1 第二章

1. 'void*' 是一种特殊的指针类，可以存放任意对象的地址。

2. const 对象必须初始化。const int buf =10

3. const 指针，其值不能被改变，所有必须初始化。
4. 顶层 const 和底层 const，顶层 const 表示指针本身是一个常量，底层 const 表示指针所指对象是一个常量。更一般的，顶层 const 可以表示任意的对象是常量，这一点对任何数据类型都适用。底层 const 则与指针和引用等复合类型的基本类型部分有关。比较特殊的是，指针类型既可以是顶层 const 也可以是底层 const。

```
int i = 0;
int *const p1 = &i;     // 不能改变p1的值，p1是一个顶层const
const int ci = 42;      // 不能改变ci的值，ci是一个顶层const
const int *p2 = &ci;    // 允许改变p2的值，p2是一个底层const
const int *const p3 = p2; // 靠右的const是顶层const，靠左的是底层const
const int &r = ci;      // 用于声明引用的const都是底层const
```

C++ 新标准引入了第二种类型说明符 **decltype**，它的作用是选择并返回操作数的数据类型。在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值：

```
decltype(f()) sum = x; // sum的类型就是函数f的返回类型
```

编译器并不实际调用 f，而是使用当调用发生时 f 的返回值的类型作为 sum 的类型。

如果 decltype 使用的表达式是一个变量，则 decltype 返回该变量的类型（包括顶层 const 和引用在内）：

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x的类型是const int
decltype(cj) y = x; // y的类型是const int&，y绑定到x
```

decltype 和引用

如果 decltype 使用的表达式不是一个变量，则 decltype 返回表达式结果对应的类型。如果表达式向 decltype 返回一个引用类型，一般来说，意味着该表达式的结果对象能作为一条赋值语句的左值：

```
// decltype的结果可以是引用类型
int i = 42, *p = &i, &r = i;
decltype(r + 0) b; // 正确，加法的结果是int，因此b是一个int
decltype(*p) c; // 错误，c是int&，必须初始化
```

如果表达式的内容是解引用操作，则 decltype 将得到引用类型。

有一种情况需要特别注意：对于 decltype 所用的表达式来说，如果变量名加上了一对括号，编译器就会把它当成一个表达式。变量是一种可以作为赋值语句左值的特殊表达式，所以这样的 decltype 就会得到引用类型：

```
// decltype的表达式如果是加上了括号的变量，结果是引用
decltype((i)) d; // 错误，d是int&，必须初始化
decltype(i) e; // 正确，e是一个int。
```

auto 和 decltype 的区别主要有三个方面：第一，auto 类型说明符用编译器计算变量的初始值来推断其类型，而 decltype 虽然也让编译器分析表达式并得到它的类型，但是不实际计算表达式的值。第二，编译器推断出来的 auto 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。例如，auto 一般会忽略掉顶层 const，而把底层 const 保留下来。与之相反，decltype 会保留变量的顶层 const。第三，与 auto 不同，decltype 的结果类型与表达式形式密切相关，如果变量名加上了一对括号，则得到的类型与不加括号时会有不同。如果 decltype 使用的是一个不加括号的变量，则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号，则编译器将推断得到引用类型。

2 第三章字符串、向量和数组

2.1 String

1. **头文件不应包含 using 声明**

2. **初始化 string 对象的方式**:

```
string s1;           // 默认初始化，s1是一个空串
string s2(s1);       // s2是s1的副本
string s2 = s1;      // 等价于s2(s1)
string s3("value");  // s3是字面值"value"的副本，不包括最后的空字符
string s3 = "value"; // 等价于s3("value")
string s4(n, 'c');   // 初始化为由n个字符c组成的串
```

3. **直接初始化和拷贝初始化**

如果使用等号（=）初始化一个变量，实际上执行的是**拷贝初始化（copy initialization）**，编译器把等号右侧的对象初始值拷贝到新创建的对象中去。如果不适用等号，则执行的是**直接初始化**（direct initialization）。

```
string s5 = "hiya";  // 拷贝初始化
string s6("hiya");   // 直接初始化
string s7(10, 'c');  // 直接初始化
```

4. getline 函数会读取换行符，但不会把它存入字符串中。getline 返回输入流。

string::size_type 类型

size 函数返回的是一个 string::size_type 类型的值。这是一个无符号的整数。

5. **处理每个字符？使用基于范围的 for 语句**

如果想对 string 对象中的每个字符做点儿什么操作，目前最好的办法是使用 C++11 新标准提供的一种语句：**范围 for**（range for）语句。这种语句遍历序列中的每个元素并对序列中的每个值执行某种操作，其语法格式是：

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

```
    statement
```

如:

```
for (auto c : str)
    // do something
```

使用下标运算符

[ ]符号叫做下标运算符，范围是[0, s.size())，越界的结果是 UB（undefined behavior，未定义行为)。

2.2 Vector

1. vector 是对象的集合，也叫容器（container)。集合中的每个对象都有一个索引，索引用于访问对象。

2. vector 是一个类模板。模板是为编译器提供的一份生成类或函数的说明。

3. vector 是模板而非类型，由 vector 生成的类型必须包含元素的类型，如:

   ```
   vector<int> v;
   ```

4. **vector 中存放的是对象，而引用不是对象，故不能存储引用。**
   ### 定义和初始化 vector 对象
   vector 模板控制着初始化向量的方法。定义 vector 对象的方法有:

- vector<T> v1，默认初始化，v1 是一个空的 vector
- vector<T> v2(v1)，v2 中包含 v1 所有元素的副本
- vector<T> v2 = v1，等价于 v2(v1)
- vector<T> v3(n, val)，v3 包含了 n 个重复的元素，每个元素的值都是 val
- vector<T> v4(n)，v4 包含了 n 个执行了值初始化的对象

- `vector<T> v5{a,b,c...}`，v5 里包含了用 a,b,c...初始化的元素

- `vector<T> v5 = {a,b,c...}`，等价于`vector<T> v5{a,b,c...}`
  值初始化
  值初始化（value initialize），是指如果是内置类型，则初始值为 0；如果是类类型，执行类默认初始化。`vector<T>(n)`中，所有元素将执行值初始化。### 向 vector 中添加元素 push_back 函数把一个元素压入 vector 对象的尾端。
  vector 的对象能高效地增长，因此更常见的情况是：创建一个空 vector，然后在运行时再利用 vector 的成员函数 push_back 向其中添加元素。
  一定不能在遍历 vector 的时候改变 vector 对象的大小。
  C++ 标准要求 vector 应该能在运行时高效快速地添加元素。因此既然 vector 对象能高效地增长，那么在定义 vector 对象的时候设定其大小就没有什么必要了，只有一种例外，即当所有元素的值都一样。一旦元素的值有所不同，更有效的办法是先定义一个空的 vector 对象，再在运行时向其中添加具体值。
  ### 其它 vector 操作如（很多和 string 类似）：

- `v.empty()`，如果 v 不含有任何元素，返回 true

- `v.size()`，返回 v 中的元素个数

- `v[n]`，返回 v 中第 n 个位置上元素的引用

- `v1 = v2`，v2 中的元素将拷贝替换 v1 的

- `v1 = {a,b,c...}`，列表中的元素将拷贝替换 v1 中的

- `v1 == v2`,`v1 != v2`，元素数量相同，对应位置的元素也相等，则相等

- <,<=,>,>=，比首个相异元素的大小，如都一样，比长度，即字典顺序
  size 返回的类型由 vector 定义的 size_type 类型。

```
vector<int>::size_type    // 正确
vector::size_type         // 错误
```

  只有当元素的值可比较时，vector 对象才能被比较。只能对确已存在的元素执行下标操作。
  ## 迭代器介绍使用迭代器（iterator）是一种通用的访问容器中元素的方法。迭代器有有效和无效之分。有效的迭代器指向某个元素，或指向尾元素的下一个位置，其它情况都属于无效。### 使用迭代器有迭代器的类型同时拥有返回迭代器的成员。

标准库容器都拥有名为 begin 和 end 的成员（函数）。其中 begin 成员负责返回指向第一个元素的迭代器。end 成员负责返回指向容器“尾元素的下一个位置”的迭代器。叫**尾后迭代器（off-the-end iterator）**。
如果容器为空，begin 和 end 都返回尾后迭代器。即：`v.begin()== v.end()`
如：

```
auto b = v.begin();
auto e = v.end();
```

迭代器运算符
标准容器迭代器的运算符：

- `*iter`，返回迭代器所指对象的引用（解引用）
- `iter->mem`，解引用 iter，并获取其成员 mem，等价于`(*iter).mem`
- `++iter`，令 iter 指示容器中的下一个元素
- `--iter`，令 iter 指示容器中的上一个元素
- `iter1 == iter2`，如果两个迭代器指示的是同一个元素，或者它们都是尾后迭代器，则相等，反之不相等
迭代器指示一个元素时，才可对其解引用。对尾后迭代器或者无效迭代器解引用的结果是 UB。
迭代器类型
标准库类型使用 iterator 和 const_iterator 来表示迭代器类型。
如：

```
vector<int>::iterator it1;
vector<int>::const_iterator it2;
```

it1 能读写元素，而 it2 只能读。
认定某个类型是迭代器类型当且仅当它支持一套操作，这套操作使得我们能访问容器的元素，或者从某个元素移动到另外一个元素。
begin 和 end 运算符 begin 和 end 返回的具体类型由对象是否是常量决定。如果对象是常量，返回 const_iterator，否则返回 iterator。
为了专门得到 const_iterator 类型的迭代器，C++11 中可以使用 cbegin 和 cend：

```
auto it = v.cbegin();
```

箭头运算符即`->`，它把解引用和成员访问两个操作结合在一起。即：`(*iter).mem`等价于`iter->mem`。

某些对 vector 对象的操作会使迭代器失效任何一种可能改变 vector 对象容量的操作，比如 push_back，都会使该 vector 对象的迭代器失效。### 迭代器运算
递增运算令迭代器每次移动一个元素，所有的标准库容器的迭代器都支持递增运算，也支持 == 和!= 运算。
string 和 vector 的迭代器提供了额外的运算符，有：

- `iter + n`，新迭代器向前移动若干个元素，它指向容器的一个元素，或是尾后迭代器
- `iter - n`，新迭代器向后移动若干个元素，它指向容器的一个元素，或是尾后迭代器
- `iter1 - iter2`，得到迭代器之间的距离，参与计算的迭代器必须是指向同一个容器中的元素或者尾元素的下一个位置
- `>,>=,<,<=`，比较迭代器所处的位置，前面的小于后面的，参与计算的迭代器必须是指向同一个容器中的元素或者尾元素的下一个位置
  迭代器的算数运算
  迭代器相减的结果的类型是 difference_type，表示右侧的迭代器要移动多少个位置才能到达左侧的。
  difference_type 是一个带符号的整数，string 和 vector 都定义了这个类型。
  迭代器相加没有意义

```
auto mid = (beg) +(end-beg)/2;//正确
auto mid =(beg+end)/2;//错误  迭代器加法不存在
```

string 类本身接受无参数的初始化方式，无论数组定义在函数体内部还是外部都被默认初始化为空串，对于内置类型 int，数组定义在函数体外部时默认初始化为 0，在 main 函数内部时，将不被初始化。

2.3 数组

1. 数组是存放相同类型的对象的容器，这些对象是匿名的。
2. 数组的大小确定不变。
3. 数组是一种内置类型。
   ### 定义和初始化内置数组数组是一种复合类型，其声明形如`a[N]`。N 叫维度，说明了数组中元素的个数，必须大于 0，且必须是一个**常量表达式**，即其值在编译期间已知。
   默认情况下，数组的元素执行默认初始化，这意味着在函数块内定义的执行默认初始化的含内置类型元素的数组，其元素的值未定义。

定义数组的时候必须指定数组的类型，不允许用 auto 关键字由初始值的列表推断类型。数组的元素应为对象，所以不存在存储引用的数组。

显式初始化数组元素即列表初始化，此时可以忽略数组的维度，维度由编译器推断出来。如：

```
int a1[10] = {0}; // 剩下的元素执行值初始化，即为0
int a2[] = {1, 2, 3};
```

字符数组的特殊性可以用字符串字面值对此类数组进行初始化。如：

```
char s[] = "hello";
```

这样初始化的数组包含结尾的空字符。

不允许拷贝和赋值这样的操作是非法的：

```
int a1[] = {1, 2, 3};
int a2[] = a1; // 非法
```

理解复杂的数组声明

4. 定义一个指针数组

```
int* a[10] = {};
```

5. 定义一个指向数组的指针：

```
int (*ptr)[10] = &a;
```

6. 定义一个绑定到数组的引用：

```
int (&a_ref)[10] = a;
```

默认情况下，类型修饰符从右向左依次绑定。不过理解数组的复杂声明时，应该由内向外理解。即从数组的名字开始按照由内向外的顺序阅读。

访问数组元素

使用数组下标的时候，通常将其定义为 size_t 类型，这是一种机器相关的无符号类型。定义在 cstddef 头文件中，是 C 标准库 stddef.h 头文件的 C++ 版本。可以使用范围 for 语句来遍历数组。

```
for (auto i : arr)
cout << i << " ";
cout << endl;
```

得到数组的大小

```
sizeof(array/sizeof(array[0];
```

检查下标的值

与 string 和 vector 一样，数组的下标是否在合理范围之内由程序员负责检查。

指针和数组在很多用到数组名字的地方，编译器都会自动地将其替换为一个**指向数组首元素的指针。**

decltype 下面得到一个数组类型：

```
int a1[10] = {};
decltype(a1) a2;
```

auto 下面得到一个整型指针：

```
int a1[10] = {};
auto a2(a1);
```

指针也是迭代器

string 和 vector 的迭代器支持的运算，指针都支持。使用递增运算符既可以让指向数组元素的指针向前移动到下一个位置上。这样可以获取数组尾元素的下一个位置的指针：

