```
    // ...
}
void File::close()
{
    if (modified)
        // …… 保存内容
}
```

通常使用内置的位运算符（参见 4.8 节，第 136 页）操作超过 1 位的位域：

856

```
File &File::open(File::modes m)
{
    mode |= READ;              // 按默认方式设置 READ
    // 其他处理
    if (m & WRITE)             // 如果打开了 READ 和 WRITE
    // 按照读/写方式打开文件
    return *this;
}
```

如果一个类定义了位域成员，则它通常也会定义一组内联的成员函数以检验或设置位域的值：

```
inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }
```

19.8.2 volatile 限定符

volatile 的确切含义与机器有关，只能通过阅读编译器文档来理解。要想让使用了 volatile 的程序在移植到新机器或新编译器后仍然有效，通常需要对该程序进行某些改变。

直接处理硬件的程序常常包含这样的数据元素，它们的值由程序直接控制之外的过程控制。例如，程序可能包含一个由系统时钟定时更新的变量。当对象的值可能在程序的控制或检测之外被改变时，应该将该对象声明为 **volatile**。关键字 volatile 告诉编译器不应对这样的对象进行优化。

volatile 限定符的用法和 const 很相似，它起到对类型额外修饰的作用：

```
volatile int display_register;    // 该 int 值可能发生改变
volatile Task *curr_task;         // curr_task 指向一个 volatile 对象
volatile int iax[max_size];       // iax 的每个元素都是 volatile
volatile Screen bitmapBuf;        // bitmapBuf 的每个成员都是 volatile
```

const 和 volatile 限定符互相没什么影响，某种类型可能既是 const 的也是 volatile 的，此时它同时具有二者的属性。

就像一个类可以定义 const 成员函数一样，它也可以将成员函数定义成 volatile 的。只有 volatile 的成员函数才能被 volatile 的对象调用。

2.4.2 节（第 56 页）描述了 const 限定符和指针的相互作用，在 volatile 限定符和指针之间也存在类似的关系。我们可以声明 volatile 指针、指向 volatile 对象的指针以及指向 volatile 对象的 volatile 指针：

```
volatile int v;         // v 是一个 volatile int
```

857

```
int *volatile vip;    // vip 是一个 volatile 指针，它指向 int
volatile int *ivp;    // ivp 是一个指针，它指向一个 volatile int
// vivp 是一个 volatile 指针，它指向一个 volatile int
volatile int *volatile vivp;

int *ip = &v;         // 错误：必须使用指向 volatile 的指针
ivp = &v;             // 正确：ivp 是一个指向 volatile 的指针
vivp = &v;            // 正确：vivp 是一个指向 volatile 的 volatile 指针
```

和 const 一样，我们只能将一个 volatile 对象的地址（或者拷贝一个指向 volatile 类型的指针）赋给一个指向 volatile 的指针。同时，只有当某个引用是 volatile 的时，我们才能使用一个 volatile 对象初始化该引用。

合成的拷贝对 volatile 对象无效

const 和 volatile 的一个重要区别是我们不能使用合成的拷贝/移动构造函数及赋值运算符初始化 volatile 对象或从 volatile 对象赋值。合成的成员接受的形参类型是（非 volatile）常量引用，显然我们不能把一个非 volatile 引用绑定到一个 volatile 对象上。

如果一个类希望拷贝、移动或赋值它的 volatile 对象，则该类必须自定义拷贝或移动操作。例如，我们可以将形参类型指定为 const volatile 引用，这样我们就能利用任意类型的 Foo 进行拷贝或赋值操作了：

```
class Foo {
public:
    Foo(const volatile Foo&); // 从一个 volatile 对象进行拷贝
    // 将一个 volatile 对象赋值给一个非 volatile 对象
    Foo& operator=(volatile const Foo&);
    // 将一个 volatile 对象赋值给一个 volatile 对象
    Foo& operator=(volatile const Foo&) volatile;
    // Foo 类的剩余部分
};
```

尽管我们可以为 volatile 对象定义拷贝和赋值操作，但是一个更深层次的问题是拷贝 volatile 对象是否有意义呢？不同程序使用 volatile 的目的各不相同，对上述问题的回答与具体的使用目的密切相关。

19.8.3 链接指示：extern "C"

C++程序有时需要调用其他语言编写的函数，最常见的是调用 C 语言编写的函数。像所有其他名字一样，其他语言中的函数名字也必须在 C++中进行声明，并且该声明必须指定返回类型和形参列表。对于其他语言编写的函数来说，编译器检查其调用的方式与处理普通 C++函数的方式相同，但是生成的代码有所区别。C++使用**链接指示**（linkage directive）指出任意非 C++函数所用的语言。

858

要想把 C++代码和其他语言（包括 C 语言）编写的代码放在一起使用，要求我们必须有权访问该语言的编译器，并且这个编译器与当前的 C++编译器是兼容的。

声明一个非 C++的函数

链接指示可以有两种形式：单个的或复合的。链接指示不能出现在类定义或函数定义的内部。同样的链接指示必须在函数的每个声明中都出现。

举个例子，接下来的声明显示了 cstring 头文件的某些 C 函数是如何声明的：

```
// 可能出现在 C++头文件<cstring>中的链接指示
// 单语句链接指示
extern "C" size_t strlen(const char *);
// 复合语句链接指示
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

链接指示的第一种形式包含一个关键字 extern，后面是一个字符串字面值常量以及一个“普通的”函数声明。

其中的字符串字面值常量指出了编写函数所用的语言。编译器应该支持对 C 语言的链接指示。此外，编译器也可能会支持其他语言的链接指示，如 extern "Ada"、extern "FORTRAN"等。

链接指示与头文件

我们可以令链接指示后面跟上花括号括起来的若干函数的声明，从而一次性建立多个链接。花括号的作用是将适用于该链接指示的多个声明聚合在一起，否则花括号就会被忽略，花括号中声明的函数名字就是可见的，就好像在花括号之外声明的一样。

多重声明的形式可以应用于整个头文件。例如，C++的 cstring 头文件可能形如：

```
// 复合语句链接指示
extern "C" {
#include <string.h>        // 操作 C 风格字符串的 C 函数
}
```

当一个#include 指示被放置在复合链接指示的花括号中时，头文件中的所有普通函数声明都被认为是由链接指示的语言编写的。链接指示可以嵌套，因此如果头文件包含带自带链接指示的函数，则该函数的链接不受影响。 859

C++从 C 语言继承的标准库函数可以定义成 C 函数，但并非必须：决定使用 C 还是 C++实现 C 标准库，是每个 C++实现的事情。

指向 extern "C"函数的指针

编写函数所用的语言是函数类型的一部分。因此，对于使用链接指示定义的函数来说，它的每个声明都必须使用相同的链接指示。而且，指向其他语言编写的函数的指针必须与函数本身使用相同的链接指示：

