# 附录 A　计数系统

人们使用很多计数系统来表示数字。有些计数系统（如罗马数字）不适合用于算术运算；而印度计数系统经过改进，并传入到欧洲后变成了阿拉伯计数系统，这种数字方便了数学、科学和商业计算。现代的计算机计数系统是基于占位符概念的，使用了最先出现在印度计数系统中的零。然而，这种原理被推广到其他计数系统。因此，虽然在日常生活中使用的是下一节将介绍的十进制，但计算领域通常使用八进制、十六进制和二进制。

## A.1　十进制数

数字的书写方式是基于 10 的幂数。例如，对于数字 2468，2 表示 2 个 1000，4 表示 4 个 100，6 表示 6 个 10，8 表示 8 个 1。

$2468 = 2 \times 1000 + 4 \times 100 + 6 \times 10 + 8 \times 1$

一千是 $10 \times 10 \times 10$ 或 10 的 3 次幂，用 $10^3$ 表示。使用这种表示法，可以这样书写上述关系：

$2468 = 2 \times 10^3 + 4 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$

因为这种数字表示法是基于 10 的幂，所以将它称作基数为 10 的表示法或十进制表示法。可以用任何数作基数。例如，C++允许使用基数 8（八进制）和基数 16（十六进制）来书写整数（请注意，$10^0$ 为 1，任何非零数的 0 次幂都为 1）。

## A.2　八进制整数

八进制数是基于 8 的幂的，所以基数为 8 的表示法用数字 0-7 来书写数字。C++用前缀 0 来表示八进制表示法。也就是说，0177 是一个八进制值。可以用 8 的幂来找到对应的十进制值：

| 八　进　制 | 十　进　制 |
|---|---|
| 177 | $= 1 \times 8^2 + 7 \times 8^1 + 7 \times 8^0$ |
| | $= 1 \times 64 + 7 \times 8 + 7 \times 1$ |
| | $= 127$ |

由于 UNIX 操作系统常使用八进制来表示值，因此 C++和 C 提供了八进制表示法。

## A.3　十六进制数

十六进制数是基于 16 的幂的。这意味着十六进制的 10 表示 16 + 0，即 16。为表示 9-16 值，需要其他一些数字，标准的十六进制表示法使用字母 a-f。C++接受这些字符的大写和小写版本，如表 A.1 所示。

**表 A.1　　十六进制数**

| 十六进制数 | 十 进 制 值 |
|---|---|
| a 或 A | 10 |
| b 或 B | 11 |
| c 或 C | 12 |
| d 或 D | 13 |
| e 或 E | 14 |
| f 或 F | 15 |

C++使用 0x 或 0X 来指示十六进制表示法。因此 0x2B3 是一个十六进制值，可使用 16 的幂来得到对应的十进制值。

| 十六进制 | 十 进 制 |
|---|---|
| 0x2B3 | $= 2 \times 16^2 + 11 \times 16^1 + 3 \times 16^0$ |
| | $= 2 \times 256 + 11 \times 16 + 3 \times 1$ |
| | $= 691$ |

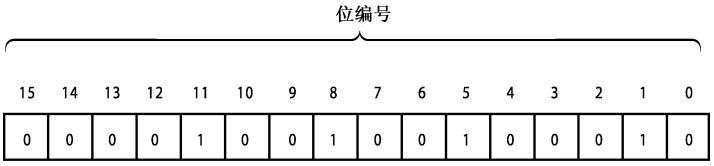
硬件文档常使用十六进制来表示诸如内存单元和端口号等值。

## A.4 二进制数

不管是使用十进制、八进制，还是十六进制表示法来书写整数，计算机都将它存储为二进制值（即基数为 2）。二进制表示法只使用两个数字——0 和 1。例如，10011011 就是二进制数。但 C++没有提供二进制表示法来书写数字的方式。二进制数是基于 2 的幂。

| 二 进 制 | 十 进 制 |
|---|---|
| 10011011 | $= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ |
| | $= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$ |
| | $= 155$ |

二进制表示法与计算机内存完全对应，在内存中，每个单元（位）都可以设置成开或关。只是将关表示为 0，将开表示为 1。内存通常是以字节为单位组织的，每个字节包含 8 位（正如第 2 章指出的，C++字节并非一定是 8 位，但本附录采用常见的做法，用字节表示八位组）。字节中的位被编号，对应于相关的 2 的幂。这样，最右侧的位编号为 0，然后是 1，依此类推。例如，图 A.1 表示一个 2 字节的整数。




$$\text{值} = 1 \times 2^{11} + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^1$$
$$= 2048 + 256 + 32 + 2$$
$$= 2338$$

图 A.1 2 字节整数值

## A.5 二进制和十六进制

十六进制表示法常用于提供更为方便的二进制数据（如内存地址或存储位标记设置的整数）视图。这样做的原因是，每个十六进制位对应于 4 位。表 A.2 说明了这种对应关系。

**表 A.2 十六进制数和对应的二进制数**

| 十六进制位 | 对应的二进制数 |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |

续表

| 十六进制位 | 对应的二进制数 |
| --- | --- |
| 9 | 1001 |
| A | 1010 |
| B | 1011 |
| C | 1100 |
| D | 1101 |
| E | 1110 |
| F | 1111 |

要将十六进制值转换为二进制，只需将每个十六进制位替换为相应的二进制数即可。例如，十六进制 0xA4 对应于二进制数 10100100。同样，可以轻松地将二进制值转换为十六进制，方法是将每 4 位转换为对应的十六进制位。例如，二进制值 10010101 将被转换为 0x95。

**Big Endian 和 Little Endian**

奇怪的是，都使用整数的二进制表示的两个计算平台对同一个值的表示可能并不相同。例如，Intel 计算机使用 Little Endian 体系结构来存储字节，而 Motorola 处理器、IBM 大型机、SPARC 处理器和 ARM 处理器使用 Big Endian 方案（但最后的两种系统可配置成使用上述任何一种方案）。

术语 Big Endian 和 Little Endian 是从“Big End In”和“Little End In”（指内存中单词（通常为两个字节）的字节顺序）衍生而来的。在 Intel 计算机（Little Endian）中，先存储低位字节，这意味着十六进制值 0xABCD 在内存中将被存储为 0xCD 0xAB。Motorola（Big Endian）计算机按相反的顺序存储，因此 0xABCD 在内存中被存储为 0xAB 0xCD。

这些术语最先出现在 Jonathan Swift 编写的《Gulliver’s Travels》一书中。为讽刺众多政治斗争的非理性，Swift 杜撰了假想国中两个喜欢争论的政治派别：Big Endians 和 Little Endians，前者坚持认为从大的一头打破鸡蛋更合理，而后者坚持认为从小的一头打破鸡蛋更合理。

作为软件工程师，应了解目标平台的词序，它会影响对通过网络传输的数据的解释方式以及数据在二进制文件中的存储方式。在上面的例子中，二字节内存模式 0xABCD 在 Little Endian 计算机上表示十进制值 52651，而在 Big Endian 计算机上表示十进制值 43981。

# 附录 B　C++保留字

C++保留了一些单词供自己和 C++库使用。程序员不应将保留字用作声明中的标识符。保留字分三类：关键字、替代标记（alternative token）和 C++库保留名称。

## B.1　C++关键字

关键字是组成编程语言词汇表的标识符，它们不能用于其他用途，如用作变量名。表 B.1 列出了 C++关键字，其中以粗体显示的关键字也是 ANSI C99 标准中的关键字，而以斜体显示的关键字是 C++11 新增的。

表 B.1　　C++关键字

| | | | | |
|---|---|---|---|---|
| alignas | alignof | asm | **auto** | bool |
| **break** | **case** | catch | **char** | char16_t |
| char32_t | class | **const** | const_cast | constexpr |
| **continue** | decltype | **default** | delete | **do** |
| **double** | dynamic_cast | **else** | **enum** | explicit |
| export | **extern** | false | **float** | **for** |
| friend | **goto** | **if** | **inline** | **int** |
| **long** | mutable | namespace | new | noexcept |
| nullptr | operator | private | protected | public |
| **register** | reinterpret_cast | **return** | **short** | **signed** |
| **sizeof** | **static** | static_assert | static_cast | **struct** |
| **switch** | template | this | thread_local | throw |
| true | try | **typedef** | typeid | typename |
| **union** | **unsigned** | using | virtual | **void** |
| **volatile** | wchar_t | **while** | | |

## B.2　替代标记

除关键字外，C++还有一些运算符的字母替代表示，它们被称为替代标记。替代标记也被保留，表 B.2 列出了替代标记及其表示的运算符。

表 B.2　　C++保留的替代标记及其含义

| 标　记 | 含　义 |
|---|---|
| and | && |
| and_eq | &= |
| bitand | & |
| bitor | \| |
| compl | ~ |
| not | ! |
| not_eq | != |
| or | \|\| |
| or_eq | \|= |
| xor | ^ |
| xor_eq | ^= |

## B.3　C++库保留名称

编译器不允许程序员将关键字和替代标记用作名称。还有另一类禁止使用（但并非绝对不能用）的名称——保留名称，它们是保留给 C++库使用的名称。如果您将这种名称用作标识符，后果将是不确定的。也就是说，可能导致编译器错误、警告、程序不能正确运行或根本不会导致任何问题。

C++语言保留了库头文件中使用的宏名。如果程序包含某个头文件，则不应将该头文件（以及该头文件包含的头文件，依此类推）中定义的宏名用作其他目的。例如，如果您直接或间接地包含了头文件<climits>，则不应将 CHAR_BIT 用作标识符，因为它已被用作该头文件中一个宏的名称。

C++语言保留了以两个下划线或下划线和大写字母打头的名称，还将以单个下划线打头的名称保留用作全局变量。因此，程序员不能在全局名称空间使用诸如__gink、__Lynx 和_lynx 等名称。

C++语言保留了在库头文件中被声明为链接性为外部的名称。对于函数，这包括函数的特征标（名称和参数列表）。例如，假设有如下代码：

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

则函数特征标 tan（double）被保留。这意味着您的程序不应声明一个原型如下所示的函数：

```
int tan(double); // don't do it
```

该原型确实与库函数 tan() 的原型不同，因为后者的返回类型为 double，但特征标部分确实相同。然而，定义下面的原型是可以的：

```
char * tan(char *); // ok
```

这是因为虽然其名称与库函数 tan()相同，但特征标不同。

## B.4　有特殊含义的标识符

C++社区讨厌新增关键字，因为它们可能与现有代码发生冲突。这就是标准委员会改变关键字 auto 的用法，并赋予其他关键字（如 virtual 和 delete）新用法的原因所在。C++11 提供了另一种避免新增关键字的机制，即使用具有特殊含义的标识符。这些标识符不是关键字，但用于实现语言功能。编译器根据上下文来判断它们是常规标识符还是用于实现语言功能：

```
class F
{
    int final;                          // #1
public:
...
    virtual void unfold() {...} = final; // #2
};
```

在上述代码中，语句#1 中的 final 是一个常规标识符，而语句#2 中的 final 使用了一种语言功能。这两种用法彼此不会冲突。

另外，C++还有很多经常出现在程序中，但不被保留的标识符。这包括头文件名、库函数名和 main（必不可少的函数的名称，程序从该函数开始执行）。只要不发生名称空间冲突，就可将这些标识符用于其他目的，但没有理由这样做。也就是说，完全可以编写下面的代码，但常识告诉您不应这样做：

```
// allowable but silly
#include <iostream>
int iostream(int a);
int main ()
{
    std::
cout << iostream(5) << '\n';
    return 0;
}

int iostream(int a)
{
    int main = a + 1;
    int cout = a -1;
    return main*cout;
}
```

# 附录 C　ASCII 字符集

计算机使用数字代码来存储字符。ASCII 码是美国最常用的编码，它是 Unicode 的一个子集（一个非常小的子集）。C++使得能够直接表示大多数字符，方法是将字符用单引号括起，例如‘A’表示字符 A。也可以用前面带反斜杠的八进制或十六进制编码来表示单个字符，例如，‘\012’和‘\0xa’表示的都是换行符（LF）。这种转义序列还可放在字符串中，如“Hello, \012my dear”。

