是字面值常量类吗? 请解释原因。

7.6 类的静态成员

有的时候类需要它的一些成员与类本身直接相关, 而不是与类的各个对象保持关联。例如, 一个银行账户类可能需要一个数据成员来表示当前的基准利率。在此例中, 我们希望利率与类关联, 而非与类的每个对象关联。从实现效率的角度来看, 没必要每个对象都存储利率信息。而且更加重要的是, 一旦利率浮动, 我们希望所有的对象都能使用新值。

声明静态成员

我们通过在成员的声明之前加上关键字 `static` 使得其与类关联在一起。和其他成员一样，静态成员可以是 `public` 的或 `private` 的。静态数据成员的类型可以是常量、引用、指针、类类型等。

举个例子，我们定义一个类，用它表示银行的账户记录：

<301>

```
class Account {
public:
    void calculate() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double);
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};
```

类的静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据。因此，每个 `Account` 对象将包含两个数据成员：`owner` 和 `amount`。只存在一个 `interestRate` 对象而且它被所有 `Account` 对象共享。

类似的，静态成员函数也不与任何对象绑定在一起，它们不包含 `this` 指针。作为结果，静态成员函数不能声明成 `const` 的，而且我们也不能在 `static` 函数体内使用 `this` 指针。这一限制既适用于 `this` 的显式使用，也对调用非静态成员的隐式使用有效。

使用类的静态成员

我们使用作用域运算符直接访问静态成员：

```
double r;
r = Account::rate(); // 使用作用域运算符访问静态成员
```

虽然静态成员不属于类的某个对象，但是我们仍然可以使用类的对象、引用或者指针来访问静态成员：

```
Account ac1;
Account *ac2 = &ac1;
// 调用静态成员函数 rate 的等价形式
r = ac1.rate(); // 通过 Account 的对象或引用
r = ac2->rate(); // 通过指向 Account 对象的指针
```

成员函数不用通过作用域运算符就能直接使用静态成员：

```
class Account {
public:
    void calculate() { amount += amount * interestRate; }
private:
    static double interestRate;
    // 其他成员与之前的版本一致
};
```

302 定义静态成员

和其他的成员函数一样，我们既可以在类的内部也可以在类的外部定义静态成员函数。当在类的外部定义静态成员时，不能重复 `static` 关键字，该关键字只出现在类内部的声明语句：

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```



和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名。`static` 关键字则只出现在类内部的声明语句中。

因为静态数据成员不属于类的任何一个对象，所以它们并不是在创建类的对象时被定义的。这意味着它们不是由类的构造函数初始化的。而且一般来说，我们不能在类的内部初始化静态成员。相反的，必须在类的外部定义和初始化每个静态成员。和其他对象一样，一个静态数据成员只能定义一次。

类似于全局变量（参见 6.1.1 节，第 184 页），静态数据成员定义在任何函数之外。因此一旦它被定义，就将一直存在于程序的整个生命周期中。

我们定义静态数据成员的方式和在类的外部定义成员函数差不多。我们需要指定对象的类型名，然后是类名、作用域运算符以及成员自己的名字：

```
// 定义并初始化一个静态成员
double Account::interestRate = initRate();
```

这条语句定义了名为 `interestRate` 的对象，该对象是类 `Account` 的静态成员，其类型是 `double`。从类名开始，这条定义语句的剩余部分就都位于类的作用域之内了。因此，我们可以直接使用 `initRate` 函数。注意，虽然 `initRate` 是私有的，我们也能用它初始化 `interestRate`。和其他成员的定义一样，`interestRate` 的定义也可以访问类的私有成员。



要想确保对象只定义一次，最好的办法是把静态数据成员的定义与其他非内联函数的定义放在同一个文件中。

静态成员的类内初始化

通常情况下，类的静态成员不应该在类的内部初始化。然而，我们可以为静态成员提供 `const` 整数类型的类内初始值，不过要求静态成员必须是字面值常量类型的 `constexpr`（参见 7.5.6 节，第 267 页）。初始值必须是常量表达式，因为这些成员本身就是常量表达式，所以它们能用在所有适合于常量表达式的地方。例如，我们可以用一个初始化了的静态数据成员指定数组成员的维度：

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);
private:
```

303

```

static constexpr int period = 30;      // period 是常量表达式
double daily_tbl[period];
};

```

如果某个静态成员的应用场景仅限于编译器可以替换它的值的情况，则一个初始化的 `const` 或 `constexpr static` 不需要分别定义。相反，如果我们将它用于值不能替换的场景中，则该成员必须有一条定义语句。

例如，如果 `period` 的唯一用途就是定义 `daily_tbl` 的维度，则不需要在 `Account` 外面专门定义 `period`。此时，如果我们忽略了这条定义，那么对程序非常微小的改动也可能造成编译错误，因为程序找不到该成员的定义语句。举个例子，当需要把 `Account::period` 传递给一个接受 `const int&` 的函数时，必须定义 `period`。

如果在类的内部提供了一个初始值，则成员的定义不能再指定一个初始值了：

```

// 一个不带初始值的静态成员的定义
constexpr int Account::period;           // 初始值在类的定义内提供

```



即使一个常量静态数据成员在类内部被初始化了，通常情况下也应该在类的外部定义一下该成员。

静态成员能用于某些场景，而普通成员不能

如我们所见，静态成员独立于任何对象。因此，在某些非静态数据成员可能非法的场合，静态成员却可以正常地使用。举个例子，静态数据成员可以是不完全类型（参见 7.3.3 节，第 249 页）。特别的，静态数据成员的类型可以就是它所属的类类型。而非静态数据成员则受到限制，只能声明成它所属类的指针或引用：

```

class Bar {
public:
    // ...
private:
    static Bar mem1;                // 正确：静态成员可以是不完全类型
    Bar *mem2;                     // 正确：指针成员可以是不完全类型
    Bar mem3;                      // 错误：数据成员必须是完全类型
};

```

静态成员和普通成员的另外一个区别是我们可以使用静态成员作为默认实参（参见 6.5.1 <304> 节，第 211 页）：

```

class Screen {
public:
    // bkground 表示一个在类中稍后定义的静态成员
    Screen& clear(char = bkground);
private:
    static const char bkground;
};

```

非静态数据成员不能作为默认实参，因为它的值本身属于对象的一部分，这么做的结果是无法真正提供一个对象以便从中获取成员的值，最终将引发错误。

7.6 节练习

练习 7.56：什么是类的静态成员？它有何优点？静态成员与普通成员有何区别？

练习 7.57：编写你自己的 Account 类。

练习 7.58：下面的静态数据成员的声明和定义有错误吗？请解释原因。

```
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

小结

< 305

类是 C++ 语言中最基本的特性。类允许我们为自己的应用定义新类型，从而使得程序更加简洁且易于修改。

类有两项基本能力：一是数据抽象，即定义数据成员和函数成员的能力；二是封装，即保护类的成员不被随意访问的能力。通过将类的实现细节设为 `private`，我们就能完成类的封装。类可以将其他类或者函数设为友元，这样它们就能访问类的非公有成员了。

类可以定义一种特殊的成员函数：构造函数，其作用是控制初始化对象的方式。构造函数可以重载，构造函数应该使用构造函数初始值列表来初始化所有数据成员。

类还能定义可变或者静态成员。一个可变成员永远都不会是 `const`，即使在 `const` 成员函数内也能修改它的值；一个静态成员可以是函数也可以是数据，静态成员存在于所有对象之外。

术语表

抽象数据类型 (abstract data type) 封装（隐藏）了实现细节的数据结构。

访问说明符 (access specifier) 包括关键字 `public` 和 `private`。用于定义成员对类的用户可见还是只对类的友元和成员可见。在类中说明符可以出现多次，每个说明符的有效范围从它自身开始，到下一个说明符为止。

聚合类 (aggregate class) 只含有公有成员的类，并且没有类内初始值或者构造函数。聚合类的成员可以用花括号括起来的初始值列表进行初始化。

类 (class) C++ 提供的自定义数据类型的机制。类可以包含数据、函数和类型成员。一个类定义一种新的类型和一个新的作用域。

类的声明 (class declaration) 首先是关键字 `class`（或者 `struct`），随后是类名以及分号。如果类已经声明而尚未定义，则它是一个不完全类型。

class 关键字 (class keyword) 用于定义类的关键字，默认情况下成员是 `private` 的。

类的作用域 (class scope) 每个类定义一个作用域。类作用域比其他作用域更加复

杂，类中定义的成员函数甚至有可能使用定义语句之后的名字。

常量成员函数 (const member function) 一个成员函数，在其中不能修改对象的普通（即既不是 `static` 也不是 `mutable`）数据成员。`const` 成员的 `this` 指针是指向常量的指针，通过区分函数是否是 `const` 可以进行重载。

构造函数 (constructor) 用于初始化对象的一种特殊的成员函数。构造函数应该给每个数据成员都赋一个合适的初始值。

构造函数初始值列表 (constructor initializer list) 说明一个类的数据成员的初始值，在构造函数体执行之前首先用初始值列表中的值初始化数据成员。未经初始值列表初始化的成员将被默认初始化。

转换构造函数 (converting constructor) 可以用一个实参调用的非显式构造函数。这样的函数隐式地将参数类型转换成类类型。

数据抽象 (data abstraction) 着重关注类型接口的一种编程技术。数据抽象令程序员可以忽略类型的实现细节，只关注类型执行的操作即可。数据抽象是面向对象编程和泛型编程的基础。

< 306

默认构造函数 (default constructor) 当没有提供任何实参时使用的构造函数。

委托构造函数 (delegating constructor) 委托构造函数的初始值列表只有一个入口，指定类的另一个构造函数执行初始化操作。

封装 (encapsulation) 分离类的实现与接口，从而隐藏了类的实现细节。在 C++ 语言中，通过把实现部分设为 `private` 完成封装的任务。

显式构造函数 (explicit constructor) 可以用一个单独的实参调用但是不能用于隐式转换的构造函数。通过在构造函数的声明之前加上 `explicit` 关键字就可以将其声明成显式构造函数。

前向声明 (forward declaration) 对尚未定义的名字的声明，通常用于表示位于类定义之前的类声明。参见“不完全类型”。

友元 (friend) 类向外部提供其非公有成员访问权限的一种机制。友元的访问权限与成员函数一样。友元可以是类，也可以是函数。

实现 (implementation) 类的成员（通常是私有的），定义了不希望为使用类类型的代码所用的数据及任何操作。

不完全类型 (incomplete type) 已经声明但是尚未定义的类型。不完全类型不能用于定义变量或者类的成员，但是用不完全类型定义指针或者引用是合法的。

接口 (interface) 类型提供的（公有）操作。通常情况下，接口不包含数据成员。

成员函数 (member function) 类的函数成员。普通的成员函数通过隐式的 `this` 指针与类的对象绑定在一起；静态成员函数不与对象绑定在一起也没有 `this` 指针。

成员函数可以重载，此时隐式的 `this` 指针参与函数匹配的过程。

可变数据成员 (mutable data member) 这种成员永远不是 `const`，即使它属于 `const` 对象。在 `const` 函数内可以修改可变数据成员。

名字查找 (name lookup) 根据名字的使用寻找匹配的声明的过程。

私有成员 (private member) 定义在 `private` 访问说明符之后的成员，只能被类的友元或者类的其他成员访问。数据成员以及仅供类本身使用而不作为接口的功能函数一般设为 `private`。

公有成员 (public member) 定义在 `public` 访问说明符之后的成员，可以被类的所有用户访问。通常情况下，只有实现类的接口的函数才被设为 `public`。

struct 关键字 (struct keyword) 用于定义类的关键字，默认情况下成员是 `public` 的。

合成默认构造函数 (synthesized default constructor) 对于没有显式地定义任何构造函数的类，编译器为其创建（合成）的默认构造函数。该构造函数检查类的数据成员，如果提供了类内初始值，就用它执行初始化操作；否则就对数据成员执行默认初始化。

this 指针 (this pointer) 是一个隐式的值，作为额外的实参传递给类的每个非静态成员函数。`this` 指针指向代表函数调用者的对象。

= default 一种语法形式，位于类内部默认构造函数声明语句的参数列表之后，要求编译器生成构造函数，而不管类是否已经有了其他构造函数。

第 II 部分

C++ 标准库

内容

第 8 章 IO 库.....	277
第 9 章 顺序容器.....	291
第 10 章 泛型算法	335
第 11 章 关联容器.....	373
第 12 章 动态内存	399

随着 C++ 版本的一次次修订，标准库也在不断成长。确实，新的 C++ 标准中有三分之二的文本都用来描述标准库。虽然我们不能深入讨论所有标准库设施，但有些核心库设施是每个 C++ 程序员都应该熟练掌握的，第二部分将介绍这些内容。

我们首先在第 8 章中介绍基本的 IO 库设施。除了使用标准库读写与控制台窗口相关的流之外，我们还将学习其他一些库类型，可以帮助我们读写命名文件以及完成到 `string` 对象的内存 IO 操作。

标准库的核心是很多容器类和一族泛型算法，这些设施能帮助我们编写简洁高效的程序。标准库会去关注那些簿记操作的细节，特别是内存管理，这样我们的程序就可以将全部注意力投入到需要求解的问题上。

我们在第 3 章中已经介绍了容器类型 `vector`，在第 9 章中将介绍更多 `vector` 相关的内容，这一章也会涉及其他顺序容器。我们还会介绍更多 `string` 类型所支持的操作，可以将 `string` 看作一种只包含字符元素的特殊容器。`string` 支持很多容器操作，但并不是全部。

第 10 章介绍泛型算法。这类算法通常在顺序容器一定范围内的元素上或其他类型的序列上进行操作。算法库为各种经典算法提供了高效的实现，如排序和搜索算法，还提供了其他一些常用操作。例如，标准库提供了 `copy` 算法，完成一个序列到另一个序列的元素拷贝；还提供了 `find` 算法，实现给定元素的查找，等等。泛型算法的通用性体现在两个层面：可应用于不同类型的序列；对序列中元素的类型限制小，大多数类型都是允许的。

标准库还提供了一些关联容器，第 11 章介绍这部分内容。关联容器中的元素是通过关键字来访问的。关联容器支持很多顺序容器的操作，也定义了一些自己特有的操作。

第 12 章是第二部分的最后一章，这一章介绍动态内存管理相关的一些语言特性和库设施。这一章介绍智能指针的一个标准版本，它是新标准库中最重要的类之一。通过使用智能指针，我们可以大幅度提高使用动态内存的代码的鲁棒性。这一章最后将给出一个较大的例子，使用了第 II 部分介绍的所有标准库设施。

第 8 章

IO 库

内容

8.1 IO 类	278
8.2 文件输入输出	283
8.3 string 流	287
小结	290
术语表	290

C++语言不直接处理输入输出，而是通过一族定义在标准库中的类型来处理 IO。这些类型支持从设备读取数据、向设备写入数据的 IO 操作，设备可以是文件、控制台窗口等。还有一些类型允许内存 IO，即，从 `string` 读取数据，向 `string` 写入数据。

IO 库定义了读写内置类型值的操作。此外，一些类，如 `string`，通常也会定义类似的 IO 操作，来读写自己的对象。

本章介绍 IO 库的基本内容。后续章节会介绍更多 IO 库的功能：第 14 章将会介绍如何编写自己的输入输出运算符，第 17 章将会介绍如何控制输出格式以及如何对文件进行随机访问。

310> 我们的程序已经使用了很多 IO 库设施了。我们在 1.2 节（第 5 页）已经介绍了大部分 IO 库设施：

- `istream`（输入流）类型，提供输入操作。
- `ostream`（输出流）类型，提供输出操作。
- `cin`, 一个 `istream` 对象，从标准输入读取数据。
- `cout`, 一个 `ostream` 对象，向标准输出写入数据。
- `cerr`, 一个 `ostream` 对象，通常用于输出程序错误消息，写入到标准错误。
- `>>` 运算符，用来从一个 `istream` 对象读取输入数据。
- `<<` 运算符，用来向一个 `ostream` 对象写入输出数据。
- `getline` 函数（参见 3.3.2 节，第 78 页），从一个给定的 `istream` 读取一行数据，存入一个给定的 `string` 对象中。

8.1 IO 类

到目前为止，我们已经使用过的 IO 类型和对象都是操纵 `char` 数据的。默认情况下，这些对象都是关联到用户的控制台窗口的。当然，我们不能限制实际应用程序仅从控制台窗口进行 IO 操作，应用程序常常需要读写命名文件。而且，使用 IO 操作处理 `string` 中的字符会很方便。此外，应用程序还可能读写需要宽字符支持的语言。

为了支持这些不同种类的 IO 处理操作，在 `istream` 和 `ostream` 之外，标准库还定义了其他一些 IO 类型，我们之前都已经使用过了。表 8.1 列出了这些类型，分别定义在三个独立的头文件中：`iostream` 定义了用于读写流的基本类型，`fstream` 定义了读写命名文件的类型，`sstream` 定义了读写内存 `string` 对象的类型。