```
int *end = &a[N];
```

不过 C++11 提供了 begin 和 end 函数，可以获取数组首元素的指针和尾后指针：

```
int a[10] = {};
int *beg_p = begin(a);
int *end_p = end(a);
```

这俩函数定义在头文件 iterator.h 中。尾后指针不能解引用和递增操作。和迭代器一样，两个指针相减的结果是它们之间的距离。参与运算的两个指针必须指向同一个数组当中的元素。

下标和指针

对数组执行下标运算其实是对指向数组元素的指针执行下标运算：

```
int i = ia[2];     // ia转换成指向数组首元素的指针
                // ia[2]得到(ia + 2)所指的元素
int *p = ia;       // p指向ia的首元素
i = *(p + 2);      // 等价于i = ia[2]
```

只要指针指向的是数组中的元素，都可以执行下标运算。

内置的下标运算符可以处理负值，这和标准库类型的下标不一样（必须是无符号的）。

C 风格字符串

C 风格的字符串即是字符串字面量，也是一种字符数组，并以空字符结尾（null terminated）。p109 列举了 C

语言标准库提供的一组函数，可以操作 C 风格字符串，他们定义在 cstring 头文件中。

c_str 函数 string 可使用 c_str 函数返回其 C 风格的字符串，如：

```
string s("hello");
const char *c_s = s.c_str();
```

无法保证返回的 C 风格字符串一直有效，因此通常在返回后再把它拷贝到另一个地方 **使用数组初始化 vector 对象**如：

```
int a[] = {1, 2, 3};
vector<int> vec(begin(a), end(a));
```

2.4 多维数组

多维数组，实际上是数组的数组。如：`int a[3][4]`，可由内而外理解，a 是一个含有 3 个元素的数组，每个元素又是一个含有 4 个元素的数组。对于二维数组，常把第一个维度看作行，第二个维度看作列。

多维数组的初始化如：

```
int a[3][4] = {
{0, 1, 2, 3},
{4, 5, 6, 7},
{8, 9, 10, 11}
};
```

列表初始化中未列出的元素执行值初始化。

多维数组的下标引用

如果表达式含有的下标运算符数量和维度一样多，该表达式的结果将是给定类型的元素；否则表达式的结果是内层数组

```
int a[3][4] = {};
int (&row)[4] = a[2]; // row绑定到a的第二个数组上
```

使用范围 for 语句处理多维数组

如果是外层循环，控制变量将得到数组类型。除了最内层的循环外，其他所有循环控制变量都应该是引用类型(因为若不是引用，编译器会认为外层控制变量是指针类型，而无法遍历一个指针)。

指针和多维数组当程序使用多维数组名字时，也会自动将其转换成指向数组首元素的指针。

多维数组的首元素是一个内层数组，故使用多维数组名将得到一个指向内层数组的指针。

即：

```
int a[2][3] = {};
```

```
int (*p)[3] = a;
```

还可以使用 auto 或者 begin 来得到指向内层数组的指针。

类型别名简化多维数组的指针

可以这样定义一个数组类型：

```
using int_arr = int[4]; // C++11
typedef int int_arr[4];
```

指针

7. 指针本身的值（value）；
8. 指针所指的对象（content）；
9. 指针本身在内存中的储存位置（address）

3 表达式

1. `*iter.empty()`和`(*iter).empty()`和`iter->empty()`的区别

2. `somevalue ? ++x,++y:--x,--y;`等价于`(somevalue ? ++x,++y:--x),--y;` # 位运算符

位运算符作用于整数类型的运算对象，并把运算对象看成是二进制位的集合。| 运算符 | 功能 | 用法 | |-|-|-| ||位求反|expr| |<<| 左移 |expr1 << expr2| |>>| 右移 |expr1 >> expr2| |&| 位与 |expr1 & expr2| |^| 位异或 |expr1 ^ expr2| ||| 位或 |expr1 | expr2|

一般来说，如果运算对象是“小整型”，则它的值会被自动提升成较大的整数类型。运算对象可以是带符号的，也可以是无符号的。如果运算对象是带符号的且它的值为负，那么位运算如何处理运算对象的“符号位”依赖于机器。
强烈建议将位运算符用于处理无符号类型。
一个提升例子就是，如果对 char 做位运算，它会先被提升为 int。
移位运算符 <>
运算符的内置含义是对其运算对象执行基于二进制位的移动操作。首先令左侧运算对象的内容按照右侧运算对象的要求移动指定位数，然后将经过移动的（可能还进行了提升）左侧运算对象的拷贝作为求值结果。其中，右侧的运算对象一定不能为负，而且必须严格小于结果的位数，否则就会产生未定义的行为。移出边界之外的位数被舍弃掉了。
左移运算符 <> 的行为依赖于左侧运算对象的类型：如果是无符号的，在左侧插入值为 0 的二进制位；如果是带符号的，在左侧插入符号位的副本或值为 0 的二进制位，如何选择视具体环境而定。
sizeof 运算符
sizeof 运算符返回一条表达式或一个类型名字所占的字节数。sizeof 运算符满足右结合律，其所得的值是一个 size_t 类型的常量表达式。它有两种形式：
- sizeof(type) - sizeof?expr 常量表达式意味着在编译期间就能得到计算。
第二种形式中，sizeof 返回的是表达式结果类型的大小。
sizeof 运算符的结果部分地依赖于其作用的类型：
- 对 char 或者类型为 char 的表达式执行 sizeof 运算，结果得 1。- 对引用类型执行 sizeof 运算得到被引用对象所占空间大小。- 对指针执行 sizeof 运算得到指针本身所占空间的大小。- 对解引用指针执行 sizeof 运算得到指针指向对象所占空间的大小，指针不需要有效。- 对数组执行 sizeof 运算得到整个数组所占空间大小。- 对 string 对象或 vector 执行 sizeof 运算只返回该类型固定部分的大小，不会计算对象中的元素占用了多少空间。# ? 显式转换 **命名的强制类型转换**
一个命名的强制类型转换有如下形式：
cast-name(expression);
其中，type 是转换的目标类型而 expression 是要转换的值。如果 type 是引用类型，则结果是左值。cast-name 是

static_cast,?dynamic_cast,?const_cast 和 reinterpret_cast 中的一种。dynamic_cast 支持运行时识别，直到 19 章（p730）才会讲解。

static_cast

任何具有明确定义的类型转换，只要不包含底层 const，都可以使用 static_cast。

```
//?进行强制类型转换以便执行浮点数除法
double?slope?=?static_cast<double>(j)?/?i;
```

当需要把一个较大的算术类型赋值给较小的类型时，static_cast 非常有用。此时，强制类型转换表示，我们知道并且不在乎潜在的精度损失。

static_cast 对于编译器无法自动执行的类型转换也非常有用。例如，我们可以使用 static_cast 找回存在于 void* 的指针中的值：

```
void?*p?=?&d;????//?正确，任何非常量对象的地址都能存入void*

//?正确，将void*转换回初始的指针类型
double?*dp?=?static_cast<double*>(p);
```

必须确保转换后所得的类型就是指针所指的类型。类型一旦不符，将产生未定义的后果。

const_cast

const_cast 只能改变运算对象的底层 const：

```
const?char?*pc;
char?*p?=?const_cast<char*>(pc);????//?正确，但是通过p写值是未定义的行
    为
```

如果对象本身是一个非常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，执行写操作就会产生未定义的后果。

const_cast 常常用于有函数重载的上下文中，这将在第 6 章介绍（p208）。

reinterpret_cast

reinterpret_cast 通常为运算对象的位模式提供较低层次上的重新解释。比如：

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

我们必须牢记 pc 所指的真实对象是一个 int 而非字符。

reinterpret_cast 非常危险，书中建议尽量避免使用。因为它本质上依赖于机器。且没有介绍应用场景。另外，书中也建议尽量避免其他的强制类型转换，强制类型转换应当在其合适的应用场景中使用。

旧式的强制类型转换

在早期版本的 C++ 语言中，显式地进行强制类型转换包含两种形式：

```
type(expr);???//?函数形式的强制类型转换
```

```
(type)expr;???//?C语言风格的强制类型转换
```

根据所涉及的类型不同，旧式的强制类型转换分别具有 const_cast,?static_cast 或 reinterpret_cast 相似的行为。与命名的强制类型转换相比，旧式的强制类型转换从表现形式上来说不那么清晰明了，容易被看漏，所以一旦转换过程出现问题，追踪起来也更加困难。# try 语句块 try 语句块的通用语法形式是：

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

当选中了某个 catch 子句处理异常之后，执行与之对应的块。catch 一旦完成，程序跳转到 try 语句块最后一个 catch 子句之后的那条语句继续执行。
try 语句块内声明的变量在 catch 子句内无法访问。
一个简要的例子：

```
while (cin >> item1 >> item2) {
    try {
        // ... 可能抛出一个异常的代码
    } catch (runtime_error err) {
        cout << err.what() << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break; // 跳出while循环
    }
}
```

4 throw 表达式

抛出异常的一个例子是：

```
throw runtime_error("Data must refer to same ISBN");
```

该异常是类型 runtime_error 的对象。抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。
- **throw 表达式（throw expression）**，异常检测部分使用 throw 表达式来表示它遇到了无法处理的问题。我们说 throw **引发（raise）**了异常。
- **try 语句块（try block）**，异常处理部分使用 try 语句块处理异常。try 语句块以关键字 try 开始，并以一个或多个 **catch 子句（catch clause）**结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。
- 一套**异常类（exception class）**，用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。
简单语句 C++ 语言中的大多数语句都以分号结束，一个表达式，比如 ival+5，末尾加上分号就变成了**表达式语句**(expression statement)，表达式语句的作用是执行表达式并丢弃掉求值结果：

```
ival + 5;    // 无意义的表达式语句
cout << ival;// 有意义的表达式语句
```

空语句

最简单的语句是**空语句**（null statement），它只有一个分号：

```
; // 空语句
```

如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。

复合语句（块）

复合语句（compound statement）是指用花括号括起来的语句和声明的序列，复合语句也被称作**块**（block）。一个块就是一个作用域。

如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。

所谓空块，是指内部没有任何语句的一对花括号。空块的作用等价于空语句：

```
while (cin >> s && s != sought)
{}  // 空块
```

5 语句作用域

可以在 if、switch、while 和 for 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了：

```
while (int i = get_num()) // 每次迭代时创建并初始化
    cout << i << endl;
i = 0;     // 错误，在循环外部无法访问
```

6 条件语句

C++ 语言提供了两种按条件执行的语句。一种是 if 语句，它根据条件决定控制流；另一种是 switch 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。
if 语句 if 语句的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。
switch 语句 switch 语句提供了一条便利的途径使得我们能够在若干固定选项中做出选择。
迭代语句迭代语句通常称之为循环，它重复执行操作直到满足某个条件才停下来。while 和 for 语句在执行循环体之前检查条件，do while 语句先执行循环体，然后再检查条件。
while 语句语法格式是：

```
while (condition)
    statement
```

只要 condition 的求值结果为真就一直执行 statement。如果 condition 第一次求值就是 false，statement 一次都不执行。
while 的条件部分可以是一个表达式或者是一个带初始化的变量声明。
使用 while 循环
当不确定到底要迭代多少次时，使用 while 循环比较合适。还有一种情况也应该使用 while 循环，这就是我们想在循环结束后访问循环控制变量。
传统的 for 语句
for 语句的语法形式是

```
for (init-statement: condition; expression)
    statement
```

6.1 范围 for 语句

C++11 新标准引入了一种更简单的 for 语句，这种语句可以遍历容器或其他序列的所有元素。**范围 for 语句**（range for statement）的语法形式是：

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

expression 必须是一个序列，比如用花括号括起来的初始值列表、数组、或者 vector 或 string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员。
declaration 定义一个变量，序列中的每个元素都能转换成该变量的类型。
每次迭代都会重新定义循环控制变量，并将其初始化成序列中的下一个值，之后才会执行 statement。
在范围 for 语句中，预存了 end() 的值。一旦在序列中添加（删除）元素，end 函数的值就可能变得无效了。因此不能通过范围 for 语句增加 vector 对象的元素。
do while 语句 do while 语句和 while 语句非常相似，唯一的区别是，do while 语句先执行循环体后检查条件。不管条件的值如何，我们都至少会执行一次循环。do while 语句的语法形式如下：