表 C.1 列出了以各种方式表示的 ASCII 字符集。在该表中，当被用作前缀时，^字符表示使用 Ctrl 键。

表 C.1　　ASCⅡ字符集

| 十　进　制 | 八　进　制 | 十六进制 | 二　进　制 | 字　　符 | ASCII 名称 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 00000000 | ^@ | NUL |
| 1 | 01 | 0x1 | 00000001 | ^A | SOH |
| 2 | 02 | 0x2 | 00000010 | ^B | STX |
| 3 | 03 | 0x3 | 00000011 | ^C | ETX |
| 4 | 04 | 0x4 | 00000100 | ^D | EOT |
| 5 | 05 | 0x5 | 00000101 | ^E | ENQ |
| 6 | 06 | 0x6 | 00000110 | ^F | ACK |
| 7 | 07 | 0x7 | 00000111 | ^G | BEL |
| 8 | 010 | 0x8 | 00001000 | ^H | BS |
| 9 | 011 | 0x9 | 00001001 | ^I，tab | HT |
| 10 | 012 | 0xa | 00001010 | ^J | LF |
| 11 | 013 | 0xb | 00001011 | ^K | VT |
| 12 | 014 | 0xc | 00001100 | ^L | FF |
| 13 | 015 | 0xd | 00001101 | ^M | CR |
| 14 | 016 | 0xe | 00001110 | ^N | SO |
| 15 | 017 | 0xf | 00001111 | ^O | SI |
| 16 | 020 | 0x10 | 00010000 | ^P | DLE |
| 17 | 021 | 0x11 | 00010001 | ^Q | DC1 |
| 18 | 022 | 0x12 | 00010010 | ^R | DC2 |
| 19 | 023 | 0x13 | 00010011 | ^S | DC3 |
| 20 | 024 | 0x14 | 00010100 | ^T | DC4 |
| 21 | 025 | 0x15 | 00010101 | ^U | NAK |
| 22 | 026 | 0x16 | 00010110 | ^V | SYN |
| 23 | 027 | 0x17 | 00010111 | ^W | ETB |
| 24 | 030 | 0x18 | 00011000 | ^X | CAN |
| 25 | 031 | 0x19 | 00011001 | ^Y | EM |
| 26 | 032 | 0x1a | 00011010 | ^Z | SUB |
| 27 | 033 | 0x1b | 00011011 | ^[，Esc | ESC |
| 28 | 034 | 0x1c | 00011100 | ^\ | FS |
| 29 | 035 | 0x1d | 00011101 | ^] | GS |
| 30 | 036 | 0x1e | 00011110 | ^^ | RS |
| 31 | 037 | 0x1f | 00011111 | ^_ | US |
| 32 | 040 | 0x20 | 00100000 | 空格 | SP |
| 33 | 041 | 0x21 | 00100001 | ! | |
| 34 | 042 | 0x22 | 00100010 | ” | |
| 35 | 043 | 0x23 | 00100011 | # | |
| 36 | 044 | 0x24 | 00100100 | $ | |
| 37 | 045 | 0x25 | 00100101 | % | |

续表

| 十　进　制 | 八　进　制 | 十六进制 | 二　进　制 | 字　　符 | ASCII 名称 |
|---|---|---|---|---|---|
| 38 | 046 | 0x26 | 00100110 | & | |
| 39 | 047 | 0x27 | 00100111 | ' | |
| 40 | 050 | 0x28 | 00101000 | ( | |
| 41 | 051 | 0x29 | 00101001 | ) | |
| 42 | 052 | 0x2a | 00101010 | * | |
| 43 | 053 | 0x2b | 00101011 | + | |
| 44 | 054 | 0x2c | 00101100 | ' | |
| 45 | 055 | 0x2d | 00101101 | - | |
| 46 | 056 | 0x2e | 00101110 | . | |
| 47 | 057 | 0x2f | 00101111 | / | |
| 48 | 060 | 0x30 | 00110000 | 0 | |
| 49 | 061 | 0x31 | 00110001 | 1 | |
| 50 | 062 | 0x32 | 00110010 | 2 | |
| 51 | 063 | 0x33 | 00110011 | 3 | |
| 52 | 064 | 0x34 | 00110100 | 4 | |
| 53 | 065 | 0x35 | 00110101 | 5 | |
| 54 | 066 | 0x36 | 00110110 | 6 | |
| 55 | 067 | 0x37 | 00110111 | 7 | |
| 56 | 070 | 0x38 | 00111000 | 8 | |
| 57 | 071 | 0x39 | 00111001 | 9 | |
| 58 | 072 | 0x3a | 00111010 | : | |
| 59 | 073 | 0x3b | 00111011 | ; | |
| 60 | 074 | 0x3c | 00111100 | < | |
| 61 | 075 | 0x3d | 00111101 | = | |
| 62 | 076 | 0x3e | 00111110 | > | |
| 63 | 077 | 0x3f | 00111111 | ? | |
| 64 | 0100 | 0x40 | 01000000 | @ | |
| 65 | 0101 | 0x41 | 01000001 | A | |
| 66 | 0102 | 0x42 | 01000010 | B | |
| 67 | 0103 | 0x43 | 01000011 | C | |
| 68 | 0104 | 0x44 | 01000100 | D | |
| 69 | 0105 | 0x45 | 01000101 | E | |
| 70 | 0106 | 0x46 | 01000110 | F | |
| 71 | 0107 | 0x47 | 01000111 | G | |
| 72 | 0110 | 0x48 | 01001000 | H | |
| 73 | 0111 | 0x49 | 01001001 | I | |
| 74 | 0112 | 0x4a | 01001010 | J | |
| 75 | 0113 | 0x4b | 01001011 | K | |
| 76 | 0114 | 0x4c | 01001100 | L | |
| 77 | 0115 | 0x4d | 01001101 | M | |
| 78 | 0116 | 0x4e | 01001110 | N | |
| 79 | 0117 | 0x4f | 01001111 | O | |
| 80 | 0120 | 0x50 | 01010000 | P | |
| 81 | 0121 | 0x51 | 01010001 | Q | |
| 82 | 0122 | 0x52 | 01010010 | R | |
| 83 | 0123 | 0x53 | 01010011 | S | |
| 84 | 0124 | 0x54 | 01010100 | T | |
| 85 | 0125 | 0x55 | 01010101 | U | |
| 86 | 0126 | 0x56 | 01010110 | V | |
| 87 | 0127 | 0x57 | 01010111 | W | |
| 88 | 0130 | 0x58 | 01011000 | X | |
| 89 | 0131 | 0x59 | 01011001 | Y | |
| 90 | 0132 | 0x5a | 01011010 | Z | |
| 91 | 0133 | 0x5b | 01011011 | [ | |

续表

| 十 进 制 | 八 进 制 | 十六进制 | 二 进 制 | 字 符 | ASCII 名称 |
|---|---|---|---|---|---|
| 92 | 0134 | 0x5c | 01011100 | \ | |
| 93 | 0135 | 0x5d | 01011101 | ] | |
| 94 | 0136 | 0x5e | 01011110 | ^ | |
| 95 | 0137 | 0x5f | 01011111 | - | |
| 96 | 0140 | 0x60 | 01100000 | ' | |
| 97 | 0141 | 0x61 | 01100001 | a | |
| 98 | 0142 | 0x62 | 01100010 | b | |
| 99 | 0143 | 0x63 | 01100011 | c | |
| 100 | 0144 | 0x64 | 01100100 | d | |
| 101 | 0145 | 0x65 | 01100101 | e | |
| 102 | 0146 | 0x66 | 01100110 | f | |
| 103 | 0147 | 0x67 | 01100111 | g | |
| 104 | 0150 | 0x68 | 01101000 | h | |
| 105 | 0151 | 0x69 | 01101001 | i | |
| 106 | 0152 | 0x6a | 01101010 | j | |
| 107 | 0153 | 0x6b | 01101011 | k | |
| 108 | 0154 | 0x6c | 01101100 | l | |
| 109 | 0155 | 0x6d | 01101101 | m | |
| 110 | 0156 | 0x6e | 01101110 | n | |
| 111 | 0157 | 0x6f | 01101111 | o | |
| 112 | 0160 | 0x70 | 01110000 | p | |
| 113 | 0161 | 0x71 | 01110001 | q | |
| 114 | 0162 | 0x72 | 01110010 | r | |
| 115 | 0163 | 0x73 | 01110011 | s | |
| 116 | 0164 | 0x74 | 01110100 | t | |
| 117 | 0165 | 0x75 | 01110101 | u | |
| 118 | 0166 | 0x76 | 01110110 | v | |
| 119 | 0167 | 0x77 | 01110111 | w | |
| 120 | 0170 | 0x78 | 01111000 | x | |
| 121 | 0171 | 0x79 | 01111001 | y | |
| 122 | 0172 | 0x7a | 01111010 | z | |
| 123 | 0173 | 0x7b | 01111011 | { | |
| 124 | 0174 | 0x7c | 01111100 | \| | |
| 125 | 0175 | 0x7d | 01111101 | } | |
| 126 | 0176 | 0x7e | 01111110 | ~ | |
| 127 | 0177 | 0x7f | 01111111 | Del | |

# 附录 D　运算符优先级

运算符优先级决定了运算符用于值的顺序。C++运算符分为 18 个优先级组，如表 D.1 所示。第 1 组中的运算符的优先级最高，第 2 组中运算符的优先级次之，依此类推。如果两个运算符被用于同一个操作数，则首先应用优先级高的运算符。如果两个运算符的优先级相同，则 C++使用结合性规则来决定哪个运算符结合得更为紧密。同一组中运算符的优先级和结合性相同，不管是从左到右（表中 L-R）还是从右到左（表中 R-L）结合。从左到右的结合性意味着首先应用最左边的运算符，而从右到左的结合性则意味着首先应用最右边的运算符。

**表 D.1　　C++运算符的优先级和结合性**

| 运　算　符 | 结　合　性 | 含　　义 |
|---|---|---|
| 优先级第 1 组 | | |
| :: | | 作用域解析运算符 |
| 优先级第 2 组 | | |
| （表达式） | | 分组 |
| () | L-R | 函数调用 |
| () | | 值构造，即 type(expr) |
| [ ] | | 数组下标 |
| -> | | 间接成员运算符 |
| . | | 直接成员运算符 |
| const_cast | | 专用的类型转换 |
| dynamic_cast | | 专用的类型转换 |
| reinterpret_cast | | 专用的类型转换 |
| static_cast | | 专用的类型转换 |
| typeid | | 类型标识 |
| ++ | | 加 1 运算符，后缀 |
| - - | | 减 1 运算符，后缀 |
| 优先级第 3 组（全是一元运算符） | | |
| ! | R-L | 逻辑非 |
| ~ | | 位非 |
| + | | 一元加号（正号） |
| - | | 一元减号（负号） |
| ++ | | 加 1 运算符，前缀 |
| - - | | 减 1 运算符，前缀 |
| & | | 地址 |
| * | | 解除引用（间接值） |
| () | | 类型转换，即(type)expr |
| sizeof | | 长度，以字节为单位 |
| new | | 动态分配内存 |
| new [ ] | | 动态分配数组 |
| delete | | 动态释放内存 |
| delete [ ] | | 动态释放数组 |

续表

| 运 算 符 | 结 合 性 | 含 义 |
|---|---|---|
| 优先级第 4 组 | | |
| . * | L-R | 成员解除引用 |
| ->* | | 间接成员解除引用 |
| 优先级第 5 组（全是二元运算符） | | |
| * | L-R | 乘 |
| / | | 除 |
| ^ | | 模（余数） |
| 优先级第 6 组（全是二元运算符） | | |
| + | L-R | 加 |
| - | | 减 |
| 优先级第 7 组 | | |
| << | L-R | 左移 |
| >> | | 右移 |
| 优先级第 8 组 | | |
| < | L-R | 小于 |
| <= | | 小于或等于 |
| >= | | 大于或等于 |
| > | | 大于 |
| 优先级第 9 组 | | |
| = = | L-R | 等于 |
| != | | 不等于 |
| 优先级第 10 组（一元运算符） | | |
| & | L-R | 按位 AND |
| 优先级第 11 组 | | |
| ^ | L-R | 按位 XOF（异或） |
| 优先级第 12 组 | | |
| \| | L-R | 按位 OR |
| 优先级第 13 组 | | |
| && | L-R | 逻辑 AND |
| 优先级第 14 组 | | |
| \|\| | L-R | 逻辑 OR |
| 优先级第 15 组 | | |
| :? | R-L | 条件 |
| 优先级第 16 组 | | |
| = | R-L | 简单赋值 |
| * = | | 乘并赋值 |
| /= | | 除并赋值 |
| %= | | 求模并赋值 |
| += | | 加并赋值 |
| -= | | 减并赋值 |
| &= | | 按位 AND 并赋值 |
| ^= | | 按位 XOR 并赋值 |
| \|= | | 按位 OR 并赋值 |
| <<= | | 左移并赋值 |
| >>= | | 右移并赋值 |

续表

| 运　算　符 | 结　合　性 | 含　　义 |
|---|---|---|
| 优先级第 17 组 | | |
| throw | L-R | 引发异常 |
| 优先级第 18 组 | | |
| , | L-R | 将两个表达式合并成一个 |

有些符号（如*或&）被用作多个运算符。在这种情况下，一种形式是一元（一个操作数），另一种形式是二元（两个操作数），编译器将根据上下文来确定使用哪种形式。对于同一个符号可以两种方式使用的情况，表 D.1 将运算符标记为一元组或二元组。

下面介绍一些优先级和结合性的例子。

对于下面的例子，编译器必须决定先将 5 和 3 相加，还是先将 5 和 6 相乘：

```
3 + 5 * 6
```

*运算符的优先级比+运算符高，所以它被首先用于 5，因此表达式变成 3 +30，即 33。

对于下面的例子，编译器必须决定先将 120 除以 6，还是先将 6 和 5 相乘：

```
120 / 6 * 5
```

/和*的优先级相同，但这些运算符从左到右结合的。这意味着首先应用操作数（6）左侧的运算符，因此表达式变为 20*5，即 100。

对于下面的例子，编译器必须决定先对 str 递增还是先对 str 解除引用：

```
char * str = "Whoa";
char ch = *str++;
```

后缀++运算符的优先级比一元运算符*高，这意味着加号运算符将对 str 进行操作，而不是对*str 进行操作。也就是说，将指针加 1，使之指向下一个字符，而不是修改被指针指向的字符。不过，由于++是后缀形式，因此将在将*str 的值赋给 ch 后，再将指针加 1。因此，上述表达式将字符 W 赋给 ch，然后移动指针 str，使之指向字符 h。

下面是一个类似的例子：

```
char * str = "Whoa";
char ch = *++str;
```

前缀++运算符和一元运算符*的优先级相同，但它们是从右到左结合的。因此，str（不是*str）将被加 1。因为++运算符是前缀形式，所以首先将 str 加 1，然后将得到的指针执行解除引用的操作。因此，str 将指向字符 h，并将字符 h 赋给 ch。