表 8.1: IO 库类型和头文件

头文件	类型
<code>iostream</code>	<code>istream</code> , <code>wistream</code> 从流读取数据
	<code>ostream</code> , <code>wostream</code> 向流写入数据
	<code>iostream</code> , <code>wiostream</code> 读写流
<code>fstream</code>	<code>ifstream</code> , <code>wifstream</code> 从文件读取数据
	<code>ofstream</code> , <code>wofstream</code> 向文件写入数据
	<code>fstream</code> , <code>wfstream</code> 读写文件
<code>sstream</code>	<code>istringstream</code> , <code>wistringstream</code> 从 <code>string</code> 读取数据
	<code>ostringstream</code> , <code>wostringstream</code> 向 <code>string</code> 写入数据
	<code>stringstream</code> , <code>wstringstream</code> 读写 <code>string</code>

311> 为了支持使用宽字符的语言，标准库定义了一组类型和对象来操纵 `wchar_t` 类型的数据（参见 2.1.1 节，第 30 页）。宽字符版本的类型和函数的名字以一个 `w` 开始。例如，`wcin`、`wcout` 和 `wcerr` 是分别对应 `cin`、`cout` 和 `cerr` 的宽字符版对象。宽字符版本的类型和对象与其对应的普通 `char` 版本的类型定义在同一个头文件中。例如，头文件 `fstream` 定义了 `ifstream` 和 `wifstream` 类型。

IO 类型间的关系

概念上，设备类型和字符大小都不会影响我们要执行的 IO 操作。例如，我们可以用 `>>` 读取数据，而不用管是从一个控制台窗口，一个磁盘文件，还是一个 `string` 读取。类似的，我们也不用管读取的字符能存入一个 `char` 对象内，还是需要一个 `wchar_t` 对象来存储。

标准库使我们能忽略这些不同类型的流之间的差异，这是通过继承机制（inheritance）实现的。利用模板（参见 3.3 节，第 87 页），我们可以使用具有继承关系的类，而不必了解继承机制如何工作的细节。我们将在第 15 章和 18.3 节（第 710 页）介绍 C++ 是如何支持继承机制的。

简单地说，继承机制使我们可以声明一个特定的类继承自另一个类。我们通常可以将一个派生类（继承类）对象当作其基类（所继承的类）对象来使用。

类型 `ifstream` 和 `istringstream` 都继承自 `istream`。因此，我们可以像使用 `istream` 对象一样来使用 `ifstream` 和 `istringstream` 对象。也就是说，我们是如何使用 `cin` 的，就可以同样地使用这些类型的对象。例如，可以对一个 `ifstream` 或 `istringstream` 对象调用 `getline`，也可以使用 `>>` 从一个 `ifstream` 或 `istringstream` 对象中读取数据。类似的，类型 `ofstream` 和 `ostringstream` 都继承自 `ostream`。因此，我们是如何使用 `cout` 的，就可以同样地使用这些类型的对象。



本节剩下部分所介绍的标准库流特性都可以无差别地应用于普通流、文件流和 `string` 流，以及 `char` 或宽字符流版本。

8.1.1 IO 对象无拷贝或赋值



如我们在 7.1.3 节（第 234 页）所见，我们不能拷贝或对 IO 对象赋值：

```
ofstream out1, out2;
out1 = out2;                                // 错误：不能对流对象赋值
ofstream print(ofstream);                    // 错误：不能初始化 ofstream 参数
out2 = print(out2);                          // 错误：不能拷贝流对象
```

由于不能拷贝 IO 对象，因此我们也不能将形参或返回类型设置为流类型（参见 6.2.1 节，第 188 页）。进行 IO 操作的函数通常以引用方式传递和返回流。读写一个 IO 对象会改变其状态，因此传递和返回的引用不能是 `const` 的。

8.1.2 条件状态

312

IO 操作一个与生俱来的问题是可能发生错误。一些错误是可恢复的，而其他错误则发生在系统深处，已经超出了应用程序可以修正的范围。表 8.2 列出了 IO 类所定义的一些函数和标志，可以帮助我们访问和操纵流的条件状态（condition state）。

表 8.2: IO 库条件状态

<code>strm::iostate</code>	<code>strm</code> 是一种 IO 类型，在表 8.1（第 278 页）中已列出。 <code>iostate</code> 是一种机器相关的类型，提供了表达条件状态的完整功能
<code>strm::badbit</code>	<code>strm::badbit</code> 用来指出流已崩溃
<code>strm::failbit</code>	<code>strm::failbit</code> 用来指出一个 IO 操作失败了

续表

<code>strm::eofbit</code>	<code>strm::eofbit</code> 用来指出流到达了文件结束
<code>strm::goodbit</code>	<code>strm::goodbit</code> 用来指出流未处于错误状态。此值保证为零
<code>s.eof()</code>	若流 s 的 <code>eofbit</code> 置位，则返回 <code>true</code>
<code>s.fail()</code>	若流 s 的 <code>failbit</code> 或 <code>badbit</code> 置位，则返回 <code>true</code>
<code>s.bad()</code>	若流 s 的 <code>badbit</code> 置位，则返回 <code>true</code>
<code>s.good()</code>	若流 s 处于有效状态，则返回 <code>true</code>
<code>s.clear()</code>	将流 s 中所有条件状态位复位，将流的状态设置为有效。返回 <code>void</code>
<code>s.clear(flags)</code>	根据给定的 <code>flags</code> 标志位，将流 s 中对应条件状态位复位。 <code>flags</code> 的类型为 <code>strm::iostate</code> 。返回 <code>void</code>
<code>s.setstate(flags)</code>	根据给定的 <code>flags</code> 标志位，将流 s 中对应条件状态位置位。 <code>flags</code> 的类型为 <code>strm::iostate</code> 。返回 <code>void</code>
<code>s.rdbuf()</code>	返回流 s 的当前条件状态，返回值类型为 <code>strm::iostate</code>

下面是一个 IO 错误的例子：

```
int ival;
cin >> ival;
```

如果我们在标准输入上键入 Boo，读操作就会失败。代码中的输入运算符期待读取一个 `int`，但却得到了一个字符 B。这样，`cin` 会进入错误状态。类似的，如果我们输入一个文件结束标识，`cin` 也会进入错误状态。

一个流一旦发生错误，其上后续的 IO 操作都会失败。只有当一个流处于无错状态时，我们才可以从它读取数据，向它写入数据。由于流可能处于错误状态，因此代码通常应该在使用一个流之前检查它是否处于良好状态。确定一个流对象的状态的最简单的方法是将它当作一个条件来使用：

```
while (cin >> word)
    // ok: 读操作成功.....
```

`while` 循环检查 `>>` 表达式返回的流的状态。如果输入操作成功，流保持有效状态，则条件为真。

查询流的状态

将流作为条件使用，只能告诉我们流是否有效，而无法告诉我们具体发生了什么。有时我们也需要知道流为什么失败。例如，在键入文件结束标识后我们的应对措施，可能与遇到一个 IO 设备错误的处理方式是不同的。

IO 库定义了一个与机器无关的 `iostate` 类型，它提供了表达流状态的完整功能。这个类型应作为一个位集合来使用，使用方式与我们在 4.8 节中（第 137 页）使用 `quiz1` 的方式一样。IO 库定义了 4 个 `iostate` 类型的 `constexpr` 值（参见 2.4.4 节，第 58 页），表示特定的位模式。这些值用来表示特定类型的 IO 条件，可以与位运算符（参见 4.8 节，第 137 页）一起使用来一次性检测或设置多个标志位。

`badbit` 表示系统级错误，如不可恢复的读写错误。通常情况下，一旦 `badbit` 被置位，流就无法再使用了。在发生可恢复错误后，`failbit` 被置位，如期望读取数值却读出一个字符等错误。这种问题通常是可以修正的，流还可以继续使用。如果到达文件结束位置，`eofbit` 和 `failbit` 都会被置位。`goodbit` 的值为 0，表示流未发生错误。如果 `badbit`、`failbit` 和 `eofbit` 任一个被置位，则检测流状态的条件会失败。

标准库还定义了一组函数来查询这些标志位的状态。操作 `good` 在所有错误位均未置位的情况下返回 `true`，而 `bad`、`fail` 和 `eof` 则在对应错误位被置位时返回 `true`。此外，在 `badbit` 被置位时，`fail` 也会返回 `true`。这意味着，使用 `good` 或 `fail` 是确定流的总体状态的正确方法。实际上，我们将流当作条件使用的代码就等价于 `!fail()`。而 `eof` 和 `bad` 操作只能表示特定的错误。

< 313

管理条件状态

流对象的 `rdstate` 成员返回一个 `iostate` 值，对应流的当前状态。`setstate` 操作将给定条件位置位，表示发生了对应错误。`clear` 成员是一个重载的成员（参见 6.4 节，第 206 页）：它有一个不接受参数的版本，而另一个版本接受一个 `iostate` 类型的参数。

`clear` 不接受参数的版本清除（复位）所有错误标志位。执行 `clear()` 后，调用 `good` 会返回 `true`。我们可以这样使用这些成员：

```
// 记住 cin 的当前状态
auto old_state = cin.rdstate(); // 记住 cin 的当前状态
cin.clear(); // 使 cin 有效
process_input(cin); // 使用 cin
cin.setstate(old_state); // 将 cin 置为原有状态
```

带参数的 `clear` 版本接受一个 `iostate` 值，表示流的新状态。为了复位单一的条件状态位，我们首先用 `rdstate` 读出当前条件状态，然后用位操作将所需位复位来生成新的状态。例如，下面的代码将 `failbit` 和 `badbit` 复位，但保持 `eofbit` 不变。

< 314

```
// 复位 failbit 和 badbit，保持其他标志位不变
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

8.1.2 节练习

练习 8.1：编写函数，接受一个 `istream&` 参数，返回值类型也是 `istream&`。此函数须从给定流中读取数据，直至遇到文件结束标识时停止。它将读取的数据打印在标准输出上。完成这些操作后，在返回流之前，对流进行复位，使其处于有效状态。

练习 8.2：测试函数，调用参数为 `cin`。

练习 8.3：什么情况下，下面的 `while` 循环会终止？

```
while (cin >> i) /* ... */
```

8.1.3 管理输出缓冲

每个输出流都管理一个缓冲区，用来保存程序读写的数据。例如，如果执行下面的代码

```
os << "please enter a value: ";
```

文本串可能立即打印出来，但也有可能被操作系统保存在缓冲区中，随后再打印。有了缓冲机制，操作系统就可以将程序的多个输出操作组合成单一的系统级写操作。由于设备的写操作可能很耗时，允许操作系统将多个输出操作组合为单一的设备写操作可以带来很大的性能提升。

导致缓冲刷新（即，数据真正写到输出设备或文件）的原因有很多：

- 程序正常结束，作为 `main` 函数的 `return` 操作的一部分，缓冲刷新被执行。

- 缓冲区满时，需要刷新缓冲，而后新的数据才能继续写入缓冲区。
- 我们可以使用操纵符如 `endl`（参见 1.2 节，第 6 页）来显式刷新缓冲区。
- 在每个输出操作之后，我们可以用操纵符 `unitbuf` 设置流的内部状态，来清空缓冲区。默认情况下，对 `cerr` 是设置 `unitbuf` 的，因此写到 `cerr` 的内容都是立即刷新的。
- 一个输出流可能被关联到另一个流。在这种情况下，当读写被关联的流时，关联到的流的缓冲区会被刷新。例如，默认情况下，`cin` 和 `cerr` 都关联到 `cout`。因此，读 `cin` 或写 `cerr` 都会导致 `cout` 的缓冲区被刷新。

315 刷新输出缓冲区

我们已经使用过操纵符 `endl`，它完成换行并刷新缓冲区的工作。IO 库中还有两个类似的操纵符：`flush` 和 `ends`。`flush` 刷新缓冲区，但不输出任何额外的字符；`ends` 向缓冲区插入一个空字符，然后刷新缓冲区：

```
cout << "hi!" << endl;    // 输出 hi 和一个换行，然后刷新缓冲区
cout << "hi!" << flush;   // 输出 hi，然后刷新缓冲区，不附加任何额外字符
cout << "hi!" << ends;    // 输出 hi 和一个空字符，然后刷新缓冲区
```

unitbuf 操纵符

如果想在每次输出操作后都刷新缓冲区，我们可以使用 `unitbuf` 操纵符。它告诉流在接下来的每次写操作之后都进行一次 `flush` 操作。而 `nounitbuf` 操纵符则重置流，使其恢复使用正常的系统管理的缓冲区刷新机制：

```
cout << unitbuf;          // 所有输出操作后都会立即刷新缓冲区
// 任何输出都立即刷新，无缓冲
cout << nounitbuf;        // 回到正常的缓冲方式
```

警告：如果程序崩溃，输出缓冲区不会被刷新

如果程序异常终止，输出缓冲区是不会被刷新的。当一个程序崩溃后，它所输出的数据很可能停留在输出缓冲区中等待打印。

当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了。否则，可能将大量时间浪费在追踪代码为什么没有执行上，而实际上代码已经执行了，只是程序崩溃后缓冲区没有被刷新，输出数据被挂起没有打印而已。

关联输入和输出流

当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流。标准库将 `cout` 和 `cin` 关联在一起，因此下面语句

```
cin >> ival;
```

导致 `cout` 的缓冲区被刷新。



交互式系统通常应该关联输入流和输出流。这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。

`tie` 有两个重载的版本（参见 6.4 节，第 206 页）：一个版本不带参数，返回指向输

出流的指针。如果本对象当前关联到一个输出流，则返回的就是指向这个流的指针，如果对象未关联到流，则返回空指针。`tie` 的第二个版本接受一个指向 `ostream` 的指针，将自己关联到此 `ostream`。即，`x.tie(&o)` 将流 `x` 关联到输出流 `o`。

我们既可以将一个 `istream` 对象关联到另一个 `ostream`，也可以将一个 `ostream` 关联到另一个 `ostream`：

```
cin.tie(&cout);           // 仅仅是用来展示：标准库将 cin 和 cout 关联在一起
// old_tie 指向当前关联到 cin 的流（如果有的话）
ostream *old_tie = cin.tie(nullptr); // cin 不再与其他流关联
// 将 cin 与 cerr 关联；这不是一个好主意，因为 cin 应该关联到 cout
cin.tie(&cerr);          // 读取 cin 会刷新 cerr 而不是 cout
cin.tie(old_tie);        // 重建 cin 和 cout 间的正常关联
```

在这段代码中，为了将一个给定的流关联到一个新的输出流，我们将新流的指针传递给了 `tie`。为了彻底解开流的关联，我们传递了一个空指针。每个流同时最多关联到一个流，但多个流可以同时关联到同一个 `ostream`。

8.2 文件输入输出



头文件 `fstream` 定义了三个类型来支持文件 IO：`ifstream` 从一个给定文件读取数据，`ofstream` 向一个给定文件写入数据，以及 `fstream` 可以读写给定文件。在 17.5.3 节中（第 676 页）我们将介绍如何对同一个文件流既读又写。

这些类型提供的操作与我们之前已经使用过的对象 `cin` 和 `cout` 的操作一样。特别是，我们可以用 IO 运算符（`<<` 和 `>>`）来读写文件，可以用 `getline`（参见 3.2.2 节，第 79 页）从一个 `ifstream` 读取数据，包括 8.1 节中（第 278 页）介绍的内容也都适用于这些类型。

除了继承自 `iostream` 类型的行为之外，`fstream` 中定义的类型还增加了一些新的成员来管理与流关联的文件。在表 8.3 中列出了这些操作，我们可以对 `fstream`、`ifstream` 和 `ofstream` 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.3: `fstream` 特有的操作

<code>fstream fstrm;</code>	创建一个未绑定的文件流。 <code>fstream</code> 是头文件 <code>fstream</code> 中定义的一个类型
<code>fstream fstrm(s);</code>	创建一个 <code>fstream</code> ，并打开名为 <code>s</code> 的文件。 <code>s</code> 可以是 <code>string</code> 类型，或者是一个指向 C 风格字符串的指针（参见 3.5.4 节，第 109 页）。这些构造函数都是 <code>explicit</code> 的（参见 7.5.4 节，第 265 页）。默认的文件模式 <code>mode</code> 依赖于 <code>fstream</code> 的类型
<code>fstream fstrm(s, mode);</code>	与前一个构造函数类似，但按指定 <code>mode</code> 打开文件
<code>fstrm.open(s)</code>	打开名为 <code>s</code> 的文件，并将文件与 <code>fstrm</code> 绑定。 <code>s</code> 可以是一个 <code>string</code> 或一个指向 C 风格字符串的指针。默认的文件 <code>mode</code> 依赖于 <code>fstream</code> 的类型。返回 <code>void</code>
<code>fstrm.close()</code>	关闭与 <code>fstrm</code> 绑定的文件。返回 <code>void</code>
<code>fstrm.is_open()</code>	返回一个 <code>bool</code> 值，指出与 <code>fstrm</code> 关联的文件是否成功打开且尚未关闭

317 8.2.1 使用文件流对象



当我们想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来。每个文件流类都定义了一个名为 `open` 的成员函数，它完成一些系统相关的操作，来定位给定的文件，并视情况打开为读或写模式。

创建文件流对象时，我们可以提供文件名（可选的）。如果提供了一个文件名，则 `open` 会自动被调用：