```
do
    statement
while (condition);
```

6.2 跳转语句

跳转语句中断当前的执行过程。C++ 语言提供了 4 种跳转语句：break, continue, goto 和 return。本章介绍前三种，return 在第六章介绍（p199 页）。
break 语句 break 语句负责终止离它最近的 while, do while, for 或 switch 语句，并从这些语句之后的第一条语句开始执行。## continue 语句 continue 语句终止最近的循环中的当前迭代并立即开始下一次迭代。continue 语句只能出现在 for, while 和 do while 循环的内部。
goto 语句
goto 语句（goto statement）的作用是从 goto 语句无条件跳转到同一函数内的另一条语句。

7 函数

7.1 函数是一个命名了的代码块，我们通过调用函数执行相应的代码。函数可以有 0 个或多个参数，而且（通常）会产生一个结果。

7.2 函数基础

一个典型的函数 (function) 定义包括以下部分: 返回类型 (return type)、函数名字、由 0 个或多个形参 (parameter) 组成的列表以及函数体。

我们通过**调用运算符（call operator）**来执行函数。调用运算符的形式是一对圆括号，它作用于一个表达式，该表达式是函数或者指向函数的指针；圆括号内是一个用逗号隔开的实参列表，我们用实参初始化函数的形参。调用表达式的类型就是函数的返回类型。

函数的调用完成两项工作：一是用实参初始化函数对应的形参，二是将控制权转移给被调用函数。此时，**主调函数**（calling function）的执行暂时被中断，**被调函数**（called function）开始执行。

当遇到一条 return 语句时函数结束执行过程。函数的返回值用于初始化调用表达式的结果。

函数返回类型大多数类型都能用作函数的返回类型。一种特殊的返回类型是 void，它表示函数不返回任何类型。函数的返回类型不能是数组类型或函数类型，但可以是指向数组或函数的指针。

局部对象在 C++ 语言中，名字有作用域，对象有**生命周期（lifetime）**，理解这两个概念非常重要:

- 名字的作用域是程序文本的一部分，名字在其中可见。
- 对象的生命周期是程序执行过程中该对象存在的一段时间。

形参和函数体内部定义的变量统称为**局部变量（local variable）**。它们仅在函数的作用域内可见。

在所有函数体之外定义的对象存在于程序的整个执行过程中。此类对象在程序启动时被创建，直到程序结束才会销毁。局部变量的生命周期依赖于定义的方式。

自动对象

对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在的块末尾时销毁它。我们把只存在于块执行期间的对象称为**自动对象（automatic object）**。

形参是一种自动对象。函数开始时为形参申请存储空间，函数一旦终止，形参就被销毁。

对于局部变量对应的自动对象来说，如果变量定义本身含有初始值，就用这个初始值进行初始化；否则执行默认初始化 (内置类型产生未定义的值)。

局部静态对象（local static object）在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁。例:

```
// 统计函数被调用了多少次
size_t count_calls()
{
    static size_t ctr = 0; // 调用结束后，这个值仍然有效
    return ++ctr;
}
```

7.2.1 函数声明

函数的名字必须在使用之前声明。类似于变量，函数只能定义一次，但可以声明多次。
函数的声明和定义唯一的区别是**声明无须函数体**，用一个分号替代即可。
函数的三要素（返回类型、函数名、形参类型）描述了函数的接口，说明了调用该函数所需的全部信息。函数声明也称作**函数原型（function prototype）**。
在头文件中进行函数声明我们建议函数在头文件中声明，在源文件中定义。这是因为如果把函数声明放在头文件中，就能确保同一函数的所有声明保持一致。而且一旦我们想改变函数的接口，只需改变一条声明即可。定义函数的源文件应该把含有函数声明的头文件包含进来，编译器负责验证函数的定义和声明是否匹配。
分离式编译 C++ 语言支持所谓的**分离式编译（separate compilation）**。分离式编译允许我们把程序分割到几个文件中去，每个文件独立编译。

7.3 参数传递

如果形参是引用类型，它将绑定到对应的实参上；否则，将实参的值拷贝后赋给形参。
当形参是引用类型时，我们说它对应的实参被**引用传递**（passed by reference）或者函数被**传引用调用**（called by reference）。
当实参的值被拷贝给形参时，形参和实参是两个相互独立的对象。我们说这样的实参被**值传递**（passed by value）或函数被**传值调用**（called by value）。
传值参数当初始化一个非引用类型的变量时，初始值被拷贝给变量。此时，对变量的改动不会影响初始值。
指针形参
当执行指针拷贝操作时，拷贝的是指针的值。拷贝之后，两个指针是不同的指针。因为指针使我们可以间接地访问它所指的对象，所以通过指针可以修改它所指对象的值。
熟悉 C 的程序员常常使用指针类型的形参访问函数外部的对象。在 C++ 语言中，建议使用引用类型的形参代替指针。

7.3.1 传引用参数

对于引用的操作实际上是作用于引用所引的对象上，引用形参也是如此。通过使用引用形参，允许函数改变一个或多个实参的值。

使用引用避免拷贝

拷贝大的类类型对象或者容器对象比较低效。甚至有的类型根本就不支持拷贝操作。此时应该使用引用形参访问该类型的对象。

如果函数无须改变引用形参的值，最好将其声明为常量引用。

使用引用形参返回额外信息

一个函数只能返回一个值，然而有时函数需要同时返回多个值，引用形参为我们一次返回多个结果提供了有效的途径。那就是通过引用形参并修改它（也就是修改了其引用的对象），从而作为结果传出。

const 形参和实参当形参是 const 时，必须注意关于顶层 const 的讨论（p57）。

当用实参初始化形参时会忽略形参的顶层 const。即当形参有顶层 const 时，传递给它常量对象或者非常量对象都是可以的。

忽略形参的顶层 const 可能产生意想不到的结果：

```
1 void fcn(const int i) {}
2 void fcn(int i) {}    // 错误：重复定义
```

在 C++ 中，允许我们定义若干具有相同名字的函数，不过前提是不同函数的形参列表有明显的区别。因为顶层 const 被忽略了，所以在上面的代码中传入两个 fcn 函数的参数可以完全一样（从而编译器不知道该调用哪一个）。

指针或引用形参与 const

我们可以使用非常量初始化一个底层 const，但是反过来不行（不能用一个常量初始化一个非底层 const）；同时一个普通的引用必须用同类型的对象初始化。

尽量使用常量引用

把函数不会改变的形参定义成（普通的）引用是一种常见错误，这么做给函数的调用者一种误导，即函数可以修改它的实参的值。此外，使用引用而非常量引用也会极大地限制函数所能接受的实参类型（比如无法传入一个常量对象了）。

比如下面这个例子将导致编译错误（p192）：

```
1 // 不良设计，第一个形参的类型应该是const string&
2 string::size_type find_char(string &s, char c, string::size_type &
     occurs);
3 //...
4 find_char("Hello World", 'o', ctr); // 无法编译通过
```

7.3.2 数组形参

当我们为函数传递一个数组时，实际上传递的是指向数组首元素的指针。

尽管不能以值传递的方式传递数组，但是我们可以把形参写成类似数组的形式：

```
1 // 每个函数都有一个const int*类型的形参
2 void print(const int*);
3 void print(const int[]);    // 可以看出来，函数的意图是作用于一个数组
```

```
4 void print(const int[10]);  // 这里的维度表示我们期望数组含有多少元素，
      实际不一定
```

```
1 和其他使用数组的代码一样，以数组作为形参的函数也必须确保使用数组时不会
      越界。
```

因为数组是以指针的形式传递给函数的，所以一开始函数并不知道数组的确切尺寸，调用者应该为此提供一些额外的信息。管理指针形参有三种常用技术。

1. 使用标记指定数组长度，如 C 风格字符串。

2. 使用标准库规范，如传递首元素和尾后元素的指针，来表示一个范围。

3. 显示传递一个表示数组大小的形参。

数组形参和 const

当函数不需要对数组元素执行写操作的时候，数组形参应该是指向 const 的指针。只有当函数确实要改变元素值的时候，才把形参定义成指向非常量的指针。

数组引用形参

C++ 语言允许将变量定义成数组的引用，基于同样的道理，形参也可以是数组的引用。此时，引用形参绑定到对应的实参上，也就是绑定到数组上。

```
1 // 正确，形参是数组的引用，维度是类型的一部分
2 void print(int (&arr)[10])
3 {
4     for (auto elem : arr)
5         cout << elem << endl;
6 }
```

但这一用法也限制了 print 函数的可用性，我们只能将函数作用于大小为 10 的数组。

传递多维数组

和所有数组一样，当将多维数组传递给函数时，真正传递的是指向数组首元素的指针，也就是一个指向数组的指针。数组第二维（以及后面所有维度）的大小都是数组类型的一部分，不能省略：

```
1 // matrix指向数组的首元素，该数组的元素是由10个整数构成的数组
2 void print(int (*matrix)[10], int rowSize) { /* ... */ }
```

```
1 `*matrix`两端的括号必不可少：`int *matrix[10]//10个指针构成的数组`；`
      int (*matrix)[10]//指向含有10个整数的数组的指针`。
```

也可以使用数组的语法定义函数，此时编译器会一如既往地忽略掉第一个维度：

```
1 // 等价定义
2 void print(int matrix[][10], int rowSize) { /* ... */ }
```

matrix 的声明看起来是一个二维数组，实际上形参是指向含有 10 个整数的数组的指针。
main: 处理命令行选项有时候我们需要给 main 函数传递实参。一种常见的情况是用户通过设置一组选项来确定函数所要执行的操作。例如：

```
prog -d -o ofile data0
```

这些命令行选项通过两个（可选的）形参传递给 main 函数。

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

第二个形参 argv 是一个数组，它的元素是指向 C 风格字符串的指针；第一个参数 argc 表示数组中字符串的数量；argc 至少为 1。
当实参传给 main 函数之后，argv 的第一个元素指向程序的名字或者一个空字符串，接下来的元素依次传递命令行提供的实参。最后一个指针之后的元素值保证为 0。
以上面的为例，argc 应该等于 5，argv 应该包含如下的 C 风格字符串：

```
argv[0] = "prog";    // 或者argv[0]也可以指向一个空字符串
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```

```
当使用argv中的实参时，一定要记得可选的实参从`argv[1]`开始；`argv[0]`保存程序的名字，而非用户的输入。
```

7.3.3 含有可变形参的函数

为了编写能处理不同数量实参的函数，C++11 新标准提供了两种主要的方法：1. 如果所有的实参类型相同，可以传递一个名为 initializer_list 的标准库类型；
2. 如果实参的类型不同，我们可以编写一种可变参数模板，其细节将在 16.4 节介绍（p618）。
C++ 还有一种特殊的形参类型（即省略符），可以用它传递可变数量的实参。这种功能一般只用于与 C 函数交互的接口程序。
initializer_list 形参
initializer_list 是一种标准库类型，用于表示某种特定类型的值的数组。initializer_list 类型定义在同名的头文件中。与 vector 不一样的是，initializer_list 对象中的元素永远是常量值，我们无法改变 initializer_list 对象中元素的值。
省略符形参
省略符形参是为了便于 C++ 程序访问某些特殊的 C 代码而设置的。
省略符形参应该仅仅用于 C 和 C++ 通用的类型。特别应该注意的是，大多数类类型的对象在传递给省略符形参时都无

法正确拷贝。
省略符形参只能出现在形参列表的最后一个位置，它的形式无外乎以下两种：

```
void foo(param_list, ...);
void foo(...);
```

7.4 返回类型和 return 语句

return 语句终止当前正在执行的函数并将控制权返回到调用该函数的地方。
return 语句有两种形式：

```
return;
return expression;
```

7.4.1 无返回值函数

没有返回值的 return 语句只能用在返回类型是 void 的函数中。返回 void 的函数不要求非得有 return 语句，因为在这类函数的最后一句后面会隐式地执行 return。
有返回值函数
只要函数的返回类型不是 void，则该函数内的每条 return 语句必须返回一个值。return 语句返回值的类型必须与函数的返回类型相同，或者能隐式地转换成函数的返回类型。
值是如何被返回的
返回一个值的方式和初始化一个变量或形参的方式完全一样：返回的值用于初始化调用点的一个临时量，该临时量就是函数调用的结果。
如果函数返回引用，则该引用仅是它所引对象的一个别名。
不要返回局部对象的引用或指针
函数完成后，它所占用的存储空间也随之被释放掉。因此，函数终止意味着局部变量的引用将指向不再有效的内存区域。
返回类类型的函数和调用运算符
调用运算符的优先级和点运算符、箭头运算符相同，并且符合左结合律。

```
//调用string对象的size成员，该string对象有shorterstring函数返回
auto sz = shorterstring(s1,s2).size();
```

引用返回左值
函数的返回类型决定函数调用是否是左值。调用一个返回引用的函数得到左值，其他返回类型得到右值。

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

```
}
int main(){
    string s("a value");
    cout<<s<<endl;
    get_val(s,0)='A';
    cout<<s<<endl;
    return 0;
}
```

列表初始化返回值

C++11 新标准规定，函数可以返回花括号包围的值的列表。此处的列表也用来对表示函数返回的临时量进行初始化。如果列表为空，临时量执行值初始化；否则，返回的值由函数的返回类型决定。

主函数 main 的返回值

我们允许 main 函数没有 return 语句直接结束，这样编译器将隐式地插入一条返回 0 的 return 语句，表示执行成功。为了使返回值与机器无关，cstdlib 头文件定义了两个预处理变量，可以用来表示成功与失败：

```
int main()
{
    if (some_failure)
        return EXIT_FAILURE;
    else
        return EXIT_SUCCESS;
}
```

递归如果函数调用了它自身，不管这种调用是直接的还是间接的，都称该函数为**递归函数**（recursive function）。

在递归函数中，一定有某条路径是不包含递归调用的；否则，函数将“永远”递归下去，换句话说，函数将不断地调用它自身直到程序栈空间耗尽为止。

main 函数不能调用它自己。

返回数组指针因为数组不能被拷贝，所以函数不能返回数组。不过，函数可以返回数组的指针或引用。使用**类型别名**（p60）可以简化这种返回类型：

```
typedef int arrT[10];     // arrT是一个类型别名，表示含有10个整数的数组
using arrT = int[10];     // arrT的等价声明
arrT* func(int i);        // func返回一个指向含有10个整数的数组的指针
```

声明一个返回数组指针的函数

返回数组指针的函数形式如下：

```
Type (*function(param_list))[dimension]
```

类似于其他数组的声明，Type 表示元素的类型，dimension 表示数组的大小。（* 表示返回的是一个指针。）例：

```
int (*func(int i))[10];
```

可以按照以下的顺序来逐层理解该声明的含义：
- func(int i)表示调用 func 函数时需要一个 int 类型的实参。
- (*func(int i))意味着我们可以对函数的调用结果执行解引用操作。
- (*func(int i))[10]表示解引用 func 的调用将得到一个大小是 10 的数组。
- int (*func(int i))[10]表示数组中的元素是 int 类型。