```
// pf 指向一个 C 函数，该函数接受一个 int 返回 void
extern "C" void (*pf)(int);
```

当我们使用 pf 调用函数时，编译器认定当前调用的是一个 C 函数。

指向 C 函数的指针与指向 C++函数的指针是不一样的类型。一个指向 C 函数的指针

不能用在执行初始化或赋值操作后指向 C++函数，反之亦然。就像其他类型不匹配的问题一样，如果我们试图在两个链接指示不同的指针之间进行赋值操作，则程序将发生错误：

```
void (*pf1)(int);                    // 指向一个 C++函数
extern "C" void (*pf2)(int);         // 指向一个 C 函数
pf1 = pf2;                           // 错误：pf1 和 pf2 的类型不同
```

有的 C++编译器会接受之前的这种赋值操作并将其作为对语言的扩展，尽管从严格意义上来看它是非法的。

链接指示对整个声明都有效

当我们使用链接指示时，它不仅对函数有效，而且对作为返回类型或形参类型的函数指针也有效：

```
// f1 是一个 C 函数，它的形参是一个指向 C 函数的指针
extern "C" void f1(void(*)(int));
```

这条声明语句指出 f1 是一个不返回任何值的 C 函数。它有一个类型是函数指针的形参，其中的函数接受一个 int 形参返回为空。这个链接指示不仅对 f1 有效，对函数指针同样有效。当我们调用 f1 时，必须传给它一个 C 函数的名字或者指向 C 函数的指针。

因为链接指示同时作用于声明语句中的所有函数，所以如果我们希望给 C++函数传入一个指向 C 函数的指针，则必须使用类型别名（参见 2.5.1 节，第 60 页）：

860

```
// FC 是一个指向 C 函数的指针
extern "C" typedef void FC(int);
// f2 是一个 C++函数，该函数的形参是指向 C 函数的指针
void f2(FC *);
```

导出 C++函数到其他语言

通过使用链接指示对函数进行定义，我们可以令一个 C++函数在其他语言编写的程序中可用：

```
// calc 函数可以被 C 程序调用
extern "C" double calc(double dparm) { /* ... */ }
```

编译器将为该函数生成适合于指定语言的代码。

值得注意的是，可被多种语言共享的函数的返回类型或形参类型受到很多限制。例如，我们不太可能把一个 C++类的对象传给 C 程序，因为 C 程序根本无法理解构造函数、析构函数以及其他类特有的操作。

对链接到 C 的预处理器的支持

有时需要在 C 和 C++中编译同一个源文件，为了实现这一目的，在编译 C++版本的程序时预处理器定义__cplusplus（两个下画线）。利用这个变量，我们可以在编译 C++程序的时候有条件地包含进来一些代码：

```
#ifdef __cplusplus
// 正确：我们正在编译 C++程序
extern "C"
#endif
int strcmp(const char*, const char*);
```

重载函数与链接指示

链接指示与重载函数的相互作用依赖于目标语言。如果目标语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持重载这些C++的函数。

C语言不支持函数重载，因此也就不难理解为什么一个C链接指示只能用于说明一组重载函数中的某一个了：

```
// 错误：两个extern "C"函数的名字相同
extern "C" void print(const char*);
extern "C" void print(int);
```

如果在一组重载函数中有一个是C函数，则其余的必定都是C++函数：

861

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// C函数可以在C或C++程序中调用
// C++函数重载了该函数，可以在C++程序中调用
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

C版本的calc函数可以在C或C++程序中调用，而使用了类类型形参的C++函数只能在C++程序中调用。上述性质与声明的顺序无关。

19.8.3 节练习

练习 19.26：说明下列声明语句的含义并判断它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

done!

862

小结

C++为解决某些特殊问题设置了一系列特殊的处理机制。

有的程序需要精确控制内存分配过程，它们可以通过在类的内部或在全局作用域中自定义 operator new 和 operator delete 来实现这一目的。如果应用程序为这两个操作定义了自己的版本，则 new 和 delete 表达式将优先使用应用程序定义的版本。

有的程序需要在运行时直接获取对象的动态类型，运行时类型识别（RTTI）为这种程序提供了语言级别的支持。RTTI 只对定义了虚函数的类有效；对没有定义虚函数的类，虽然也可以得到其类型信息，但只是静态类型。

当我们定义指向类成员的指针时，在指针类型中包含了该指针所指成员所属类的类型信息。成员指针可以绑定到该类当中任意一个具有指定类型的成员上。当我们解引用成员指针时，必须提供获取成员所需的对象。

C++定义了另外几种聚集类型：

- 嵌套类，定义在其他类的作用域中，嵌套类通常作为外层类的实现类。
- union，是一种特殊的类，它可以定义几个数据成员但是在任意时刻只有一个成员有值，union 通常嵌套在其他类的内部。
- 局部类，定义在函数的内部，局部类的所有成员都必须定义在类内，局部类不能含有静态数据成员。

C++支持几种固有的不可移植的特性，其中位域和 volatile 使得程序更容易访问硬件；链接指示使得程序更容易访问用其他语言编写的代码。

术语表

匿名 union（anonymous union） 未命名的 union，不能用于定义对象。匿名 union 的成员也是外层作用域的成员。匿名 union 不能包含成员函数，也不能包含私有成员或受保护的成员。

位域（bit-field） 特殊的类成员，该成员含有一个整型值以指定为其分配的二进制位数。如果可能的话，在类中连续定义的位域将被压缩在一个普通的整数值当中。

判别式（discriminant） 是一种使用一个对象判断 union 的当前值类型的编程技术。

dynamic_cast 是一个运算符，执行从基类向派生类的带检查的强制类型转换。当基类中至少含有一个虚函数时，该运算符负责检查指针或引用所绑定的对象的动态类型。如果对象类型与目标类型（或其派生类）一致，则类型转换完成。否则，指针转换将返回一个值为 0 的指针；引用转换将抛出一个异常。

863

枚举类型（enumeration） 将一组整型常量命名后聚合在一起形成的类型。

枚举成员（enumerator） 是枚举类型的成员。枚举成员是常量，可以用在任何需要整型常量的地方。

free 是定义在 cstdlib 中的低层函数，负责释放内存。free 只能释放由 malloc 分配的内存。

链接指示（linkage directive） 支持 C++程序调用其他语言编写的函数的一种机制。所有编译器都应支持调用 C++和 C 函数，至于是否支持其他语言则由编译器决定。

局部类（local class） 定义在函数中的类。局部类只有在其外层函数内可见。局部类

的所有成员都必须定义在类的内部。局部类不能含有静态成员。局部类成员不能访问外层函数的非静态变量，只能访问类型名字、静态变量或枚举成员。

malloc 是定义在 cstdlib 中的低层函数，负责分配内存。malloc 分配的内存必须由 free 释放。

mem_fn 是一个标准库类模板，根据指向成员函数的指针生成一个可调用对象。

嵌套类（nested class） 定义在其他类内部的类，嵌套类定义在它的外层作用域中：在外层类的作用域中嵌套类的名字必须唯一，在外层类之外可以被重用。在外层类之外访问嵌套类需要用作用域运算符指明嵌套类所属的范围。

嵌套类型（nested type） “嵌套类”的同义词。

不可移植（nonportable） 固有的与机器有关的特性，当程序转移到其他机器或编译器上时需要修改代码。

operator delete 是一个标准库函数，用于释放由 operator new 分配的未指明类型的、未构造的内存空间。相应的，operator delete[]释放由 operator new[]为数组分配的内存。

operator new 是一个标准库函数，用于分配一个给定大小的、未指明类型的、未构造的内存空间。标准库函数 operator new[] 为数组分配原始内存。与 allocator 类相比，这两个标准库函数提供的内存分配机制更低级。现代的 C++程序应该使用 allocator 而不是这两个函数。

定位 new 表达式（placement new expression） 是 new 的一种特殊形式，在给定的内存中构造对象。它不分配内存，而是根据实参指定在哪儿构造对象。它是对 allocator 类的 construct 成员的行为的一种低级模拟。

成员指针（pointer to member） 其中既包含类类型，也包含指针所指的成员类型。成员指针的定义必须同时指定类的名字以及指针所指的成员类型：