```
ifstream in(ifile);           // 构造一个 ifstream 并打开给定文件
ofstream out;                 // 输出文件流未关联到任何文件
```

这段代码定义了一个输入流 `in`，它被初始化为从文件读取数据，文件名由 `string` 类型的参数 `ifile` 指定。第二条语句定义了一个输出流 `out`，未与任何文件关联。在新 C++ 标准中，文件名既可以是库类型 `string` 对象，也可以是 C 风格字符数组（参见 3.5.4 节，第 109 页）。旧版本的标准库只允许 C 风格字符数组。

C++ 11

用 `fstream` 代替 `iostream&`

我们在 8.1 节（第 279 页）已经提到过，在要求使用基类型对象的地方，我们可以用继承类型的对象来替代。这意味着，接受一个 `iostream` 类型引用（或指针）参数的函数，可以用一个对应的 `fstream`（或 `sstream`）类型来调用。也就是说，如果有一个函数接受一个 `ostream&` 参数，我们在调用这个函数时，可以传递给它一个 `ofstream` 对象，对 `istream&` 和 `ifstream` 也是类似的。

例如，我们可以用 7.1.3 节中的 `read` 和 `print` 函数来读写命名文件。在本例中，我们假定输入和输出文件的名字是通过传递给 `main` 函数的参数来指定的（参见 6.2.5 节，第 196 页）：

```
ifstream input(argv[1]);      // 打开销售记录文件
ofstream output(argv[2]);     // 打开输出文件
Sales_data total;             // 保存销售总额的变量
if (read(input, total)) {     // 读取第一条销售记录
    Sales_data trans;         // 保存下一条销售记录的变量
    while(read(input, trans)) { // 读取剩余记录
        if (total.isbn() == trans.isbn()) // 检查 isbn
            total.combine(trans);       // 更新销售总额
        else {
            print(output, total) << endl; // 打印结果
            total = trans;             // 处理下一本
        }
    }
    print(output, total) << endl; // 打印最后一本书的销售额
} else                         // 文件中无输入数据
    cerr << "No data?!" << endl;
```

除了读写的是命名文件外，这段程序与 229 页的加法程序几乎是完全相同的。重要的部分是对 `read` 和 `print` 的调用。虽然两个函数定义时指定的形参分别是 `istream&` 和 `ostream&`，但我们可以向它们传递 `fstream` 对象。

318 成员函数 `open` 和 `close`

如果我们定义了一个空文件流对象，可以随后调用 `open` 来将它与文件关联起来：

```
ifstream in(ifile);           // 构筑一个 ifstream 并打开给定文件
ofstream out;                 // 输出文件流未与任何文件相关联
out.open(ifile + ".copy");    // 打开指定文件
```

如果调用 `open` 失败, `failbit` 会被置位(参见 8.1.2 节, 第 280 页)。因为调用 `open` 可能失败, 进行 `open` 是否成功的检测通常是一个好习惯:

```
if (out)      // 检查 open 是否成功
    // open 成功, 我们可以使用文件了
```

这个条件判断与我们之前将 `cin` 用作条件相似。如果 `open` 失败, 条件会为假, 我们就不会去使用 `out` 了。

一旦一个文件流已经打开, 它就保持与对应文件的关联。实际上, 对一个已经打开的文件流调用 `open` 会失败, 并会导致 `failbit` 被置位。随后的试图使用文件流的操作都会失败。为了将文件流关联到另外一个文件, 必须首先关闭已经关联的文件。一旦文件成功关闭, 我们可以打开新的文件:

```
in.close();                // 关闭文件
in.open(ifile + "2");      // 打开另一个文件
```

如果 `open` 成功, 则 `open` 会设置流的状态, 使得 `good()` 为 `true`。

自动构造和析构

考虑这样一个程序, 它的 `main` 函数接受一个要处理的文件列表(参见 6.2.5 节, 第 196 页)。这种程序可能会有如下的循环:

```
// 对每个传递给程序的文件执行循环操作
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p); // 创建输出流并打开文件
    if (input) {         // 如果文件打开成功, “处理”此文件
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} // 每个循环步 input 都会离开作用域, 因此会被销毁
```

每个循环步构造一个新的名为 `input` 的 `ifstream` 对象, 并打开它来读取给定的文件。像之前一样, 我们检查 `open` 是否成功。如果成功, 将文件传递给一个函数, 该函数负责读取并处理输入数据。如果 `open` 失败, 打印一条错误信息并继续处理下一个文件。

因为 `input` 是 `while` 循环的局部变量, 它在每个循环步中都要创建和销毁一次(参见 5.4.1 节, 第 165 页)。当一个 `fstream` 对象离开其作用域时, 与之关联的文件会自动关闭。在下一步循环中, `input` 会再次被创建。



当一个 `fstream` 对象被销毁时, `close` 会自动被调用。

8.2.1 节练习

练习 8.4: 编写函数, 以读模式打开一个文件, 将其内容读入到一个 `string` 的 `vector` 中, 将每一行作为一个独立的元素存于 `vector` 中。

练习 8.5: 重写上面的程序, 将每个单词作为一个独立的元素进行存储。

练习 8.6: 重写 7.1.1 节的书店程序(第 229 页), 从一个文件中读取交易记录。将文件名作为一个参数传递给 `main`(参见 6.2.5 节, 第 196 页)。



8.2.2 文件模式

每个流都有一个关联的文件模式 (file mode)，用来指出如何使用文件。表 8.4 列出了文件模式和它们的含义。

表 8.4：文件模式

in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行 IO

无论用哪种方式打开文件，我们都可以指定文件模式，调用 `open` 打开文件时可以，用一个文件名初始化流来隐式打开文件时也可以。指定文件模式有如下限制：

- 只可以对 `ofstream` 或 `fstream` 对象设定 `out` 模式。
- 只可以对 `ifstream` 或 `fstream` 对象设定 `in` 模式。
- 只有当 `out` 也被设定时才可设定 `trunc` 模式。
- 只要 `trunc` 没被设定，就可以设定 `app` 模式。在 `app` 模式下，即使没有显式指定 `out` 模式，文件也总是以输出方式被打开。
- 默认情况下，即使我们没有指定 `trunc`，以 `out` 模式打开的文件也会被截断。为了保留以 `out` 模式打开的文件的内容，我们必须同时指定 `app` 模式，这样只会将数据追加写到文件末尾；或者同时指定 `in` 模式，即打开文件同时进行读写操作（参见 17.5.3 节，第 676 页，将介绍对同一个文件既进行输入又进行输出的方法）。
- `ate` 和 `binary` 模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用。

每个文件流类型都定义了一个默认的文件模式，当我们未指定文件模式时，就使用此默认模式。与 `ifstream` 关联的文件默认以 `in` 模式打开；与 `ofstream` 关联的文件默认以 `out` 模式打开；与 `fstream` 关联的文件默认以 `in` 和 `out` 模式打开。

320 以 `out` 模式打开文件会丢弃已有数据

默认情况下，当我们打开一个 `ofstream` 时，文件的内容会被丢弃。阻止一个 `ofstream` 清空给定文件内容的方法是同时指定 `app` 模式：

```
// 在这几条语句中, file1 都被截断
ofstream out("file1"); // 隐含以输出模式打开文件并截断文件
ofstream out2("file1", ofstream::out); // 隐含地截断文件
ofstream out3("file1", ofstream::out | ofstream::trunc);
// 为了保留文件内容, 我们必须显式指定 app 模式
ofstream app("file2", ofstream::app); // 隐含为输出模式
ofstream app2("file2", ofstream::out | ofstream::app);
```



保留被 `ofstream` 打开的文件中已有数据的唯一方法是显式指定 `app` 或 `in` 模式。

每次调用 open 时都会确定文件模式

对于一个给定流，每当打开文件时，都可以改变其文件模式。

```
ofstream out; // 未指定文件打开模式
out.open("scratchpad"); // 模式隐含设置为输出和截断
out.close(); // 关闭 out，以便我们将其用于其他文件
out.open("precious", ofstream::app); // 模式为输出和追加
out.close();
```

第一个 open 调用未显式指定输出模式，文件隐式地以 out 模式打开。通常情况下，out 模式意味着同时使用 trunc 模式。因此，当前目录下名为 scratchpad 的文件的内容将被清空。当打开名为 precious 的文件时，我们指定了 append 模式。文件中已有的数据都得以保留，所有写操作都在文件末尾进行。



在每次打开文件时，都要设置文件模式，可能是显式地设置，也可能是隐式地设置。当程序未指定模式时，就使用默认值。

8.2.2 节练习

练习 8.7：修改上一节的书店程序，将结果保存到一个文件中。将输出文件名作为第二个参数传递给 main 函数。

练习 8.8：修改上一题的程序，将结果追加到给定的文件末尾。对同一个输出文件，运行程序至少两次，检验数据是否得以保留。

8.3 string 流

< 321

sstream 头文件定义了三个类型来支持内存 IO，这些类型可以向 string 写入数据，从 string 读取数据，就像 string 是一个 IO 流一样。

istringstream 从 string 读取数据，**ostringstream** 向 string 写入数据，而头文件 **stringstream** 既可从 string 读数据也可向 string 写数据。与 fstream 类型类似，头文件 sstream 中定义的类型都继承自我们已经使用过的 iostream 头文件中定义的类型。除了继承得来的操作，sstream 中定义的类型还增加了一些成员来管理与流相关联的 string。表 8.5 列出了这些操作，可以对 stringstream 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.5: stringstream 特有的操作

<code>sstream strm;</code>	<code>strm</code> 是一个未绑定的 <code>stringstream</code> 对象。 <code>sstream</code> 是头文件 <code>sstream</code> 中定义的一个类型
<code>sstream strm(s);</code>	<code>strm</code> 是一个 <code>sstream</code> 对象，保存 <code>string s</code> 的一个拷贝。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
<code>strm.str()</code>	返回 <code>strm</code> 所保存的 <code>string</code> 的拷贝
<code>strm.str(s)</code>	将 <code>string s</code> 拷贝到 <code>strm</code> 中。返回 <code>void</code>

8.3.1 使用 istringstream

当我们的某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词

时，通常可以使用 `istringstream`。

考虑这样一个例子，假定有一个文件，列出了一些人和他们的电话号码。某些人只有一个号码，而另一些人则有多个——家庭电话、工作电话、移动电话等。我们的输入文件看起来可能是这样的：

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

文件中每条记录都以一个人名开始，后面跟随一个或多个电话号码。我们首先定义一个简单的类来描述输入数据：

```
// 成员默认为公有；参见 7.2 节（第 240 页）
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

类型 `PersonInfo` 的对象会有一个成员来表示人名，还有一个 `vector` 来保存此人的所有电话号码。

322 我们的程序会读取数据文件，并创建一个 `PersonInfo` 的 `vector`。`vector` 中每个元素对应文件中的一条记录。我们在一个循环中处理输入数据，每个循环步读取一条记录，提取出一个人名和若干电话号码：

```
string line, word; // 分别保存来自输入的一行和单词
vector<PersonInfo> people; // 保存来自输入的所有记录
// 逐行从输入读取数据，直至 cin 遇到文件尾（或其他错误）
while (getline(cin, line)) {
    PersonInfo info; // 创建一个保存此记录数据的对象
    istringstream record(line); // 将记录绑定到刚读入的行
    record >> info.name; // 读取名字
    while (record >> word) // 读取电话号码
        info.phones.push_back(word); // 保持它们
    people.push_back(info); // 将此记录追加到 people 末尾
}
```

这里我们用 `getline` 从标准输入读取整条记录。如果 `getline` 调用成功，那么 `line` 中将保存着从输入文件而来的一条记录。在 `while` 中，我们定义了一个局部 `PersonInfo` 对象，来保存当前记录中的数据。

接下来我们将一个 `istringstream` 与刚刚读取的文本行进行绑定，这样就可以在此 `istringstream` 上使用输入运算符来读取当前记录中的每个元素。我们首先读取人名，随后用一个 `while` 循环读取此人的电话号码。

当读取完 `line` 中所有数据后，内层 `while` 循环就结束了。此循环的工作方式与前面章节中读取 `cin` 的循环很相似，不同之处是，此循环从一个 `string` 而不是标准输入读取数据。当 `string` 中的数据全部读出后，同样会触发“文件结束”信号，在 `record` 上的下一个输入操作会失败。

我们将刚刚处理好的 `PersonInfo` 追加到 `vector` 中，外层 `while` 循环的一个循环步就随之结束了。外层 `while` 循环会持续执行，直至遇到 `cin` 的文件结束标识。

8.3.1 节练习

练习 8.9: 使用你为 8.1.2 节（第 281 页）第一个练习所编写的函数打印一个 `istringstream` 对象的内容。

练习 8.10: 编写程序，将来自一个文件中的行保存在一个 `vector<string>` 中。然后使用一个 `istringstream` 从 `vector` 读取数据元素，每次读取一个单词。

练习 8.11: 本节的程序在外层 `while` 循环中定义了 `istringstream` 对象。如果 `record` 对象定义在循环之外，你需要对程序进行怎样的修改？重写程序，将 `record` 的定义移到 `while` 循环之外，验证你设想的修改方法是否正确。

练习 8.12: 我们为什么没有在 `PersonInfo` 中使用类内初始化？

8.3.2 使用 `ostringstream`

< 323

当我们逐步构造输出，希望最后一起打印时，`ostringstream` 是很有用的。例如，对上一节的例子，我们可能想逐个验证电话号码并改变其格式。如果所有号码都是有效的，我们希望输出一个新的文件，包含改变格式后的号码。对于那些无效的号码，我们不会将它们输出到新文件中，而是打印一条包含人名和无效号码的错误信息。

由于我们不希望输出有无效电话号码的人，因此对每个人，直到验证完所有电话号码后才可以进行输出操作。但是，我们可以先将输出内容“写入”到一个内存 `ostringstream` 中：

```
for (const auto &entry : people) { // 对 people 中每一项
    ostringstream formatted, badNums; // 每个循环步创建的对象
    for (const auto &nums : entry.phones) { // 对每个数
        if (!valid(nums)) {
            badNums << " " << nums; // 将数的字符串形式存入 badNums
        } else
            // 将格式化的字符串“写入” formatted
            formatted << " " << format(nums);
    }
    if (badNums.str().empty()) // 没有错误的数
        os << entry.name << " "
        << formatted.str() << endl; // 打印名字 和格式化的数
    else // 否则，打印名字和错误的数
        cerr << "input error: " << entry.name
        << " invalid number(s) " << badNums.str() << endl;
}
```

在此程序中，我们假定已有两个函数，`valid` 和 `format`，分别完成电话号码验证和改变格式的功能。程序最有趣的部分是对字符串流 `formatted` 和 `badNums` 的使用。我们使用标准的输出运算符(`<<`)向这些对象写入数据，但这些“写入”操作实际上转换为 `string` 操作，分别向 `formatted` 和 `badNums` 中的 `string` 对象添加字符。

8.3.2 节练习

练习 8.13: 重写本节的电话号码程序，从一个命名文件而非 `cin` 读取数据。

练习 8.14: 我们为什么将 `entry` 和 `nums` 定义为 `const auto&`？

324 小结

C++ 使用标准库类来处理面向流的输入和输出：

- `iostream` 处理控制台 IO
- `fstream` 处理命名文件 IO
- `stringstream` 完成内存 `string` 的 IO

类 `fstream` 和 `stringstream` 都是继承自类 `iostream` 的。输入类都继承自 `istream`，输出类都继承自 `ostream`。因此，可以在 `istream` 对象上执行的操作，也可在 `ifstream` 或 `istringstream` 对象上执行。继承自 `ostream` 的输出类也有类似情况。

每个 IO 对象都维护一组条件状态，用来指出此对象上是否可以进行 IO 操作。如果遇到了错误——例如在输入流上遇到了文件末尾，则对象的状态变为失效，所有后续输入操作都不能执行，直至错误被纠正。标准库提供了一组函数，用来设置和检测这些状态。