使用尾置返回类型

C++ 新标准提供了另一种简化上述 func 声明的方法，就是使用**尾置返回类型（trailing return type）**。任何函数的定义都能使用尾置返回，但是这种形式对于返回类型比较复杂的函数最有效。

尾置返回类型跟在形参列表后面并以一个-> 符号开头。为了表示函数真正的返回类型跟在形参列表之后，我们在本应该出现返回类型的地方放置一个**auto**：

```
// func接受一个int类型的实参，返回一个指针，该指针指向含有10个整数的数组
auto func(int i) -> int(*)[10];
```

使用 decltype

如果我们知道函数返回的指针将指向哪个数组，就可以使用 decltype 关键字声明返回类型（即获得一个数组类型）。例：

```
int odd[] = {1, 3, 5, 7, 9};
int even[] = {0, 2, 4, 6, 8};
// 返回一个指针，该指针指向含有5个整数的数组
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even;
}
```

```
decltype并不负责把数组类型转换成对应的指针，所以decltype的结果只是一个数组，要想表示arrptr返回指针还必须在函数声明时加一个`*`的符号。
```

7.5 函数重载

如果同一作用域内的几个函数名字相同但形参列表不同，我们称之为**重载（overload）函数**。比如：

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

这些函数接受的形参类型不一样，但是执行的操作非常类似。当调用这些函数时，编译器会根据传递的实参类型推断想要的是哪个函数。
函数的名字仅仅是让编译器知道它调用的是哪个函数，而函数重载可以在一定程度上减轻程序员起名字、记名字的负担。
main 函数不能重载。
不允许两个函数除了返回类型以外其他所有的要素都相同。比如：

```
Record lookup(const Account&);
bool lookup(const Account&);     // 错误，与上一个函数相比只有返回类型不同
```

```
 my note: 返回类型不同的函数，也可以是重载的。只要函数名相同而形参有明显的不同。
```

重载和 const 形参

顶层 const 不影响传入函数的对象。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来：

```
Record lookup(Phone);
Record lookup(const Phone);     // 重复声明
```

如果形参是某种类型的指针或引用，则通过区分其指向的是常量对象还是非常量对象可以实现函数重载，此时的 const 是底层的：

```
Record lookup(Account&);        // 此函数作用于Account的引用
Record lookup(const Account&); // 新函数，作用于常量引用
```

这种情况下，当我们传递一个非常量对象时，编译器会优先选用非常量版本的函数（尽管传给常量版本的也可以）。

const_cast 和重载

const_cast 在重载函数的情境中最有用。比如这两个重载函数：

```
// 比较两个string对象的长度，返回较短的那个引用
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}

// 重载
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1), const_cast<const string&>(s2));
```

```
    return const_cast<string&>(r);
}
```

下面重载的版本中，首先将它的实参强制转换成了对 const 的引用，然后调用了 shorterString 函数的 const 版本。const 版本返回对 const string 的引用，这个引用事实上绑定在一个非常量实参上。因此，可以再将其转换回普通的 const&，这显然是安全的。

传入非常量的实参将调用非常量的版本。

调用重载的函数

定义了一组重载函数后，我们需要以合理的实参调用它们。**函数匹配（function matching）**是指一个过程，在这个过程中我们把函数调用与一组重载函数中的某一个关联起来。编译器首先将调用的实参与重载集合中的每一个函数的形参进行比较，然后根据比较的结果决定到底调用哪个函数。

当调用重载函数时有三种可能的结果：

- 编译器找到一个与实参**最佳匹配（best match）**的函数，并生成调用该函数的代码。- 找不到任何一个函数与调用的实参匹配，此时编译器发出**无匹配（no match）**的错误信息。- 有多于一个函数可以匹配，但是每一个都不是明显的最佳选择。此时也将发生错误，称为**二义性调用（ambiguous call）**。### 重载与作用域

一般来说，将函数声明置于局部作用域内不是一个明智的选择。

如果我们在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体。对于函数而言也是如此。如果在内层作用域声明了一个函数，那么外层的同名的函数都将变得不可见，因此无法找到外层的重载版本。

特殊用途语言特性 ### 默认实参这样一种形参，在函数的很多次调用中它们都被赋予一个相同的值，此时，我们把这个反复出现的值称为函数的**默认实参**（default argument）。调用含有默认实参的函数时，可以包含该实参，也可以省略该实参。如：

```
typedef string::size_type sz;
string screen(sz ht = 24, sz wid = 80, char backrnd = ' ');
```

一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值。

使用默认实参调用函数

如果我们想使用默认实参，只要在调用函数的时候省略该实参就可以了。如：

```
string window;
window = screen();     // 等价于screen(24, 80, ' ');
window = stcreen(66); // 等价于screen(66, 80, ' ');
```

函数调用时实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参。

当设计含有默认实参的函数时，其中一项任务是合理设置形参的顺序，尽量让不怎么使用默认值的形参出现在前面，而让那些经常使用默认值的形参出现在后面。

默认实参初始值

局部变量不可以作为默认实参。另外只要表达式的类型可以转换成形参类型，该表达式就可以作为默认实参。

如：

```
int g_a = 0;
void f(int a = g_a);
```

7.5.1 内联函数和 constexpr 函数

调用普通函数比直接写其语句要慢，这是因为调用函数包含一些额外的工作。

内联函数可以避免函数调用的开销

将函数指定为内联函数（inline），通常就是将它在每个调用点上“内联地”展开。

内联说明只是向编译器发出一个请求，编译器可以选择忽略这个请求。

内联机制用于优化规模小，流程直接，频繁调用的函数。

constexpr 函数

是指能用于**常量表达式**的函数。

函数的返回类型及所有形参都得是字面值类型，且函数体内必须有且只有一条 return 语句。如：

```
constexpr int new_sz() { return 8; }
constexpr int foo = new_sz();
```

constexpr 函数被隐式地指定为内联函数。

把内联函数和 constexpr 函数放在头文件内

这是因为内联函数和 constexpr 函数可以多次定义，且必须完全一致。所以把它们都定义在头文件内。

调试帮助程序可以包含一些用于调试的代码，但是这些代码只在开发程序时使用。当应用程序编写完成准备发布时，要先屏蔽掉调试代码。这种方法用到两项预处理功能：assert 和 NDEBUG。

assert 预处理宏

assert 是一种**预处理宏（preprocessor macro）**。所谓预处理宏其实是一个预处理变量，它的行为有点类似于内联函数。assert 宏使用一个表达式作为它的条件：

```
assert(expr);
```

首先对 expr 求值，如果表达式为假（即 0），assert 输出信息并终止程序的执行。如果表达式为真（即非 0），assert 什么也不做。

assert 宏定义在 cassert 头文件中。预处理名字由预处理器而非编译器管理，因此我们可以直接使用预处理名字而无需提供 using 声明。

assert 宏常用于检查“不能发生”的条件。**NDEBUG 预处理变量**

assert 的行为依赖于一个名为 NDEBUG 的预处理变量的状态。如果定义了 NDEBUG，则 assert 什么也不做。默认状态下没有定义 NDEBUG，此时 assert 将执行运行时检查。

我们可以使用一个 #define 语句定义 NDEBUG，从而关闭调试状态。或者使用编译器提供的命令行选项定义预处理变量：

```
$ CC -D NDEBUG main.c
```

这条命令的作用等价于在 main.c 文件的一开始写 #define NDEBUG。
我们可以把 assert 当成调试程序的一种辅助手段，但是不能用它代替真正的运行时逻辑检查，也不能代替程序本身应该包含的错误检查。
除了用于 assert，也可以使用 NDEBUG 编写自己的调试代码。
比如：

```
void print(const int ia[], size_t size)
{
#ifndef NDEBUG
    // __func__是编译器定义的一个局部静态变量，用于存放函数的名字
    cerr << __func__ << "": array size is: " << size << endl;
#endif

// ...
}
```

编译器为每个函数都定义了__func__，除此之外，预处理器还定义了 4 个对于调试程序很有用的名字：
- __FILE__, 存放文件名的字符串字面值。
- __LINE__, 存放当前行号的整型字面值。
- __TIME__, 存放文件编译时间的字符串字面值。
- __DATA__, 存放文件编译日期的字符串字面值。
函数匹配以下述这组函数及其调用为例, 讲述编译器如何确定调用哪个重载函数:

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);     // 调用void f(double, double);
```

确定候选函数和可行函数
函数匹配的第一步是选定本次调用对应的重载函数集，集合中的函数成为**候选函数（candidate function）**。候选函数具备两个特征：
1. 与被调用函数同名。
2. 其声明在调用点可见。
第二步考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为**可行函数（viable function）**。可行函数也有两个特征：
1. 其形参数量与本次调用提供的实参数量相等。
2. 每个实参的类型与对应的形参类型相同，或者能转换成形参的类型。如果没有找到可行函数，编译器将报告无匹配函数的错误。**寻找最佳匹配（如果有的话）**
第三步是从可行函数中选择与本次调用最匹配的函数。在这一过程中，逐一检查函数调用提供的实参，寻找形参类型与

实参类型最匹配的那个可行函数。
如果有且只有一个函数满足下列条件，则匹配成功：
- 该函数每个实参的匹配都不劣于其他可行函数需要的匹配。
- 至少有一个实参的匹配优于其他可行函数提供的匹配。
如果编译器检查了每一个可行函数，没有一个能脱颖而出，则会报告二义性调用错误。
实参类型转换为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级，具体排序如下所示：
1. 精确匹配，包括以下情况：
- 实参类型和形参类型相同。
- 实参从数组类型或函数类型转换成对应的指针类型。
- 向实参添加顶层 const 或者从实参中删除顶层 const。
2. 通过 const 转换实现的匹配（p143）。
3. 通过类型提升实现的匹配（p142）。
4. 通过算数类型转换或指针转换实现的匹配（p142）。
5. 通过类类型转换实现的匹配（参见 14.9 节，p514）。**需要类型提升和算术类型转换的匹配**
函数匹配和 const 实参

```
int calc(char*,char*)
int calc(const char*,const char*)
//区别是他们的指针类型的形参是否指向了常量，属于底层const，合法定义
```

```
int calc(char*,char*)
int calc(char* const,char* const)
//区别是他们的指针类型的形参是否是常量，属于顶层const,非法定义
```

7.6 函数指针

函数指针指向的是函数而非对象。函数的类型由它的返回类型和形参类型共同决定，与函数名无关。例如：

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

该函数的类型是：`bool (const string&, const string&);`
要想声明一个指向该函数的指针，只需要将函数名替换成指针即可：

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

使用函数指针
当我们把函数名作为一个值使用的时候，该函数名自动转换成指针（指向该函数的）。
例如，可以这样给把函数地址赋值给指针：

```
pf = lengthCompare; // pf指向名为lengthCompare的函数
pf = &lengthCompare; // 等价的赋值语句，取地址符是可选的
```

可以直接对指向函数的指针调用该函数，无须解引用指针：

```
bool b1 = pf("Hello", "Hi");
bool b2 = (*pf)("Hello", "Hi"); // 等价调用
bool b3 = lengthCompare("Hello", "Hi"); // 等价调用
```

可以给函数指针赋一个 nullptr 或 0，表示没有指向任何函数。

重载函数的指针

当使用了重载函数时，编译器必须确定一个能和指针类型精确匹配的函数，即返回类型和形参列表都要一样。

函数指针形参

不能定义函数类型的形参，但是形参可以是指向函数的指针。

当把函数名作为实参使用，它会自动转换成指针。

定义一个函数（以及指针）类型的方法有：

- typedef

```
typedef bool Func(int); // Func是函数类型
typedef bool (*FuncP)(int); // FuncP是函数指针类型
```

- decltype

  假如已经有了一个函数：`bool Foo(int);`

  ```
  decltype(Foo) Func;
  decltype(Foo) *FuncP;
  ```

- using

  ```
  using Func = bool(int);
  using FuncP = bool(*)(int);
  ```

  返回指针函数的指针和数组类似，虽然不能返回一个函数，但是能返回指向函数类型的指针。然而，我们必须把返回类型写成指针形式，编译器不会自动地将函数返回类型当成对应的指针类型处理。与往常一样，要想声明一个返回函数指针的函数，最简单的办法是使用类型别名：

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

  其中我们使用类型别名将 F 定义成函数类型，将 PF 定义成指向函数类型的指针。必须时刻注意的是，和函数类型的形参不一样，返回类型不会自动地转换成指针。我们必须显式地将返回类型指定为指针：

```
PF f1(int);//正确：PF是指向函数的指针，f1返回指向函数的指针
F f1(int);//错误：F是函数类型，f1不能返回一个函数
F *f1(int);//正确：显式地指定返回类型是指向函数的指针
```

出于完整性的考虑，有必要提醒读者我们还可以使用尾置返回类型的方式 (参见 6.3.3 节，第 206 页) 声明一个返回函数指针的函数：

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

将 auto 和 decltype 用于函数指针类型

8 第七章类

类的基本思想是**数据抽象** (data abstraction) 和**封装** (encapsulation)。
数据抽象就是**接口 (interface) 与实现 (implementation) 分离**的技术。
接口就是暴露给用户的操作，比如公有的成员函数。
实现就是数据成员、接口的实现、私有的成员函数。
通过**抽象数据类型 (abstract data type)**，来实现数据抽象和封装。
定义抽象数据类型
封装就是隐藏，抽象数据类型隐藏了自己的成员变量，外部只能使用其接口来间接访问其成员。
定义成员函数
类内的所有成员必须声明在类的内部。
类的成员函数可以定义在类的内部，也可以定义在类的外部。
定义在类内部的函数是隐式的 inline 函数。
引入 this
当调用一个成员函数时，实际上是替某个对象调用它。
成员函数通过名为 **this** 的隐式参数来访问此对象。this 指向了此对象的地址。
在成员函数内部，可以省略 this 来访问成员。
this 是一个常量指针，不能够修改其值。
当成员函数中调用另一个成员函数时，将隐式传递 this 指针。