注意，表 D.1 在“优先级”行中使用一元或二元来区分使用同一个符号的两个运算符，如一元地址运算符和二元按位 AND 运算符。

附录 B 列出了一些运算符的替代表示。

# 附录 E　其他运算符

为了避免篇幅过长，有三组运算符没有在本书正文部分介绍。第一组是按位运算符，能够操纵值中的各个位；这些运算符是从 C 语言继承而来的；第二组运算符是两个成员解除引用运算符，它们是 C++新增的；第三组是 C++11 新增的运算符：alignof 和 noexcept。本附录将简要地对这些运算符做一总结。

## E.1　按位运算符

按位运算符对整数值的位进行操作。例如，左移运算符将位向左移，按位非运算符将所有的 1 变成 0，所有的 0 变成 1，C++共有 6 个这样的运算符：<<、>>、~、&、|和^。

### E.1.1　移位运算符

左移运算符的语法如下：

```
value << shift
```

其中，value 是要被操作的整数值，shift 是要移动的位数。例如，下面的代码将值 13 的所有位都向左移 3 位：

```
13 << 3
```

腾出的位置用 0 填充，超出边界的位被丢弃（参见图 E.1）。

由于每个位都表示右边一位的 2 倍（参见附录 A），所以左移一位相当于乘以 2。同样，左移 2 位相当于乘以 $2^2$，左移 n 位相当于乘以 $2^n$。因此，13<<3 的值为 $13 \times 2^3$，即 104。

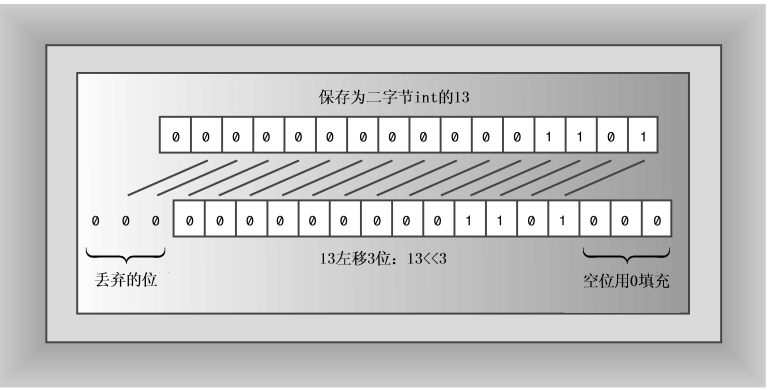



图 E.1　左移运算符

左移运算符提供了通常可以在汇编语言中找到的功能。不过，左移运算符在汇编语言中会直接修改寄存器的内容，而 C++左移运算符生成一个新值，而不修改原来的值。例如，请看下面的代码：

```
int x = 20;
int y = x << 3;
```

上述代码不会修改 x 的值。表达式 x<<3 使用 x 的值来生成一个新值，就像 x+3 会生成一个新值，而不会修改 x 一样。

如果要用左移运算符来修改变量的值，则还必须使用赋值运算符。可以使用常规的赋值运算符或<<=运算符（该运算符将移动与赋值结合在一起）：

```
= x << 4; // regular assignment
y <<= 2; // shift and assign
```

正如所期望的，右移运算符（>>）将位向右移，其语法如下：

```
value >> shift
```

其中，value 是要移动的整数值，shift 是要移动的位数。例如，下面的代码将值 17 中所有的位向右移两位：

```
17 >> 2
```

对于无符号整数，腾出的位置用 0 填充，超过边界的位被删除。对于有符号整数，腾出的位置可能用 0 填充，也可能用原来最左边的位填充，这取决于 C++实现（图 E.2 是一个用 0 填充的例子）。

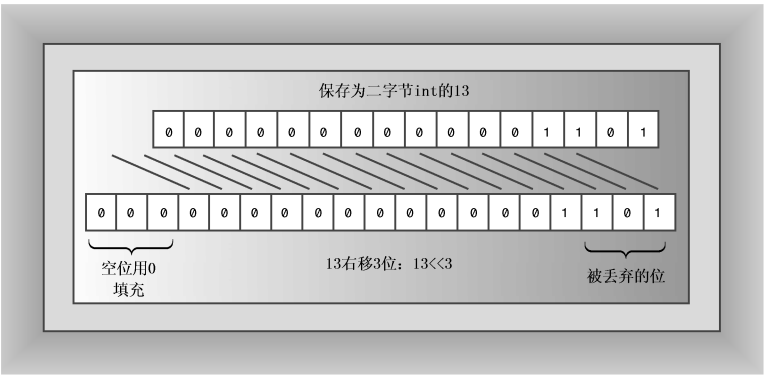



图 E.2　右移运算符

向右移动一位相当于除以 2。向右移动 n 位相当于除以 $2^n$。

C++还定义了一个“右移并赋值”运算符，如果要用移动后的值替换变量的值，可以这样做：

```
int q = 43;
q >>= 2;              // replace 43 by 43 >> 2, or 10
```

在有些系统上，使用左移运算符（右移运算符）实现将整数乘（除）以 2 的速度比使用乘（除）法运算符更快，但由于编译器在优化代码方面越来越好，因此这种差异正在减小。

### E.1.2　逻辑按位运算符

逻辑按位运算符类似于常规的逻辑运算符，只是它们用于值的每一位，而不是整个值。例如，请看常规的非运算符（!）和位非（或求反）运算符（～）。!运算符将 true（或非零值）转换为 false，将 false 值转换为 true。～运算符将每一位转换为它的反面（1 转换为 0，0 转换为 1）。例如，对于 unsigned char 值 3：

```
unsigned char x = 3;
```

表达式!x 的值为 0。要知道~x 的值，先把它写成二进制形式：00000011。然后将每个 0 转换为 1，将每个 1 转换为 0。这样将得到值 11111100，在十进制中，为 252（图 E.3 是一个 16 位的例子）。新值是原值的补值。


图 E.3　按位非运算符

按位运算符 OR（|）对两个整数值进行操作，生成一个新的整数值。如果被操作的两个值的对应位至少有一个为 1，则新值中相应位为 1，否则为 0（参见图 E.4）。

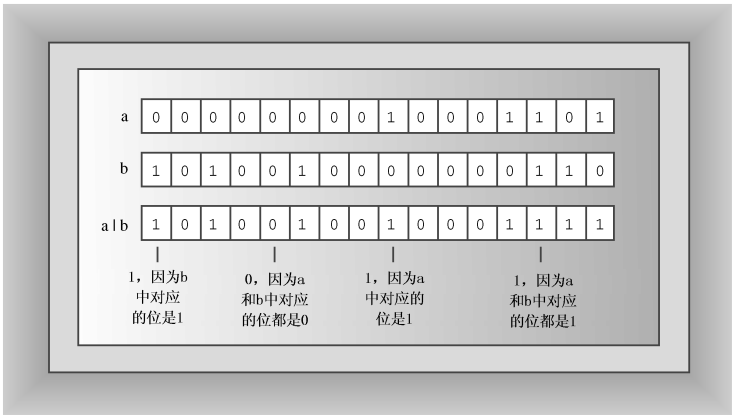



图 E.4 按位运算符 OR

表 E.1 对 | 运算符的操作方式进行了总结。

**表 E.1 b1|b2 的值**

| 位 值 | b1 = 0 | b1 = 1 |
|---|---|---|
| b2 = 0 | 0 | 1 |
| b2 = 1 | 1 | 1 |

运算符| =组合了按位运算符 OR 与赋值运算符的功能：

```
a |= b; // set a to a | b
```

按位运算符 XOR（^）将两个整数值结合起来，生成一个新的整数值。如果原始值中对应的位有一个（而不是两个）为 1，则新值中相应位为 1；如果对应的位都为 0 或 1，则新值中相应位为 0（参见图 E.5）。

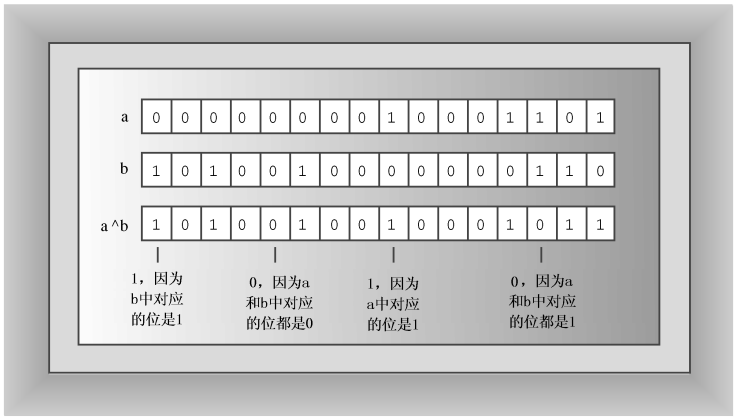



图 E.5 按位运算符 XOR

表 E.2 总结了^运算符的结合方式。

**表 E.2 b1^b2 的值**

| 位 值 | b1 = 0 | b1 = 1 |
|---|---|---|
| b2 = 0 | 0 | 1 |
| b2 = 1 | 1 | 0 |

^ =运算符结合了按位运算符 XOR 和赋值运算符的功能：

```
a ^= b; // set a to a ^ b
```

按位运算符 AND（&）将两个整数结合起来，生成一个新的整数值。如果原始值中对应位都为 1，则新值中相应位为 1，否则为 0（参见图 E.6）。

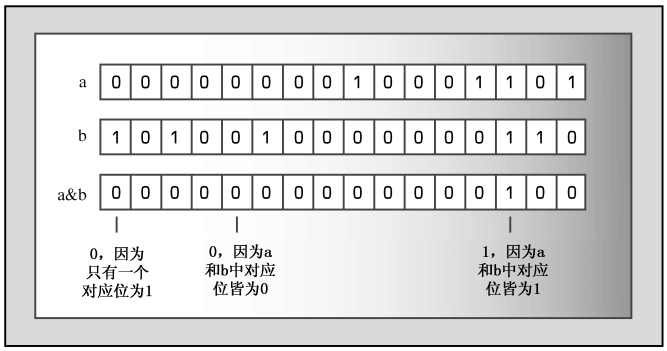



图 E.6　按位运算符 AND

表 E.3 总结了&运算符是如何运算的。

**表 E.3　　b1&b2 的值**

| 位　　值 | b1 = 0 | b2 = 1 |
|---|---|---|
| b2 = 0 | 0 | 0 |
| b2 = 1 | 0 | 1 |

& =运算符结合了按位运算符 AND 和赋值运算符的功能：

```
a &= b; // set a to a & b
```

### E.1.3　按位运算符的替代表示

对于几种按位运算符，C++提供了替代表示，如表 E.4 所示。它们适用于字符集中不包含传统按位运算符的区域。

**表 E.4　　按位运算符的替代表示**

| 标 准 表 示 | 替 代 表 示 |
|---|---|
| & | bitand |
| & = | and_eq |
| \| | bitor |
| \| = | or_eq |
| ～ | compl |
| ^ | xor |
| ^ = | xor_eq |

这些替代表示让您能够编写下面这样的语句：

```
b = compl a bitand b; // same as b = ~a & b;
c = a xor b;          // same as c = a ^ c;
```

### E.1.4　几种常用的按位运算符技术

控制硬件时，常涉及打开/关闭特定的位或查看它们的状态。按位运算符提供了执行这种操作的途径。下面简要地介绍一下这些方法。

在下面的示例中，lottabits 表示一个值，bit 表示特定位的值。位从右到左进行编号，从 0 开始，因此，第 n 位的值为 $2^n$。例如，只有第 3 位为 1 的整数的值为 $2^3$（8）。一般来说，正如附录 A 介绍的，各个位都对应于 2 的幂。因此我们使用术语位（bit）表示 2 的幂；这对应于特定位为 1，其他所有位都为 0 的情况。

1. 打开位

下面两项操作打开 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits | bit;
lottabits |= bit;
```

它们都将对应的位设置为 1，而不管这一位以前的值是多少。这是因为对 1 和 1 或者 0 和 1 执行 OR 操作时，都将得到 1。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 做 OR 操作将得到 0，对 1 和 0 做 OR 操作将生成 1。

2. 切换位

下面两项操作切换 lottabits 中对应于 bit 表示的位。也就是说，如果位是关闭的，则将被打开；如果位是打开的，将被关闭：

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

对 0 和 1 执行 XOR 操作的结果为 1，因此将关闭已打开的位；对 1 和 1 执行 XOR 操作的结果为 0，因此将打开已关闭的位。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 执行 XOR 操作的结果为 0，对 1 和 0 执行 XOR 操作的结果为 1。

3. 关闭位

下面的操作将关闭 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits & ~bit;
```

该语句关闭相应的位，而不管它以前的状态如何。首先，运算符～bit 将原来为 1 的位设置为 0，原来为 0 的位设置为 1。对 0 和任意值执行 AND 操作都将得到 0，因此关闭相应的位。lottabits 中其他所有位都保持不变，这是因为对 1 和任意值执行 AND 操作时，该位的值将保持不变。

下面是一种更简洁的方法：

```
lottabits &= ~bit;
```

4. 测试位的值

如果要确定 lottabits 中对应于 bit 的位是否为 1，则下面的测试不一定管用：

```
if (lottabits == bit) // no good
```

这是因为即使 lottabits 中对应的位为 1，而其他位也可能为 1。仅当对应的位为 1，而其他位皆为 0 时，上述等式才为 true。因此修补的方式是，首先对 lottabits 和 bit 执行 AND 操作，这样生成的值的对应位保持不变，因为对 1 和任何值执行 AND 操作都将保持该值不变；而其他位都为 0，因为对 0 和任何值执行 AND 操作的结果都为 0。正确的测试如下：

```
if (lottabits & bit == bit) // testing a bit
```