术语表

条件状态 (condition state) 可被任何流类使用的一组标志和函数，用来指出给定流是否可用。

文件模式 (file mode) 类 `fstream` 定义的一组标志，在打开文件时指定，用来控制文件如何被使用。

文件流 (file stream) 用来读写命名文件的流对象。除了普通的 `iostream` 操作，文件流还定义了 `open` 和 `close` 成员。成员函数 `open` 接受一个 `string` 或一个 C 风格字符串参数，指定要打开的文件名，它还可以接受一个可选的参数，指明文件打开模式。成员函数 `close` 关闭流所关联的文件，调用 `close` 后才可以调用 `open` 打开另一个文件。

fstream 用于同时读写一个相同文件的文件流。默认情况下，`fstream` 以 `in` 和 `out` 模式打开文件。

ifstream 用于从输入文件读取数据的文件流。默认情况下，`ifstream` 以 `in` 模式打开文件。

继承 (inheritance) 程序设计功能，令一个类型可以从另一个类型继承接口。类 `ifstream` 和 `istringstream` 继承自 `istream`，`ofstream` 和 `ostringstream` 继承自 `ostream`。第 15 章将介绍继承。

istringstream 用来从给定 `string` 读取数据的字符串流。

ofstream 用来向输出文件写入数据的文件流。默认情况下，`ofstream` 以 `out` 模式打开文件。

字符串流 (string stream) 用于读写 `string` 的流对象。除了普通的 `iostream` 操作外，字符串流还定义了一个名为 `str` 的重载成员。调用 `str` 的无参版本会返回字符串流关联的 `string`。调用时传递给它一个 `string` 参数，则会将字符串流与该 `string` 的一个拷贝相关联。

stringstream 用于读写给定 `string` 的字符串流。

第 9 章

顺序容器

内容

9.1 顺序容器概述	292
9.2 容器库概览	294
9.3 顺序容器操作	305
9.4 vector 对象是如何增长的	317
9.5 额外的 string 操作	320
9.6 容器适配器	329
小结	332
术语表	332

本章是第 3 章内容的扩展，完成本章的学习后，对标准库顺序容器知识的掌握就完整了。元素在顺序容器中的顺序与其加入容器时的位置相对应。标准库还定义了几种关联容器，关联容器中元素的位置由元素相关联的关键字值决定。我们将在第 11 章中介绍关联容器特有的操作。

所有容器类都共享公共的接口，不同容器按不同方式对其进行扩展。这个公共接口使容器的学习更加容易——我们基于某种容器所学习的内容也都适用于其他容器。每种容器都提供了不同的性能和功能的权衡。

326

一个容器就是一些特定类型对象的集合。顺序容器（sequential container）为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。与之相对的，我们将在第 11 章介绍的有序和无序关联容器，则根据关键字的值来存储元素。

标准库还提供了三种容器适配器，分别为容器操作定义了不同的接口，来与容器类型适配。我们将在本章末尾介绍适配器。



本章的内容基于 3.2 节、3.3 节和 3.4 节中已经介绍的有关容器的知识，我们假定读者已经熟悉了这几节的内容。



9.1 顺序容器概述

表 9.1 列出了标准库中的顺序容器，所有顺序容器都提供了快速顺序访问元素的能力。但是，这些容器在以下方面都有不同的性能折中：

- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

表 9.1：顺序容器类型

<code>vector</code>	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢
<code>deque</code>	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快
<code>list</code>	双向链表。只支持双向顺序访问。在 <code>list</code> 中任何位置进行插入/删除操作速度都很快
<code>forward_list</code>	单向链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快
<code>array</code>	固定大小数组。支持快速随机访问。不能添加或删除元素
<code>string</code>	与 <code>vector</code> 相似的容器，但专门用于保存字符。随机访问快。在尾部插入/删除速度快

除了固定大小的 `array` 外，其他容器都提供高效、灵活的内存管理。我们可以添加和删除元素，扩张和收缩容器的大小。容器保存元素的策略对容器操作的效率有着固有的，有时是重大的影响。在某些情况下，存储策略还会影响特定容器是否支持特定操作。

327

例如，`string` 和 `vector` 将元素保存在连续的内存空间中。由于元素是连续存储的，由元素的下标来计算其地址是非常快速的。但是，在这两种容器的中间位置添加或删除元素就会非常耗时：在一次插入或删除操作后，需要移动插入/删除位置之后的所有元素，来保持连续存储。而且，添加一个元素有时可能还需要分配额外的存储空间。在这种情况下，每个元素都必须移动到新的存储空间中。

`list` 和 `forward_list` 两个容器的设计目的是令容器任何位置的添加和删除操作都很快。作为代价，这两个容器不支持元素的随机访问：为了访问一个元素，我们只能遍历整个容器。而且，与 `vector`、`deque` 和 `array` 相比，这两个容器的额外内存开销也很大。

`deque` 是一个更为复杂的数据结构。与 `string` 和 `vector` 类似，`deque` 支持快速

的随机访问。与 `string` 和 `vector` 一样，在 `deque` 的中间位置添加或删除元素的代价（可能）很高。但是，在 `deque` 的两端添加或删除元素都是很快的，与 `list` 或 `forward_list` 添加删除元素的速度相当。

`forward_list` 和 `array` 是新 C++ 标准增加的类型。与内置数组相比，`array` 是一种更安全、更容易使用的数组类型。与内置数组类似，`array` 对象的大小是固定的。因此，`array` 不支持添加和删除元素以及改变容器大小的操作。`forward_list` 的设计目标是达到与最好的手写的单向链表数据结构相当的性能。因此，`forward_list` 没有 `size` 操作，因为保存或计算其大小就会比手写链表多出额外的开销。对其他容器而言，`size` 保证是一个快速的常量时间的操作。

C++
11



新标准库的容器比旧版本快得多，原因我们将在 13.6 节（第 470 页）解释。新标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好（通常会更好）。现代 C++ 程序应该使用标准库容器，而不是更原始的数据结构，如内置数组。

确定使用哪种顺序容器



通常，使用 `vector` 是最好的选择，除非你有很好的理由选择其他容器。

以下是一些选择容器的基本原则：

- 除非你有很好的理由选择其他容器，否则应使用 `vector`。
- 如果你的程序有很多小的元素，且空间的额外开销很重要，则不要使用 `list` 或 `forward_list`。
- 如果程序要求随机访问元素，应使用 `vector` 或 `deque`。
- 如果程序要求在容器的中间插入或删除元素，应使用 `list` 或 `forward_list`。
- 如果程序需要在头尾位置插入或删除元素，但不会在中间位置进行插入或删除操作，则使用 `deque`。
- 如果程序只有在读取输入时才需要在容器中间位置插入元素，随后需要随机访问元素，则
 - 首先，确定是否真的需要在容器中间位置添加元素。当处理输入数据时，通常可以很容易地向 `vector` 追加数据，然后再调用标准库的 `sort` 函数（我们将在 10.2.3 节介绍 `sort`（第 343 页））来重排容器中的元素，从而避免在中间位置添加元素。
 - 如果必须在中间位置插入元素，考虑在输入阶段使用 `list`，一旦输入完成，将 `list` 中的内容拷贝到一个 `vector` 中。

328

如果程序既需要随机访问元素，又需要在容器中间位置插入元素，那该怎么办？答案取决于在 `list` 或 `forward_list` 中访问元素与 `vector` 或 `deque` 中插入/删除元素的相对性能。一般来说，应用中占主导地位的操作（执行的访问操作更多还是插入/删除更多）决定了容器类型的选择。在此情况下，对两种容器分别测试应用的性能可能就是必要的了。

Best
Practices

如果你不确定应该使用哪种容器，那么可以在程序中只使用 `vector` 和 `list` 公共的操作：使用迭代器，不使用下标操作，避免随机访问。这样，在必要时选择使用 `vector` 或 `list` 都很方便。

9.1 节练习

练习 9.1：对于下面的程序任务，`vector`、`deque` 和 `list` 哪种容器最为适合？解释你的选择的理由。如果没有哪一种容器优于其他容器，也请解释理由。

- (a) 读取固定数量的单词，将它们按字典序插入到容器中。我们将在下一章中看到，关联容器更适合这个问题。
- (b) 读取未知数量的单词，总是将新单词插入到末尾。删除操作在头部进行。
- (c) 从一个文件读取未知数量的整数。将这些数排序，然后将它们打印到标准输出。



9.2 容器库概览

容器类型上的操作形成了一种层次：

- 某些操作是所有容器类型都提供的（参见表 9.2，第 295 页）。
- 另外一些操作仅针对顺序容器（参见表 9.3，第 299 页）、关联容器（参见表 11.7，第 388 页）或无序容器（参见表 11.8，第 395 页）。
- 还有一些操作只适用于一小部分容器。

329 在本节中，我们将介绍对所有容器都适用的操作。本章剩余部分将聚焦于仅适用于顺序容器的操作。关联容器特有的操作将在第 11 章介绍。

一般来说，每个容器都定义在一个头文件中，文件名与类型名相同。即，`deque` 定义在头文件 `deque` 中，`list` 定义在头文件 `list` 中，以此类推。容器均定义为模板类（参见 3.3 节，第 86 页）。例如对 `vector`，我们必须提供额外信息来生成特定的容器类型。对大多数，但不是所有容器，我们还需要额外提供元素类型信息：

```
list<Sales_data>      // 保存 Sales_data 对象的 list
deque<double>          // 保存 double 的 deque
```

对容器可以保存的元素类型的限制

顺序容器几乎可以保存任意类型的元素。特别是，我们可以定义一个容器，其元素的类型是另一个容器。这种容器的定义与任何其他容器类型完全一样：在尖括号中指定元素类型（此种情况下，是另一种容器类型）：

```
vector<vector<string>> lines; // vector 的 vector
```

此处 `lines` 是一个 `vector`，其元素类型是 `string` 的 `vector`。



较旧的编译器可能需要在两个尖括号之间键入空格，例如，
`vector<vector<string>>`。

虽然我们可以在容器中保存几乎任何类型，但某些容器操作对元素类型有其自己的特殊要求。我们可以为不支持特定操作需求的类型定义容器，但这种情况下就只能使用那些没有特殊要求的容器操作了。

例如，顺序容器构造函数的一个版本接受容器大小参数（参见 3.3.1 节，第 88 页），它使用了元素类型的默认构造函数。但某些类没有默认构造函数。我们可以定义一个保存这种类型对象的容器，但我们在构造这种容器时不能只传递给它一个元素数目参数：

```
// 假定 noDefault 是一个没有默认构造函数的类型
vector<noDefault> v1(10, init);           // 正确：提供了元素初始化器
vector<noDefault> v2(10);                  // 错误：必须提供一个元素初始化器
```

当后面介绍容器操作时，我们还会注意到每个容器操作对元素类型的其他限制。

表 9.2: 容器操作

330

类型别名	
iterator	此容器类型的迭代器类型
const_iterator	可以读取元素，但不能修改元素的迭代器类型
size_type	无符号整数类型，足够保存此种容器类型最大可能容器的大小
difference_type	带符号整数类型，足够保存两个迭代器之间的距离
value_type	元素类型
reference	元素的左值类型；与 value_type&含义相同
const_reference	元素的 const 左值类型（即，const value_type&）
构造函数	
C c;	默认构造函数，构造空容器（array，参见第 301 页）
C c1(c2);	构造 c2 的拷贝 c1
C c(b, e);	构造 c，将迭代器 b 和 e 指定的范围内的元素拷贝到 c (array 不支持)
C c{a, b, c...};	列表初始化 c
赋值与 swap	
c1 = c2	将 c1 中的元素替换为 c2 中元素
c1 = {a, b, c...}	将 c1 中的元素替换为列表中元素（不适用于 array）
a.swap(b)	交换 a 和 b 的元素
swap(a, b)	与 a.swap(b) 等价
大小	
c.size()	c 中元素的数目（不支持 forward_list）
c.max_size()	c 可保存的最大元素数目
c.empty()	若 c 中存储了元素，返回 false，否则返回 true
添加/删除元素（不适用于 array）	
注：在不同容器中，这些操作的接口都不同	
c.insert(args)	将 args 中的元素拷贝进 c
c.emplace(init)	使用 init 构造 c 中的一个元素
c.erase(args)	删除 args 指定的元素
c.clear()	删除 c 中的所有元素，返回 void
关系运算符	
==, !=	所有容器都支持相等（不等）运算符
<, <=, >, >=	关系运算符（无序关联容器不支持）
获取迭代器	
c.begin(), c.end()	返回指向 c 的首元素和尾元素之后位置的迭代器
c.cbegin(), c.cend()	返回 const_iterator

续表

反向容器的额外成员（不支持 <code>forward_list</code> ）	
<code>reverse_iterator</code>	按逆序寻址元素的迭代器
<code>const_reverse_iterator</code>	不能修改元素的逆序迭代器
<code>c.rbegin(), c.rend()</code>	返回指向 <code>c</code> 的尾元素和首元素之前位置的迭代器
<code>c.rbegin(), c.rend()</code>	返回 <code>const_reverse_iterator</code>

9.2 节练习

练习 9.2：定义一个 `list` 对象，其元素类型是 `int` 的 `deque`。

331

9.2.1 迭代器



与容器一样，迭代器有着公共的接口：如果一个迭代器提供某个操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。例如，标准容器类型上的所有迭代器都允许我们访问容器中的元素，而所有迭代器都是通过解引用运算符来实现这个操作的。类似的，标准库容器的所有迭代器都定义了递增运算符，从当前元素移动到下一个元素。

表 3.6（第 96 页）列出了容器迭代器支持的所有操作，其中有一个例外不符合公共接口特点——`forward_list` 迭代器不支持递减运算符（`--`）。表 3.7（第 99 页）列出了迭代器支持的算术运算，这些运算只能应用于 `string`、`vector`、`deque` 和 `array` 的迭代器。我们不能将它们用于其他任何容器类型的迭代器。

迭代器范围



迭代器范围的概念是标准库的基础。

一个迭代器范围（`iterator range`）由一对迭代器表示，两个迭代器分别指向同一个容器中的元素或者是尾元素之后的位置（`one past the last element`）。这两个迭代器通常被称为 `begin` 和 `end`，或者是 `first` 和 `last`（可能有些误导），它们标记了容器中元素的一个范围。

虽然第二个迭代器常常被称为 `last`，但这种叫法有些误导，因为第二个迭代器从来都不会指向范围中的最后一个元素，而是指向尾元素之后的位置。迭代器范围中的元素包含 `first` 所表示的元素以及从 `first` 开始直至 `last`（但不包含 `last`）之间的所有元素。

这种元素范围被称为左闭合区间（`left-inclusive interval`），其标准数学描述为

`[begin, end)`

表示范围自 `begin` 开始，于 `end` 之前结束。迭代器 `begin` 和 `end` 必须指向相同的容器。`end` 可以与 `begin` 指向相同的位置，但不能指向 `begin` 之前的位置。

对构成范围的迭代器的要求

如果满足如下条件，两个迭代器 `begin` 和 `end` 构成一个迭代器范：

- 它们指向同一个容器中的元素，或者是容器最后一个元素之后的位置，且
- 我们可以通过反复递增 `begin` 来到达 `end`。换句话说，`end` 不在 `begin` 之前。



编译器不会强制这些要求。确保程序符合这些约定是程序员的责任。

使用左闭合范围蕴含的编程假定

标准库使用左闭合范围是因为这种范围有三种方便的性质。假定 `begin` 和 `end` 构成 [332](#) 一个合法的迭代器范围，则

- 如果 `begin` 与 `end` 相等，则范围为空
- 如果 `begin` 与 `end` 不等，则范围至少包含一个元素，且 `begin` 指向该范围中的第一个元素
- 我们可以对 `begin` 递增若干次，使得 `begin==end`

这些性质意味着我们可以像下面的代码一样用一个循环来处理一个元素范围，而这是安全的：

```
while (begin != end) {  
    *begin = val; // 正确：范围非空，因此 begin 指向一个元素  
    ++begin;      // 移动迭代器，获取下一个元素  
}
```

给定构成一个合法范围的迭代器 `begin` 和 `end`，若 `begin==end`，则范围为空。在此情况下，我们应该退出循环。如果范围不为空，`begin` 指向此非空范围的一个元素。因此，在 `while` 循环体中，可以安全地解引用 `begin`，因为 `begin` 必然指向一个元素。最后，由于每次循环对 `begin` 递增一次，我们确定循环最终会结束。

9.2.1 节练习

练习 9.3： 构成迭代器范围的迭代器有何限制？

练习 9.4： 编写函数，接受一对指向 `vector<int>` 的迭代器和一个 `int` 值。在两个迭代器指定的范围内查找给定的值，返回一个布尔值来指出是否找到。

练习 9.5： 重写上一题的函数，返回一个迭代器指向找到的元素。注意，程序必须处理未找到给定值的情况。

练习 9.6： 下面程序有何错误？你应该如何修改它？

```
list<int> lst1;  
list<int>::iterator iter1 = lst1.begin(),  
                     iter2 = lst1.end();  
while (iter1 < iter2) /* ... */
```

9.2.2 容器类型成员

每个容器都定义了多个类型，如表 9.2 所示（第 295 页）。我们已经使用过其中三种：`size_type`（参见 3.2.2 节，第 79 页）、`iterator` 和 `const_iterator`（参见 3.4.1 节，第 97 页）。

除了已经使用过的迭代器类型，大多数容器还提供反向迭代器。简单地说，反向迭代器就是一种反向遍历容器的迭代器，与正向迭代器相比，各种操作的含义也都发生了颠倒。例如，对一个反向迭代器执行 `++` 操作，会得到上一个元素。我们将在 10.4.3 节（第 363 页）

[333](#)