```
1  std::string isbn() const {return this->bookNo;}
```

引入 const 成员函数
参数列表之后，添加 const 关键字，表明传入的 this 指针是一个指向常量对象的指针。故此成员函数内，不能修改成员变量的内容。
const 对象只能调用 const 版本的成员函数（因此如果函数不修改成员变量，那么为了提高灵活性，应该把函数声明成 const 版本的）。
C++ 语言的做法是允许把 const 关键字放在成员函数的参数列表之后，此时，紧跟在参数列表后面的 const 表示 this 是一个指向常量的指针。像这样使用 const 的成员函数被称作常量成员函数（const member function）。
常量对象，以及常量对象的引用或指针都只能调用常量成员函数。
类作用域和成员函数
类本身就是一个作用域。
成员函数的定义必须包含其所属的类名（使用作用域运算符）。

如果成员函数声明为 const 版本的，其定义时，也要在参数列表后加 const。
成员函数体可以随意使用类中的成员，无须在意成员出现的顺序，这是因为编译器分两步处理类：首先编译成员的声明，然后才轮到成员函数体。

定义一个返回 this 对象的函数

可以使用如下语句返回 this 对象：

```
return *this;
```

返回类型使用引用类型，表明返回的就是 this 所指的对象。
一般来说，当我们定义的函数类似于某个内置运算符时，应该令函数的行为尽量模仿这个运算符。比如说内置的赋值运算符把它的左侧运算对象当成左值返回，这种情况下，函数就可以返回 this 对象的引用。
定义类相关的非成员函数
有些函数也提供了操作类对象的方法，但他们不属于类的成员函数。
可以把这些函数放到类的头文件中声明。这些函数也可以看成是类的接口。
有可能会把这些函数声明称友元，从而方便它们直接操作成员变量。
构造函数
类通过一个或几个特殊的成员函数初始化其成员变量，这些函数叫**构造函数（constructor）**。
每当类对象被创建，构造函数就会被执行。
构造函数名和类名一致，无返回类型，可能有多个（参数个数差异），不能是 const 的。
对于 const 对象，构造函数执行完毕后，它才获得 const 属性。

合成的默认构造函数

如果对象没有初始值，它将执行默认初始化。
类通过**默认构造函数 (default constructor)** 来执行默认初始化。如果没有显示定义过构造函数，编译器就会自动生成一个，叫做合成的默认构造函数。
合成的默认构造函数根据如下规则初始化类成员：
- 如果存在类内初始值，使用它来初始化成员
- 否则，对成员执行默认初始化

某些类不能依赖合成的默认构造函数

所谓不能依赖，就是不可以让编译器生成默认构造函数，要自己定义一个。其原因可能是：
- 如果定义了自己的构造函数，那么编译器就不会生成默认的构造函数，此类就没有了默认构造函数。
- 默认构造函数可能执行的是错误的操作，比如内置类型若没有类内初始值，则进行默认初始化，其值未定义。
- 有时候，编译器无法生成默认构造函数，比如类成员中有类，而此类有可能没有默认构造函数。

=default 的含义

C++11 中，使用这种语句来让编译器生成一个默认构造函数：

```
SalesData() = default;
```

```
这种情况下，应当对内置类型的数据成员提供类内初始值，否则应当使用构造函数初始值列表形式的默认构造函数。
```

构造函数初始值列表

```
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){}
```

参数列表后，函数体前的一部分内容叫构造函数初始值列表（constructor initialize list)。
它负责为对象的成员变量赋初值。
如果成员不在初始化列表中，它用类内初始值初始化（如果存在)，否则执行默认初始化。
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。**在类的外部定义构造函数**

```
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
}
```

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。
拷贝、赋值和析构
拷贝构造函数，当初始化变量时以值传递或函数返回一个对象时，会发生拷贝。
赋值运算，当使用了赋值运算符时，会发生对象的赋值操作。
析构函数，当一个变量不在存在时，会执行析构。
这些操作如果不显示定义，编译器就会合成一个，合成的拷贝赋值版本只是做了浅拷贝操作。
某些类不能依赖合成的版本
如果类中有成员绑定了外部的对象（比如动态内存)，那么就不可依赖合成的版本。
可使用容器管理必要的存储空间，当发生拷贝等操作时，容器也会执行正确的拷贝。
访问控制与封装
使用**访问说明符（access specifiers)**加强类的封装性。
- public 说明符之后的成员对外可见，外部可访问，public 成员定义类的接口。
- private 说明符之后的成员对内可见，外部无法访问，即隐藏了实现细节。

class 和 struct
其区别仅仅在于默认的访问权限。class 默认为 private，struct 默认是 public。
作为接口，应当是 public 的，而实现细节（数据成员或相关函数）应当为 private 的。

友元

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）。即在函数或类前面加 friend 关键字。

友元声明只能出现在类的内部。它并非函数声明，函数声明还要在别的地方声明。

一般来说，最好在类定义的开始或结束前的位置集中声明友元。

“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可以随时改变，而无须调整用户级别的代码。

类的其它特性

类成员再探 **定义一个类型成员**

可以在类的内部定义一个类型（使用 typedef 或 using），这个类型也有访问限制。

通常放在类的开头位置。

令成员作为内联函数

规模较小的成员函数适合声明成内联函数（定义时在前面加 inline 即可）。

如果定义在类内的函数，默认就是 inline 的。

inline 成员函数通常定义到类的头文件中，即声明和定义在同一个文件中。

重载成员函数

和普通函数的重载规则一样。只要参数的数量 or 类型有区别，就可以重载。

如果是 const 版本的成员函数（传入 const this），那么也可以重载。因为本质上，其隐式参数 this 的类型改变了。

类数据成员的初始值

可以给类数据成员一个类内初始值。使用等号或者花括号。

返回 *this 的成员函数返回引用的函数是左值的，意味着这些函数（返回 *this）返回的是对象本身而非对象的副本。

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。

但是如此一来（const 成员函数返回 const 引用），就无法继续让返回的对象调用非常量版本的成员函数。一个解决的办法就是**重载一个非常量版本的接口**，定义一个私有的常量版本的函数，负责具体工作，而非常量版本的接口负责调用它，并返回非常量引用。

建议：对于公共代码使用私有功能函数。

类类型

每个类是一个唯一的类型，即使其内容完全一样。

类的声明

可以暂时声明类而不定义它，这叫前置声明（forward declaration）。

这种类型，在没有定义前是一个不完全类型（incomplete type）。这种类型只能在有限的情况下使用：

- 定义指向这种类型的指针 or 引用
- 声明以不完全类型为参数 or 返回值的函数

要创建一个类的对象，则必须已经定义好了这个类，这是因为编译器需要知道类的存储空间大小。

只有被定义，才能访问其成员。

声明一个前置类型的方法：

```
class A;
struct B;
namespace game
{
    class C;    // 前置声明一个在命名空间中的类
}
```

8.0.1 友元再探

类可以把普通函数定义成友元，也可以把类，类的成员函数定义成友元。
友元类有权访问本类的非公有成员。

8.1 类的作用域

一个类就是一个作用域。
类的作用域之外，普通的成员只能通过对象、引用 or 指针访问。对于类型成员的访问，需要使用域运算符::来访问。
名字查找与类的作用域
编译器处理完类的全部声明后，才会处理成员函数的定义。因此成员函数体中可以使用类中定义的任何位置的名字。
成员函数中的名字查找
按如下方式解析：
- 在块内查找声明
- 在类内查找，所有成员都可以被考虑
- 在类的外围作用域中查找
构造函数再探
构造函数初始值列表
如果没有在构造函数的初始值列表中显示初始化成员，那么该成员将执行默认初始化。
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始值。

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

```
ConstRef::ConstRef(int ii){
    i = ii;// 正确
    ci = ii; //错误，不能给const赋值
    ri = i; //错误：ri未被初始化
}
//正确形式
ConstRef::ConstRef(int ii)::i(ii),ci(ii),ri(i){}
```

成员初始化的顺序

成员的初始化顺序和它们在类内的定义顺序一致。

而非其在初始值列表中的顺序，初始值列表只是做了初始化的工作。所以要让初始值列表中的成员顺序与定义顺序一致。

最好使构造函数初始值的顺序与成员声明的顺序一致，尽量避免用某些成员初始化其他成员。**有默认实参的构造函数**

如果构造函数的所有实参都有默认实参，那么它实际上也同时定义了默认构造函数。

委托构造函数

C++11 可以定义委托构造函数（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他自己的初始化过程，或者说它把它自己的一些职责委托给了其他构造函数。

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。

即先执行受委托的构造函数内容，再执行自己的。

默认构造函数的作用

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。

默认构造函数在以下情况发生：

- 不使用初始值定义一个非静态变量或者数组时
- 当类含有类类型的成员且使用合成的默认构造函数时
- 当类类型的成员没有在构造函数初始值列表中显式初始化时

值初始化在以下情况下发生：

- 数组初始化时，若提供的初始值少于数组大小时
- 不使用初始值定义一个局部静态变量时
- 书写形如 T() 的表达式显式请求值初始化时

隐式的类类型转换

如果构造函数只接受一个实参，则它实际上定义了**转换构造函数（converting constructor）**。

即定义了一个隐式转换机制。如 string 的接受一个 const char* 版本的构造函数。

使用 explicit 阻止这种隐式转换机制，explicit 只能放到类内声明构造函数里。

只允许一步类类型转换

聚合类

聚合类（aggregate class）使得用户可以直接访问其成员。当类满足如下条件时，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 - 没有基类，没有 virtual 函数可以使用花括号括起来的成员初始值列表来初始化聚合类对象。

字面值常量类（ Literal Classes）

类也可以是字面值类型。

这样的类可以含有 constexpr 函数成员，且符合 constexpr 函数的所有要求，且是隐式 const 的。
数据成员都是字面值类型的聚合类是字面值常量类。
如果不是聚合类，满足如下条件也是一个字面值常量类：
- 数据成员都是字面值类型
- 至少含有一个 constexpr 构造函数
- 如果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；类类型成员必须使用自己的 constexpr 构造函数
- 类必须使用析构函数的默认定义

声明静态成员

在声明前加 static 关键字。
静态成员可以是 public 或 private。数据成员可以是常量，引用，指针，类类型等。
对象不包含与静态数据成员有关的数据。
静态函数不包含 this 指针。

使用类的静态成员

使用作用域运算符访问静态成员。
类的对象、引用或指针可以访问静态成员。
类的成员函数可以直接访问静态成员。

定义静态成员

static 只能出现在类的内部，不能出现在外部。
静态数据成员不属于类的对象，不是有构造函数初始化的。静态数据成员定义在函数体之外，一旦定义，就一直存在于程序的整个生命周期中。

```
1  double T::a = 1; // 定义并初始化一个静态成员
```

静态成员的类内初始化

通常，不应该在类内初始化静态数据成员。
不过，可以为静态成员提供 const 整数类型的类内初始值，且要求静态成员必须是字面值常量类型。

9 IO 库

9.1 IO 类

为了支持不同种类的 IO 处理操作，标准库定义了这几种类型：
- 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

它们分别定义在同名的头文件中。

IO 类型间的关系

类型 ifstream 和 istringstream 都继承自 istream。我们可以像使用 istream 对象一样来使用它们。对于 ostream 也是如此。

IO 对象无拷贝或赋值

由于不能拷贝 IO 对象，因此也不能将形参或返回类型设置为流类型。进行 IO 操作的函数通常以引用方式传递或返回流。

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。

条件状态 IO 类定义了一些函数和标志，可以帮助我们访问和操纵流的条件状态。见 p279。一个 IO 错误的例子：

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

如果试图在标准输入上键入 Boo，读操作就会失败，cin 进入错误状态。

如果输入一个文件结束符标识，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。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
```

9.1.1 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。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。这样可以带来很大的性能提升。
导致缓冲区刷新的原因有：
- 程序正常结束
- 缓冲区满时
- 使用操纵符，如 endl，来显式刷新缓冲区
- 读 cin 或写 cerr，都会导致 cout 的缓冲区被刷新

刷新输出缓冲区

IO 库还提供了两个操纵符用于刷新缓冲区：
- flush 刷新缓冲区，但不输出任何额外字符
- ends 向缓冲区插入一个空字符，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。

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

```
如果程序崩溃，输出缓冲区不会刷新
```

9.2 文件输入输出

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。见 p283。

使用文件流对象

当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，如果提供了一个文件名，则 open 会被自动调用：

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

```
当一个fstream对象被销毁时，close会自动被调用。
```

9.2.1 文件模式

每个流都有一个关联的文件模式，用来指出如何使用文件。见 p286。
每个文件流类型都定义了一个默认的文件模式，当未指定文件模式时，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。

以 out 模式打开文件会丢失已有数据
默认情况下，当我们打开一个 ofstream 时，文件的内容会被丢弃。
阻止丢弃的方法是同时指定 app 模式：

```
1 ofstream out("file1");     // 文件被截断
2 ofstream app("file2", ofstream::app);     // 保留文件内容，写操作在文件
      末尾进行
```