```
T C::*pmem = &C::member;
```

该语句将 pmem 定义为一个指针，它可以指向类 C 的成员，并且该成员的类型是 T，然后初始化 pmem 令其指向类 C 的名为 member 的成员。要使用该指针，我们必须提供 C 的一个对象或指针：

```
classobj.*pmem;
classptr->*pmem;
```

从 classptr 所指的对象 classobj 中获取 member。 864

运行时类型识别（run-time type identification） 是 C++的一种特性，允许在运行时获取指针或引用的动态类型。RTTI 运算符包括 typeid 和 dynamic_cast，为含有虚函数的类的指针或引用提供动态类型。当作用于其他类型时，返回的结果是指针或引用的静态类型。

限定作用域的枚举类型（scoped enumeration） 是一种新的枚举类型，它的枚举成员不能被外层作用域直接访问。

typeid 运算符（typeid operator） 是一个一元运算符，返回标准库类型 type_info 的引用，表示给定表达式的类型。当表达式是某个含有虚函数的类型的对象时，返回表达式的动态类型；此类表达式在运行时求值。如果表达式的类型是指针、引用或其他未定义虚函数的类型，则返回指针、引用或对象的静态类型；此类表达式不会被求值。

type_info typeid 运算符返回的标准库类型。type_info 的细节因机器而异，但是必须提供一组操作，其中名为 name 的函数负责返回一个表示类型名字的字符串。type_info 对象不能被拷贝、移动或赋值。

联合（union） 是一种和类有些相似的类型。可以包含多个数据成员，但是同一时刻只能有一个成员有值。联合可以有包括

构造函数和析构函数在内的成员函数。联合不能被用作基类。在 C++11 新标准中，联合可以含有类类型的成员，前提是这些类自定义了拷贝控制成员。对于这样的联合来说，如果它们没有定义自己的拷贝控制成员，则编译器将为它们生成删除的版本。

不限定作用域的枚举类型（unscoped enumeration） 该枚举类型的枚举成员在枚举类型的外层作用域中可以访问。

volatile 是一种类型限定符，告诉编译器变量可能在程序的直接控制之外发生改变。它起到一种标示的作用，令编译器不对代码进行优化操作。

done!

done!

done!

附录 A
标准库

内容

本附录介绍了标准库中算法和随机数部分的一些额外细节。这里还提供了一个我们使用过的所有标准库名字的列表，列表中给出了每个名字所在的头文件。

在第 10 章中我们使用过一些较常用的算法，并且描述了算法之下的架构。在本附录中，我们将列出所有标准库算法，按它们执行的操作的种类来组织。

在 17.4 节（第 660 页）中我们描述了随机数库的架构，并使用了几个分布类型。库中定义了若干随机数引擎和 20 种不同的分布。在本附录中，我们将列出所有引擎和分布类型。

done!

866

A.1 标准库名字和头文件

本书中大多数代码没有给出编译程序所需的实际#include 指令。为了方便读者，表 A.1 列出了本书程序用到的标准库名字以及它们所在的头文件。

表 A.1：标准库名字和头文件

名字	头文件
abort	<cstdlib>
accumulate	<numeric>
allocator	<memory>
array	<array>
auto_ptr	<memory>
back_inserter	<iterator>
bad_alloc	<new>
bad_array_new_length	<new>
bad_cast	<typeinfo>
begin	<iterator>
bernoulli_distribution	<random>
bind	<functional>
bitset	<bitset>
boolalpha	<iostream>
cerr	<iostream>
cin	<iostream>
cmatch	<regex>
copy	<algorithm>
count	<algorithm>
count_if	<algorithm>
cout	<iostream>
cref	<functional>
csub_match	<regex>
dec	<iostream>
default_float_engine	<iostream>
default_random_engine	<random>
deque	<deque>
domain_error	<stdexcept>
end	<iterator>
endl	<iostream>
ends	<iostream>
equal_range	<algorithm>
exception	<exception>
fill	<algorithm>
fill_n	<algorithm>
find	<algorithm>

续表

名字	头文件
find_end	<algorithm>
find_first_of	<algorithm>
find_if	<algorithm>
fixed	<iostream>
flush	<iostream>
for_each	<algorithm>
forward	<utility>
forward_list	<forward_list>
free	<cstdlib>
front_inserter	<iterator>
fstream	<fstream>
function	<functional>
get	<tuple>
getline	<string>
greater	<functional>
hash	<functional>
hex	<iostream>
hexfloat	<iostream>
ifstream	<fstream>
initializer_list	<initializer_list>
inserter	<iterator>
internal	<iostream>
ios_base	<ios_base>
isalpha	<cctype>
islower	<cctype>
isprint	<cctype>
ispunct	<cctype>
isspace	<cctype>
istream	<iostream>
istream_iterator	<iterator>
istringstream	<sstream>
isupper	<cctype>
left	<iostream>
less	<functional>
less_equal	<functional>
list	<list>
logic_error	<stdexcept>
lower_bound	<algorithm>
lround	<cmath>
make_move_iterator	<iterator>
make_pair	<utility>

867

续表

名字	头文件
make_shared	<memory>
make_tuple	<tuple>
malloc	<cstdlib>
map	<map>
max	<algorithm>
max_element	<algorithm>
mem_fn	<functional>
min	<algorithm>
move	<utility>
multimap	<map>
multiset	<set>
negate	<functional>
noboolalpha	<iostream>
normal_distribution	<random>
noshowbase	<iostream>
noshowpoint	<iostream>
noskipws	<iostream>
not1	<functional>
nothrow	<new>
nothrow_t	<new>
nounitbuf	<iostream>
nouppercase	<iostream>
nth_element	<algorithm>
oct	<iostream>
ofstream	<fstream>
ostream	<iostream>
ostream_iterator	<iterator>
ostringstream	<sstream>
out_of_range	<stdexcept>
pair	<utility>
partial_sort	<algorithm>
placeholders	<functional>
placeholders::_1	<functional>
plus	<functional>
priority_queue	<queue>
ptrdiff_t	<cstddef>
queue	<queue>
rand	<random>
random_device	<random>
range_error	<stdexcept>
ref	<functional>

868

续表

名字	头文件
regex	<regex>
regex_constants	<regex>
regex_error	<regex>
regex_match	<regex>
regex_replace	<regex>
regex_search	<regex>
remove_pointer	<type_traits>
remove_reference	<type_traits>
replace	<algorithm>
replace_copy	<algorithm>
reverse_iterator	<iterator>
right	<iostream>
runtime_error	<stdexcept>
scientific	<iostream>
set	<set>
set_difference	<algorithm>
set_intersection	<algorithm>
set_union	<algorithm>
setfill	<iomanip>
setprecision	<iomanip>
setw	<iomanip>
shared_ptr	<memory>
showbase	<iostream>
showpoint	<iostream>
size_t	<cstddef>
skipws	<iostream>
smatch	<regex>
sort	<algorithm>
sqrt	<cmath>
sregex_iterator	<regex>
ssub_match	<regex>
stable_sort	<algorithm>
stack	<stack>
stoi	<string>
strcmp	<cstring>
strcpy	<cstring>
string	<string>
stringstream	<sstream>
strlen	<cstring>
strncpy	<cstring>
strtod	<string>

869

续表

名字	头文件
swap	<utility>
terminate	<exception>
time	<ctime>
tolower	<cctype>
toupper	<cctype>
transform	<algorithm>
tuple	<tuple>
tuple_element	<tuple>
tuple_size	<tuple>
type_info	<typeinfo>
unexpected	<exception>
uniform_int_distribution	<random>
uniform_real_distribution	<random>
uninitialized_copy	<memory>
uninitialized_fill	<memory>
unique	<algorithm>
unique_copy	<algorithm>
unique_ptr	<memory>
unitbuf	<iostream>
unordered_map	<unordered_map>
unordered_multimap	<unordered_map>
unordered_multiset	<unordered_set>
unordered_set	<unordered_set>
upper_bound	<algorithm>
uppercase	<iostream>
vector	<vector>
weak_ptr	<memory>

A.2 算法概览

标准库定义了超过 100 个算法。要想高效使用这些算法需要了解它们的结构而不是单纯记忆每个算法的细节。因此，我们在第 10 章中关注标准库算法架构的描述和理解。在本节中，我们将简要描述每个算法，在下面的描述中，

- beg 和 end 是表示元素范围的迭代器（参见 9.2.1 节，第 296 页）。几乎所有算法都对一个由 beg 和 end 表示的序列进行操作。
- beg2 是表示第二个输入序列开始位置的迭代器。end2 表示第二个序列的末尾位置（如果有的话）。如果没有 end2，则假定 beg2 表示的序列与 beg 和 end 表示的序列一样大。beg 和 beg2 的类型不必匹配，但是，必须保证对两个序列中的元素都可以执行特定操作或调用给定的可调用对象。
- dest 是表示目的序列的迭代器。对于给定输入序列，算法需要生成多少元素，目的序列必须保证能保存同样多的元素。

- unaryPred 和 binaryPred 是一元和二元谓词（参见 10.3.1 节，第 344 页），分别接受一个和两个参数，都是来自输入序列的元素，两个谓词都返回可用作条件的类型。
- comp 是一个二元谓词，满足关联容器中对关键字序的要求（参见 11.2.2 节，第 378 页）。 871
- unaryOp 和 binaryOp 是可调用对象（参见 10.3.2 节，第 346 页），可分别使用来自输入序列的一个和两个实参来调用。

A.2.1 查找对象的算法

这些算法在一个输入序列中搜索一个指定值或一个值的序列。

每个算法都提供两个重载的版本，第一个版本使用底层类型的相等运算符（==）来比较元素；第二个版本使用用户给定的 unaryPred 和 binaryPred 比较元素。

简单查找算法

这些算法查找指定值，要求输入迭代器（input iterator）。