介绍更多关于反向迭代器的内容。

剩下的就是类型别名了，通过类型别名，我们可以在不了解容器中元素类型的情况下使用它。如果需要元素类型，可以使用容器的 `value_type`。如果需要元素类型的一个引用，可以使用 `reference` 或 `const_reference`。这些元素相关的类型别名在泛型编程中非常有用，我们将在 16 章中介绍相关内容。

为了使用这些类型，我们必须显式使用其类名：

```
// iter 是通过 list<string> 定义的一个迭代器类型
list<string>::iterator iter;
// count 是通过 vector<int> 定义的一个 difference_type 类型
vector<int>::difference_type count;
```

这些声明语句使用了作用域运算符（参见 1.2 节，第 7 页）来说明我们希望使用 `list<string>` 类的 `iterator` 成员及 `vector<int>` 类定义的 `difference_type`。

9.2.2 节练习

练习 9.7：为了索引 `int` 的 `vector` 中的元素，应该使用什么类型？

练习 9.8：为了读取 `string` 的 `list` 中的元素，应该使用什么类型？如果写入 `list`，又该使用什么类型？



9.2.3 begin 和 end 成员

`begin` 和 `end` 操作（参见 3.4.1 节，第 95 页）生成指向容器中第一个元素和尾元素之后位置的迭代器。这两个迭代器最常见的用途是形成一个包含容器中所有元素的迭代器范围。

如表 9.2（第 295 页）所示，`begin` 和 `end` 有多个版本：带 `r` 的版本返回反向迭代器（我们将在 10.4.3 节（第 363 页）中介绍相关内容）；以 `c` 开头的版本则返回 `const` 迭代器：

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin(); // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin(); // list<string>::const_reverse_iterator
```

不以 `c` 开头的函数都是被重载过的。也就是说，实际上有两个名为 `begin` 的成员。一个是 `const` 成员（参见 7.1.2 节，第 231 页），返回容器的 `const_iterator` 类型。另一个是非常量成员，返回容器的 `iterator` 类型。`rbegin`、`end` 和 `rend` 的情况类似。当我们对一个非常量对象调用这些成员时，得到的是返回 `iterator` 的版本。只有在对一个 `const` 对象调用这些函数时，才会得到一个 `const` 版本。与 `const` 指针和引用类似，可以将一个普通的 `iterator` 转换为对应的 `const_iterator`，但反之不行。

以 `c` 开头的版本是 C++ 新标准引入的，用以支持 `auto`（参见 2.5.2 节，第 61 页）与 `begin` 和 `end` 函数结合使用。过去，没有其他选择，只能显式声明希望使用哪种类型的迭代器：

```
// 显式指定类型
list<string>::iterator it5 = a.begin();
```

334

C++
11

```
list<string>::const_iterator it6 = a.begin();
// 是 iterator 还是 const_iterator 依赖于 a 的类型
auto it7 = a.begin(); // 仅当 a 是 const 时, it7 是 const_iterator
auto it8 = a.cbegin(); // it8 是 const_iterator
```

当 auto 与 begin 或 end 结合使用时, 获得的迭代器类型依赖于容器类型, 与我们想要如何使用迭代器毫不相干。但以 c 开头的版本还是可以获得 const_iterator 的, 而不管容器的类型是什么。



当不需要写访问时, 应使用 cbegin 和 cend。

9.2.3 节练习

练习 9.9: begin 和 cbegin 两个函数有什么不同?

练习 9.10: 下面 4 个对象分别是什么类型?

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

9.2.4 容器定义和初始化



每个容器类型都定义了一个默认构造函数 (参见 7.1.4 节, 第 236 页)。除 array 之外, 其他容器的默认构造函数都会创建一个指定类型的空容器, 且都可以接受指定容器大小和元素初始值的参数。

表 9.3: 容器定义和初始化

C c;	默认构造函数。如果 C 是一个 array, 则 c 中元素按默认方式初始化; 否则 c 为空
C c1(c2)	c1 初始化为 c2 的拷贝。c1 和 c2 必须是相同类型 (即, 它们必须是相同的容器类型, 且保存的是相同的元素类型; 对于 array 类型, 两者还必须具有相同大小)
C c{a, b, c...}	c 初始化为初始化列表中元素的拷贝。列表中元素的类型必须与 C 的元素类型相容。对于 array 类型, 列表中元素数目必须等于或小于 array 的大小, 任何遗漏的元素都进行值初始化 (参见 3.3.1 节, 第 88 页)
C c(b, e)	c 初始化为迭代器 b 和 e 指定范围中的元素的拷贝。范围内元素的类型必须与 c 的元素类型相容 (array 不适用)
只有顺序容器 (不包括 array) 的构造函数才能接受大小参数	
C seq(n)	seq 包含 n 个元素, 这些元素进行了值初始化; 此构造函数是 explicit 的 (参见 7.5.4 节, 第 265 页)。(string 不适用)
C seq(n, t)	seq 包含 n 个初始化为值 t 的元素

将一个容器初始化为另一个容器的拷贝

将一个新容器创建为另一个容器的拷贝的方法有两种: 可以直接拷贝整个容器, 或者

(array 除外) 拷贝由一个迭代器对指定的元素范围。

为了创建一个容器为另一个容器的拷贝, 两个容器的类型及其元素类型必须匹配。不过, 当传递迭代器参数来拷贝一个范围时, 就不要求容器类型是相同的了。而且, 新容器和原容器中的元素类型也可以不同, 只要能将要拷贝的元素转换 (参见 4.11 节, 第 141 页) 为要初始化的容器的元素类型即可。

```
335 // 每个容器有三个元素, 用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};

list<string> list2(authors);      // 正确: 类型匹配
deque<string> authList(authors); // 错误: 容器类型不匹配
vector<string> words(articles);  // 错误: 容器类型必须匹配
// 正确: 可以将 const char* 元素转换为 string
forward_list<string> words(articles.begin(), articles.end());
```



当将一个容器初始化为另一个容器的拷贝时, 两个容器的容器类型和元素类型都必须相同。

接受两个迭代器参数的构造函数用这两个迭代器表示我们想要拷贝的一个元素范围。与以往一样, 两个迭代器分别标记想要拷贝的第一个元素和尾元素之后的位置。新容器的大小与范围中元素的数目相同。新容器中的每个元素都用范围内对应元素的值进行初始化。

由于两个迭代器表示一个范围, 因此可以使用这种构造函数来拷贝一个容器中的子序列。例如, 假定迭代器 `it` 表示 `authors` 中的一个元素, 我们可以编写如下代码

```
// 拷贝元素, 直到 (但不包括) it 指向的元素
deque<string> authList(authors.begin(), it);
```

336> 列表初始化



在新标准中, 我们可以对一个容器进行列表初始化 (参见 3.3.1 节, 第 88 页)

```
// 每个容器有三个元素, 用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

当这样做时, 我们就显式地指定了容器中每个元素的值。对于除 array 之外的容器类型, 初始化列表还隐含地指定了容器的大小: 容器将包含与初始值一样多的元素。

与顺序容器大小相关的构造函数

除了与关联容器相同的构造函数外, 顺序容器 (array 除外) 还提供另一个构造函数, 它接受一个容器大小和一个 (可选的) 元素初始值。如果我们不提供元素初始值, 则标准库会创建一个值初始化器 (参见 3.3.1 节, 第 88 页):

```
vector<int> ivec(10, -1);           // 10 个 int 元素, 每个都初始化为 -1
list<string> svec(10, "hi!");       // 10 个 strings; 每个都初始化为 "hi!"
forward_list<int> ivec(10);         // 10 个元素, 每个都初始化为 0
deque<string> svec(10);            // 10 个元素, 每个都是空 string
```

如果元素类型是内置类型或者是具有默认构造函数（参见 9.2 节，第 294 页）的类类型，可以只为构造函数提供一个容器大小参数。如果元素类型没有默认构造函数，除了大小参数外，还必须指定一个显式的元素初始值。



只有顺序容器的构造函数才接受大小参数，关联容器并不支持。

标准库 array 具有固定大小

与内置数组一样，标准库 array 的大小也是类型的一部分。当定义一个 array 时，除了指定元素类型，还要指定容器大小：

```
array<int, 42>           // 类型为：保存 42 个 int 的数组  
array<string, 10>         // 类型为：保存 10 个 string 的数组
```

为了使用 array 类型，我们必须同时指定元素类型和大小：

```
array<int, 10>::size_type i;      // 数组类型包括元素类型和大小  
array<int>::size_type j;          // 错误：array<int>不是一个类型
```

由于大小是 array 类型的一部分，array 不支持普通的容器构造函数。这些构造函数都会确定容器的大小，要么隐式地，要么显式地。而允许用户向一个 array 构造函数传递大小参数，最好情况下也是多余的，而且容易出错。

array 大小固定的特性也影响了它所定义的构造函数的行为。与其他容器不同，一个默认构造的 array 是非空的：它包含了与其大小一样多的元素。这些元素都被默认初始化（参见 2.2.1 节，第 40 页），就像一个内置数组（参见 3.5.1 节，第 102 页）中的元素那样。如果我们对 array 进行列表初始化，初始值的数目必须等于或小于 array 的大小。如果初始值数目小于 array 的大小，则它们被用来初始化 array 中靠前的元素，所有剩余元素都会进行值初始化（参见 3.3.1 节，第 88 页）。在这两种情况下，如果元素类型是一个类类型，那么该类必须有一个默认构造函数，以使值初始化能够进行：

```
array<int, 10> ia1;           // 10 个默认初始化的 int  
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // 列表初始化  
array<int, 10> ia3 = {42};    // ia3[0] 为 42, 剩余元素为 0
```

值得注意的是，虽然我们不能对内置数组类型进行拷贝或对象赋值操作（参见 3.5.1 节，第 102 页），但 array 并无此限制：

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};  
int cpy[10] = digs;                 // 错误：内置数组不支持拷贝或赋值  
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};  
array<int, 10> copy = digits; // 正确：只要数组类型匹配即合法
```

与其他容器一样，array 也要求初始值的类型必须与要创建的容器类型相同。此外，array 还要求元素类型和大小也都一样，因为大小是 array 类型的一部分。

9.2.4 节练习

练习 9.11：对 6 种创建和初始化 vector 对象的方法，每一种都给出一个实例。解释每个 vector 包含什么值。

练习 9.12：对于接受一个容器创建其拷贝的构造函数，和接受两个迭代器创建拷贝的构造函数，解释它们的不同。

练习 9.13: 如何从一个 `list<int>` 初始化一个 `vector<double>`? 从一个 `vector<int>` 又该如何创建? 编写代码验证你的答案。

9.2.5 赋值和 swap

表 9.4 中列出的与赋值相关的运算符可用于所有容器。赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝:

```
c1 = c2;           // 将 c1 的内容替换为 c2 中元素的拷贝
c1 = {a,b,c};    // 赋值后, c1 大小为 3
```

第一个赋值运算后, 左边容器将与右边容器相等。如果两个容器原来大小不同, 赋值运算后两者的大小都与右边容器的原大小相同。第二个赋值运算后, `c1` 的 `size` 变为 3, 即花括号列表中值的数目。

338 与内置数组不同, 标准库 `array` 类型允许赋值。赋值号左右两边的运算对象必须具有相同的类型:

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // 所有元素值均为 0
a1 = a2; // 替换 a1 中的元素
a2 = {0}; // 错误: 不能将一个花括号列表赋予数组
```

由于右边运算对象的大小可能与左边运算对象的大小不同, 因此 `array` 类型不支持 `assign`, 也不允许用花括号包围的值列表进行赋值。

表 9.4: 容器赋值运算

<code>c1=c2</code>	将 <code>c1</code> 中的元素替换为 <code>c2</code> 中元素的拷贝。 <code>c1</code> 和 <code>c2</code> 必须具有相同的类型
<code>c={a,b,c...}</code>	将 <code>c1</code> 中元素替换为初始化列表中元素的拷贝 (<code>array</code> 不适用)
<code>swap(c1,c2)</code>	交换 <code>c1</code> 和 <code>c2</code> 中的元素。 <code>c1</code> 和 <code>c2</code> 必须具有相同的类型。 <code>swap</code> 通常比从 <code>c2</code> 向 <code>c1</code> 拷贝元素快得多
<code>assign</code> 操作不适用于关联容器和 <code>array</code>	
<code>seq.assign(b,e)</code>	将 <code>seq</code> 中的元素替换为迭代器 <code>b</code> 和 <code>e</code> 所表示的范围中的元素。迭代器 <code>b</code> 和 <code>e</code> 不能指向 <code>seq</code> 中的元素
<code>seq.assign(il)</code>	将 <code>seq</code> 中的元素替换为初始化列表 <code>il</code> 中的元素
<code>seq.assign(n,t)</code>	将 <code>seq</code> 中的元素替换为 <code>n</code> 个值为 <code>t</code> 的元素



赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而 `swap` 操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效 (容器类型为 `array` 和 `string` 的情况除外)。

使用 `assign` (仅顺序容器)

赋值运算符要求左边和右边的运算对象具有相同的类型。它将右边运算对象中所有元素拷贝到左边运算对象中。顺序容器 (`array` 除外) 还定义了一个名为 `assign` 的成员, 允许我们从一个不同但相容的类型赋值, 或者从容器的一个子序列赋值。`assign` 操作用参数所指定的元素 (的拷贝) 替换左边容器中的所有元素。例如, 我们可以用 `assgin` 实现将一个 `vector` 中的一段 `char *` 值赋予一个 `list` 中的 `string`:

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // 错误：容器类型不匹配
// 正确：可以将 const char* 转换为 string
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

这段代码中对 `assign` 的调用将 `names` 中的元素替换为迭代器指定的范围中的元素的拷贝。339 `assign` 的参数决定了容器中将有多少个元素以及它们的值都是什么。



由于其旧元素被替换，因此传递给 `assign` 的迭代器不能指向调用 `assign` 的容器。

`assign` 的第二个版本接受一个整型值和一个元素值。它用指定数目且具有相同给定值的元素替换容器中原有的元素：

```
// 等价于 slist1.clear();
// 后跟 slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1);           // 1 个元素，为空 string
slist1.assign(10, "Hiya!");      // 10 个元素，每个都是 "Hiya!"
```

使用 `swap`

`swap` 操作交换两个相同类型容器的内容。调用 `swap` 之后，两个容器中的元素将会交换：

```
vector<string> svec1(10); // 10 个元素的 vector
vector<string> svec2(24); // 24 个元素的 vector
swap(svec1, svec2);
```

调用 `swap` 后，`svec1` 将包含 24 个 `string` 元素，`svec2` 将包含 10 个 `string`。除 `array` 外，交换两个容器内容的操作保证会很快——元素本身并未交换，`swap` 只是交换了两个容器的内部数据结构。



除 `array` 外，`swap` 不对任何元素进行拷贝、删除或插入操作，因此可以保证在常数时间内完成。

元素不会被移动的事实意味着，除 `string` 外，指向容器的迭代器、引用和指针在 `swap` 操作之后都不会失效。它们仍指向 `swap` 操作之前所指向的那些元素。但是，在 `swap` 之后，这些元素已经属于不同的容器了。例如，假定 `iter` 在 `swap` 之前指向 `svec1[3]` 的 `string`，那么在 `swap` 之后它指向 `svec2[3]` 的元素。与其他容器不同，对一个 `string` 调用 `swap` 会导致迭代器、引用和指针失效。

与其他容器不同，`swap` 两个 `array` 会真正交换它们的元素。因此，交换两个 `array` 所需的时间与 `array` 中元素的数目成正比。

因此，对于 `array`，在 `swap` 操作之后，指针、引用和迭代器所绑定的元素保持不变，但元素值已经与另一个 `array` 中对应元素的值进行了交换。

在新标准库中，容器既提供成员函数版本的 `swap`，也提供非成员版本的 `swap`。而早期标准库版本只提供成员函数版本的 `swap`。非成员版本的 `swap` 在泛型编程中是非常重要的。统一使用非成员版本的 `swap` 是一个好习惯。

340

9.2.5 节练习

练习 9.14: 编写程序，将一个 `list` 中的 `char *` 指针（指向 C 风格字符串）元素赋值给一个 `vector` 中的 `string`。



9.2.6 容器大小操作

除了一个例外，每个容器类型都有三个与大小相关的操作。成员函数 `size`（参见 3.2.2 节，第 78 页）返回容器中元素的数目；`empty` 当 `size` 为 0 时返回布尔值 `true`，否则返回 `false`；`max_size` 返回一个大于或等于该类型容器所能容纳的最大元素数的值。`forward_list` 支持 `max_size` 和 `empty`，但不支持 `size`，原因我们将在下一节解释。

9.2.7 关系运算符

每个容器类型都支持相等运算符（`==` 和 `!=`）；除了无序关联容器外的所有容器都支持关系运算符（`>`、`>=`、`<`、`<=`）。关系运算符左右两边的运算对象必须是相同类型的容器，且必须保存相同类型的元素。即，我们只能将一个 `vector<int>` 与另一个 `vector<int>` 进行比较，而不能将一个 `vector<int>` 与一个 `list<int>` 或一个 `vector<double>` 进行比较。