9.3 string 流

sstream 头文件定义了三个类型来支持内存 IO：
- istringstream 从 string 读取数据。
- ostringstream 向 string 写入数据。
- stringstream 既可以从 string 读数据，也可以向 string 写数据。

sstream 增加了一些成员来管理与流相关联的 string。见 p287。
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。
使用 ostringstream
当我们逐步构造输出，希望最后一期打印时，ostringstream 是很有用的。
顺序容器
顺序容器概述
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
顺序容器有：vector, deque, list, forward_list, array, string。
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，在其中间添加或删除元素就会非常耗时，因为这需要移动插入或删除位置之后的所有元素。而且，添加元素可能导致分配额外的存储空间，这种情况下，每个元素都会移动到新的存储空间中。
list 和 forward_list 两个容器添加和删除操作都很快速。作为代价，它们不支持元素的随机访问，为了访问一个元素，只能遍历整个容器。与 vector、deque 和 array 相比，这两个容器的额外内存开销也很大。
deque 支持快速随机访问，在 deque 的中间位置插入或删除元素的代价（可能）很高。但是，在 deque 的两端添加

或删除元素都是很快的。

forward_1ist 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。

确定使用哪种容器

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。

容器库概览 **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。

迭代器

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。比如解引用操作。

表 3.6（96 页）列出了容器迭代器支持的所有操作。表 3.7（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。

迭代器范围

迭代器范围由一对迭代器表示，通常被称为 begin 和 end，它们标记了容器中元素的一个范围。这个范围被称为左闭合区间：[begin, end)

使用左闭合区间蕴含的编程假定

假定 begin 和 end 构成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

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。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，且都可以指定容器大小和元素初始值的参数。

将一个容器初始化为另一个容器的拷贝

方法有两种：

- 直接拷贝整个容器，两个容器的类型和元素的类型都必须匹配。
- 拷贝一个迭代器范围，容器类型不一定匹配，且元素类型只要能够转换即可。

```
1 //每个容器有三个元素，用给定的初始化器进行初始化
2 list<string> authors={"Milton", "Shakespeare", "Austen"};
3 vector<const char*> articles={"a", "an", "the"};
4 1ist<string>1ist2(authors);//正确：类型匹配
5 deque<string>authList(authors);//错误：容器类型不匹配
6 vector<string>words(articles);//错误：容器类型必须匹配
7 //正确：可以将const char*元素转换为
8 string forward_list<string> words(articles.begin(), articles.end());
```

列表初始化

```
list<const char *> articles = {"a","an","the"};
```

标准库 array 具有固定大小

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;
```

9.3.1 赋值和 swap

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的元素的拷贝。

```
c1 = c2;
ca = {a,b,c};
```

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>al={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
array<int, 10>a2={0};//所有元素值均为0
al=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 asign，也不允许用花括号包围的值列表进行赋值。

使用 assign(仅顺序容器)

赋值运算要求两边容器类型和元素类型相同。顺序容器（除了 array）还定义了一个名为 assign 的成员，允许从一个相容的序列中赋值。

使用 swap

调用 swap 操作后，两个容器中的元素将会交换。

除了 array，交换两个容器的操作保证会很快，因为元素本身并未交换，swap 只是交换了两个容器的内部数据结构。

容器大小操作

每个容器都支持这些大小相关的操作：

- 成员函数 size，返回容器中元素的数目，forward_list 不支持；
- empty，当 size 为 0 时返回 true，否则返回 false；
- max_size，返回一个大于或等于该容器所能容纳的最大元素数的值，这是一个很大的值。

关系运算符

每个容器都支持相等运算符（== 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。

比较两个容器实际上是进行元素的逐对比较。

只有当元素类型定义了相应的比较运算符时，才可以使用关系运算符比较两个容器。

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、添加及删除。

向顺序容器添加元素标准库容器提供了灵活的内存管理。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，p305。

```
这些操作会改变容器的大小；array不支持这些操作。
forward_list有自己专有版本的insert和emplace；参见9.3.4节（第312页）。
forward_1ist不支持 push_back和emplace_back。
vector和string不支持push front和emplace front。
c. push back(t);
c. emplace back(args);
c. push_front(t);
c. emplace_front(args);
c. insert(p,t);
c. emplace(p, args);
c. insert(p,n,t)c. insert(p,b,e);
c. insert(p, il);
```

```
向一个deque、string或vector插入元素会使所有指向容器的迭代器、引用和指针失效。
将元素插入到deque、string或vector中的任何位置都是合法的。然而，这样做可能很耗时。
```

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。

访问元素

表9.6（p310）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。

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
```

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。保证下标合法是程序员的责任，

编译器不检查越界错误。
如果想确保下标是合法的，可以使用 at 成员函数。at 成员函数类似下标运算符，如果下标越界，at 会抛出一个 out_of_range 异常。
删除元素删除 deque 中除首尾之外的任何元素都会使所有迭代器、引用、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。
删除元素之前，程序员必须确保它们是存在的。

```
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 popback；vector和string不支持pop_front。
c.pop_back()
c.pop_front()
c.erase(p)
c.erase(b,e)
c.clear()
```

9.3.2 改变容器大小

可以使用 resize 来增大或缩小容器。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。
resize 接受一个可选的元素指参数，用来初始化新添加的元素。如果未提供，新元素进行值初始化。

```
c.resize(n);
c.resize(n,t);
```

9.3.3 容器操作可能使迭代器失效

使用失效的迭代器、引用、或指针是一种严重的错误。
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，那么所有的迭代器都会失效。如果空间未重新分配，指向插入位置之前的元素的迭代器仍有效，但之后的迭代器会失效。
- 对于 list 和 forward_list，指向容器的迭代器仍有效。
当从容器中删除元素后：
- 对于 list 和 forward_list，指向容器其他位置的迭代器仍有效。
- 对于 string 和 vector，被删除元素之前的元素的迭代器仍有效。
vector 对象是如何增长的 **管理容量的成员函数**

```
//shrink to_fit 只适用于vector、string 和deque。
//capacity和 reserve 只适用于vector和string。
```

```
c.shrink_to_fit();//请将 capacity()减少为与size()相同大小
c.capacity();//不重新分配内存空间的话，c可以保存多少元素
C.reserve(n);//分配至少能容纳n个元素的内存空间
```

9.4 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。
构造 string 的其他方法
使用下面这些方法可以构造 string：
以下 n, len2, pos2 都是无符号值。
| 方法 | 说明 | |-|-| |string s(cp, n)|s 是 cp 指向的数组中前 n 个字符的拷贝 | |string s(s2, pos2)|s 是 string s2 从下标 pos2 开始的字符拷贝 | |string s(s2, pos2, len2)|s 是 string s2 从下标 pos2 开始 len2 个字符的拷贝，不管 len2 的值是多少，构造函数至多拷贝 s2.size() - pos2 个字符 | **substr 操作**
substr 返回一个 string，它是原始 string 的一部分或全部的拷贝。
`s.substr(pos, n)` 返回一个 string，包含 s 中从 pos 开始的 n 个字符的拷贝。pos 默认为 0,n 默认为 s.size() - pos，即拷贝从 pos 开始的所有字符。
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, erase 操作。除此之外，它还定义了额外的 insert 和 erase 版本。即使用下标的版本。

```
s.insert(s.size(),5,"!");//在s末尾插入五个!
```

这些函数都拥有许多重载的版本。
assign 版本还接受 C 风格字符串：**需要以空格结尾**
append 和 replace 是额外的成员函数，append 在 string 末尾进行插入操作，replace 替换内容，它是调用 erase 和 insert 的一种简写形式：

```
string s("C++ Primer 4th Ed.");
//从位置11开始，删除三个字符并插入Fifth;
s.replace(11,3,"Fifth)
```

9.4.1 string 搜索操作

string 提供了 6 个搜索函数，它们都有 4 个重载版本。它们都返回一个 string::size_type 的值作为匹配位置（下标）。如果搜索失败，返回 string::npos，其值为 -1。
可以给函数一个搜索的起始位置 pos，它默认值是 0：`auto pos = s.find_first_of(numbers, pos);`

```
string name("guohaoxin01236578");
auto pos1 = name.find("guo");//pos1==0返回字符串guo第一次出现的位置
numbers = "0123456789";
auto pos2 = name.find_first_of(numbers);//寻找numbers字符串中任意字符出现的位置，find_first_not_of
```

指定从哪里开始搜索

```
string size_type pos = 0;
while((pos=name.find_first_of(numbers,pos))!=string::npos){
    cout<<"found number at index:"<<pos<<" element is "<<name[pos]<<endl;
    ++pos;//移动到下一字符
}
```

9.4.2 compare 函数

这是字符串比较函数，和 C 标准库的 strcmp 很相似。### 数值转换
标准库提供了数值转换的函数。

```
to_string(val)
stoi/l/ul/ll/ull/f/d/ld//转换成int、double、float
```

如果 string 不能转换成一个数值，那么会抛出一个 invalid_argument 的异常。如果转换得到的数值无法用任何类型来表示，则抛出一个 out_of_range 异常。
容器适配器三个容器适配器：stack(栈适配器),queue,priority_queue(队列适配器)。
定义一个适配器 stack stk;

10 泛型算法

标准库并未给每个容器都定义成员函数来实现一些特殊的操作，如查找元素、替换或删除元素、重排元素等。而是定义了一组泛型算法。它们实现了一些经典算法的公共接口，可以用于不同类型的元素和多种容器类型，包括内置的数组类型。

10.1 概述

大多数算法定义在头文件 algorithm 中，头文件 numeric 中定义了一组数值泛型算法。
通常，算法并不直接操作容器，而是遍历由两个迭代器指定的一个元素范围来进行操作。
算法不依赖于容器，但依赖于元素类型的操作。比如，find 用元素类型的 == 运算符完成序列中的元素与给定值的比较。
大多数算法提供了一种方法，允许我们使用自定义的操作来代替默认的运算符（即使用谓词）。
迭代器令算法不依赖于容器，但算法依赖于元素类型的操作。
初识泛型算法
附录 A 按照操作方式列出了所有的算法。
除了少数例外，标准库算法都对一个范围内的元素进行操作。我们将此元素范围称为“输入范围”。
理解算法的最基本的方法就是了解它们是否读取元素、改变元素或是重排元素顺序。
只读算法
一些算法只会读取其输入范围内的元素，而从不改变元素。比如 find、accumulate。

```
int sum = accumulate(vec.cbegin(),vec.cend(),0);//求和， 和的初值为0；
string sum =accumulate(v.cbegin(),v.cend(),string(""));//string定义了字
    符串的 “+” 法，
//错误， const char *上没有定义+运算符
string sum =accumulate(v.cbegin(),v.cend(),"");
```

操作两个序列的算法
举一个列子：equal 算法，它比较两个序列中的元素。此算法接受三个迭代器：前两个表示第一个序列中的元素的范围，第三个表示第二个序列的首元素：

```
// roster2中的元素数目应该至少与roster1一样多
```

```
2  equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

这样的算法基于一个非常重要的假设：它假定第二个序列至少与第一个序列一样长。

写容器元素的算法

一些算法将新值赋予序列中的元素。当我们使用这类算法时，必须注意确保序列原大小至少不小于我们要求算法写入元素数目（note：如容器大小足够）。

这样的算法比如 fill。

介绍 back_inserter

一种保证算法有足够元素空间来容纳输出数据的方法是使用**插入迭代器**（insert iterator）。插入迭代器是一种向容器中添加元素的迭代器。当我们通过一个插入迭代器赋值时，一个与赋值号右侧值相等的元素被添加到容器中。

拷贝算法

拷贝 (copy) 算法是另一个向目的位置迭代器指向的输出序列中的元素写入数据的算法。此算法接受三个迭代器，前两个表示一个输入范围，第三个表示目的序列的起始位置。此算法将输入范围中的元素拷贝到目的序列中。传递给 copy 的目的序列至少要包含与输入序列一样多的元素，这一点很重要。

我们可以用 copy 实现内置数组的拷贝，如下面代码所示：