实际应用中，程序员常将上述测试简化为：

```
if (lottabits & bit)        // testing a bit
```

因为 bit 中有一位为 1，而其他位都为 0，因此 lottabits & bit 的结果要么为 0（测试结果为 false），要么为 bit（非零值，测试结果为 true）。

## E.2 成员解除引用运算符

C++允许定义指向类成员的指针，对这种指针进行声明或解除引用时，需要使用一种特殊的表示法。为说明需要使用的特殊表示法，先来看一个样本类：

```
class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};
```

如果没有具体的对象，则 inches 成员只是一个标签。也就是说，这个类将 inches 定义为一个成员标识符，但要为它分配内存，必须声明这个类的一个对象：

```
Example ob; // now ob.inches exists
```

因此，可以结合使用标识符 inches 和特定的对象，来引用实际的内存单元（对于成员函数，可以省略对象名，但对象被认为是指针指向的对象）。

C++允许这样定义一个指向标识符 inches 的成员指针：

```
int Example::*pt = &Example::inches;
```

这种指针与常规指针有所差别。常规指针指向特定的内存单元，而 pt 指针并不指向特定的内存单元，因为

声明中没有指出具体的对象。指针 pt 指的是 inches 成员在任意 Example 对象中的位置。和标识符 inches 一样，pt 被设计为与对象标识符一起使用。实际上，表达式*pt 对标识符 inches 的角色做了假设，因此，可以使用对象标识符来指定要访问的对象，使用 pt 指针来指定该对象的 inches 成员。例如，类方法可以使用下面的代码：

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl; // display inches member of ob1
cout << ob2.*pt << endl; // display inches member of ob2
cout << po->*pt << endl; // display inches member of *po
```

其中，*和->*都是成员解除引用运算符（member dereferencing operator）。声明对象（如 obl）后，ob1.*pi 指的将是 ob1 对象的 inches 成员。同样，pq->*pt 指的是 pq 指向的对象的 inches 成员。

改变上述示例中使用的对象，将改变使用的 inches 成员。不过也可以修改 pt 指针本身。由于 feet 的类型与 inches 相同，因此可以将 pt 重新设置为指向 feet 成员（而不指向 inches 成员），这样 ob1.*pt 将是 ob1 的 feet 成员：

```
pt = &Example::feet;     // reset pt
cout << ob1.*pt << endl; // display feet member of ob1
```

实际上，*pt 相当于一个成员名，因此可用于标识（相同类型的）其他成员。

也可以使用成员指针来标识成员函数，其语法稍微复杂点。对于不带任何参数、返回值为 void 的函数，声明一个指向该函数的指针的代码如下：

```
void (*pf)(); // pf points to a function
```

声明指向成员函数的指针时，必须指出该函数所属的类。例如，下面的代码声明了一个指向 Example 类方法的指针：

```
void (Example::*pf)() const; // pf points to an Example member function
```

这表明 pf 可用于可使用 Example 方法的地方。注意，Example::*pf 必须放在括号中，可以将特定成员函数的地址赋给该指针：

```
pf = &Example::show_inches;
```

注意，和普通函数指针的赋值情况不同，这里必须使用地址运算符。完成赋值操作后，便可以使用一个对象来调用该成员函数：

```
Example ob3(20);
(ob3.*pf)(); // invoke show_inches() using the ob3 object
```

必须将 ob3.*pf 放在括号中，以明确地指出，该表达式表示的是一个函数名。

由于 show_feet()的原型与 show_inches()相同，因此也可以使用 pf 来访问 show_feet()方法：

```
pf = &Example::show_feet;
(ob3.*pf)();       // apply show_feet() to the ob3 object
```

程序清单 E.1 中的类定义包含一个 use_ptr()方法，该方法使用成员指针来访问 Example 类的数据成员和函数成员。

**程序清单 E.1　memb_pt.cpp**

```
// memb_pt.cpp -- dereferencing pointers to class members
#include <iostream>
using namespace std;

class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};

Example::Example()
{
    feet = 0;
    inches = 0;
```

```
}

Example::Example(int ft)
{
    feet = ft;
    inches = 12 * feet;
}

Example::~Example()
{
}
void Example::show_in() const
{
    cout << inches << " inches\n";
}

void Example::show_ft() const
{
    cout << feet << " feet\n";
}

void Example::use_ptr() const
{
    Example yard(3);
    int Example::*pt;
    pt = &Example::inches;
    cout << "Set pt to &Example::inches:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    pt = &Example::feet;
    cout << "Set pt to &Example::feet:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    void (Example::*pf)() const;
    pf = &Example::show_in;
    cout << "Set pf to &Example::show_in:\n";
    cout << "Using (this->*pf)(): ";
    (this->*pf)();
    cout << "Using (yard.*pf)(): ";
    (yard.*pf)();
}

int main()
{
    Example car(15);
    Example van(20);
    Example garage;

    cout << "car.use_ptr() output:\n";
    car.use_ptr();
    cout << "\nvan.use_ptr() output:\n";
    van.use_ptr();

    return 0;
}
```

下面是程序清单 E.1 中程序的运行情况：

```
car.use_ptr() output:
Set pt to &Example::inches:
this->pt: 180
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 15
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 180 inches
Using (yard.*pf)(): 36 inches

van.use_ptr() output:
Set pt to &Example::inches:
this->pt: 240
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 20
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 240 inches
Using (yard.*pf)(): 36 inches
```

这个例子在编译期间给指针赋值。在更复杂的类中，可以使用指向数据成员和方法的成员指针，以便在运行阶段确定与指针关联的成员。

## E.3　alignof（C++11）

计算机系统可能限制数据在内存中的存储方式。例如，一个系统可能要求 double 值存储在编号为偶数的内存单元中，而另一个系统可能要求其起始地址为 8 个整数倍。运算符 alignof 将类型作为参数，并返回一个整数，指出要求的对齐方式。例如，对齐要求可能决定结构中信息的组织方式，如程序清单 E.2 所示。

**程序清单 E.2　align.cpp**

```
// align.cpp -- checking alignment
#include <iostream>
using namespace std;
struct things1
{
    char ch;
    int a;
    double x;
};
struct things2
{
    int a;
    double x;
    char ch;
};

int main()
{
    things1 th1;
    things2 th2;
    cout << "char alignment: " << alignof(char) << endl;
    cout << "int alignment: " << alignof(int) << endl;
    cout << "double alignment: " << alignof(double) << endl;
    cout << "things1 alignment: " << alignof(things1) << endl;
    cout << "things2 alignment: " << alignof(things2) << endl;
    cout << "things1 size: " << sizeof(things1) << endl;
    cout << "things2 size: " << sizeof(things2) << endl;
    return 0;
}
```

下面是该程序在一个系统中的输出：

```
char alignment: 1
int alignment: 4
double alignment: 8
things1 alignment: 8
things2 alignment: 8
things1 size: 16
things2 size: 24
```

两个结构的对齐要求都是 8。这意味着结构长度将是 8 的整数倍，这样创建结构数组时，每个元素的起始位置都是 8 的整数倍。在程序清单 E.2 中，每个结构的所有成员只占用 13 位，但结构要求占用的位数为 8 的整数倍，这意味着需要填充一些位。在每个结构中，double 成员的对齐要求为 8 的整数倍，但在结构 thing1 和 thing2 中，成员的排列顺序不同，这导致 thing2 需要更多的内部填充，以便其边界处于正确的位置。

## E.4　noexcept（C++11）

关键字 noexcept 用于指出函数不会引发异常。它也可用作运算符，判断操作数（表达式）是否可能引发异常；如果操作数可能引发异常，则返回 false，否则返回 true。例如，请看下面的声明：

```
int hilt(int);
int halt(int) noexcept;
```

表达式 noexcept(hilt) 的结果为 false，因为 hilt() 的声明未保证不会引发异常，但 noexcept(halt) 的结果为 true。

# 附录 F　模板类 string

本附录的技术性较强，但如果您只想了解模板类 string 的功能，可以将重点放在对各种 string 类方法的描述上。

string 类是基于下述模板定义的：

```
template<class charT, class traits = char_traits<charT>,
                 class Allocator = allocator<charT> >
class basic_string {...};
```

其中，chatT 是存储在字符串中的类型；traits 参数是一个类，它定义了类型要被表示为字符串时，所必须具备的特征。例如，它必须有 length()方法，该方法返回被表示为 charT 数组的字符串的长度。这种数组结尾用 charT(0)值（广义的空值字符）表示。（表达式 charT(0)将 0 转换为 charT 类型。它可以像类型为 char 时那样为零，也可以是 charT 的一个构造函数创建的对象）。这个类还包含用于对值进行比较等操作的方法。Allocator 参数是用于处理字符串内存分配的类。默认的 allocator<char>模板按标准方式使用 new 和 delete。

有 4 种预定义的具体化：

```
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

上述具体化又使用下面的具体化：

```
char_traits<char>
allocator<char>
char_traits<char16_t>
allocator<char_16>
char_traits<char_32>
allocator<char_32>
char_traits<wchar_t>
allocator<wchar_t>
```

除 char 和 wchar_t 外，还可以通过定义 traits 类和使用 basic_string 模板来为其他一些类型创建一个 string 类。

## F.1　13 种类型和一个常量

模板 basic_string 定义了几种类型，供以后定义方法时使用：

```
typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

traits 是对应于特定类型（如 char_traits<char>）的模板参数；traits_type 将成为该特定类型的 typedef。下述表示法意味着 char_type 是 traits 表示的类中定义的一个类型名：

```
typedef typename traits::char_type value_type;
```

关键字 typename 告诉编译器，表达式 trait::char_type 是一种类型。例如，对于 string 具体化，value_type 为 char。

size_type 与 size_of 的用法相似，只是它根据存储的类型返回字符串的长度。对于 string 具体化，将根据 char 返回字符串的长度，在这种情况下，size_type 与 size_of 等效。size_type 是一种无符号类型。

differrence_type 用于度量字符串中两个元素之间的距离（单位为元素的长度）。通常，它是底层类型 size_type 有符号版本。

对于 char 具体化来说，pointer 的类型为 char *，而 reference 的类型为 char &类型。然而，如果要为自己设计的类型创建具体化，则这些类型（pointer 和 reference）可以指向类，与基本指针和引用有相同的特征。

为将标准模板库（STL）算法用于字符串，该模板定义了一些迭代器类型：

```
typedef (models random access iterator)          iterator;
typedef (models random access iterator)          const_iterator;
typedef std::reverse_iterator<iterator>          reverse_iterator;
typedef std::reverse_iterator<const_iterator>    const_reverse_iterator;
```

该模板还定义了一个静态常量：

```
static const size_type npos = -1;
```

由于 size_type 是无符号的，因此将-1 赋给 npos 相当于将最大的无符号值赋给它，这个值比可能的最大数组索引大 1。

## F.2　数据信息、构造函数及其他

可以根据其效果来描述构造函数。由于类的私有部分可能依赖于实现，因此可根据公用接口中可用的数据来描述这些效果。表 F.1 列出了一些方法，它们的返回值可用来描述构造函数和其他方法的效果。注意，其中的大部分术语来自 STL。

**表 F.1　一些 string 数据方法**

| 方　法 | 返　回　值 |
|---|---|
| begin() | 指向字符串第一个字符的迭代器 |
| cbegin() | 一个 const_iterator，指向字符串中的第一个字符（C++11） |
| end() | 为超尾值的迭代器 |
| cend() | 为超尾值的 const_iterator（C++11） |
| rbegin() | 为超尾值的反转迭代器 |
| crbegin() | 为超尾值的反转 const_iterator（C++11） |
| rend() | 指向第一个字符的反转迭代器 |
| crend() | 指向第一个字符的反转 const_iterator（C++11） |
| size() | 字符串中的元素数，等于 begin()到 end()之间的距离 |
| length() | 与 size()相同 |
| capacity() | 给字符串分配的元素数。这可能大于实际的字符数，capacity() – size()的值表示在字符串末尾附加多少字符后需要分配更多的内存 |
| max_size() | 字符串的最大长度 |
| data() | 一个指向数组第一个元素的 const charT*指针，其第一个 size()元素等于*this 控制的字符串中对应的元素，其下一个元素为 charT 类型的 charT(0)字符（字符串末尾标记）。当 string 对象本身被修改后，该指针可能无效 |
| c_str() | 一个指向数组第一个元素的 const charT*指针，其第一个 size()元素等于*this 控制的字符串中对应的元素，其下一个元素是 charT 类型的 charT（0）字符（字符串尾标识）。当 string 对象本身被修改后，该指针可能无效 |
| get_allocator() | 用于为字符串 object 分配内存的 allocator 对象的副本 |

请注意 begin()、rend()、data()和 c_str()之间的差别。它们都与字符串的第一个字符相关，但相关的方式不同。begin()和 rend()方法返回一个迭代器，正如第 16 章讨论的，这是一种广义指针。具体地说，begin()返回一个正向迭代器模型，而 rend()返回反转迭代器的一个副本。这两种方法都引用了 string 对象管理的字符串（由于 string 类使用动态内存分配，因此实际的 string 内容不一定位于对象中，因此，我们使用术语“管理”来描述对象和字符串之间的关系）。可以将返回迭代器的方法用于基于迭代器的 STL 算法中。例如，可以使用 STL reverse()函数来反转字符串的内容：

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

而 data()和 c_str()方法返回常规指针。另外，返回的指针将指向存储字符串字符的数组的第一个元素。

该数组可能（但不一定）是 string 对象管理的字符串的副本（string 对象采用的内部表示可以是数组，但不一定非得是数组）。由于返回的指针可能指向原始数据，而原始数据是 const，因此不能用它们来修改数据。另外，当字符串被修改后，将不能保证这些指针是有效的，这表明它们可能指向原始数据。data()和 c_str()的区别在于，c_str()指向的数组以空值字符（或与之等价的其他字符）结束，而 data()只是确保实际的字符串字符是存在的。因此，c_str()方法期望接受一个 C-风格字符串参数：

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

同样，data()和 size()可用作这种函数的参数，即接受指向数组元素的指针和表示要处理的元素数目的值：

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

C++实现可能将 string 对象的字符串表示为动态分配的 C-风格字符串，并使用 char*指针来实现正向迭代器。在这种情况下，实现可能让 begin()、data()和 c_str()都返回同样的指针，但返回指向 3 个不同的数据对象的引用也是合法的（虽然更复杂）。

在 C++11 中，模板类 basic_string 有 11 个构造函数（在 C++98 中只有 6 个）和一个析构函数：

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const basic_string& str);
basic_string(const basic_string& str, const Allocator&);
basic_string(const basic_string& str, size_type pos,
    size_type n = npos, const Allocator& a = Allocator());
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string&& str, const Allocator&);
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
    const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
~basic_string();
```