比较两个容器实际上是进行元素的逐对比较。这些运算符的工作方式与 `string` 的关系运算（参见 3.2.2 节，第 79 页）类似：

- 如果两个容器具有相同大小且所有元素都两两对应相等，则这两个容器相等；否则两个容器不等。
- 如果两个容器大小不同，但较小容器中每个元素都等于较大容器中的对应元素，则较小容器小于较大容器。
- 如果两个容器都不是另一个容器的前缀子序列，则它们的比较结果取决于第一个不相等的元素的比较结果。

下面的例子展示了这些关系运算符是如何工作的：

```
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 和 v2 在元素[2]处不同：v1[2] 小于等于 v2[2]
v1 < v3 // false; 所有元素都相等，但 v3 中元素数目更少
v1 == v4 // true; 每个元素都相等，且 v1 和 v4 大小相同
v1 == v2 // false; v2 元素数目比 v1 少
```

341

容器的关系运算符使用元素的关系运算符完成比较



只有当其元素类型也定义了相应的比较运算符时，我们才可以使用关系运算符来比较两个容器。

容器的相等运算符实际上是使用元素的 `==` 运算符实现比较的，而其他关系运算符是使用元素的 `<` 运算符。如果元素类型不支持所需运算符，那么保存这种元素的容器就不能使用相应的关系运算。例如，我们在第 7 章中定义的 `Sales_data` 类型并未定义 `==` 和 `<` 运算。因此，就不能比较两个保存 `Sales_data` 元素的容器：

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // 错误: Sales_data 没有<运算符
```

9.2.7 节练习

练习 9.15: 编写程序，判定两个 `vector<int>` 是否相等。

练习 9.16: 重写上一题的程序，比较一个 `list<int>` 中的元素和一个 `vector<int>` 中的元素。

练习 9.17: 假定 `c1` 和 `c2` 是两个容器，下面的比较操作有何限制（如果有的话）？

```
if (c1 < c2)
```

9.3 顺序容器操作

顺序容器和关联容器的不同之处在于两者组织元素的方式。这些不同之处直接关系到了元素如何存储、访问、添加以及删除。上一节介绍了所有容器都支持的操作（罗列于表 9.2（第 295 页））。本章剩余部分将介绍顺序容器所特有的操作。

9.3.1 向顺序容器添加元素



除 `array` 外，所有标准库容器都提供灵活的内存管理。在运行时可以动态添加或删除元素来改变容器大小。表 9.5 列出了向顺序容器（非 `array`）添加元素的操作。

表 9.5: 向顺序容器添加元素的操作

这些操作会改变容器的大小；`array` 不支持这些操作。

`forward_list` 有自己专有的 `insert` 和 `emplace`；参见 9.3.4 节（第 312 页）。

`forward_list` 不支持 `push_back` 和 `emplace_back`。

`vector` 和 `string` 不支持 `push_front` 和 `emplace_front`。

<code>c.push_back(t)</code>	在 <code>c</code> 的尾部创建一个值为 <code>t</code> 或由 <code>args</code> 创建的元素。返回 <code>void</code>
<code>c.emplace_back(args)</code>	

<code>c.push_front(t)</code>	在 <code>c</code> 的头部创建一个值为 <code>t</code> 或由 <code>args</code> 创建的元素。返回 <code>void</code>
<code>c.emplace_front(args)</code>	

<code>c.insert(p, t)</code>	在迭代器 <code>p</code> 指向的元素之前创建一个值为 <code>t</code> 或由 <code>args</code> 创建的元素。返回指向新添加的元素的迭代器
<code>c.emplace(p, args)</code>	

<code>c.insert(p, n, t)</code>	在迭代器 <code>p</code> 指向的元素之前插入 <code>n</code> 个值为 <code>t</code> 的元素。返回指向新添加的第一个元素的迭代器；若 <code>n</code> 为 0，则返回 <code>p</code>

<code>c.insert(p, b, e)</code>	将迭代器 <code>b</code> 和 <code>e</code> 指定的范围内的元素插入到迭代器 <code>p</code> 指向的元素之前。 <code>b</code> 和 <code>e</code> 不能指向 <code>c</code> 中的元素。返回指向新添加的第一个元素的迭代器；若范围为空，则返回 <code>p</code>

<code>c.insert(p, il)</code>	<code>il</code> 是一个花括号包围的元素值列表。将这些给定值插入到迭代器 <code>p</code> 指向的元素之前。返回指向新添加的第一个元素的迭代器；若列表为空，则返回 <code>p</code>



向一个 `vector`、`string` 或 `deque` 插入元素会使所有指向容器的迭代器、引用和指针失效。

当我们使用这些操作时，必须记得不同容器使用不同的策略来分配元素空间，而这些策略直接影响性能。在一个 `vector` 或 `string` 的尾部之外的任何位置，或是一个 `deque` 的首尾之外的任何位置添加元素，都需要移动元素。而且，向一个 `vector` 或 `string` 添加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移动到新的空间中。

342→

使用 `push_back`

在 3.3.2 节（第 90 页）中，我们看到 `push_back` 将一个元素追加到一个 `vector` 的尾部。除 `array` 和 `forward_list` 之外，每个顺序容器（包括 `string` 类型）都支持 `push_back`。

例如，下面的循环每次读取一个 `string` 到 `word` 中，然后追加到容器尾部：

```
// 从标准输入读取数据，将每个单词放到容器末尾
string word;
while (cin >> word)
    container.push_back(word);
```

对 `push_back` 的调用在 `container` 尾部创建了一个新的元素，将 `container` 的 `size` 增大了 1。该元素的值为 `word` 的一个拷贝。`container` 的类型可以是 `list`、`vector` 或 `deque`。

由于 `string` 是一个字符容器，我们也可以用 `push_back` 在 `string` 末尾添加字符：

```
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s'); // 等价于 word += 's'
}
```

关键概念：容器元素是拷贝

当我们用一个对象来初始化容器时，或将一个对象插入到容器中时，实际上放入到容器中的是对象值的一个拷贝，而不是对象本身。就像我们将一个对象传递给非引用参数（参见 3.2.2 节，第 79 页）一样，容器中的元素与提供值的对象之间没有任何关联。随后对容器中元素的任何改变都不会影响到原始对象，反之亦然。

使用 `push_front`

除了 `push_back`，`list`、`forward_list` 和 `deque` 容器还支持名为 `push_front` 的类似操作。此操作将元素插入到容器头部：

```
list<int> ilist;
// 将元素添加到 ilist 开头
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

此循环将元素 0、1、2、3 添加到 `ilist` 头部。每个元素都插入到 `list` 的新的开始位置（new beginning）。即，当我们插入 1 时，它会被放置在 0 之前，2 被放置在 1 之前，依此类推。因此，在循环中以这种方式将元素添加到容器中，最终会形成逆序。在循环执行完毕后，`ilist` 保存序列 3、2、1、0。

343→

注意，`deque` 像 `vector` 一样提供了随机访问元素的能力，但它提供了 `vector` 所

不支持的 `push_front`。`deque` 保证在容器首尾进行插入和删除元素的操作都只花费常数时间。与 `vector` 一样，在 `deque` 首尾之外的位置插入元素会很耗时。

在容器中的特定位置添加元素

`push_back` 和 `push_front` 操作提供了一种方便地在顺序容器尾部或头部插入单个元素的方法。`insert` 成员提供了更一般的添加功能，它允许我们在容器中任意位置插入 0 个或多个元素。`vector`、`deque`、`list` 和 `string` 都支持 `insert` 成员。`forward_list` 提供了特殊版本的 `insert` 成员，我们将在 9.3.4 节（第 312 页）中介绍。

每个 `insert` 函数都接受一个迭代器作为其第一个参数。迭代器指出了在容器中什么位置放置新元素。它可以指向容器中任何位置，包括容器尾部之后的下一个位置。由于迭代器可能指向容器尾部之后不存在的元素的位置，而且在容器开始位置插入元素是很有用的功能，所以 `insert` 函数将元素插入到迭代器所指定的位置之前。例如，下面的语句

```
clist.insert(iter, "Hello!"); // 将"Hello!"添加到 iter 之前的位置
```

将一个值为 "Hello" 的 `string` 插入到 `iter` 指向的元素之前的位置。

虽然某些容器不支持 `push_front` 操作，但它们对于 `insert` 操作并无类似的限制（插入开始位置）。因此我们可以将元素插入到容器的开始位置，而不必担心容器是否支持 `push_front`：

```
vector<string> svec;
list<string> slist;

// 等价于调用 slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");

// vector 不支持 push_front，但我们可以插入到 begin() 之前
// 警告：插入到 vector 末尾之外的任何位置都可能很慢
svec.insert(svec.begin(), "Hello!");
```



将元素插入到 `vector`、`deque` 和 `string` 中的任何位置都是合法的。然而，这样做可能很耗时。

插入范围内元素

除了第一个迭代器参数之外，`insert` 函数还可以接受更多的参数，这与容器构造函数类似。其中一个版本接受一个元素数目和一个值，它将指定数量的元素添加到指定位置之前，这些元素都按给定值初始化：

```
svec.insert(svec.end(), 10, "Anna");
```

这行代码将 10 个元素插入到 `svec` 的末尾，并将所有元素都初始化为 `string` "Anna"。

接受一对迭代器或一个初始化列表的 `insert` 版本将给定范围中的元素插入到指定位置之前：

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// 将 v 的最后两个元素添加到 slist 的开始位置
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：迭代器表示要拷贝的范围，不能指向与目的位置相同的容器
slist.insert(slist.begin(), slist.begin(), slist.end());
```

如果我们传递给 `insert` 一对迭代器，它们不能指向添加元素的目标容器。

在新标准下，接受元素个数或范围的 `insert` 版本返回指向第一个新加入元素的迭代器。(在旧版本的标准库中，这些操作返回 `void`。)如果范围为空，不插入任何元素，`insert` 操作会将第一个参数返回。

345 使用 `insert` 的返回值

通过使用 `insert` 的返回值，可以在容器中一个特定位置反复插入元素：

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // 等价于调用 push_front
```



理解这个循环是如何工作的非常重要，特别是理解这个循环为什么等价于调用 `push_front` 尤为重要。

在循环之前，我们将 `iter` 初始化为 `lst.begin()`。第一次调用 `insert` 会将我们刚刚读入的 `string` 插入到 `iter` 所指向的元素之前的位置。`insert` 返回的迭代器恰好指向这个新元素。我们将此迭代器赋予 `iter` 并重复循环，读取下一个单词。只要继续有单词读入，每步 `while` 循环就会将一个新元素插入到 `iter` 之前，并将 `iter` 改变为新加入元素的位置。此元素为（新的）首元素。因此，每步循环将一个新元素插入到 `list` 首元素之前的位置。

使用 `emplace` 操作

C++ 11 新标准引入了三个新成员——`emplace_front`、`emplace` 和 `emplace_back`，这些操作构造而不是拷贝元素。这些操作分别对应 `push_front`、`insert` 和 `push_back`，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

当调用 `push` 或 `insert` 成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。而当我们调用一个 `emplace` 成员函数时，则是将参数传递给元素类型的构造函数。`emplace` 成员使用这些参数在容器管理的内存空间中直接构造元素。例如，假定 `c` 保存 `Sales_data` (参见 7.1.4 节，第 237 页) 元素：

```
// 在 c 的末尾构造一个 Sales_data 对象
// 使用三个参数的 Sales_data 构造函数
c.emplace_back("978-0590353403", 25, 15.99);
// 错误：没有接受三个参数的 push_back 版本
c.push_back("978-0590353403", 25, 15.99);
// 正确：创建一个临时的 Sales_data 对象传递给 push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```

其中对 `emplace_back` 的调用和第二个 `push_back` 调用都会创建新的 `Sales_data` 对象。在调用 `emplace_back` 时，会在容器管理的内存空间中直接创建对象。而调用 `push_back` 则会创建一个局部临时对象，并将其压入容器中。

`emplace` 函数的参数根据元素类型而变化，参数必须与元素类型的构造函数相匹配：

```
// iter 指向 c 中一个元素，其中保存了 Sales_data 元素
```

```
c.emplace_back(); // 使用 Sales_data 的默认构造函数
c.emplace(iter, "999-99999999"); // 使用 Sales_data(string)
// 使用 Sales_data 的接受一个 ISBN、一个 count 和一个 price 的构造函数
c.emplace_front("978-0590353403", 25, 15.99);
```



`emplace` 函数在容器中直接构造元素。传递给 `emplace` 函数的参数必须与元素类型的构造函数相匹配。

9.3.1 节练习

练习 9.18: 编写程序，从标准输入读取 `string` 序列，存入一个 `deque` 中。编写一个循环，用迭代器打印 `deque` 中的元素。

练习 9.19: 重写上题的程序，用 `list` 替代 `deque`。列出程序要做出哪些改变。

练习 9.20: 编写程序，从一个 `list<int>` 拷贝元素到两个 `deque` 中。值为偶数的所有元素都拷贝到一个 `deque` 中，而奇数值元素都拷贝到另一个 `deque` 中。

练习 9.21: 如果我们将第 308 页中使用 `insert` 返回值将元素添加到 `list` 中的循环程序改写为将元素插入到 `vector` 中，分析循环将如何工作。

练习 9.22: 假定 `iv` 是一个 `int` 的 `vector`，下面的程序存在什么错误？你将如何修改？

```
vector<int>::iterator iter = iv.begin(),
                     mid = iv.begin() + iv.size() / 2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
```

9.3.2 访问元素



表 9.6 列出了我们可以用来在顺序容器中访问元素的操作。如果容器中没有元素，访问操作的结果是未定义的。

包括 `array` 在内的每个顺序容器都有一个 `front` 成员函数，而除 `forward_list` 之外的所有顺序容器都有一个 `back` 成员函数。这两个操作分别返回首元素和尾元素的引用：

```
// 在解引用一个迭代器或调用 front 或 back 之前检查是否有元素
if (!c.empty()) {
    // val 和 val2 是 c 中第一个元素值的拷贝
    auto val = *c.begin(), val2 = c.front();
    // val3 和 val4 是 c 中最后一个元素值的拷贝
    auto last = c.end();
    auto val3 = *(--last); // 不能递减 forward_list 迭代器
    auto val4 = c.back(); // forward_list 不支持
}
```

此程序用两种不同方式来获取 `c` 中的首元素和尾元素的引用。直接的方法是调用 `front` 和 `back`。而间接的方法是通过解引用 `begin` 返回的迭代器来获得首元素的引用，以及通过递减然后解引用 `end` 返回的迭代器来获得尾元素的引用。

< 347

这个程序有两点值得注意：迭代器 `end` 指向的是容器尾元素之后的（不存在的）元

素。为了获取尾元素，必须首先递减此迭代器。另一个重要之处是，在调用 `front` 和 `back`（或解引用 `begin` 和 `end` 返回的迭代器）之前，要确保 `c` 非空。如果容器为空，`if` 中操作的行为将是未定义的。

表 9.6：在顺序容器中访问元素的操作

at 和下标操作只适用于 <code>string</code> 、 <code>vector</code> 、 <code>deque</code> 和 <code>array</code> 。
<code>back</code> 不适用于 <code>forward_list</code> 。
<code>c.back()</code> 返回 <code>c</code> 中尾元素的引用。若 <code>c</code> 为空，函数行为未定义
<code>c.front()</code> 返回 <code>c</code> 中首元素的引用。若 <code>c</code> 为空，函数行为未定义
<code>c[n]</code> 返回 <code>c</code> 中下标为 <code>n</code> 的元素的引用， <code>n</code> 是一个无符号整数。若 <code>n >= c.size()</code> ，则函数行为未定义
<code>c.at(n)</code> 返回下标为 <code>n</code> 的元素的引用。如果下标越界，则抛出一 <code>out_of_range</code> 异常



对一个空容器调用 `front` 和 `back`，就像使用一个越界的下标一样，是一种严重的程序设计错误。

访问成员函数返回的是引用

在容器中访问元素的成员函数（即，`front`、`back`、下标和 `at`）返回的都是引用。如果容器是一个 `const` 对象，则返回值是 `const` 的引用。如果容器不是 `const` 的，则返回值是普通引用，我们可以用来改变元素的值：

```
if (!c.empty()) {
    c.front() = 42;           // 将 42 赋予 c 中的第一个元素
    auto &v = c.back();       // 获得指向最后一个元素的引用
    v = 1024;                // 改变 c 中的元素
    auto v2 = c.back();       // v2 不是一个引用，它是 c.back() 的一个拷贝
    v2 = 0;                  // 未改变 c 中的元素
}
```

与往常一样，如果我们使用 `auto` 变量来保存这些函数的返回值，并且希望使用此变量来改变元素的值，必须记得将变量定义为引用类型。

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`）也都提供下标运算符（参见 3.3.3 节，第 91 页）。就像我们已经看到的那样，下标运算符接受一个下标参数，返回容器中该位置的元素的引用。给定下标必须“在范围内”（即，大于等于 0，且小于容器的大小）。保证下标有效是程序员的责任，下标运算符并不检查下标是否在合法范围内。使用越界的下标是一种严重的程序设计错误，而且编译器并不检查这种错误。

如果我们希望确保下标是合法的，可以使用 `at` 成员函数。`at` 成员函数类似下标运算符，但如果下标越界，`at` 会抛出一个 `out_of_range` 异常（参见 5.6 节，第 173 页）：