```
find(beg, end, val)
find_if(beg, end, unaryPred)
find_if_not(beg, end, unaryPred)
count(beg, end, val)
count_if(beg, end, unaryPred)
```

find 返回一个迭代器，指向输入序列中第一个等于 val 的元素。

find_if 返回一个迭代器，指向第一个满足 unaryPred 的元素。

find_if_not 返回一个迭代器，指向第一个令 unaryPred 为 false 的元素。上述三个算法在未找到元素时都返回 end。

count 返回一个计数器，指出 val 出现了多少次；count_if 统计有多少个元素满足 unaryPred。

```
all_of(beg, end, unaryPred)
any_of(beg, end, unaryPred)
none_of(beg, end, unaryPred)
```

这些算法都返回一个 bool 值，分别指出 unaryPred 是否对所有元素都成功、对任意一个元素成功以及对所有元素都不成功。如果序列为空，any_of 返回 false，而 all_of 和 none_of 返回 true。

查找重复值的算法

下面这些算法要求前向迭代器（forward iterator），在输入序列中查找重复元素。

```
adjacent_find(beg, end)
adjacent_find(beg, end, binaryPred)
```

返回指向第一对相邻重复元素的迭代器。如果序列中无相邻重复元素，则返回 end。

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred)
```

返回一个迭代器，从此位置开始有 count 个相等元素。如果序列中不存在这样的子序列，

则返回 end。

872 **查找子序列的算法**

在下面的算法中，除了 find_first_of 之外，都要求两个前向迭代器。find_first_of 用输入迭代器表示第一个序列，用前向迭代器表示第二个序列。这些算法搜索子序列而不是单个元素。

```
search(beg1, end1, beg2, end2)
search(beg1, end1, beg2, end2, binaryPred)
```

返回第二个输入范围（子序列）在第一个输入范围中第一次出现的位置。如果未找到子序列，则返回 end1。

```
find_first_of(beg1, end1, beg2, end2)
find_first_of(beg1, end1, beg2, end2, binaryPred)
```