有些新增的构造函数以不同的方式处理参数。例如，C++98 包含如下复制构造函数：

```
basic_string(const basic_string& str, size_type pos = 0,
    size_type n = npos, const Allocator& a = Allocator());
```

而 C++11 用三个构造函数取代了它——上述列表中的第 2～4 个，这提高了编码效率。真正新增的只有移动构造函数（使用右值引用的构造函数，这在第 18 章讨论过）以及使用 initializer_list 参数的构造函数。

注意到大多数构造函数构造函数都有一个下面这样的参数：

```
const Allocator& a = Allocator()
```

Allocator 是用于管理内存的 allocator 类的模板参数名；Allocator()是这个类的默认构造函数。因此，在默认情况下，构造函数将使用 allocator 对象的默认版本，但它们使得能够选择使用 allocator 对象的其他版本。下面分别介绍这些构造函数。

### F.2.1 默认构造函数

默认构造函数的原型如下：

```
explicit basic_string(const Allocator& a = Allocator());
```

通常，接受 allocator 类的默认参数，并使用该构造函数来创建空字符串：

```
string bean;
wstring theory;
```

调用该默认构造函数后，将存在下面的关系：

- data()方法返回一个非空指针，可以将该指针加上 0；
- size()方法返回 0；
- capacity()的返回值不确定。

将 data()返回的值赋给指针 str 后，第一个条件意味着 str + 0 是有效的。

### F.2.2 使用 C-风格字符串的构造函数

使用 C-风格字符串的构造函数让您能够将 string 对象初始化为一个 C-风格字符串；从更普遍的意义上看，它使得能够将 charT 具体化初始化为一个 charT 数组：

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

为确定要复制的字符数，该构造函数将 traits::length()方法用于 s 指向的数组（s 不能为空指针）。例如，下面的语句使用指定的字符串来初始化 toast 对象：

```
string toast("Here's looking at you, kid.");
```

char 类型的 traits::length()方法将使用空值字符来确定要复制多少个字符。

该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指针，该指针指向数组 s 的一个副本的第一个元素；
- size()方法返回的值等于 trains::length()的值；
- capacity()方法返回一个至少等于 size()的值。

### F.2.3　使用部分 C-风格字符串的构造函数

使用部分 C-风格字符串的构造函数让您能够使用 C-风格字符串的一部分来初始化 string 对象；从更广泛的意义上说，该构造函数使得能够使用 charT 数组的一部分来初始化 charT 具体化：

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

该构造函数将 s 指向的数组中的 n 个字符复制到构造的对象中。请注意，如果 s 包含的字符数少于 n，则复制过程将不会停止。如果 n 大于 s 的长度，该构造函数将把字符串后面的内存内容解释为 charT 类型的数据。

该构造函数要求 s 不能是空值指针，同时 n<npos（npos 是一个静态类常量，它是字符串可能包含的最大元素数目）。如果 n 等于 npos，该构造函数将引发一个 out_of_range 异常（由于 n 的类型为 size_type，而 npos 是 size_type 的最大值，因此 n 不能大于 npos）；否则，在该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指针，该指针指向数组 s 的副本的第一个元素；
- size()方法返回 n；
- capacity()方法返回一个至少等于 size()的值。

### F.2.4　使用左值引用的构造函数

复制构造函数类似于下面这样：

```
basic_string(const basic_string& str);
```

它使用一个 string 参数初始化一个新的 string 对象：

```
string mel("I'm ok!");
string ida(mel);
```

其中，ida 将是 mel 管理的字符串副本。

下一个构造函数要求您指定一个分配器：

```
basic_string(const basic_string& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data()方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data()指向的；
- size()方法返回 str.size()的值；
- capacity()方法返回一个至少等于 size()的值。

再下一个构造函数让您能够指定多项内容：

```
basic_string(const basic_string& str, size_type pos, size_type n = npos,
    const Allocator& a = Allocator());
```

第二个参数（pos）指定了源字符串中的位置，将从这个位置开始进行复制：

```
string att("Telephone home.");
string et(att, 4);
```

位置编号从 0 开始，因此，位置 4 是字符 p。所以，et 被初始化为“phone home”。

第 3 个参数 n 是可选的，它指定要复制的最大字符数目，因此下面的语句将 pt 初始化为字符串“phone”：

```
string att("Telephone home.");
string pt(att, 4, 5);
```

然而，该构造函数不能跨越源字符串的结尾，例如，下面的语句将在复制句点后停止：

```
string pt(att, 4, 200)
```

因此，该构造函数实际复制的字符数量等于 n 和 str.size()-pos 中较小的一个。

该构造函数要求 pos 不大于 str.size()，也就是说，被复制的初始位置必须位于源字符串中。如果情况并

非如此，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，copy_len 将是 n 和 str.size()-pos 中较小的一个，并存在下面的关系：

- data()方法返回一个指向字符串的指针，该字符串包含 copy_len 个元素，这些元素是从 str 的 pos 位置开始复制而得到的；
- size()方法返回 copy_len；
- capacity()方法返回一个不小于 size()的值。

### F.2.5 使用右值引用的构造函数（C++11）

C++11 给 string 类添加了移动语义。正如第 18 章介绍的，这意味着添加一个移动构造函数，它使用右值引用而不是左值引用：

```
basic_string(basic_string&& str) noexcept;
```

在实参为临时对象时将调用这个构造函数：

```
string one("din");     // C-style string constructor
string two(one);       // copy constructor - one is an lvalue
string three(one+two); // move constructor, sum is an rvalue
```

正如第 18 章讨论的，three 将获取 operator + ()创建的对象的所有权，而不是将该对象复制给 three，再销毁原始对象。

第二个使用右值引用的构造函数让您能够指定分配器：

```
basic_string(const basic_string&& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data()方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data()指向的；
- size()方法返回 str.size()的值；
- capacity()方法返回一个至少等于 size()的值。

### F.2.6 使用一个字符的 n 个副本的构造函数

使用一个字符的 n 个副本的构造函数创建一个由 n 个 c 组成的 string 对象：

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

该构造函数要求 n<npos。如果 n 等于 npos，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指向字符串第一个元素的指针，该字符串由 n 个元素组成，其中每个元素的值都为 c；
- size()方法返回 n；
- capacity()方法返回不小于 size()的值。

### F.2.7 使用区间的构造函数

使用区间的构造函数使用一个用迭代器定义的、STL-风格的区间：

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
             const Allocator& a = Allocator());
```

begin 迭代器指向源字符串中要复制的第一个元素，end 指向要复制的最后一个元素的后面。

这种构造函数可用于数组、字符串或 STL 容器：

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
    input.push_back(ch);
string str_input(input.begin(), input.end());
```

在第一种用法中，InputIterator 的类型为 const char *；在第二种用法中，InputIterator 的类型为 vector<char>::iterator。

调用该构造函数后，将存在下面的关系：

- data()方法返回一个指向字符串的第一个元素的指针，该字符串是通过复制区间[begin，end）中的

元素得到的；

- size()方法返回 begin 到 end 之间的距离（度量距离时，使用的单位为对迭代器解除引用得到的数据类型的长度）；
- capacity()方法返回一个不小于 size()的值。

### F.2.8　使用初始化列表的构造函数（C++11）

这个构造函数接受一个 initializer_list<charT>参数：

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

可将一个用大括号括起的字符列表作为参数：

```
string slow({'s', 'n', 'a', 'i', 'l'});
```

这并非初始化 string 的最方便方式，但让 string 的接口类似于 STL 容器类。

initializer_list 类包含成员函数 begin()和 end()，调用该构造函数的影响与调用使用区间的构造函数相同：

```
basic_string(il.begin(), il.end(), a);
```

### F.2.9　内存杂记

有些方法用于处理内存，如清除内存的内容、调整字符串长度或容量。表 F.2 列出了一些与内存相关的方法。

**表 F.2　　一些与内存有关的方法**

| 方　法 | 作　用 |
|---|---|
| void resize(size_type n) | 如果 n>npos，将引发 out_of_range 异常；否则，将字符串的长度改为 n，如果 n<size()，则截短字符串，如果 n>size()，则使用 charT(0)中的字符填充字符串 |
| void resize(size_type n, charT c) | 如果 n>npos，将引发 out_of_range 异常；否则，将字符串长度改为 n，如果 n<size()，则截短字符串，如果 n>size()，则使用字符 c 填充字符串 |
| void reserve(size_type res_arg = 0) | 将 capacity()设置为大于或等于 res_arg。由于这将重新分配字符串，因此以前的引用、迭代器和指针将无效 |
| void shrink_to_fit() | 请求让 capacity()的值与 size()相同，这是 C++11 新增的 |
| void clear() noexcept | 删除字符串中所有的字符 |
| bool empty()const noexcept | 如果 size()==0，则返回 true |

## F.3　字符串存取

有 4 种方法可以访问各个字符，其中两种方法使用[ ]运算符，另外两种方法使用 at()方法：

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

第一个 operator[ ]()方法使得能够使用数组表示法来访问字符串的元素，可用于检索或更改值。第二个 operator[ ]()方法可用于 const 对象，但只能用于检索值：

```
string word("tack");
cout << word[0]; // display the t
word[3] = 't';    // overwrite the k with a t
const ward("garlic");
cout << ward[2]; // display the r
```

at()方法提供了相似的访问功能，只是索引是通过函数参数提供的：

```
string word("tack");
cout << word.at(0); // display the t
```

差别在于（除语法差别外）：at()方法执行边界检查，如果 pos>=size()，将引发 out_of_range 异常。pos 的类型为 size_type，是无符号的，因此 pos 的值不能为负；而 operator[ ]()方法不进行边界检查，因此，如果 pos>=size()，则其行为将是不确定的（如果 pos= =size()，const 版本将返回空值字符的等价物）。

因此，可以在安全性（使用 at()检测异常）和执行速度（使用数组表示）之间进行选择。

还有一个这样的函数，它返回原始字符串的子字符串：

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

它返回一个字符串——这是从 pos 开始，复制 n 个字符（或到字符串尾部）得到的。例如，下面的代码将 pet 初始化为“donkey”：

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

C++11 新增了如下四个存取方法：

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

其中 front()方法访问 string 的第一个元素，相当于 operator[] (0)；back()方法访问 string 的最后一个元素，相当于 operator[] (size() - 1)。

## F.4 基本赋值

在 C++11 中，有 5 个重载的赋值方法，在 C++98 的基础上增加了两个：

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(basic_string&& str) noexcept; // C++11
basic_string& operator=(initializer_list<charT>);      // C++11
```

第一个方法将一个 string 对象赋给另一个；第二个方法将 C-风格字符串赋给 string 对象；第三个方法将一个字符赋给 string 对象；第四个方法使用移动语义，将一个右值 string 对象赋给一个 string 对象；第五个方法让您能够使用初始化列表进行赋值。因此，下面的操作都是可能的：

```
string name("George Wash");
string pres, veep, source, join, awkward;
pres = name;
veep = "Road Runner";
source = 'X';
join = name + source; // now with move semantics!
awkward = {'C','l','o','u','s','e','a','u'};
```

## F.5 字符串搜索

string 类提供了 6 种搜索函数，其中每个函数都有 4 个原型。下面简要地介绍它们。

### F.5.1 find()系列

在 C++11 中，find()的原型如下：

```
size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const noexcept;
```

第一个返回 str 在调用对象中第一次出现时的起始位置。搜索从 pos 开始，如果没有找到子字符串，将返回 npos。

下面的代码在一个字符串中查找字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);           // sets loc1 to 1
size_type loc2 = longer.find(shorter, loc1 + 1); // sets loc2 to 16
```

由于第二条搜索语句从位置 2 开始（That 中的 a），因此它找到的第一个 hat 位于字符串尾部。要测试是否失败，可使用 string::npos 值：

```
if (loc1 == string::npos)
    cout << "Not found\n";
```

第二个方法完成同样的工作，但它使用字符数组而不是 string 对象作为子字符串：

```
size_type loc3 = longer.find("is"); //sets loc3 to 5
```

第三个方法完成相同的工作，但它只使用字符串 s 的前 n 个字符。这与使用 basic_string（const charT * s，size_type n）构造函数，然后将得到的对象用作第一种格式的 find()的 string 参数的效果完全相同。例如，下面的代码搜索子字符串“fun”：

```
size_type loc4 = longer.find("funds", 3); //sets loc4 to 10
```