```
vector<string> svec;      // 空 vector
cout << svec[0];           // 运行时错误：svec 中没有元素！
cout << svec.at(0);        // 抛出一个 out_of_range 异常
```

9.3.2 节练习

练习 9.23: 在本节第一个程序(第 309 页)中,若 `c.size()` 为 1, 则 `val`、`val2`、`val3` 和 `val4` 的值会是什么?

练习 9.24: 编写程序, 分别使用 `at`、下标运算符、`front` 和 `begin` 提取一个 `vector` 中的第一个元素。在一个空 `vector` 上测试你的程序。

9.3.3 删除元素



与添加元素的多种方式类似,(非 `array`)容器也有多种删除元素的方式。表 9.7 列出了这些成员函数。

表 9.7: 顺序容器的删除操作

这些操作会改变容器的大小, 所以不适用于 `array`。

`forward_list` 有特殊版本的 `erase`, 参见 9.3.4 节(第 312 页)。

`forward_list` 不支持 `pop_back`; `vector` 和 `string` 不支持 `pop_front`。

`c.pop_back()` 删除 `c` 中尾元素。若 `c` 为空, 则函数行为未定义。函数返回 `void`

`c.pop_front()` 删除 `c` 中首元素。若 `c` 为空, 则函数行为未定义。函数返回 `void`

`c.erase(p)` 删除迭代器 `p` 所指定的元素, 返回一个指向被删元素之后元素的迭代器, 若 `p` 指向尾元素, 则返回尾后(`off-the-end`)迭代器。若 `p` 是尾后迭代器, 则函数行为未定义

`c.erase(b, e)` 删除迭代器 `b` 和 `e` 所指定范围内的元素。返回一个指向最后一个被删元素之后元素的迭代器, 若 `e` 本身就是尾后迭代器, 则函数也返回尾后迭代器

`c.clear()` 删除 `c` 中的所有元素。返回 `void`



WARNING 删除 `deque` 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向 `vector` 或 `string` 中删除点之后位置的迭代器、引用和指针都会失效。



WARNING 删除元素的成员函数并不检查其参数。在删除元素之前, 程序员必须确保它(们)是存在的。

pop_front 和 pop_back 成员函数

`pop_front` 和 `pop_back` 成员函数分别删除首元素和尾元素。与 `vector` 和 `string` 不支持 `push_front` 一样, 这些类型也不支持 `pop_front`。类似的, `forward_list` 不支持 `pop_back`。与元素访问成员函数类似, 不能对一个空容器执行弹出操作。

这些操作返回 `void`。如果你需要弹出的元素的值, 就必须在执行弹出操作之前保存它:

```
while (!ilist.empty()) {
    process(ilist.front()); // 对 ilist 的首元素进行一些处理
    ilist.pop_front();      // 完成处理后删除首元素
}
```

349 从容器内部删除一个元素

成员函数 `erase` 从容器中指定位置删除元素。我们可以删除由一个迭代器指定的单个元素，也可以删除由一对迭代器指定的范围内的所有元素。两种形式的 `erase` 都返回指向删除的(最后一个)元素之后位置的迭代器。即，若 `j` 是 `i` 之后的元素，那么 `erase(i)` 将返回指向 `j` 的迭代器。

例如，下面的循环删除一个 `list` 中的所有奇数元素：

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2)           // 若元素为奇数
        it = lst.erase(it); // 删除此元素
    else
        ++it;
```

每个循环步中，首先检查当前元素是否是奇数。如果是，就删除该元素，并将 `it` 设置为我们所删除的元素之后的元素。如果`*it` 为偶数，我们将 `it` 递增，从而在下一步循环检查下一个元素。

删除多个元素

接受一对迭代器的 `erase` 版本允许我们删除一个范围内的元素：

```
// 删除两个迭代器表示的范围内的元素
// 返回指向最后一个被删元素之后位置的迭代器
elem1 = slist.erase(elem1, elem2); // 调用后, elem1 == elem2
```

迭代器 `elem1` 指向我们要删除的第一个元素，`elem2` 指向我们要删除的最后一个元素之后的位置。

350

为了删除一个容器中的所有元素，我们既可以调用 `clear`，也可以用 `begin` 和 `end` 获得的迭代器作为参数调用 `erase`：

```
slist.clear(); // 删除容器中所有元素
slist.erase(slist.begin(), slist.end()); // 等价调用
```

9.3.3 节练习

练习 9.25：对于第 312 页中删除一个范围内的元素的程序，如果 `elem1` 与 `elem2` 相等会发生什么？如果 `elem2` 是尾后迭代器，或者 `elem1` 和 `elem2` 皆为尾后迭代器，又会发生什么？

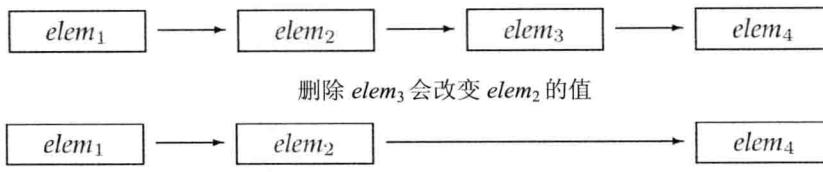
练习 9.26：使用下面代码定义的 `ia`，将 `ia` 拷贝到一个 `vector` 和一个 `list` 中。使用单迭代器版本的 `erase` 从 `list` 中删除奇数元素，从 `vector` 中删除偶数元素。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```



9.3.4 特殊的 `forward_list` 操作

为了理解 `forward_list` 为什么有特殊版本的添加和删除操作，考虑当我们从一个单向链表中删除一个元素时会发生什么。如图 9.1 所示，删除一个元素会改变序列中的链接。在此情况下，删除 `elem3` 会改变 `elem2`，`elem2` 原来指向 `elem3`，但删除 `elem3` 后，`elem2` 指向了 `elem4`。

图 9.1: `forward_list` 的特殊操作

当添加或删除一个元素时，删除或添加的元素之前的那个元素的后继会发生改变。为了添加或删除一个元素，我们需要访问其前驱，以便改变前驱的链接。但是，`forward_list` 是单向链表。在一个单向链表中，没有简单的方法来获取一个元素的前驱。出于这个原因，在一个 `forward_list` 中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。这样，我们总是可以访问到被添加或删除操作所影响的元素。

由于这些操作与其他容器上的操作的实现方式不同，`forward_list` 并未定义 `insert`、`emplace` 和 `erase`，而是定义了名为 `insert_after`、`emplace_after` 和 `erase_after` 的操作（参见表 9.8）。例如，在我们的例子中，为了删除 `elem3`，应该用指向 `elem2` 的迭代器调用 `erase_after`。为了支持这些操作，`forward_list` 也定义了 `before_begin`，它返回一个首前（off-the-beginning）迭代器。这个迭代器允许我们在链表首元素之前并不存在的元素“之后”添加或删除元素（亦即在链表首元素之前添加删除元素）。

<351

表 9.8: 在 `forward_list` 中插入或删除元素的操作

<code>lst.before_begin()</code>	返回指向链表首元素之前不存在的元素的迭代器。此迭代器不能解引用。
<code>lst.cbefore_begin()</code>	<code>cbefore_begin()</code> 返回一个 <code>const_iterator</code>
<code>lst.insert_after(p, t)</code>	在迭代器 <code>p</code> 之后的位置插入元素。 <code>t</code> 是一个对象， <code>n</code> 是数量，
<code>lst.insert_after(p, n, t)</code>	<code>b</code> 和 <code>e</code> 是表示范围的一对迭代器（ <code>b</code> 和 <code>e</code> 不能指向 <code>lst</code> 内），
<code>lst.insert_after(p, b, e)</code>	<code>il</code> 是一个花括号列表。返回一个指向最后一个插入元素的迭代器。如果范围为空，则返回 <code>p</code> 。若 <code>p</code> 为尾后迭代器，则函数行为未定义
<code>lst.insert_after(p, il)</code>	
<code>emplace_after(p, args)</code>	使用 <code>args</code> 在 <code>p</code> 指定的位置之后创建一个元素。返回一个指向这个新元素的迭代器。若 <code>p</code> 为尾后迭代器，则函数行为未定义
<code>lst.erase_after(p)</code>	删除 <code>p</code> 指向的位置之后的元素，或删除从 <code>b</code> 之后直到（但不包含） <code>e</code> 之间的元素。返回一个指向被删元素之后元素的迭代器，若不存在这样的元素，则返回尾后迭代器。如果 <code>p</code> 指向 <code>lst</code> 的尾元素或者是一个尾后迭代器，则函数行为未定义
<code>lst.erase_after(b, e)</code>	

当在 `forward_list` 中添加或删除元素时，我们必须关注两个迭代器——一个指向我们要处理的元素，另一个指向其前驱。例如，可以改写第 312 页中从 `list` 中删除奇数元素的循环程序，将其改为从 `forward_list` 中删除元素：

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin();           // 表示 flst 的“首前元素”
auto curr = flst.begin();                 // 表示 flst 中的第一个元素
while (curr != flst.end()) {              // 仍有元素要处理
    if (*curr % 2)                      // 若元素为奇数
        curr = flst.erase_after(prev);    // 删除它并移动 curr
    else {
        prev = curr;                   // 移动迭代器 curr，指向下一个元素，prev 指向
```

```

    ++curr;      // curr 之前的元素
}
}

```

此例中，`curr` 表示我们要处理的元素，`prev` 表示 `curr` 的前驱。调用 `begin` 来初始化 `curr`，这样第一步循环就会检查第一个元素是否是奇数。我们用 `before_begin` 来初始化 `prev`，它返回指向 `curr` 之前不存在的元素的迭代器。

当找到奇数元素后，我们将 `prev` 传递给 `erase_after`。此调用将 `prev` 之后的元素删除，即，删除 `curr` 指向的元素。然后我们将 `curr` 重置为 `erase_after` 的返回值，使得 `curr` 指向序列中下一个元素，`prev` 保持不变，仍指向（新）`curr` 之前的元素。如果 `curr` 指向的元素不是奇数，在 `else` 中我们将两个迭代器都向前移动。

9.3.4 节练习

练习 9.27：编写程序，查找并删除 `forward_list<int>` 中的奇数元素。

练习 9.28：编写函数，接受一个 `forward_list<string>` 和两个 `string` 共三个参数。函数应在链表中查找第一个 `string`，并将第二个 `string` 插入到紧接着第一个 `string` 之后的位置。若第一个 `string` 未在链表中，则将第二个 `string` 插入到链表末尾。

9.3.5 改变容器大小

如表 9.9 所描述，我们可以用 `resize` 来增大或缩小容器，与往常一样，`array` 不支持 `resize`。如果当前大小大于所要求的大小，容器后部的元素会被删除；如果当前大小小于新大小，会将新元素添加到容器后部：

```

list<int> ilist(10, 42);      // 10 个 int: 每个的值都是 42
ilist.resize(15);            // 将 5 个值为 0 的元素添加到 ilist 的末尾
ilist.resize(25, -1);        // 将 10 个值为 -1 的元素添加到 ilist 的末尾
ilist.resize(5);             // 从 ilist 末尾删除 20 个元素

```

`resize` 操作接受一个可选的元素值参数，用来初始化添加到容器中的元素。如果调用者未提供此参数，新元素进行值初始化（参见 3.3.1 节，第 88 页）。如果容器保存的是类类型元素，且 `resize` 向容器添加新元素，则我们必须提供初始值，或者元素类型必须提供一个默认构造函数。

表 9.9：顺序容器大小操作

`resize` 不适用于 `array`

`c.resize(n)` 调整 `c` 的大小为 `n` 个元素。若 `n < c.size()`，则多出的元素被丢弃。若必须添加新元素，对新元素进行值初始化

`c.resize(n, t)` 调整 `c` 的大小为 `n` 个元素。任何新添加的元素都初始化为值 `t`



如果 `resize` 缩小容器，则指向被删除元素的迭代器、引用和指针都会失效；对 `vector`、`string` 或 `deque` 进行 `resize` 可能导致迭代器、指针和引用失效。

9.3.5 节练习

< 353

练习 9.29: 假定 `vec` 包含 25 个元素, 那么 `vec.resize(100)` 会做什么? 如果接下来调用 `vec.resize(10)` 会做什么?

练习 9.30: 接受单个参数的 `resize` 版本对元素类型有什么限制 (如果有的话)?

9.3.6 容器操作可能使迭代器失效



向容器中添加元素和从容器中删除元素的操作可能会使指向容器元素的指针、引用或迭代器失效。一个失效的指针、引用或迭代器将不再表示任何元素。使用失效的指针、引用或迭代器是一种严重的程序设计错误, 很可能引起与使用未初始化指针一样的问题 (参见 2.3.2 节, 第 49 页)

在向容器添加元素后:

- 如果容器是 `vector` 或 `string`, 且存储空间被重新分配, 则指向容器的迭代器、指针和引用都会失效。如果存储空间未重新分配, 指向插入位置之前的元素的迭代器、指针和引用仍有效, 但指向插入位置之后元素的迭代器、指针和引用将会失效。
- 对于 `deque`, 插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素, 迭代器会失效, 但指向存在的元素的引用和指针不会失效。
- 对于 `list` 和 `forward_list`, 指向容器的迭代器 (包括尾后迭代器和首前迭代器)、指针和引用仍有效。

当我们从一个容器中删除元素后, 指向被删除元素的迭代器、指针和引用会失效, 这应该不会令人惊讶。毕竟, 这些元素都已经被销毁了。当我们删除一个元素后:

- 对于 `list` 和 `forward_list`, 指向容器其他位置的迭代器 (包括尾后迭代器和首前迭代器)、引用和指针仍有效。
- 对于 `deque`, 如果在首尾之外的任何位置删除元素, 那么指向被删除元素外其他元素的迭代器、引用或指针也会失效。如果是删除 `deque` 的尾元素, 则尾后迭代器也会失效, 但其他迭代器、引用和指针不受影响; 如果是删除首元素, 这些也不会受影响。
- 对于 `vector` 和 `string`, 指向被删元素之前元素的迭代器、引用和指针仍有效。

注意: 当我们删除元素时, 尾后迭代器总是会失效。



WARNING

使用失效的迭代器、指针或引用是严重的运行时错误。

建议: 管理迭代器

< 354

当你使用迭代器 (或指向容器元素的引用或指针) 时, 最小化要求迭代器必须保持有效的程序片段是一个好的方法。

由于向迭代器添加元素和从迭代器删除元素的代码可能会使迭代器失效, 因此必须保证每次改变容器的操作之后都正确地重新定位迭代器。这个建议对 `vector`、`string` 和 `deque` 尤为重要。

编写改变容器的循环程序

添加/删除 vector、string 或 deque 元素的循环程序必须考虑迭代器、引用和指针可能失效的问题。程序必须保证每个循环步中都更新迭代器、引用或指针。如果循环中调用的是 insert 或 erase，那么更新迭代器很容易。这些操作都返回迭代器，我们可以用来更新：

```
// 傻瓜循环，删除偶数元素，复制每个奇数元素
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // 调用 begin而不是 cbegin，因为我们要改变 vi
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter); // 复制当前元素
        iter += 2; // 向前移动迭代器，跳过当前元素以及插入到它之前的元素
    } else
        iter = vi.erase(iter); // 删除偶数元素
    // 不应向前移动迭代器，iter 指向我们删除的元素之后的元素
}
```

此程序删除 vector 中的偶数值元素，并复制每个奇数值元素。我们在调用 insert 和 erase 后都更新迭代器，因为两者都会使迭代器失效。

在调用 erase 后，不必递增迭代器，因为 erase 返回的迭代器已经指向序列中下一个元素。调用 insert 后，需要递增迭代器两次。记住，insert 在给定位置之前插入新元素，然后返回指向新插入元素的迭代器。因此，在调用 insert 后，iter 指向新插入元素，位于我们正在处理的元素之前。我们将迭代器递增两次，恰好越过了新添加的元素和正在处理的元素，指向下一个未处理的元素。

不要保存 end 返回的迭代器

当我们添加/删除 vector 或 string 的元素后，或在 deque 中首元素之外任何位置添加/删除元素后，原来 end 返回的迭代器总是会失效。因此，添加或删除元素的循环程序必须反复调用 end，而不能在循环之前保存 end 返回的迭代器，一直当作容器末尾使用。通常 C++ 标准库的实现中 end() 操作都很快，部分就是因为这个原因。

例如，考虑这样一个循环，它处理容器中的每个元素，在其后添加一个新元素。我们希望循环能跳过新添加的元素，只处理原有元素。在每步循环之后，我们将定位迭代器，使其指向下一个原有元素。如果我们试图“优化”这个循环，在循环之前保存 end() 返回的迭代器，一直用作容器末尾，就会导致一场灾难：