```
1  int al[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2  int a2[sizeof(a1)/sizeof(*al)];//a2与al大小一样
3  //ret指向拷贝到a2的尾元素之后的位置
4  auto ret=copy(begin(a1), end(al), a2);//把a1的内容拷贝给a2
```

copy 返回的是其目的位置迭代器（递增后）的值。即，ret 恰好指向拷贝到 a2 的尾元素之后的位置。

重排元素的算法

某些算法会重排容器中元素的顺序，比如 sort，它利用元素类型的 return type { function body } 其中，capture list是一个lambda所在函数中定义的局部变量的列表。 可以忽略返回类型，这时会自动推断返回类型。c++ auto func = { return 42; }; ### lambda捕获和返回 当定义一个lambda时，编译器生成一个与lambda对应的新的（未命名的）类类型。当向一个函数传递一个lambda时，同时定义了一个新类型和该类型的一个对象。类似地，当使用**auto**定义一个用lambda初始化的变量时，定义了一个从lambda生成的类型的对象。

默认情况下，从lambda生成的类都包含一个对应该lambda所捕获的变量的数据成员。类似任何普通类的数据成员，lambda的数据成员也在lambda对象创建时被初始化。 变量捕获的方式可以是值或引用。值捕获是变量的拷贝，引用捕获是变量的引用。 当以引用方式捕获一个变量时，必须保证在lambda执行时变量是存在的。

建议： 尽量保持lambda的变量捕获简单化。如果可能的话，应该避免捕获指针或引用。见p351。

隐式捕获 可以让编译器根据lambda体中的代码来推断要使用哪些变量。为了指示编译器推断捕获列表，应在捕获列表中写一个&或=。&告诉编译器采用捕获引用方式，=则表示采用值捕获方式。 如：c++ // sz 为隐式捕获，值捕获方式 wc = find_if(words.begin(), words.end(), = { return s.size() >= sz; }); 详见lambda捕获列表，p352。 **可变lambda** 默认情况下，对于一个值拷贝的变量，lambda不会改变其值。如果希望改变，必须在参数列表后加上关键字**mutable**。c++ void fcn3() { size_t v1 = 42; // f 可以改变它捕获的变量的值 auto f = v1 mutable { return ++v1; }; v1 = 0; auto j = f(); // j 为 43 } ### 参数绑定 对于那种只在一两个地方使用的简单操作，lambda表达式是最有用的。如果需要在很多地方

使用相同的操作，或者一个操作需要很多语句完成，通常应该定义一个函数。 如果lambda的捕获列表为空，通常可以用函数来代替它。但如果捕获列表不为空就不能直接代替了。 **标准库bind函数** 为了解决这个问题，可以使用一个新的名为bind的标准库函数，它定义在头文件functional中。它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。c++ auto newCallable = bind(callable, arg_list); newCallable本身是一个可调用对象，arg_list是一个逗号分隔的参数列表，对应给定的callable参数。即，当我们调用newCallable时，newCallable会调用callable，并传递给它arg_list中的参数。 arg_list中的参数可能包含形如`_n`的名字，这些参数是“占位符”，表示newCallable的参数。比如：`_1`为newCallable的第一个参数，`_2`为第二个参数。 **使用placeholders名字** 名字`_n`都定义在一个名为placeholders的命名空间中，这个命名空间本身定义在std命名空间中。 一种简单的**using**语句是：c++ using namespace names-pace_name; 这种形式说明希望所有来自namespace_name的名字都可以在我们的程序中直接使用。如：c++ using namespace std::placeholders; 这使得placeholders定义的所有名字都可用。 ## 再探迭代器 除了每个容器的迭代器，标准库在头文件iterator中还定义了额外几种迭代器。 - 插入迭代器：这些迭代器被绑定到一个容器上，可以用来向容器插入元素。 - 流迭代器：这些迭代器被绑定到输入或输出流上，可以来遍历所关联的IO流。 - 反向迭代器：这些迭代器向后而不是向前移动。 - 移动迭代器：不拷贝其中的元素，而是移动它们。将在13.6.2节（p480页）介绍。 ### 插入迭代器 插入器是一种迭代器适配器，它接受一个容器，生成一个迭代器，能实现向给定容器添加元素。当我们通过一个插入迭代器进行赋值时，该迭代器调用容器操作来向给定容器的指定位置插入一个元素。c++ it = t; // 在 it 指定的当前位置插入值 t。插入迭代器有三种类型，差异在于元素插入的位置： - back_inserter，创建一个使用push_back的迭代器。 - front_inserter，创建一个使用push_front的迭代器。 - inserter，创建一个使用insert的迭代器。此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器。元素将被插入到给定迭代器所表示的元素之前。c++ list lst = {1,2,3,4}; liatlst2,lst3;//空的 list //copy 完成后 lst2 包含 4 3 2 1 copy(lst.begin(),lst.end(),front_inserter(lst2)); //opy 完成后 lst3 包含 1 2 3 4 copy(lst.begin(),lst.end(),inserter(lst3,lst3.begin()));

```
### iostream迭代器
istream_iterator读取输入流，ostream_iterator向一个输出流写数据。这些迭代器将它们对应的流当作一个特定类型的元素序列来处理。
通过使用流迭代器，我们可以使用泛型算法从流对象读取数据以及向其写入数据。
详细操作见p359。
### 反向迭代器
反向迭代器就是在容器中从尾元素向首元素反向移动的迭代器。对于反向迭代器，递增（以及递减）操作的含义会颠倒过来。
可以通过rbegin，rend，crbegin，crend成员函数来获得反向迭代器。这些成员函数返回指向容器尾元素和首元素之前一个位置的迭代器。
## 泛型算法结构
任何算法的最基本的特性是它要求其迭代器提供哪些操作。算法所要求的迭代器操作可以分为5个迭代器类别。

|迭代器|要求|
|-|-|
```

```
| 输入迭代器 | 只读，不写；单遍扫描，只能递增 |
| 输出迭代器 | 只写，不读；单遍扫描，只能递增 |
| 前向迭代器 | 可读写；多遍扫描，只能递增 |
| 双向迭代器 | 可读写；多遍扫描，可递增递减 |
| 随机访问迭代器 | 可读写，多遍扫描，支持全部迭代器运算 |

### 5类迭代器
类似容器，迭代器也定义了一组公共操作。一些操作所有迭代器都支持，另一些只有特定类别的迭代器才支持。
如ostream_iterator只支持递增、解引用和赋值。vector、string、deque的迭代器除了这些操作，还支持递减、关系和算术运算。
除了输出迭代器之外，一个高层类别的迭代器支持低层类别迭代器的所有操作。
### 算法的形参模式
大多数算法具有如下4种形式之一：
```

alg(beg, end, other args); alg(beg, end, dest, other args); alg(beg, end, beg2, other args); alg(beg, end, beg2, end2, other args);

```
其中，alg是算法名字，beg和end表述输入范围。几乎所有算法都有一个输入范围。
**接受单个目标迭代器的算法**
dest参数是一个表示算法可以写入目的位置的迭代器。算法假定（assume）：按其需要写入数据，不管写入多少个元素都是安全的。
一般dest被绑定到一个插入迭代器或是一个ostream_iterator。插入迭代器会将新元素添加到容器中，因为保证空间是足够的。
**接受第二个输入序列的算法**
接受beg2或beg2和end2的算法用这些迭代器表示第二个输入范围。
接受单独beg2的算法假定从beg2开始的序列与beg和end所表示的范围至少一样大。
### 算法命名规范
除了参数规范，算法还遵循一套命名和重载规范。
**一些算法使用重载形式传递一个谓词**
函数的一个版本用元素类型的运算符来比较元素；另一个版本接受一个额外的谓词参数，来代替<或==：
```

unique(beg, end); unique(beg, end, comp);

```
**\_if版本的算法**
接受一个元素值的算法通常有另一个不同名的（不是重载的）版本，该版本接受一个谓词代替元素值。接受谓词参数的算法都有附加的\_if前缀：
```

find(beg, end, val); find_if(beg, end, pred);

```
**区分拷贝元素的版本和不拷贝的版本**
默认情况下，重排元素的算法将重排后的元素写回给定的输入序列中。这些算法还提供另一个版本，将元素写到一个指定的输出目的位置。
```

reverse(beg, end); reverse_copy(beg, end, dest);

```
## 特定容器的算法
链表类型list定义了几个成员函数形式的算法。通用版本的sort要求随机访问迭代器，因此不能用于list。
链表类型定义的其他算法的通用版本可以用于链表，但代价太高。这些算法需要交换输入序列中的元素。一个链表可以通过改变元素间的链接而不是真的交换它们的值来快速“交换”元素。因此，这些链表版本的算法的性能比对应的通用版本好得多。
这些算法见p369。
**链表特有的操作会改变容器**
多数链表特有的算法与通用版本的很相似，但不完全相同，其中一个至关重要的区别是链表版本的会修改底层的容器。例如，remove 的链表版本会删除指定的元素，unique 的链表版本会删除第二个和后继的重复元素。
    对于通用版本的，如 std::remove ，不会删除容器的元素。它只会迁移元素。之后需要调用 erase 才能执行确切的删除动作。
```

```
# 关联容器

关联容器与顺序容器有着根本的不同：

- 关联容器中的元素是按关键字来保存和访问的。

- 顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。

关联容器支持高效的关键字查找和访问，有两个主要的关联容器：

- map，其元素是一些关键字-值对，关键字起到索引作用，值则表示与之相关的数据。

- set，每个元素只包含一个关键字。

标准库提供8个关联容器，如表11.1所示。这8个容器间的不同体现在三个维度上：每个容器
1. 或者是一个set，或者是一个map；
2. 或者要求不重复的关键字，或者允许重复关键字；
3. 按顺序保存元素，或无序保存。

允许重复关键字的容器的名字中都包含单词multi；不保持关键字按顺序存储的容器的名字都以单词unordered开头。因此一个unordered multi set是一个允许重复关键字，元素无序保存的集合，而一个set则是一个要求不重复关键字，有序存储的集合。无序容器使用哈希函数来组织元素。


## 使用关联容器

map是关键字-值对的集合，通常被称为**关联数组**。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。我们通过一个关键字而不是位置来查找值。与之相对，set就是关键字的简单集合。
```

//统计每个单词在输入中出现的次数 map>word) ++word_count[word]; for(const auto &a:word_count) //打印结果 cout<<a.first<<“occurs”<<w.second<<((w.second>1)?“times”:“time”< word_count; // string 到 size_t 的空 map set exclude = {“The”, “the”, “And”, “and”};

```
string word;
while (cin >> word) {
    // 只统计不在exclude中的单词
    if (exclude.find(word) == exclude.end())
        ++word_count[word]; // 提取word的计数器并将其加1
}
```

```
7
8  for (const auto &w : word_count) // 对map中的每个元素
9      // 打印结果
10     cout << w.first << " occurs time: " << w.second << endl;
```

‘‘‘

10.2 关联容器概述

关联容器（有序的和无序的）都支持 9.2 节（第 294 页）中介绍的普通容器操作。关联容器不支持顺序容器的位置相关的操作，例如 push_front。

除了与顺序容器相同的操作之外，关联容器还支持一些顺序容器不支持的操作（见 p388）和类型别名（见 p381）。

关联容器的迭代器都是双向的。

10.2.1 定义关联容器

```
1  map<string, size_t> word_count; // 空容器
2  set<string> exclude = {"the", "but", "and"}; // 列表初始化
3
4  // 三个元素；authors将姓映射为名
5  map<string, string> authors = {
6      {"Joyce", "James"},
7      {"Austen", "Jane"},
8      {"Dickens", "Charles"}
9  };
```

初始化 multimap 或 multiset

一个 map 或 set 中的关键字必须是唯一的，即，对于一个给定的关键字，只能有一个元素的关键字等于它。

multimap 和 multiset 没有此限制，它们都允许多个元素具有相同的关键字（这些元素会相邻存储）。

10.2.2 关键字类型的要求

对于有序容器，关键字类型必须定义元素比较的方法，默认情况下，标准库使用关键字类型的 bookstore(compareIsbn); ‘‘‘

10.2.3 pair 类型

pair 类型定义在头文件 utility 中。

一个 pair 保存两个数据成员，pair 是一个用来生成特定类型的模板。

```
pair<string, string> anon; // 保存两个string
pair<string, vector<int>> line; // 保存string和vector<int>
```

pair 的默认构造函数对数据成员进行值初始化。也可以为每个成员提供初始化器：

```
pair<string, string> author{"James", "Joyce"};
```

pair 的数据成员是 public 的，两个成员分别是 first，second。

创建 pair 对象的函数

```
pair<string, int>
process(vector<string> &v)
{
    // 处理v
    if (!v.empty())
        return {v.back(), v.back().size()}; // 列表初始化
    else
        return pair<string, int>(); // 隐式构造返回值
}
```

10.3 关联容器操作

除了表 9.2(第 295 页) 中列出的类型，关联容器还定义了这些类型：

- key_type, 此容器类型的关键字类型
- mapped_type, 每个关键字关联的类型，只适用于 map
- value_type, 对于 set，与 key_type 相同，对于 map, 为pair<const key_type, mapped_type>

10.3.1 关联容器迭代器

当解引用一个关联容器迭代器时，我们会得到一个类型为容器的 value_type 的值的引用。对 map 而言，value_type 是一个 pair 类型。

```
必须记住，一个map的value_type是一个pair，我们可以改变pair的值，但不能改变关键字成员的值。
```

set 的迭代器是 const 的

与不能改名 map 元素的关键字一样，一个 set 中的关键字也是 const 的。可以用一个 set 迭代器来读取元素的值，但不能修改。

遍历关联容器

map 和 set 类型都支持 begin 和 end 操作，我们可以利用这些函数获取迭代器，然后用迭代器来遍历容器。

```
auto map_it = word_count.cbegin();
while (map_it != word_count.cend()) {
    // ...
    ++map_it; // 递增迭代器，移动到下一个元素
}
```

!!!note 当使用一个迭代器遍历一个 map、multimap、set 或 multiset 时，迭代器按关键字升序遍历元素。

关联容器和算法

我们通常不对关联容器使用泛型算法。更多讨论见书本 p383。

10.3.2 添加元素

关联容器的 insert 成员向容器中添加一个元素或一个元素范围。由于 map 和 set 包含不重复的关键字，因此插入一个已存在的元素对容器没有任何影响。

向 map 添加元素

对一个 map 进行 insert 操作时，必须记住元素类型是 pair。

```
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
word_count. insert(map<string, size_t>:: value_type(word,1));
```

检测 insert 的返回值

insert（或 emplace）返回的值依赖于容器类型和参数。对于不包含重复关键字的容器，添加单一元素的 insert 和 emplace 版本返回一个 pair，告诉我们插入操作是否成功。**pair 的 first 成员是一个迭代器**，指向具有给定关键字的元素；second 成员是一个 bool 值，指出元素是插入成功还是已经存在于容器中。如果关键字已在容器中，则 insert 什么事情也不做，且返回值中的 bool 部分为 false。如果关键字不存在，元素被插入容器中，且 bool 值为 true。

展开递增语句

```
++ret.first->second;
`++((ret.first)->second)`//等价的表达式
```

-. ret 保存 insert 返回的值，是一个 pair。-. ret.first 是 pair 的第一个成员，是一个 map 迭代器，指向具有给定关键字的元素。-. ret.first-> 解引用此迭代器，提取 map 中的元素，元素也是一个 pair。-. ret.first->second map 中元素的值部分。-. ++ret.first->second 递增此值。

向 multiset 或 multimap 添加元素

由于一个 multi 容器中的关键字不必唯一，在这些类型上调用 insert 总会插入一个元素：