返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置。如果未找到匹配元素，则返回 end1。

```
find_end(beg1, end1, beg2, end2)
find_end(beg1, end1, beg2, end2, binaryPred)
```

类似 search，但返回的是最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1。

A.2.2 其他只读算法

这些算法要求前两个实参都是输入迭代器。

equal 和 mismatch 算法还接受一个额外的输入迭代器，表示第二个范围的开始位置。这两个算法都提供两个重载的版本。第一个版本使用底层类型的相等运算符（==）比较元素，第二个版本则用用户指定的 unaryPred 或 binaryPred 比较元素。

```
for_each(beg, end, unaryOp)
```

对输入序列中的每个元素应用可调用对象（参见 10.3.2 节，第 346 页）unaryOp。unaryOp 的返回值（如果有的话）被忽略。如果迭代器允许通过解引用运算符向序列中的元素写入值，则 unaryOp 可能修改元素。

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

比较两个序列中的元素。返回一个迭代器的 pair（参见 11.2.3 节，第 379 页），表示两个序列中第一个不匹配的元素。如果所有元素都匹配，则返回的 pair 中第一个迭代器为 end1，第二个迭代器指向 beg2 中偏移量等于第一个序列长度的位置。

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

确定两个序列是否相等。如果输入序列中每个元素都与从 beg2 开始的序列中对应元素相等，则返回 true。

873 ## A.2.3 二分搜索算法

这些算法都要求前向迭代器，但这些算法都经过了优化，如果我们提供随机访问迭代

器（random-access iterator）的话，它们的性能会好得多。从技术上讲，无论我们提供什么类型的迭代器，这些算法都会执行对数次的比较操作。但是，当使用前向迭代器时，这些算法必须花费线性次数的迭代器操作来移动到序列中要比较的元素。

这些算法要求序列中的元素已经是有序的。它们的行为类似关联容器的同名成员（参见 11.3.5 节，第 389 页）。equal_range、lower_bound 和 upper_bound 算法返回迭代器，指向给定元素在序列中的正确插入位置——插入后还能保持有序。如果给定元素比序列中的所有元素都大，则会返回尾后迭代器。

每个算法都提供两个版本：第一个版本用元素类型的小于运算符（<）来检测元素；第二个版本则使用给定的比较操作。在下列算法中，“x 小于 y”表示 x<y 或 comp(x,y) 成功。

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp)
```

返回一个迭代器，表示第一个小于等于 val 的元素，如果不存在这样的元素，则返回 end。

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp)
```

返回一个迭代器，表示第一个大于 val 的元素，如果不存在这样的元素，则返回 end。

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp)
```

返回一个 pair（参见 11.2.3 节，第 379 页），其 first 成员是 lower_bound 返回的迭代器，second 成员是 upper_bound 返回的迭代器。

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp)
```

返回一个 bool 值，指出序列中是否包含等于 val 的元素。对于两个值 x 和 y，当 x 不小于 y 且 y 也不小于 x 时，认为它们相等。

A.2.4 写容器元素的算法

很多算法向给定序列中的元素写入新值。这些算法可以从不同角度加以区分：通过表示输入序列的迭代器类型来区分；或者通过是写入输入序列中元素还是写入给定目的位置来区分。

只写不读元素的算法

874

这些算法要求一个输出迭代器（output iterator），表示目的位置。_n 结尾的版本接受第二个实参，表示写入的元素数目，并将给定数目的元素写入到目的位置中。

```
fill(beg, end, val)
fill_n(dest, cnt, val)
generate(beg, end, Gen)
generate_n(dest, cnt, Gen)
```

给输入序列中每个元素赋予一个新值。fill 将值 val 赋予元素；generate 执行生成器对象 Gen() 生成新值。生成器是一个可调用对象（参见 10.3.2 节，第 346 页），每次调用会生成一个不同的返回值。fill 和 generate 都返回 void。_n 版本返回一个迭代器，指向写入到输出序列的最后一个元素之后的位置。

使用输入迭代器的写算法

这些算法读取一个输入序列，将值写入到一个输出序列中。它们要求一个名为 dest 的输出迭代器，而表示输入范围的迭代器必须是输入迭代器。

```
copy(beg, end, dest)
copy_if(beg, end, dest, unaryPred)
copy_n(beg, n, dest)
```

从输入范围将元素拷贝到 dest 指定的目的序列。copy 拷贝所有元素，copy_if 拷贝那些满足 unaryPred 的元素，copy_n 拷贝前 n 个元素。输入序列必须有至少 n 个元素。

```
move(beg, end, dest)
```

对输入序列中的每个元素调用 std:: move（参见 13.6.1 节，第 472 页），将其移动到迭代器 dest 开始的序列中。

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

调用给定操作，并将结果写到 dest 中。第一个版本对输入范围中每个元素应用一元操作。第二个版本对两个输入序列中的元素应用二元操作。

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val)
```

将每个元素拷贝到 dest，将指定的元素替换为 new_val。第一个版本替换那些==old_val 的元素。第二个版本替换那些满足 unaryPred 的元素。

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp)
```

两个输入序列必须都是有序的。将合并后的序列写入到 dest 中。第一个版本用<运算符比较元素；第二个版本则使用给定比较操作。

875

使用前向迭代器的写算法

这些算法要求前向迭代器，由于它们是向输入序列写入元素，迭代器必须具有写入元素的权限。

```
iter_swap(iter1, iter2)
swap_ranges(beg1, end1, beg2)
```

交换 iter1 和 iter2 所表示的元素，或将输入范围中所有元素与 beg2 开始的第二个序列中所有元素进行交换。两个范围不能有重叠。iter_swap 返回 void，swap_ranges 返回递增后的 beg2，指向最后一个交换元素之后的位置。

```
replace(beg, end, old_val, new_val)
replace_if(beg, end, unaryPred, new_val)
```

用 new_val 替换每个匹配元素。第一个版本使用==比较元素与 old_val，第二个版本替换那些满足 unaryPred 的元素。

使用双向迭代器的写算法

这些算法需要在序列中有反向移动的能力，因此它们要求双向迭代器（bidirectional iterator）。

```
copy_backward(beg, end, dest)
```

move_backward(beg, end, dest)

从输入范围中拷贝或移动元素到指定目的位置。与其他算法不同，dest 是输出序列的尾后迭代器（即，目的序列恰在 dest 之前结束）。输入范围中的尾元素被拷贝或移动到目的序列的尾元素，然后是倒数第二个元素被拷贝/移动，依此类推。元素在目的序列中的顺序与在输入序列中相同。如果范围为空，则返回值为 dest；否则，返回值表示从*beg 中拷贝或移动的元素。

inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp)

将同一个序列中的两个有序子序列合并为单一的有序序列。beg 到 mid 间的子序列和 mid 到 end 间的子序列被合并，并被写入到原序列中。第一个版本使用<比较元素，第二个版本使用给定的比较操作，返回 void。