```
// 灾难：此循环的行为是未定义的
auto begin = v.begin(),
end = v.end(); // 保存尾迭代器的值是一个坏主意
while (begin != end) {
    // 做一些处理
    // 插入新值，对 begin 重新赋值，否则的话它就会失效
    ++begin; // 向前移动 begin，因为我们想在此元素之后插入元素
    begin = v.insert(begin, 42); // 插入新值
    ++begin; // 向前移动 begin 跳过我们刚刚加入的元素
}
```

此代码的行为是未定义的。在很多标准库实现上，此代码会导致无限循环。问题在于我们将 end 操作返回的迭代器保存在一个名为 end 的局部变量中。在循环体中，我们向容器

中添加了一个元素，这个操作使保存在 `end` 中的迭代器失效了。这个迭代器不再指向 `v` 中任何元素，或是 `v` 中尾元素之后的位置。



如果在一个循环中插入/删除 deque、string 或 vector 中的元素，不要缓存 end 返回的迭代器。

必须在每次插入操作后重新调用 `end()`，而不能在循环开始前保存它返回的迭代器：

```
// 更安全的方法：在每个循环步添加/删除元素后都重新计算 end
while (begin != v.end()) {
    // 做一些处理
    ++begin; // 向前移动 begin，因为我们想在此元素之后插入元素
    begin = v.insert(begin, 42); // 插入新值
    ++begin; // 向前移动 begin，跳过我们刚刚加入的元素
}
```

9.3.6 节练习

练习 9.31：第 316 页中删除偶数值元素并复制奇数值元素的程序不能用于 `list` 或 `forward_list`。为什么？修改程序，使之也能用于这些类型。

练习 9.32：在第 316 页的程序中，向下面语句这样调用 `insert` 是否合法？如果不合法，为什么？

```
iter = vi.insert(iter, *iter++);
```

练习 9.33：在本节最后一个例子中，如果不将 `insert` 的结果赋予 `begin`，将会发生什么？编写程序，去掉此赋值语句，验证你的答案。

练习 9.34：假定 `vi` 是一个保存 `int` 的容器，其中有偶数值也有奇数值，分析下面循环的行为，然后编写程序验证你的分析是否正确。

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
    ++iter;
```

9.4 vector 对象是如何增长的



为了支持快速随机访问，`vector` 将元素连续存储——每个元素紧挨着前一个元素存储。通常情况下，我们不必关心一个标准库类型是如何实现的，而只需关心它如何使用。然而，对于 `vector` 和 `string`，其部分实现渗透到了接口中。

假定容器中元素是连续存储的，且容器的大小是可变的，考虑向 `vector` 或 `string` 中添加元素会发生什么：如果没有空间容纳新元素，容器不可能简单地将它添加到内存中其他位置——因为元素必须连续存储。容器必须分配新的内存空间来保存已有元素和新元素，将已有元素从旧位置移动到新空间中，然后添加新元素，释放旧存储空间。如果我们每添加一个新元素，`vector` 就执行一次这样的内存分配和释放操作，性能会慢到不可接受。

为了避免这种代价，标准库实现者采用了可以减少容器空间重新分配次数的策略。当

不得不获取新的内存空间时，`vector` 和 `string` 的实现通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用，可用来保存更多的新元素。这样，就不需要每次添加新元素都重新分配容器的内存空间了。

这种分配策略比每次添加新元素时都重新分配容器内存空间的策略要高效得多。其实际性能也表现得足够好——虽然 `vector` 在每次重新分配内存空间时都要移动所有元素，但使用此策略后，其扩张操作通常比 `list` 和 `deque` 还要快。

管理容量的成员函数

如表 9.10 所示，`vector` 和 `string` 类型提供了一些成员函数，允许我们与它的实现中内存分配部分互动。`capacity` 操作告诉我们容器在不扩张内存空间的情况下可以容纳多少个元素。`reserve` 操作允许我们通知容器它应该准备保存多少个元素。

表 9.10：容器大小管理操作

<code>shrink_to_fit</code> 只适用于 <code>vector</code> 、 <code>string</code> 和 <code>deque</code> 。	
<code>capacity</code> 和 <code>reserve</code> 只适用于 <code>vector</code> 和 <code>string</code> 。	
<code>c.shrink_to_fit()</code>	请将 <code>capacity()</code> 减少为与 <code>size()</code> 相同大小
<code>c.capacity()</code>	不重新分配内存空间的话， <code>c</code> 可以保存多少元素
<code>c.reserve(n)</code>	分配至少能容纳 <code>n</code> 个元素的内存空间



`reserve` 并不改变容器中元素的数量，它仅影响 `vector` 预先分配多大的内存空间。

357

只有当需要的内存空间超过当前容量时，`reserve` 调用才会改变 `vector` 的容量。如果需求大小大于当前容量，`reserve` 至少分配与需求一样大的内存空间（可能更大）。

如果需求大小小于或等于当前容量，`reserve` 什么也不做。特别是，当需求大小小于当前容量时，容器不会退回内存空间。因此，在调用 `reserve` 之后，`capacity` 将会大于或等于传递给 `reserve` 的参数。

这样，调用 `reserve` 永远也不会减少容器占用的内存空间。类似的，`resize` 成员函数（参见 9.3.5 节，第 314 页）只改变容器中元素的数目，而不是容器的容量。我们同样不能使用 `resize` 来减少容器预留的内存空间。

C++ 11

在新标准库中，我们可以调用 `shrink_to_fit` 来要求 `deque`、`vector` 或 `string` 退回不需要的内存空间。此函数指出我们不再需要任何多余的内存空间。但是，具体的实现可以选择忽略此请求。也就是说，调用 `shrink_to_fit` 也并不保证一定退回内存空间。

capacity 和 size

理解 `capacity` 和 `size` 的区别非常重要。容器的 `size` 是指它已经保存的元素的数目；而 `capacity` 则是在不分配新的内存空间的前提下它最多可以保存多少元素。

下面的代码展示了 `size` 和 `capacity` 之间的相互作用：

```
vector<int> ivec;
// size 应该为 0; capacity 的值依赖于具体实现
cout << " ivec: size: " << ivec.size()
     << " capacity: " << ivec.capacity() << endl;
// 向 ivec 添加 24 个元素
```

```

for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

// size 应该为 24; capacity 应该大于等于 24, 具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl

```

当在我们的系统上运行时, 这段程序得到如下输出:

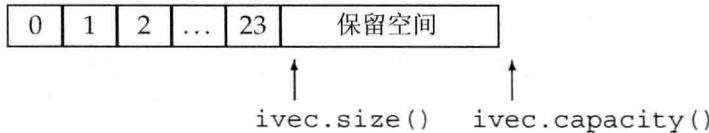
```

ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32

```

我们知道一个空 vector 的 size 为 0, 显然在我们的标准库实现中一个空 vector 的 capacity 也为 0。当向 vector 中添加元素时, 我们知道 size 与添加的元素数目相等。而 capacity 至少与 size 一样大, 具体会分配多少额外空间则视标准库具体实现而定。在我们的标准库实现中, 每次添加 1 个元素, 共添加 24 个元素, 会使 capacity 变为 32。358

可以想象 ivec 的当前状态如下图所示:



现在可以预分配一些额外空间:

```

ivec.reserve(50); // 将 capacity 至少设定为 50, 可能会更大
// size 应该为 24; capacity 应该大于等于 50, 具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

```

程序的输出表明 reserve 严格按照我们需求的大小分配了新的空间:

```
ivec: size: 24 capacity: 50
```

接下来可以用光这些预留空间:

```

// 添加元素用光多余容量
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);
// capacity 应该未改变, size 和 capacity 不相等
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

```

程序输出表明此时我们确实用光了预留空间, size 和 capacity 相等:

```
ivec: size: 50 capacity: 50
```

由于我们只使用了预留空间, 因此没有必要为 vector 分配新的空间。实际上, 只要没有操作需求超出 vector 的容量, vector 就不能重新分配内存空间。

如果我们现在再添加一个新元素, vector 就不得不重新分配空间:

```

ivec.push_back(42); // 再添加一个元素
// size 应该为 51; capacity 应该大于等于 51, 具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;

```

这段程序的输出为

359 ivec: size: 51 capacity: 100

这表明 `vector` 的实现采用的策略似乎是在每次需要分配新内存空间时将当前容量翻倍。

可以调用 `shrink_to_fit` 来要求 `vector` 将超出当前大小的多余内存退回给系统：

```
ivec.shrink_to_fit(); // 要求归还内存
// size 应该未改变; capacity 的值依赖于具体实现
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

调用 `shrink_to_fit` 只是一个请求，标准库并不保证退还内存。



每个 `vector` 实现都可以选择自己的内存分配策略。但是必须遵守的一条原则是：只有当迫不得已时才可以分配新的内存空间。

只有在执行 `insert` 操作时 `size` 与 `capacity` 相等，或者调用 `resize` 或 `reserve` 时给定的大小超过当前 `capacity`，`vector` 才可能重新分配内存空间。会分配多少超过给定容量的额外空间，取决于具体实现。

虽然不同的实现可以采用不同的分配策略，但所有实现都应遵循一个原则：确保用 `push_back` 向 `vector` 添加元素的操作有高效率。从技术角度说，就是通过在一个初始为空的 `vector` 上调用 n 次 `push_back` 来创建一个 n 个元素的 `vector`，所花费的时间不能超过 n 的常数倍。

9.4 节练习

练习 9.35：解释一个 `vector` 的 `capacity` 和 `size` 有何区别。

练习 9.36：一个容器的 `capacity` 可能小于它的 `size` 吗？

练习 9.37：为什么 `list` 或 `array` 没有 `capacity` 成员函数？

练习 9.38：编写程序，探究在你的标准库实现中，`vector` 是如何增长的。

练习 9.39：解释下面程序片段做了什么：

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);
svec.resize(svec.size() + svec.size() / 2);
```

练习 9.40：如果上一题中的程序读入了 256 个词，在 `resize` 之后容器的 `capacity` 可能是多少？如果读入了 512 个、1000 个或 1048 个词呢？

360 9.5 额外的 `string` 操作

除了顺序容器共同的操作之外，`string` 类型还提供了一些额外的操作。这些操作中的大部分要么是提供 `string` 类和 C 风格字符数组之间的相互转换，要么是增加了允许我们用下标代替迭代器的版本。

标准库 `string` 类型定义了大量函数。幸运的是，这些函数使用了重复的模式。由于函数过多，本节初次阅读可能令人心烦，因此读者可能希望快速浏览本节。当你了解 `string` 支持哪些类型的操作后，就可以在需要使用一个特定操作时回过头来仔细阅读。

9.5.1 构造 `string` 的其他方法



除了我们在 3.2.1 节（第 76 页）已经介绍过的构造函数，以及与其他顺序容器相同的构造函数（参见表 9.3，第 299 页）外，`string` 类型还支持另外三个构造函数，如表 9.11 所示。

表 9.11：构造 `string` 的其他方法

<code>n, len2 和 pos2</code> 都是无符号值	
<code>string s(cp, n)</code>	<code>s</code> 是 <code>cp</code> 指向的数组中前 <code>n</code> 个字符的拷贝。此数组至少应该包含 <code>n</code> 个字符
<code>string s(s2, pos2)</code>	<code>s</code> 是 <code>string s2</code> 从下标 <code>pos2</code> 开始的字符的拷贝。若 <code>pos2>s2.size()</code> ，构造函数的行为未定义
<code>string s(s2, pos2, len2)</code>	<code>s</code> 是 <code>string s2</code> 从下标 <code>pos2</code> 开始 <code>len2</code> 个字符的拷贝。若 <code>pos2>s2.size()</code> ，构造函数的行为未定义。不管 <code>len2</code> 的值是多少，构造函数至多拷贝 <code>s2.size()-pos2</code> 个字符

这些构造函数接受一个 `string` 或一个 `const char*` 参数，还接受（可选的）指定拷贝多少个字符的参数。当我们传递给它们的是一个 `string` 时，还可以给定一个下标来指出从哪里开始拷贝：

```
const char *cp = "Hello World!!!";      // 以空字符结束的数组
char noNull[] = {'H', 'i'};              // 不是以空字符结束
string s1(cp); // 拷贝 cp 中的字符直到遇到空字符; s1 == "Hello World!!!"
string s2(noNull, 2);                  // 从 noNull 拷贝两个字符; s2 == "Hi"
string s3(noNull);                   // 未定义: noNull 不是以空字符结束
string s4(cp + 6, 5);                // 从 cp[6] 开始拷贝 5 个字符; s4 == "World"
string s5(s1, 6, 5);                 // 从 s1[6] 开始拷贝 5 个字符; s5 == "World"
string s6(s1, 6);                   // 从 s1[6] 开始拷贝，直至 s1 末尾; s6 == "World!!!"
string s7(s1, 6, 20);                // 正确，只拷贝到 s1 末尾; s7 == "World!!!"
string s8(s1, 16);                  // 抛出一个 out_of_range 异常
```

通常当我们从一个 `const char*` 创建 `string` 时，指针指向的数组必须以空字符结尾，拷贝操作遇到空字符时停止。如果我们还传递给构造函数一个计数值，数组就不必以空字符结尾。如果我们未传递计数值且数组也未以空字符结尾，或者给定计数值大于数组大小，则构造函数的行为是未定义的。

361

当从一个 `string` 拷贝字符时，我们可以提供一个可选的开始位置和一个计数值。开始位置必须小于或等于给定的 `string` 的大小。如果位置大于 `size`，则构造函数抛出一个 `out_of_range` 异常（参见 5.6 节，第 173 页）。如果我们传递了一个计数值，则从给定位置开始拷贝这么多个字符。不管我们要求拷贝多少个字符，标准库最多拷贝到 `string` 结尾，不会更多。

substr 操作

`substr` 操作（参见表 9.12）返回一个 `string`，它是原始 `string` 的一部分或全部的拷贝。可以传递给 `substr` 一个可选的开始位置和计数值：

```

string s("hello world");
string s2 = s.substr(0, 5);           // s2 = hello
string s3 = s.substr(6);             // s3 = world
string s4 = s.substr(6, 11);         // s3 = world
string s5 = s.substr(12);           // 抛出一个 out_of_range 异常

```

如果开始位置超过了 `string` 的大小，则 `substr` 函数抛出一个 `out_of_range` 异常（参见 5.6 节，第 173 页）。如果开始位置加上计数值大于 `string` 的大小，则 `substr` 会调整计数值，只拷贝到 `string` 的末尾。

表 9.12：子字符串操作

<code>s.substr(pos, n)</code>	返回一个 <code>string</code> ，包含 <code>s</code> 中从 <code>pos</code> 开始的 <code>n</code> 个字符的拷贝。 <code>pos</code> 的默认值为 0。 <code>n</code> 的默认值为 <code>s.size() - pos</code> ，即拷贝从 <code>pos</code> 开始的所有字符
-------------------------------	--

9.5.1 节练习

练习 9.41：编写程序，从一个 `vector<char>` 初始化一个 `string`。

练习 9.42：假定你希望每次读取一个字符存入一个 `string` 中，而且知道最少需要读取 100 个字符，应该如何提高程序的性能？



9.5.2 改变 `string` 的其他方法

`string` 类型支持顺序容器的赋值运算符以及 `assign`、`insert` 和 `erase` 操作（参见 9.2.5 节，第 302 页；9.3.1 节，第 306 页；9.3.3 节，第 311 页）。除此之外，它还定义了额外的 `insert` 和 `erase` 版本。

除了接受迭代器的 `insert` 和 `erase` 版本外，`string` 还提供了接受下标的版本。下标指出了开始删除的位置，或是 `insert` 到给定值之前的位置：

```

s.insert(s.size(), 5, '!'); // 在 s 末尾插入 5 个感叹号
s.erase(s.size() - 5, 5); // 从 s 删除最后 5 个字符

```



标准库 `string` 类型还提供了接受 C 风格字符数组的 `insert` 和 `assign` 版本。例如，我们可以将由空字符结尾的字符数组 `insert` 到或 `assign` 给一个 `string`：

```

const char *cp = "Stately, plump Buck";
s.assign(cp, 7);           // s == "Stately"
s.insert(s.size(), cp + 7); // s == "Stately, plump Buck"

```

此处我们首先通过调用 `assign` 替换 `s` 的内容。我们赋予 `s` 的是从 `cp` 指向的地址开始的 7 个字符。要求赋值的字符数必须小于或等于 `cp` 指向的数组中的字符数（不包括结尾的空字符）。

接下来在 `s` 上调用 `insert`，我们的意图是将字符插入到 `s[size()]` 处（不存在的）元素之前的位置。在此例中，我们将 `cp` 开始的 7 个字符（至多到结尾空字符之前）拷贝到 `s` 中。

我们也可以指定将来自其他 `string` 或子字符串的字符插入到当前 `string` 中或赋予当前 `string`：

```

string s = "some string", s2 = "some other string";
s.insert(0, s2); // 在 s 中位置 0 之前插入 s2 的拷贝

```