第四个方法的功能与第一个相同，但它使用一个字符而不是 string 对象作为子字符串：

```
size_type loc5 = longer.find('a'); //sets loc5 to 2
```

### F.5.2　rfind()系列

rfind()方法的原型如下：

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;
```

这些方法与相应 find()方法的工作方式相似，但它们搜索字符串最后一次出现的位置，该位置位于 pos 之前（包括 pos）。如果没有找到，该方法将返回 npos。

下面的代码从字符串末尾开始查找子字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);           // sets loc1 to 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // sets loc2 to 1
```

### F.5.3　find_first_of()系列

find_first_of()方法的原型如下：

```
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find()方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符首次出现的位置。

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter); // sets loc1 to 10
size_type loc2 = longer.find_first_of("fat");   // sets loc2 to 2
```

在 longer 中，首次出现的 fluke 中的字符是 funny 中的 f，而首次出现的 fat 中的字符是 That 中的 a。

### F.5.4　find_last_of()系列

find_last_of()方法的原型如下：

```
size_type find_last_of (const basic_string& str,
                        size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 rfind()方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符出现的最后位置。

下面的代码在一个字符串中查找字符串“hat”和“any”中字母最后出现的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter); // sets loc1 to 18
size_type loc2 = longer.find_last_of("any");   // sets loc2 to 17
```

在 longer 中，最后出现的 hat 中的字符是 hat 中的 t，而最后出现的 any 中的字符是 hat 中的 a。

### F.5.5　find_first_not_of()系列

find_first_not_of()方法的原型如下：

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos,
                            size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find_first_of()方法的工作方式相似，但它们搜索第一个不位于子字符串中的字符。

下面的代码在字符串中查找第一个没有出现在“This”和“Thatch”中的字母：

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);  // sets loc1 to 2
size_type loc2 = longer.find_first_not_of("Thatch"); // sets loc2 to 4
```

在 longer 中，That 中的 a 是第一个在 This 中没有出现的字符，而字符串 longer 中的第一个空格是第一个没有在 Thatch 中出现的字符。

### F.5.6 find_last_not_of()系列

find_last_not_of()方法的原型如下：

```
size_type find_last_not_of (const basic_string& str,
                            size_type pos = npos) const noexcept;
size_type find_last_not_of (const charT* s, size_type pos,
                            size_type n) const;
size_type find_last_not_of (const charT* s,
                            size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 find_last_of()方法的工作方式相似，但它们搜索的是最后一个没有在子字符串中出现的字符。

下面的代码在字符串中查找最后一个没有出现在“That.”中的字符：

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);     // sets loc1 to 15
size_type loc2 = longer.find_last_not_of(shorter, 10); // sets loc2 to 10
```

在 longer 中，最后的空格是最后一个没有出现在 shorter 中的字符，而 longer 字符串中的 f 是搜索到位置 10 时，最后一个没有出现在 shorter 中的字符。

## F.6 比较方法和函数

string 类提供了用于比较 2 个字符串的方法和函数。下面是方法的原型：

```
int compare(const basic_string& str) const noexcept;

int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str,
size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1, const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2 ) const;
```

这些方法使用 traits::compare()方法，后者是为用于字符串的字符类型定义的。如果根据 traits::compare()提供的顺序，第一个字符串位于第二个字符串之前，则第一个方法将返回一个小于 0 的值；如果这两个字符串相同，则它将返回 0；如果第一个字符串位于第二个字符串的后面，则它将返回一个大于 0 的值。如果较长的字符串的前半部分与较短的字符串相同，则较短的字符串将位于较长的字符串之前。

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 is < 0
int a12 = s1.compare(s2); // a12 is > 0
```

第二个方法与第一个方法相似，但它进行比较时，只使用第一个字符串中从位置 pos1 开始的 n1 个字符。

下面的示例将字符串 s1 的前 4 个字符同字符串 s2 进行比较：

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 is 0
```

第三个方法与第一个方法相似，但它使用第一个字符串中从 pos1 位置开始的 n1 个字符和第二个字符串中从 pos2 位置开始的 n2 个字符进行比较。例如，下面的语句将对 stout 中的 out 和 about 中的 out 进行比较：

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3); // a3 is 0
```

第四个方法与第一个方法相似，但它将一个字符数组而不是 string 对象作为第二个字符串。

第五和六个方法与第三个方法相似，但将一个字符串数组而不是 string 对象作为第二个字符串。

非成员比较函数是重载的关系运算符：

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

每一个运算符都被重载，使之将 string 对象与 string 对象进行比较、将 string 对象与 C-风格字符串进行比较、将 C-风格字符串与 string 对象进行比较。它们都是根据 compare()方法定义的，因此提供了一种在表示方面更为方便的比较方式。

## F.7　字符串修改方法

string 类提供了多个用于修改字符串的方法，其中绝大多数都拥有大量的重载版本，因此可用于 string 对象、字符串数组、单个字符和迭代器区间。

### F.7.1　用于追加和相加的方法

可以使用重载的+ =运算符或 append()方法将一个字符串追加到另一个字符串的后面。如果得到的字符串长于最大字符串长度，将引发 length_error 异常。+=运算符使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面：

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

append()方法也使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面。此外，通过指定初始位置和追加的字符数，或者通过指定区间，还可以追加 string 对象的一部分。通过指定要使用字符串中的多少个字符，可以追加字符串的一部分。追加字符的版本使得能够指定要复制该字符的多少个实例。下面是各种 append()方法的原型：

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c); // append n copies of c
void push_back(charT c);                        // appends 1 copy of c
```

下面是几个示例：

```
string test("The");
test.append("ory"); // test is "Theory"
test.append(3,'!'); // test is "Theory!!!"
```

operator+()函数被重载，以便能够拼接字符串。该重载函数不修改字符串，而是创建一个新的字符串，该字符串是通过将第二个字符串追加到第一个字符串后面得到的。加法函数不是成员函数，它们使得能够将 string 对象和 string 对象、string 对象和字符串数组、字符串数组和 string 对象、string 对象和字符以及字符和 string 对象相加。下面是一些例子：

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce"; // st3 is "reduce"
string st4 = 't' + st2;   // st4 is "train"
string st5 = st1 + st2;   // st5 is "redrain"
```

### F.7.2　其他赋值方法

除了基本的赋值运算符外，string 类还提供了 assign()方法，该方法使得能够将整个字符串、字符串的一部分或由相同字符组成的字符序列赋给 string 对象。下面是各种 assign()方法的原型：

```
basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str) noexcept; // C++11
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
```

```
basic_string& assign(size_type n, charT c);    // assign n copies of c
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>); // C++11
```

下面是几个例子：

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5); // test is "et tu"
test.assign(6, '#");      // test is "######"
```

接受右值引用作为参数的 assign()方法是 C++11 新增的，它支持移动语义；另一个新增的 assign()方法让您能够将 initializer_list 赋给 string 对象。

### F.7.3 插入方法

insert()方法使得能够将 string 对象、字符串数组或几个字符插入到 string 对象中。这个方法与 append()方法相似，但它还接受另一个指定插入位置的参数，该参数可以是位置，也可以是迭代器。数据将被插入到插入点的前面。有几种方法返回一个指向得到的字符串的引用。如果 pos1 超过了目标字符串结尾，或者 pos2 超过了要插入的字符串结尾，该方法将引发 out_of_range 异常。如果得到的字符串长于最大长度，该方法将引发 length_error 异常。下面是各种 insert()方法的原型：

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>); // C++11
```

例如，下面的代码将字符串“former”字符串插入到“The banker.”中 b 的前面：

```
string st3("The banker.");
st3.insert(4, "former ");
```

而下面的代码将字符串“ waltzed”（不包括!，它是第 9 个字符）插入到“The former banker.”末尾的句号之前：

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

### F.7.4 清除方法

erase()方法从字符串中删除字符，其原型如下：

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator position);
iterator erase(const_iterator first, iterator last);
void pop_back();
```

第一种格式将从 pos 位置开始，删除 n 个字符或删除到字符串尾。第二种格式删除迭代器位置引用的字符，并返回指向下一个元素的迭代器；如果后面没有其他元素，则返回 end()。第三种格式删除区间[first，last）中的字符，即从 first（包括）到 last（不包括）之间的字符；它返回最后一个迭代器，该迭代器指向最后一个被删除的元素后面的一个元素。最后，方法 pop_back()删除字符串中的最后一个字符。

### F.7.5 替换方法

各种 replace()方法都指定了要替换的字符串部分和用于替换的内容。可以使用初始位置和字符数目或迭代器区间来指定要替换的部分。替换内容可以是 string 对象、字符串数组，也可以是特定字符的多个实例。对于用于替换的 string 对象和数组，可以通过指定特定部分（使用位置和计数或只使用计数）或迭代器区间做进一步的修改。下面是各种 replace()方法的原型：

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
```

```
                        const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                        const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2,
                        const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                        size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                         InputIterator j1, InputIterator j2);
basic_string& replace(const_iteraor i1, const_iteator i2,
                        initializer)list<charT> il);
```

下面是一个例子：

```
string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); // replace right with left
```

注意，您可以使用 find()来找出要在 replace()中使用的位置：

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
    s3.replace(pos, s1.size(), s2);
```

上述代码将 old 替换为 mature。

### F.7.6　其他修改方法：copy()和 swap()

copy()方法将 string 对象或其中的一部分复制到指定的字符串数组中：

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

其中，s 指向目标数组，n 是要复制的字符数，pos 指出从 string 对象的什么位置开始复制。复制将一直进行下去，直到复制了 n 个字符或到达 string 对象的最后一个字符。函数返回复制的字符数，该方法不追加空值字符，同时由程序员负责检查数组的长度是否足够存储复制的内容。

**警告：**copy()方法不追加空值字符，也不检查目标数组的长度是否足够。

swap()方法使用一个时间恒定的算法来交换两个 string 对象的内容：

```
void swap(basic_string& str);
```

## F.8　输出和输入

string 类重载了<<运算符来显示 string 对象，该运算符返回 istream 对象的引用，因此可以拼接输出：

```
string claim("The string class has many features.");
cout << claim << endl;
```

string 类重载了>>运算符，使得能够将输入读入到字符串中：

```
string who;
cin >> who;
```

到达文件尾、读取字符串允许的最大字符数或遇到空白字符后，输入将终止（空白的定义取决于字符集以及 charT 表示的类型）。

有两个 getline()函数，第一个的原型如下：

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
           basic_string<charT,traits,Allocator>& str, charT delim);
```

这个函数将输入流 is 中的字符读入到字符串 str 中，直到遇到定界字符 delim、到达文件尾或者到达字符串的最大长度。delim 字符将被读取（从输入流中删除），但不被存储。第二个版本没有第三个参数，同时使用换行符（或其广义形式），而不是 delim：

```
string str1, str2;
getline(cin, str1);      // read to end-of-line
getline(cin, str2, '.'); // read to period
```

# 附录 G　标准模板库方法和函数

标准模板库（STL）旨在提供通用算法的高效实现，它通过通用函数（可用于满足特定算法要求的任何容器）和方法（可用于特定容器类实例）来表达这些算法。本附录假设您对 STL 有一定的了解（如通过阅读第 16 章）。例如，本章假设您了解迭代器和构造函数。

## G.1　STL 和 C++11

C++11 对 C++语言做了大量修改，本书无法全面介绍，同样 C++11 对 STL 也做了大量修改，本附录无法全面介绍。然而，可以对新增的内容做一总结。

C++11 给 STL 新增了多个元素。首先，它新增了多个容器；其次，给旧容器新增了多项功能；第三，在算法系列中新增了一些模板函数。本附录介绍了所有这些变化，对前两类变化有大致了解将很有帮助。

### G.1.1　新增的容器

C++11 新增了如下容器：array、forward_list、unordered_st 以及无序关联容器 unordered_multiset、unordered_map 和 unordered_multimap。

array 容器一旦声明，其长度就是固定的，它使用静态（栈）内存，而不是动态分配的内存。提供它旨在替代数组；array 受到的限制比 vector 多，但效率更高。

容器 list 是一种双向链表，除两端的节点外，每个节点都链接到它前面和后面的节点。forward_list 是一种单向链表，除最后一个节点外，每个节点都链接到下一个节点。相对于 list，它更紧凑，但受到的限制更多。

与 set 和其他关联容器一样，无序关联容器让您能够使用键快速检索数据，差别在于关联容器使用的底层数据结构为树，而无序关联容器使用的是哈希表。

### G.1.2　对 C++98 容器所做的修改

C++11 对容器类的方法做了三项主要修改。

首先，新增的右值引用使得能够给容器提供移动语义（参见第 18 章）。因此，STL 现在给容器提供了移动构造函数和移动赋值运算符，这些方法将右值引用作为参数。

其次，由于新增了模板类 initilizer_list（参见第 18 章），因此新增了将 initilizer_list 作为参数的构造函数和赋值运算符。这使得可以编写类似于下面的代码：

```
vector<int> vi{100, 99, 97, 98};
vi = {96, 99, 94, 95, 102};
```

第三，新增的可变参数模板（variadic template）和函数参数包（parameter pack）使得可以提供就地创建（emplacement）方法。这意味着什么呢？与移动语义一样，就地创建旨在提高效率。请看下面的代码段：

```
class Items
{
    double x;
    double y;
    int m;
public:
    Items();                                    // #1
    Items (double xx, double yy, int mm); // #2
    ...
};
...
vector<Items> vt(10);
```

```
...
vt.push_back(Items(8.2, 2.8, 3)); //
```