A.2.5 划分与排序算法

对于序列中的元素进行排序，排序和划分算法提供了多种策略。

每个排序和划分算法都提供稳定和不稳定版本（参见 10.3.1 节，第 345 页）。稳定算法保证保持相等元素的相对顺序。由于稳定算法会做更多工作，可能比不稳定版本慢得多并消耗更多内存。

划分算法

876

一个划分算法将输入范围中的元素划分为两组。第一组包含那些满足给定谓词的元素，第二组则包含不满足谓词的元素。例如，对于一个序列中的元素，我们可以根据元素是否是奇数或者单词是否以大写字母开头等来划分它们。这些算法都要求双向迭代器。

is_partitioned(beg, end, unaryPred)

如果所有满足谓词 unaryPred 的元素都在不满足 unaryPred 的元素之前，则返回 true。若序列为空，也返回 true。

partition_copy(beg, end, dest1, dest2, unaryPred)

将满足 unaryPred 的元素拷贝到 dest1，并将不满足 unaryPred 的元素拷贝到 dest2。返回一个迭代器 pair（11.2.3 节，第 379 页），其 first 成员表示拷贝到 dest1 的元素的末尾，second 表示拷贝到 dest2 的元素的末尾。输入序列与两个目的序列都不能重叠。

partition_point(beg, end, unaryPred)

输入序列必须是已经用 unaryPred 划分过的。返回满足 unaryPred 的范围的尾后迭代器。如果返回的迭代器不是 end，则它指向的元素及其后的元素必须都不满足 unaryPred。

stable_partition(beg, end, unaryPred)
partition(beg, end, unaryPred)

使用 unaryPred 划分输入序列。满足 unaryPred 的元素放置在序列开始，不满足的元素放在序列尾部。返回一个迭代器，指向最后一个满足 unaryPred 的元素之后的位置，如果所有元素都不满足 unaryPred，则返回 beg。

排序算法

这些算法要求随机访问迭代器。每个排序算法都提供两个重载的版本。一个版本用元

素的<运算符来比较元素，另一个版本接受一个额外参数来指定排序关系（11.2.2 节，第 378 页）。partial_sort_copy 返回一个指向目的位置的迭代器，其他排序算法都返回 void。

partial_sort 和 nth_element 算法都只进行部分排序工作，它们常用于不需要排序整个序列的场合。由于这些算法工作量更少，它们通常比排序整个输入序列的算法更快。

```
sort(beg, end)
stable_sort(beg, end)
sort(beg, end, comp)
stable_sort(beg, end, comp)
```

排序整个范围。

877

```
is_sorted(beg, end)
is_sorted(beg, end, comp)
is_sorted_until(beg, end)
is_sorted_until(beg, end, comp)
```

is_sorted 返回一个 bool 值，指出整个输入序列是否有序。is_sorted_until 在输入序列中查找最长初始有序子序列，并返回子序列的尾后迭代器。

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

排序 mid-beg 个元素。即，如果 mid-beg 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置。当 partial_sort 完成后，从 beg 开始直至 mid 之前的范围中的元素就都已排好序了。已排序范围中的元素都不会比 mid 后的元素更大。未排序区域中元素的顺序是未指定的。

```
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

排序输入范围中的元素，并将足够多的已排序元素放到 destBeg 和 destEnd 所指示的序列中。如果目的范围的大小大于等于输入范围，则排序整个输入序列并存入从 destBeg 开始的范围。如果目的范围大小小于输入范围，则只拷贝输入序列中与目的范围一样多的元素。

算法返回一个迭代器，指向目的范围中已排序部分的尾后迭代器。如果目的序列的大小小于或等于输入范围，则返回 destEnd。

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

参数 nth 必须是一个迭代器，指向输入序列中的一个元素。执行 nth_element 后，此迭代器指向的元素恰好是整个序列排好序后此位置上的值。序列中的元素会围绕 nth 进行划分：nth 之前的元素都小于等于它，而之后的元素都大于等于它。

A.2.6 通用重排操作

这些算法重排输入序列中元素的顺序。前两个算法 remove 和 unique，会重排序列，使得排在序列第一部分的元素满足某种标准。它们返回一个迭代器，标记子序列的末尾。其他算法，如 reverse、rotate 和 random_shuffle 都重排整个序列。

这些算法的基本版本都进行“原址”操作，即，在输入序列自身内部重排元素。三个

重排算法提供“拷贝”版本。这些_copy版本完成相同的重排工作，但将重排后的元素写入到一个指定目的序列中，而不是改变输入序列。这些算法要求输出迭代器来表示目的序列。

使用前向迭代器的重排算法

878

这些算法重排输入序列。它们要求迭代器至少是前向迭代器。

```
remove(beg, end, val)
remove_if(beg, end, unaryPred)
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
```

从序列中“删除”元素，采用的办法是用保留的元素覆盖要删除的元素。被删除的是那些==val或满足unaryPred的元素。算法返回一个迭代器，指向最后一个删除元素的尾后位置。

```
unique(beg, end)
unique(beg, end, binaryPred)
unique_copy(beg, end, dest)
unique_copy_if(beg, end, dest, binaryPred)
```

重排序列，对相邻的重复元素，通过覆盖它们来进行“删除”。返回一个迭代器，指向不重复元素的尾后位置。第一个版本用==确定两个元素是否相同，第二个版本使用谓词检测相邻元素。

```
rotate(beg, mid, end)
rotate_copy(beg, mid, end, dest)
```

围绕mid指向的元素进行元素转动。元素mid成为首元素，随后是mid+1到end之前的元素，再接着是beg到mid之前的元素。返回一个迭代器，指向原来在beg位置的元素。

使用双向迭代器的重排算法

由于这些算法要反向处理输入序列，它们要求双向迭代器。

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

翻转序列中的元素。reverse返回void，reverse_copy返回一个迭代器，指向拷贝到目的序列的元素的尾后位置。

使用随机访问迭代器的重排算法

由于这些算法要随机重排元素，它们要求随机访问迭代器。

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
shuffle(beg, end, Uniform_rand)
```

混洗输入序列中的元素。第二个版本接受一个可调用对象参数，该对象必须接受一个正整数值，并生成0到此值的包含区间内的一个服从均匀分布的随机整数。shuffle的第三个参数必须满足均匀分布随机数生成器的要求（参见17.4节，第659页）。所有版本都返回void。

879

A.2.7 排列算法

排列算法生成序列的字典序排列。对于一个给定序列，这些算法通过重排它的一个排列来生成字典序中下一个或前一个排列。算法返回一个 bool 值，指出是否还有下一个或前一个排列。