```
multimap<string, string> authors;
// 插入第一个元素
authors.insert({"Barth, John", "Sot-Weed Factor"});
// 正确，添加第二个元素
authors.insert({"Barth, John"}, "Lost in the Funhouse");
```

对允许重复关键字的容器，接受单个元素的 insert 操作返回一个指向新元素的迭代器。

10.3.3 删除元素

关联容器定义了三个版本的 erase：

- 与顺序容器一样，传递给 erase 一个迭代器或一个迭代器范围来删除一个元素或一个元素范围。
- 接受一个 key_type 参数，删除所有匹配给定关键字的元素（如果存在的话），返回实际删除的元素的数量。

对于保存不重复关键字的容器，erase 的返回值总是 0 或 1。

对允许重复关键词的容器，删除的元素的数量可能大于 1。

10.3.4 map 的下标操作

map 和 unordered_map 容器提供了下标运算符和一个对应的 at 函数。

set 类型不支持下标操作，不能对一个 multimap 或一个 unordered_multimap 进行下标操作，因为这些容器中可能有多个值与一个关键字相关联。

map 下标运算符接受一个索引获取与此关键字相关联的值，如果关键字不在 map 中，会为它创建一个元素并插入到 map 中，关联值将进行值初始化。

使用下标操作的返回值

当对一个 map 进行下标操作时，会获得一个 mapped_type 对象。

当解引用一个 map 迭代器时，会得到一个 value_type 对象。

!!!note 与 vector 与 string 不同，map 的下标运算符返回的类型与解引用 map 迭代器得到的类型不同。

10.3.5 访问元素

如果我们关心的只不过是一个特定元素是否已在容器中，使用 find 比较好。

对于不允许重复关键字的容器，可能使用 find 还是 count 没什么区别。

`c.find(k)`返回一个迭代器,`c.count(k)`返回关键词等于 k 的元素的数量。

对于允许重复关键字的容器，count 会统计有多少个元素有相同的关键字。

```
1 c.count(k);
2 c.lower_bound(k);//返回一个迭代器，指向第一个关键词不小于k的元素
3 c.upper_bound(k);//返回一个迭代器，指向第一个关键词大于k的元素
4 c.equal_bound(k);//返回一个迭代器pair
```

10.4 无序容器

无序容器不是使用比较运算符来组织元素，而是使用一个哈希函数和关键字类型的 == 运算符。

在关键字类型的元素没有明显的序关系的情况下，无序容器是非常有用的。

11 动态内存

我们的程序到目前为止只使用过静态内存或栈内存。

- 静态内存用来保存局部 static 对象、类 static 数据成员以及定义在任何函数之外的变量。
- 栈内存用来保存定义在函数内的非 static 对象。

分配在静态或栈内存中的对象由编译器自动创建和销毁。

- 对于栈对象，仅在其定义的程序块运行时才存在。
- static 对象在使用之前分配，在程序结束时销毁。

除了静态内存和栈内存，每个程序还拥有一个内存池，这部分内存被称作自由空间或堆（heap）。程序用堆来存储动态分配（dynamically allocate）的对象。

动态对象的生存周期由程序来控制，当动态对象不再使用时，我们的代码必须显示地销毁它们。

11.1 动态内存与智能指针

C++ 中，动态内存的管理是通过一对运算符来完成的：

- new，在动态内存中为对象分配空间并返回一个指向该对象的指针。
- delete，接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。

为了更容易（同时也更安全）地使用动态内存，新的标准提供了两种智能指针（smart pointer）类型来管理动态对象。

智能指针的行为类似常规指针，重要的区别是它负责自动释放所指向的对象。两种智能指针的区别在于管理底层指针的方式：

- shared_ptr 允许多个指针指向同一个对象；
- unique_ptr 则“独占”所指向的对象。
- 标准库还定义了一个名为 weak_ptr 的伴随类，它是一种弱引用，指向 shared_ptr 所管理的对象。

这些类型定义在 memory 头文件中。

11.1.1 shared_ptr 类

智能指针也是模板，当创建一个智能指针时，必须提供指向的类型：

```
1 shared_ptr<string> p1; // shared_ptr, 可以指向string
```

默认初始化的智能指针中保存着一个空指针。

解引用一个智能指针返回它指向的对象。如果在一个条件判断中使用智能指针，效果就是检测它是否为空：

```
1 if (p1) *p1 = "hi";
```

make_shared 函数

最安全的分配和使用动态内存的方法是调用标准库函数 make_shared。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 shared_ptr。

```
1 // 指向一个值为42的int的shared_ptr
2 shared_ptr<int> p3 = make_shared<int>(42);
3
4 // p6指向一个动态分配的空vector<string>
5 auto p6 = make_shared<vector<string>>();
```

类似顺序容器的 emplace 成员，make_shared 用其参数来构造给定类型的对象。如果我们不传递任何参数，对象就会进行值初始化。

shared_ptr 的拷贝和赋值

每个 shared_ptr 都会记录有多少个其他 shared_ptr 指向相同的对象：

```
1 auto p = make_shared<int>(42); // p指向的对象只有p一个引用者
2 auto q(p); // p和q指向相同的对象，此对象有两个引用者
```

可以认为每个 shared_ptr 都有一个关联的计数器，通常称其为**引用计数** (reference count)。无论何时我们拷贝一个 shared_ptr，计数器都会递增。当我们给 shared_ptr 赋予一个新值或是 shared_ptr 被销毁时，计数器就会递减。

一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象。

!!!note 到底是由一个计数器还是其他数据结构来记录有多少指针共享对象，完全由标准库的具体实现决定。关键是智能指针类能记录有多少个 shared_ptr 指向相同的对象，并能在恰当的时候自动释放对象。

使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要多少对象

2. 程序不知道所需对象的准确类型

3. 程序需要在多个对象间共享数据

容器类是出于第一种原因而使用动态内存的典型例子，我们将在第 15 章看到出于第二种原因的例子。本章介绍出于第三种原因的例子。

11.1.2 直接管理内存

C++ 提供了 new 运算符分配内存，delete 运算符释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 new 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi指向一个动态分配的、未初始化的无名对象
```

默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的值将是未定义的，而类类型将使用默认构造函数进行初始化。

可以使用直接初始化方式来初始化一个动态分配的对象：

```
int *pi = new int(1024);

vector<int> *pv = new vector<int>{1, 2, 3};
```

动态分配的 const 对象

用 new 分配 const 对象是合法的：

```
const int *pci = new const int(1024);
```

类似其他任何 const 对象，一个动态分配的 const 对象必须进行初始化。

内存耗尽

一旦一个程序用光了它所有可用的内存，new 表达式就会失败（并返回一个空指针）。默认情况下，如果 new 不能分配所要求的内存空间，它会抛出一个类型为 bad_alloc 的异常。

我们可以改变使用 new 的方式来阻止它抛出异常：

```
// 如果分配失败，new返回一个空指针
int *p1 = new int; // 如果分配失败，new抛出std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new返回一个空指针
```

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 delete 表达式（delete expression）来将动态内存归还给系统。

```
delete p; // p必须指向一个动态分配的对象或是一个空指针
```

指针值和 delete 释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的。

```
int i, *pil=&i, *pi2=nullptr;
double*pd = new double(33), *pd2=pd;
delete i; //错误：i不是一个指针
delete pi1; //未定义；pi1指向一个局部变量
delete pd; //正确
delete pd2; //未定义：pd2指向的内存已经被释放了
delete pi2; //正确：释放一个空指针总是没有错误的
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担—调用者必须记得释放内存：

```
//factory返回一个指针，指向一个动态分配的对象
Foo* factory(T arg){
    //视情况处理arg
    return new Foo(arg);//调用者负责释放此内存
}
```

```
使用new和delete管理动态内存存在三个常见问题：
1.忘记delete内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长时间后，真正耗尽内存时，才能检测到这种错误。
2.使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这种错误。
3.同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了delete操作，对象的内存就被归还给自由空间了。如果我们随后又delete第二个指针，自由空间就可能被破坏。相对于查找和修正这些错误来说，制造出这些错误要简单得多。
```

11.1.3 shared_ptr 和 new 结合使用

如果不初始化一个智能指针，它就会被初始化为一个空指针。还可以用 new 返回的指针来初始化智能指针：

```
shared_ptr<int> p2(new int(42)); // p2指向一个值为42的int
```

接受指针参数的智能指针构造函数是 explicit 的，因此必须使用直接初始化形式来初始化一个智能指针：

```
shared_ptr<int> p1 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024));  // 正确：使用了直接初始化形式
```

p1 的初始化隐式地要求编译器用一个 new 返回的 int* 来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```
shared ptr<int>clone(int p){
return new int(p); //错误：隐式转换为shared ptr<int>
}
```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```
shared ptr<int>clone(int p){
//正确：显式地用int*创建 shared ptr<int>
return shared_ptr<int>(new int(p));
}
```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象（可以提供自己的操作来替代 delete）。

更多关于智能指针使用的讨论见 p412。

11.1.4 智能指针和异常

程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个对象
    // 这段代码抛出一个异常，且在f中未被捕获
} // 函数结束时shared_ptr自动释放内存
```

无论是否发生了异常，局部对象都会被销毁，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

如果使用了内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42); // 动态分配一个新对象
    // 这段代码抛出一个异常，且在f中未被捕获
    delete ip; // 在退出以前释放内存
}
```

如果在 new 和 delete 之间发生了异常，且异常未在 f 中被捕获，则内存就永远不会被释放了。

使用我们自己的释放操作

这里给一个简单的定义删除器的例子，而具体的讨论见书本 p416。

```
auto deleter = [](int* p)
{
    std::cout << "delete data: " << *p << std::endl;
    delete p;
};

std::shared_ptr<int> p(new int(42), deleter);
```

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

-. 不使用相同的内置指针值初始化（或 reset）多个智能指针。-. 不 delete get () 返回的指针。-. 不使用 get () 初始化或 reset 另一个智能指针。-. 如果你使用 get () 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。-. 如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

11.1.5 unique_ptr

与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。

```
unique_ptr<double> p1; // 可以指向一个double的unique_ptr
unique_ptr<int> p2(new int(42)); // p2指向一个值为42的int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作。

更多有关 unique_ptr 操作的讨论见 p418。

11.1.6 weak_ptr

weak_ptr 是一种不控制所指对象生存期的智能指针，它指向一个 shared_ptr 管理的对象。将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。即使有 weak_ptr 指向对象，对象还是会被释放。

当我们创建一个 weak_ptr 时，要用一个 shared_ptr 来初始化它：

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp若共享p；p的引用计数未改变
```

由于对象可能不存在，我们不能使用 weak_ptr 直接访问对象，而必须调用 lock。如果存在，lock 返回一个指向共享对象的 shared_ptr。否则返回一个空 shared_ptr。

```
if (shared_ptr<int> np = wp.lock()) { // 如果np不为空则条件成立
    // 在if中，np与p共享对象
}
```

11.2 动态数组

C++ 语言和标准库提供了两种一次分配一个对象数组的方法：

- 一种 new 表达式语法，可以分配并初始化一个对象数组。
- 标准库中包含一个名为 allocator 的类，允许我们将分配和初始化分离。使用 allocator 通常会提供更好的性能和更灵活的内存管理能力。

!!!note 大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。

11.2.1 new 和数组

为了让 new 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目：

```
// 调用get_size确定分配多少个int
int *pia = new int[get_size()]; // pia指向第一个int
```

方括号中的大小必须是整型，但不必是常量。

分配一个数组会得到一个元素类型的指针

当用 new 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。

!!!note 要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，new 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化，方法是在大小之后跟一对空括号：

```
int *pia = new int[10]; // 10个未初始化的int
int *pia2 = new int[10](); // 10个值初始化为0的int
```

新标准中，我们还可以提供一个元素初始化器的花括号列表：

```
// 10个int分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
```

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 delete——在指针前加上一个空方括号对：

```
delete p; // p必须指向一个动态分配的对象或为空
delete [] pa; // pa必须指向一个动态分配的数组或为空
```

数组的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

智能指针和动态数组

标准库提供了一个可以管理 new 分配的数组的 unique_ptr 版本：

```
// up指向一个包含10个未初始化int的数组
unique_ptr<int[]> up(new int[10]);
up.release(); // 自动用delete[]销毁其指针
```

> my note: 这里似乎有错误，release 方法据 p418 介绍，是放弃对指针的控制权，返回指针。并不销毁原来指向的对象。另一个事例见：http://zh.cppreference.com/w/cpp/memory/unique_ptr/release

当 unique_ptr 销毁时，会自动销毁其指向的对象。

11.2.2 allocator 类

new 和 delete 有一些灵活性上的局限：

- new 将内存分配和对象构造组合在了一起。
- delete 将对象析构和内存释放组合在了一起。

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作。

allocator 类

标准库 allocator 类定义在头文件 memory 中，它帮助我们将内存分配和对象构造分离开来。它分配的内存是原始的、未构造的。

allocator 也是模板，为了定义一个 allocator 对象，我们必须指明这个 allocator 可以分配的对象类型。当一个 allocator 对象分配内存时，它会根据给定对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc; // 可以分配string的allocator对象
auto const p = alloc.allocate(n); // 分配n个未初始化的string
```

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。

```
auto q = p; // q指向最后构造元素之后的位置
alloc.construct(q++); // *q为空字符串
alloc.construct(q++, "hi"); // *q为hi!
```

还未构造对象的情况下就使用原始内存是错误的。

当我们用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。

```
while (q != p)
    alloc.destroy(--q); // 释放我们真正构造的string
```

一旦元素被销毁后，就可以重新用这部分内存来保存其他 string，也可以将其归还给系统。释放内存通过调用 deallocate 来完成：

```
alloc.deallocate(p, n);
```

我们传递给 deallocate 的指针不能为空，它必须指向由 allocate 分配的内存。而且，传递给 deallocate 的大小参数必须与调用 allocated 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库为 allocator 类定义了两个伴随算法，可以在未初始化内存中创建对象。见 p429。