调用 insert()将导致内存分配函数在 vt 末尾创建一个默认 Items 对象。接下来，构造函数 Items()创建一个临时 Items 对象，该对象被复制到 vt 的开头，然后被删除。在 C++11 中，您可以这样做：

```
vi.emplace_back(8.2, 2.8, 3);
```

方法 emplace_back()是一个可变参数模板，将一个函数参数包作为参数：

```
template <class... Args> void emplace_back(Args&&... args);
```

上述三个实参（8.2、2.8 和 3）将被封装到参数 args 中。参数 args 被传递给内存分配函数，而内存分配函数将其展开，并使用接受三个参数的 Items 构造函数（#2），而不是默认构造函数（#1）。也就是说，它使用 Items(args…)，这里将展开为 Items(8.2, 2.8, 3)。因此，将在矢量中就地创建所需的对象，而不是创建一个临时对象，再将其复制到矢量中。

STL 在多个就地创建方法中使用了这种技术。

## G.2　大部分容器都有的成员

所有容器都定义了表 G.1 列出的类型。在这个表中，x 为容器类型，如 vector<int>；T 为存储在容器中的类型，如 int。表 G.1 中的示例阐明了含义

**表 G.1　　为所有容器定义的类型**

| 类　　型 | 值 |
|---|---|
| x::value-type | T，元素类型 |
| x::reference | T & |
| x::const_reference | const T & |
| x::iterator | 指向 T 的迭代器类型，行为与 T*相似 |
| x::const_iterator | 指向 const T 的迭代器类型，行为与 const T *相似 |
| x::different_type | 用于表示两个迭代器之间距离的符号整型，如两个指针的差 |
| x::size_type | 无符号整型 size_type 可以表示数据对象的长度、元素数目和下标 |

类定义使用 typedef 定义这些成员。可以使用这些类型来声明适当的变量。例如，下面的代码使用迂回的方式，将由 string 对象组成的矢量中的第一个“bonus”替换为“bogus”，以演示如何使用成员类型来声明变量。

```
using namespace std;
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
    input.push_back(temp);
vector<string>::iterator want=
    find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
    vector<string>::reference r = *want;
    r = "bogus";
}
```

上述代码使 r 成为一个指向（want 指向的）input 中元素的引用。同样，继续前面的例子，可以编写下面这样的代码：

```
vector<string>::value_type s1 = input[0]; // s1 is type string
vector<string>::reference s2 = input[1]; // s2 is type string &
```

这将导致 s1 为一个新 string 对象，它是 input[0]的拷贝；而 s2 为指向 input[1]的引用。在这个例子中，由于已经知道模板是基于 string 类型的，因此编写下面的等效代码将更简单：

```
string s1 = input[0];   // s1 is type string
string & s2 = input[1]; // s2 is type string &
```

然而，还可以在更通用的代码中使用表 G.1 中较精致（其中容器和元素的类型是通用的）的类型。例如，假设希望 min()函数将一个指向容器的引用作为参数，并返回容器中最小的项目。这假设为用于实例化模板的值类型定义了<运算符，而不想使用 STL min_element()算法，这种算法使用迭代器接口。由于参数可能是 vector<int>、list<strint>或 deque<double>，因此需要使用带模板参数（如 Bag）的模板来表示容器

（也就是说，Bag 是一个模板类型，可能被实例化为 vector<int>、list<string>或其他一些容器类型）。因此，函数的参数类型应为 const Bag & b。返回类型是什么呢？应为容器的值类型，即 Bag::value_type。然而，在这种情况下，Bag 只是一个模板参数，编译器无法知道 value_type 成员实际上是一种类型。但可以使用 typename 关键字来指出，类成员是 typedef：

```
vector<string>::value_type st; // vector<string> a defined class
typename Bag::value_type m;     // Bag an as yet undefined type
```

对于上述第一个定义，编译器能够访问 vector 模板定义，该定义指出，value_type 是一个 typedef；对于第二个定义，typename 关键字指出，无论 Bag 将会是什么，Bag::value-type 都将是类型的名称。这些考虑因素导致了下面的定义：

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
    typename Bag::const_iterator it;
    typename Bag::value_type m = *b.begin();
    for (it = b.begin(); it != b.end(); ++it)
        if (*it < m)
            m = *it;
    return m;
}
```

这样，便可以这样使用该模板函数：