为了理解什么是下一个或前一个排列，考虑下面这个三字符的序列：abc。它有六种可能的排列：abc、acb、bac、bca、cab 及 cba。这些排列是按字典序递增序列出的。即，abc 是第一个排列，这是因为它的第一个元素小于或等于任何其他排列的首元素，并且它的第二个元素小于任何其他首元素相同的排列。类似的，acb 排在下一位，原因是它以 a 开头，小于任何剩余排列的首元素。同理，以 b 开头的排列也都排在以 c 开头的排列之前。

对于任意给定的排列，基于单个元素的一个特定的序，我们可以获得它的前一个和下一个排列。给定排列 bca，我们知道其前一个排列为 bac，下一个排列为 cab。序列 abc 没有前一个排列，而 cba 没有下一个排列。

这些算法假定序列中的元素都是唯一的，即，没有两个元素的值是一样的。

为了生成排列，必须既向前又向后处理序列，因此算法要求双向迭代器。

```
is_permutation(beg1, end1, beg2)
is_permutation(beg1, end1, beg2, binaryPred)
```

如果第二个序列的某个排列和第一个序列具有相同数目的元素，且元素都相等，则返回 true。第一个版本用==比较元素，第二个版本使用给定的 binaryPred。

```
next_permutation(beg, end)
next_permutation(beg, end, comp)
```

如果序列已经是最后一个排列，则 next_permutation 将序列重排为最小的排列，并返回 false。否则，它将输入序列转换为字典序中下一个排列，并返回 true。第一个版本使用元素的<运算符比较元素，第二个版本使用给定的比较操作。

```
prev_permutation(beg, end)
prev_permutation(beg, end, comp)
```

类似 next_premutation，但将序列转换为前一个排列。如果序列已经是最小的排列，则将其重排为最大的排列，并返回 false。

880

A.2.8 有序序列的集合算法

集合算法实现了有序序列上的一般集合操作。这些算法与标准库 set 容器不同，不要与 set 上的操作相混淆。这些算法提供了普通顺序容器（vector、list 等）或其他序列（如输入流）上的类集合行为。

这些算法顺序处理元素，因此要求输入迭代器。他们还接受一个表示目的序列的输出迭代器，唯一的例外是 includes。这些算法返回递增后的 dest 迭代器，表示写入 dest 的最后一个元素之后的位置。

每种算法都有重载版本，第一个使用元素类型的<运算符，第二个使用给定的比较操作。

```
includes(beg, end, beg2, end2)
includes(beg, end, beg2, end2, comp)
```

如果第二个序列中每个元素都包含在输入序列中，则返回 true。否则返回 false。

```
set_union(beg, end, beg2, end2, dest)
set_union(beg, end, beg2, end2, dest, comp)
```

对两个序列中的所有元素，创建它们的有序序列。两个序列都包含的元素在输出序列中只出现一次。输出序列保存在 dest 中。

```
set_intersection(beg, end, beg2, end2, dest)
set_intersection(beg, end, beg2, end2, dest, comp)
```

对两个序列都包含的元素创建一个有序序列。结果序列保存在 dest 中。

```
set_difference(beg, end, beg2, end2, dest)
set_difference(beg, end, beg2, end2, dest, comp)
```

对出现在第一个序列中，但不在第二个序列中的元素，创建一个有序序列。

```
set_symmetric_difference(beg, end, beg2, end2, dest)
set_symmetric_difference(beg, end, beg2, end2, dest, comp)
```

对只出现在一个序列中的元素，创建一个有序序列。

A.2.9 最小值和最大值

这些算法使用元素类型的<运算符或给定的比较操作。第一组算法对值而非序列进行操作。第二组算法接受一个序列，它们要求输入迭代器。

```
min(val1, val2)
min(val1, val2, comp)
min(init_list)
min(init_list, comp)
max(val1, val2)
max(val1, val2, comp)
max(init_list)
max(init_list, comp)
```

881

返回 val1 和 val2 中的最小值/最大值，或 initializer_list 中的最小值/最大值。两个实参的类型必须完全一致。参数和返回类型都是 const 的引用，意味着对象不会被拷贝。

```
minmax(val1, val2)
minmax(val1, val2, comp)
minmax(init_list)
minmax(init_list, comp)
```

返回一个 pair(参见 11.2.3 节，第 379 页)，其 first 成员为提供的值中的较小者，second 成员为较大者。initializer_list 版本返回一个 pair，其 first 成员为 list 中的最小值，second 为最大值。

```
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
minmax_element(beg, end)
minmax_element(beg, end, comp)
```

min_element 和 max_element 分别返回指向输入序列中最小和最大元素的迭代器。minmax_element 返回一个 pair，其 first 成员为最小元素，second 成员为最大元素。

字典序比较

此算法比较两个序列，根据第一对不相等的元素的相对大小来返回结果。算法使用元素类型的<运算符或给定的比较操作。两个序列都要求用输入迭代器给出。

```
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)
```

如果第一个序列在字典序中小于第二个序列，则返回 true。否则，返回 false。如果一个序列比另一个短，且所有元素都与较长序列的对应元素相等，则较短序列在字典序中更小。如果序列长度相等，且对应元素都相等，则在字典序中任何一个都不大于另外一个。

A.2.10 数值算法

数值算法定义在头文件 numeric 中。这些算法要求输入迭代器；如果算法输出数据，则使用输出迭代器表示目的位置。

882

```
accumulate(beg, end, init)
accumulate(beg, end, init, binaryOp)
```

返回输入序列中所有值的和。和的初值从 init 指定的值开始。返回类型与 init 的类型相同。第一个版本使用元素类型的+运算符，第二个版本使用指定的二元操作。

```
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, binOp1, binOp2)
```

返回两个序列的内积，即，对应元素的积的和。两个序列一起处理，来自两个序列的元素相乘，乘积被累加起来。和的初值由 init 指定，init 的类型确定了返回类型。

第一个版本使用元素类型的乘法（*）和加法（+）运算符。第二个版本使用给定的二元操作，使用第一个操作代替加法，第二个操作代替乘法。

```
partial_sum(beg, end, dest)
partial_sum(beg, end, dest, binaryOp)
```

将新序列写入 dest，每个新元素的值都等于输入范围中当前位置和之前位置上所有元素之和。第一个版本使用元素类型的+运算符；第二个版本使用指定的二元操作。算法返回递增后的 dest 迭代器，指向最后一个写入元素之后的位置。

```
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, binaryOp)
```

将新序列写入 dest，每个新元素（除了首元素之外）的值都等于输入范围中当前位置和前一个位置元素之差。第一个版本使用元素类型的-运算符，第二个版本使用指定的二元操作。

```
iota(beg, end, val)
```

将 val 赋予首元素并递增 val。将递增后的值赋予下一个元素，继续递增 val，然后将递增后的值赋予序列中的下一个元素。继续递增 val 并将其新值赋予输入序列中的后续元素。

A.3 随机数

标准库定义了一组随机数引擎类和适配器，使用不同数学方法生成伪随机数。标准库还定义了一组分布模板，根据不同的概率分布生成随机数。引擎和分布类型的名字都与它们的数学性质相对应。

这些类如何生成随机数的细节已经大大超出了本书的范围。在本节中，我们将列出这些引擎和分布类型，但读者需要查询其他资料来学习如何使用这些类型。

A.3.1 随机数分布

883

除了总是生成 bool 类型的 bernoulli_distribution 外，其他分布类型都是模板。每个模板都接受单个类型参数，它指出了分布生成的结果类型。

分布类与我们已经用过的其他类模板不同，它们限制了我们可以为模板类型指定哪些类型。一些分布模板只能用来生成浮点数，而其他模板只能用来生成整数。

在下面的描述中，我们通过将类型说明为 *template_name*<RealT>来指出分布生成浮点数。对这些模板，我们可以用 float、double 或 long double 代替 RealT。类似的，IntT 表示要求一个内置整型类型，但不包括 bool 类型或任何 char 类型。可以用来代替 IntT 的类型是 short、int、long、long long、unsigned short、unsigned int、unsigned long 或 unsigned long long。

分布模板定义了一个默认模板类型参数（参见 17.4.2 节，第 664 页）。整型分布的默认参数是 int，生成浮点数的模板的默认参数是 double。

每个分布的构造函数都有这种分布特定的参数。某些参数指出了分布的范围。这些范围与迭代器范围不同，都是包含的。

均匀分布

```
uniform_int_distribution<IntT> u(m, n);
uniform_real_distribution<RealT> u(x, y);
```

生成指定类型的，在给定包含范围内的值。m（或 x）是可以返回的最小值；n（或 y）是最大值。m 默认为 0；n 默认为类型 IntT 对象可以表示的最大值。x 默认为 0.0，y 默认为 1.0。

伯努利分布

```
bernoulli_distribution b(p);
```

以给定概率 p 生成 true；p 的默认值为 0.5。

```
binomial_distribution<IntT> b(t, p);
```

分布是按采样大小为整型值 t，概率为 p 生成的；t 的默认值为 1，p 的默认值为 0.5。

```
geometric_distribution<IntT> g(p);
```

每次试验成功的概率为 p；p 的默认值为 0.5。

```
negative_binomial_distribution<IntT> nb(k, p);
```

k（整型值）次试验成功的概率为 p；k 的默认值为 1，p 的默认值为 0.5。

泊松分布

```
poisson_distribution<IntT> p(x);
```

均值为 double 值 x 的分布。

884

```
exponential_distribution<RealT> e(lam);
```

指数分布，参数 lambda 通过浮点值 lam 给出；lam 的默认值为 1.0。

```
gamma_distribution<RealT> g(a, b);
```

alpha（形状参数）为 a，beta（尺度参数）为 b；两者的默认值均为 1.0。

```
weibull_distribution<RealT> w(a, b);
```

形状参数为 a，尺度参数为 b 的分布；两者的默认值均为 1.0。

```
extreme_value_distribution<RealT> e(a, b);
```

a 的默认值为 0.0，b 的默认值为 1.0。

正态分布

```
normal_distribution<RealT> n(m, s);
```

均值为 m，标准差为 s；m 的默认值为 0.0，s 的默认值为 1.0。

```
lognormal_distribution<RealT> ln(m, s);
```

均值为 m，标准差为 s；m 的默认值为 0.0，s 的默认值为 1.0。

```
chi_squared_distribution<RealT> c(x);
```

自由度为 x；默认值为 1.0。

```
cauchy_distribution<RealT> c(a, b);
```

位置参数 a 和尺度参数 b 的默认值分别为 0.0 和 1.0。

```
fisher_f_distribution<RealT> f(m, n);
```

自由度为 m 和 n；默认值均为 1。

```
student_t_distribution<RealT> s(n);
```

自由度为 n；n 的默认值均为 1。

抽样分布

```
discrete_distribution<IntT> d(i, j);
discrete_distribution<IntT> d{il};
```

i 和 j 是一个权重序列的输入迭代器，il 是一个权重的花括号列表。权重必须能转换为 double。

```
piecewise_constant_distribution<RealT> pc(b, e, w);
```

b、e 和 w 是输入迭代器。

```
piecewise_linear_distribution<RealT> pl(b, e, w);
```

b、e 和 w 是输入迭代器。

A.3.2 随机数引擎

标准库定义了三个类，实现了不同的算法来生成随机数。标准库还定义了三个适配器，可以修改给定引擎生成的序列。引擎和引擎适配器类都是模板。与分布的参数不同，这些引擎的参数更为复杂，且需深入了解特定引擎使用的数学知识。我们在这里列出所有引擎，885以便读者对它们有所了解，但介绍如何生成这些类型超出了本书的范围。

标准库还定义了几个从引擎和适配器类型构造的类型。default_random_engine类型是一个参数化的引擎类型的类型别名，参数化所用的变量的目的是在通常情况下获得好的性能。标准库还定义了几个类，它们都是一个引擎或适配器的完全特例化版本。标准库定义的引擎和特例化版本如下：

default_random_engine

某个其他引擎类型的类型别名，目的是用于大多数情况。

linear_congruential_engine

minstd_rand0 的乘数为 16807，模为 2147483647，增量为 0。

minstd_rand 的乘数为 48271，模为 2147483647，增量为 0。

mersenne_twister_engine

mt19937 为 32 位无符号梅森旋转生成器。

mt19937_64 为 64 位无符号梅森旋转生成器。

subtract_with_carry_engine

ranlux24_base 为 32 位无符号借位减法生成器。

ranlux48_base 为 64 位无符号借位减法生成器。

discard_block_engine

引擎适配器，将其底层引擎的结果丢弃。用要使用的底层引擎、块大小和旧块大小来参数化。

ranlux24 使用 ranlux24_base 引擎，块大小为 223，旧块大小为 23。

ranlux48 使用 ranlux48_base 引擎，块大小为 389，旧块大小为 11。

independent_bits_engine

引擎适配器，生成指定位数的随机数。用要使用的底层引擎、结果的位数以及保存生成的二进制位的无符号整型类型来参数化。指定的位数必须小于指定的无符号类型所能保存的位数。

shuffle_order_engine

引擎适配器，返回的就是底层引擎生成的数，但返回的顺序不同。用要使用的底层引擎和要混洗的元素数目来参数化。

knuth_b 使用 minstd_rand0 和表大小 256。

done!

done!

索引

粗体页码指的是第一次定义该术语的页码，*斜体*页码指的是各章“术语表”定义该术语的页码。

C++11 的新特性

Symbols

A