```
vector<int> temperatures;
// input temperature values into the vector
int coldest = min(temperatures);
```

temperatures 参数将使得 Bag 被谓词为 vector<int>，而 typename Bag::value-type 被谓词为 vector <int>::value_type，进而为 int。

所有的容器都还可以包含表 G.2 列出的成员函数或操作。其中，X 是容器类型，如 vector<int>；而 T 是存储在容器中的类型，如 int。另外，a 和 b 是类型为 X 的值；u 是标识符；r 是类型为 X 的非 const 值；rv 是类型为 X 的非 const 右值，而移动操作是 C++11 新增的。

**表 G.2 为所有容器定义的操作**

| 操　作 | 描　述 |
|---|---|
| X u; | 创建一个名为 u 的空对象 |
| X() | 创建一个空对象 |
| X(a) | 创建对象 x 的拷贝 |
| X u(a) | u 是 a 的拷贝（复制构造函数） |
| X u = a; | u 是 a 的拷贝（复制构造函数） |
| r = a | r 等于 a 的值（复制赋值） |
| X u(rv) | u 等于 rv 的原始值（移动构造函数） |
| X u = rv | u 等于 rv 的原始值（移动构造函数） |
| a = rv | u 等于 rv 的原始值（移动赋值） |
| (&a)->~X() | 对 a 的每个元素执行析构函数 |
| begin() | 返回一个指向第一个元素的迭代器 |
| end() | 返回一个指向超尾的迭代器 |
| cbegin() | 返回一个指向第一个元素的 const 迭代器 |
| cend() | 返回一个指向超尾的 const 迭代器 |
| size() | 返回元素数目 |
| maxsize() | 返回容器的最大可能长度 |
| empty() | 如果容器为空，则返回 true |
| swap() | 交换两个容器的内容 |
| = = | 如果两个容器的长度相同、包含的元素相同且元素排列的顺序相同，则返回 true |
| ! = | a!=b 返回!(a= =b) |

使用双向或随机迭代器的容器（vector、list、deque、array、set 和 map）是可反转的，它们提供了表 G.3 所示的方法。

表 G.3　为可反转容器定义的类型和操作

| 操　作 | 描　述 |
|---|---|
| X::reverse_iterator | 指向类型 T 的反向迭代器 |
| X::const_reverse_iterator | 指向类型 T 的 const 反向迭代器 |
| a.rbegin() | 返回一个反向迭代器，指向 a 的超尾 |
| a.rend() | 返回一个指向 a 的开头的反向迭代器 |
| a.crbegin() | 返回一个 const 反向迭代器，指向 a 的超尾 |
| a.crend() | 返回一个指向 a 的开头的 const 反向迭代器 |

无序集合（set）和无序映射（map）无需支持表 G.4 所示的可选容器操作，但其他容器必须支持。

表 G.4　可选的容器操作

| 操　作 | 描　述 |
|---|---|
| < | 如果 a 按词典顺序排在 b 之前，则 a<b 返回 true |
| > | a>b 返回 b<a |
| <= | a<=b 返回!(a>b) |
| >= | a>=b 返回!(a<b) |

容器的>运算符假设已经为值类型定义了>运算符。词典比较是一种广义的按字母顺序排序，它逐元素地比较两个容器，直到两个容器中对应的元素相同时为止。在这种情况下，元素对的顺序将决定容器的顺序。例如，如果两个容器的前 10 个元素都相同，但第一个容器的第 11 个元素比第二个容器的第 11 个元素小，则第一个容器将排在第二个容器之前。如果两个容器中的元素一直相同，直到其中一个容器中的元素用完，则较短的容器将排在较长的容器之前。

## G.3　序列容器的其他成员

模板类 vector、forward_list、list、deque 和 array 都是序列容器，它们都前面列出的方法，但 forward_list 不是可反转的，不支持表 G.3 所示的方法。序列容器以线性顺序存储一组类型相同的值。如果序列包含的元素数是固定的，通常选择使用 array；否则，应首先考虑使用 vector，它让 array 的随机存取功能以及添加和删除元素的功能于一身。然而，如果经常需要在序列中间添加元素，应考虑使用 list 或 forward_list。如果添加和删除操作主要是在序列两端进行的，应考虑使用 deque。

array 对象的长度是固定的，因此无法使用众多序列方法。表 G.5 列出除 array 外的序列容器可用的其他方法（forward_list 的 resize()方法的定义稍有不同）。同样，其中 X 是容器类型，如 vector<int>；T 是存储在容器中的类型，如 int；a 是类型为 X 的值；t 是类型为 x::value_type 的左值或 const 右值；i 和 j 是输入迭代器；[i, j]是有效的区间；il 是类型为 initilizer_list<value_type>的对象；p 是指向 a 的有效 const 迭代器；q 是可解除引用的有效 const 迭代器；[q1, q2]是有效的 const 迭代器区间；n 是 x::size_type 类型的整数；Args 是模板参数包，而 args 是形式为 Args&&的函数参数包。

表 G.5　为序列容器定义的其他操作

| 操　作 | 描　述 |
|---|---|
| X(n, t) | 创建一个序列容器，它包含 t 的 n 个拷贝 |
| X a(n, t) | 创建一个名为 a 的序列容器，它包含 t 的 n 个拷贝 |
| X(i, j) | 使用区间[i, j] 内的值创建一个序列容器 |
| X a(i, j) | 使用区间[i, j) 内的值创建一个名为 a 的序列容器 |
| X(il) | 创建一个序列容器，并将其初始化为 il 的内容 |
| a = il; | 将 il 的值复制到 a 中 |
| a.emplace(p, args); | 在 p 前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数 |

续表

| 操 作 | 描 述 |
|---|---|
| a.insert(p, t) | 在 p 之前插入 t 的拷贝，并返回指向该拷贝的迭代器。T 的默认值为 T()，即在没有显式初始化时，用于 T 类型的值 |
| a.insert(p, rv) | 在 p 之前插入 rv 的拷贝，并返回指向该拷贝的迭代器；可能使用移动语义 |
| a.insert(p, n, t) | 在 p 之前插入 t 的 n 个拷贝 |
| a.insert(p, i, j) | 在 p 之前插入[i, j)区间内元素的拷贝 |
| a.insert(p, il) | 等价于 a.insert(p, il.begin(), il.end()) |
| a.resize(n) | 如果 n > a.size()，则在 a.end()之前插入 n - a.size()个元素；用于新元素的值为没有显式初始化时，用于 T 类型的值；如果 n < a.size()，则删除第 n 个元素之后的所有元素 |
| a.resize(n, t) | 如果 n > a.size()，则在 a.end()之前插入 t 的 n - a.size()个拷贝；如果 n<a.size()，则删除第 n 个元素之后的所有元素 |
| a.assign(i, j) | 使用区间[i，j)内的元素拷贝替换 a 当前的内容 |
| a.assign(n, t) | 使用 t 的 n 个拷贝替换 a 的当前内容。t 的默认值为 T()，即在没有显式初始化时，用于 T 类型的值 |
| a.assign(il) | 等价于 a.assign(il.begin(), il.end()) |
| a.erase(q) | 删除 q 指向的元素；返回一个指向 q 后面的元素的迭代器 |
| a.erase(q1, q2) | 删除区间[q1, q2]内的元素；返回一个迭代器，该迭代器指向 q2 原来指向的元素 |
| a.clear() | 与 erase(a.begin(), a.end()) 等效 |
| a.front() | 返回*a.begin()（第一个元素） |

表 G.6 列出了一些序列类（vector、forward_list、list 和 deque）都有的方法。

**表 G.6　　为某些序列定义的操作**

| 操 作 | 描 述 | 容 器 |
|---|---|---|
| a.back() | 返回*a.end()（最后一个元素） | vector、list、deque |
| a.push_back(t) | 将 t 插入到 a.end()前面 | vector、list、deque |
| a.push_back(rv) | 将 rv 插入到 a.end()前面；可能使用移动语义 | vector、list、deque |
| a.pop_back() | 删除最后一个元素 | vector、list、deque |
| a.emplace_back(args) | 追加一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数 | vector、list、deque |
| a.push_front(t) | 将 t 的拷贝插入到第一个元素前面 | forward_list、list、deque |
| a.push_front(rv) | 将 rv 的拷贝插入到第一个元素前面；可能使用移动语义 | forward_list、list、deque |
| a.emplace_front() | 在最前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数 | forward_list、list、deque |
| a.pop_front() | 删除第一个元素 | forward_list、list |
| a[n] | 返回*(a.begin()+ n) | vector、deque、array |
| a.at(n) | 返回*(a.begin()+ n)；如果 n>a.size，则引发 out_of_range 异常 | vector、deque、array |

模板 vector 还包含表 G.7 列出的方法。其中，a 是 vector 容器，n 是 x::size_type 型整数。

**表 G.7　　vector 的其他操作**

| 操 作 | 描 述 |
|---|---|
| a.capacity() | 返回在不要求重新分配内存的情况下，矢量能存储的元素总量 |
| a.reserve(n) | 提醒 a 对象：至少需要存储 n 个元素的内存。调用该方法后，容量至少为 n 个元素。如果 n 大于当前的容量，则需要重新分配内存。如果 n 大于 a.max_size()，该方法将引发 length_error 异常 |

模板 list 还包含表 G.8 列出的方法。其中，a 和 b 是 list 容器；T 是存储在链表中的类型，如 int；t 是类型为 T 的值；i 和 j 是输入迭代器；q2 和 p 是迭代器；q 和 q1 是可解除引用的迭代器；n 是 x::size_type 型整数。该表使用了标准的 STL 表示法[i, j)，这指的是从 i 到 j（不包括 j）的区间。

表 G.8　list 的其他操作

| 方　法 | 描　述 |
|---|---|
| a.splice(p, b) | 将链表 b 的内容移到链表 a 中，并将它们插在 p 之前 |
| a.splice(p, b, i) | 将 i 指向的链表 b 中的元素移到链表 a 的 p 位置之前 |
| a.splice(p, b, i, j) | 将链表 b 中[i，j)区间内的元素移到链表 a 的 p 位置之前 |
| a.remove(const T& t) | 删除链表 a 中值为 t 的所有元素 |
| a.remove_if(Predicate pred) | 如果 i 是指向链表 a 中元素的迭代器，则删除 pred(*i)为 true 的所有值（Predicate 是布尔值函数或函数对象，参见第 15 章） |
| a.unique() | 删除连续的相同元素组中除第一个元素之外的所有元素 |
| a.unique(BinaryPredicate bin_pred) | 删除连续的 bin_pred(*i, *(i - 1))为 true 的元素组中除第一个元素之外的所有元素（BinaryPredicate 是布尔值函数或函数对象，参见第 15 章） |
| a.merge(b) | 使用为值类型定义的<运算符，将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则 a 中的元素将放在前面。合并后，链表 b 为空 |
| a.merge(b, Compare comp) | 使用 comp 函数或函数对象将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则链表 a 中的元素将放在前面。合并后，链表 b 为空 |
| a.sort() | 使用<运算符对链表 a 进行排序 |
| a.sort(Compare comp) | 使用 comp 函数或函数对象对链表 a 进行排序 |
| a.reverse() | 将链表 a 中的元素顺序反转 |

forward_list 的操作与此类似，但由于模板类 forward_list 的迭代器不能后移，有些方法必须调整。因此，用 insert_after()、erase_after ()和 splice_after()替代了 insert()、erase()和 splice()，这些方法都对迭代器后面而不是前面的元素进行操作。

## G.4　set 和 map 的其他操作

关联容器（集合和映射是这种容器的模型）带有模板参数 Key 和 Compare，这两个参数分别表示用来对内容进行排序的键类型和用于对键值进行比较的函数对象（被称为比较对象）。对于 set 和 multiset 容器，存储的键就是存储的值，因此键类型与值类型相同。对于 map 和 multimap 容器，存储的值（模板参数 T）与键类型（模板参数 Key）相关联，值类型为 pair<const Key, T>。关联容器有其他成员来描述这些特性，如表 G.9 所示。

表 G.9　为关联容器定义的类型

| 类　型 | 值 |
|---|---|
| X::key_type | Key，键类型 |
| X::key_compare | Compare，默认为 less<key_type> |
| X::value_compare | 二元谓词类型，与 set 和 multiset 的 key_compare 相同，为 map 或 multimap 容器中的 pair<const Key, T>值提供了排序功能 |
| X::mapped_type | T，关联数据类型（仅限于 map 和 multimap） |

关联容器提供了表 G.10 列出的方法。通常，比较对象不要求键相同的值是相同的；等价键（equivalent key）意味着两个值（可能相同，也可能不同）的键相同。在该表中，X 为容器类，a 是类型为 X 的对象。如果 X 使用唯一键（即为 set 或 map），则 a_uniq 将是类型为 X 的对象。如果 x 使用多个键（即为 multiset 或 multimap），则 a_eq 将是类型为 X 的对象。和前面一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用的迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.10 为 set、multiset、map 和 multimap 定义的操作

| 操 作 | 描 述 |
|---|---|
| X(i, j, c) | 创建一个空容器，插入区间[i, j]中的元素，并将 c 用作比较对象 |
| X a(i, j, c) | 创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 c 用作比较对象 |
| X(i, j) | 创建一个空容器，插入区间[i, j]中的元素，并将 Compare()用作比较对象 |
| X a(i, j) | 创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 Compare()用作比较对象 |
| X(il); | 等价于 X(il.begin(), il.end()) |
| a = il | 将区间[il.begin(), il.end()]的内容赋给 a |
| a.key_comp() | 返回在构造 a 时使用的比较对象 |
| a.value_comp() | 返回一个 value_compare 对象 |
| a_uniq.insert(t) | 当且仅当 a 不包含具有相同键的值时，将 t 值插入到容器 a 中。该方法返回一个 pair<iterator, bool>值。如果进行了插入，则 bool 的值为 true，否则为 false。iterator 指向键与 t 相同的元素 |
| a_eq.insert(t) | 插入 t 并返回一个指向其位置的迭代器 |
| a.insert(p, t) | 将 p 作为 insert()开始搜索的位置，将 t 插入。如果 a 是键唯一的容器，则当且仅当 a 不包含拥有相同键的元素时，才插入；否则，将进行插入。无论是否进行了插入，该方法都将返回一个迭代器，该迭代器指向拥有相同键的位置 |
| a.insert(i, j) | 将区间[i, j]中的元素插入到 a 中 |
| a.insert(il) | 将 initializer_list il 中的元素插入到 a 中 |
| a_uniq.emplace(args) | 类似于 a_uniq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数 |
| a_eq.emplace(args) | 类似于 a_eq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数 |
| a.emplace_hint(args) | 类似于 a.insert(p, t)，但使用参数列表与参数包 args 的内容匹配的构造函数 |
| a.erase(k) | 删除 a 中键与 k 相同的所有元素，并返回删除的元素数目 |
| a.erase(q) | 删除 q 指向的元素 |
| a.erase(q1, q2) | 删除区间[q1, q2)中的元素 |
| a.clear() | 与 erase(a.begin(), a.end())等效 |
| a.find(k) | 返回一个迭代器，该迭代器指向键与 k 相同的元素；如果没有找到这样的元素，则返回 a.end() |
| a.count(k) | 返回键与 k 相同的元素的数量 |
| a.lower_bound(k) | 返回一个迭代器，该迭代器指向第一个键不小于 k 的元素 |
| a.upper_bound(k) | 返回一个迭代器，该迭代器指向第一个键大于 k 的元素 |
| a.equal_range(k) | 返回第一个成员为 a.lower_bound(k)，第二个成员为 a.upper_bound(k)的值对 |
| a.operator[ ](k) | 返回一个引用，该引用指向与键 k 关联的值（仅限于 map 容器） |

## G.4 无序关联容器（C++11）

前面说过，无序关联容器（unordered_set、unordered_multiset、unordered_map 和 unordered_multimap）使用键和哈希表，以便能够快速存取数据。下面简要地介绍这些概念。哈希函数（hash function）将键转换为索引值。例如，如果键为 string 对象，哈希函数可能将其中每个字符的数字编码相加，再计算结果除以 13 的余数，从而得到一个 0～12 的索引。而无序容器将使用 13 个桶（bucket）来存储 string，所有索引为 4 的 string 都将存储在第 4 个桶中。如果您要在容器中搜索键，将对键执行哈希函数，进而只在索引对应的桶中搜索。理想情况下，应有足够多的桶，每个桶只包含为数不多的 string。

C++11 库提供了模板 hash<key>，无序关联容器默认使用该模板。为各种整型、浮点型、指针以及一些模板类（如 string）定义了该模板的具体化。

表 G.11 列出了用于这些容器的类型。

无序关联容器的接口类似于关联容器。具体地说，表 G.10 也适用于无序关联容器，但存在如下例外：不需要方法 lower_bound()和 upper_bound()，构造函数 X(i, j, c) 亦如此。常规关联容器是经过排序的，这让它们能够使用表示“小于”概念的比较谓词。这种比较不适用于无序关联容器，因此它们使用基于概念“等于”的比较谓词。

表 G.11　为无序关联容器定义的类型

| 类　型 | 值 |
|---|---|
| X::key_type | Key，键类型 |
| X::key_equal | Pred，一个二元谓词，检查两个类型为 Key 的参数是否相等 |
| X::hasher | Hash，一个这样的二元函数对象，即如果 hf 的类型为 Hash，k 的类型为 Key，则 hf(k) 的类型为 std::size_t |
| X::local_iterator | 一个类型与 X::iterator 相同的迭代器，但只能用于一个桶 |
| X::const_local_iterator | 一个类型与 X::const_iterator 相同的迭代器，但只能用于一个桶 |
| X::mapped_type | T，关联数据类型（仅限于 map 和 multimap） |

除表 G.10 所示的方法外，无序关联容器还包含其他一些必不可少的方法，如表 G.12 所示。在该表中，X 为无序关联容器类，a 是类型为 X 的对象，b 可能是类型为 X 的常量对象，a_uniq 是类型为 unordered_set 或 unordered_map 的对象，a_eq 是类型为 unordered_multiset 或 unordered_multimap 的对象，hf 是类型为 hasher 的值，eq 是类型为 key_equal 的值，n 是类型为 size_type 的值，z 是类型为 float 的值。与以前一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.12　为无序关联容器定义的操作

| 操　作 | 描　述 |
|---|---|
| X(n, hf, eq) | 创建一个至少包含 n 个桶的空容器，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略了 eq，则将 key_equal()用作键值相等谓词；如果也省略了 hf，则将 hasher()用作哈希函数 |
| X a(n, hf, eq) | 创建一个名为 a 的空容器，它至少包含 n 个桶，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略 eq，则将 key_equal()用作键值相等谓词；如果也省略了 hf，则将 hasher()用作哈希函数 |
| X(i, j, n, hf, eq) | 创建一个至少包含 n 个桶的空容器，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal()用作键值相等谓词；如果省略了 hf，将 hasher()用作哈希函数；如果省略了 n，则包含桶数不确定 |
| X a(i, j, n, hf, eq) | 创建一个名为 a 的的空容器，它至少包含 n 个桶，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal()用作键值相等谓词；如果省略了 hf，将 hasher()用作哈希函数；如果省略了 n，则包含桶数不确定 |
| b.hash_function() | 返回 b 使用的哈希函数 |
| b.key_eq() | 返回创建 b 时使用的键值相等谓词 |
| b.bucket_count() | 返回 b 包含的桶数 |
| b.max_bucket_count () | 返回一个上限数，它指定了 b 最多可包含多少个桶 |
| b.bucket(k) | 返回键值为 k 的元素所属桶的索引 |
| b.bucket_size(n) | 返回索引为 n 的桶可包含的元素数 |
| b.begin(n) | 返回一个迭代器，它指向索引为 n 的桶中的第一个元素 |
| b.end(n) | 返回一个迭代器，它指向索引为 n 的桶中的最后一个元素 |
| b.cbegin(n) | 返回一个常量迭代器，它指向索引为 n 的桶中的第一个元素 |
| b.cend(n) | 返回一个常量迭代器，它指向索引为 n 的桶中的最后一个元素 |
| b.load_factor() | 返回每个桶包含的平均元素数 |
| b.max_load_factor() | 返回负载系数的最大可能取值；超过这个值后，容器将增加桶 |
| b.max_load_factor(z) | 可能修改最大负载系统，建议将它设置为 z |
| a.rehash(n) | 将桶数调整为不小于 n，并确保 a.bucket_count() > a.size() / a.max_load_factor() |
| a.reserve(n) | 等价于 a.rehash(ceil(n/a.max_load_factor()))，其中 ceil(x)返回不小于 x 的最小整数 |

## G.5　STL 函数

STL 算法库（由头文件 algorithm 和 numeric 支持）提供了大量基于迭代器的非成员模板函数。正如第

16 章介绍的，选择的模板参数名指出了特定参数应模拟的概念。例如，ForwardIterator 用于指出，参数至少应模拟正向迭代器的要求；Predicate 用于指出，参数应是一个接受一个参数并返回 bool 值的函数对象。C++标准将算法分成 4 组：非修改式序列操作、修改式序列操作、排序和相关运算符以及数值操作（C++11 将数值操作从 STL 移到了 numeric 库中，但这不影响它们的用法）。序列操作（sequence operation）表明，函数将接受两个迭代器作为参数，它们定义了要操作的区间或序列。修改式（mutating）意味着函数可以修改容器的内容。

### G.5.1 非修改式序列操作

表 G.13 对非修改式序列操作进行了总结。这里没有列出参数，而重载函数只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

**表 G.13 非修改式序列操作**

| 函 数 | 描 述 |
|---|---|
| all_of() | 如果对于所有元素的谓词测试都为 true，则返回 true。这是 C++11 新增的 |
| any_of() | 只要对于任何一个元素的谓词测试为 true，就返回 true。这是 C++11 新增的 |
| none_of() | 如果对于所有元素的谓词测试都为 false，则返回 true。这是 C++11 新增的 |
| for_each() | 将一个非修改式函数对象用于区间中的每个成员 |
| find() | 在区间中查找某个值首次出现的位置 |
| find_if() | 在区间中查找第一个满足谓词测试条件的值 |
| find_if_not() | 在区间中查找第一个不满足谓词测试条件的值。这是 C++11 新增的 |
| find_end() | 在序列中查找最后一个与另一个序列匹配的值。匹配时可以使用等式或二元谓词 |
| find_first_of() | 在第二个序列中查找第一个与第一个序列的值匹配的元素。匹配时可以使用等式或二元谓词 |
| adjacent_find | 查找第一个与其后面的元素匹配的元素。匹配时可以使用等式或二元谓词 |
| count() | 返回特定值在区间中出现的次数 |
| count_if() | 返回特定值与区间中的值匹配的次数，匹配时使用二元谓词 |
| mismatch() | 查找区间中第一个与另一个区间中对应元素不匹配的元素，并返回指向这两个元素的迭代器。匹配时可以使用等式或二元谓词 |
| equal() | 如果一个区间中的每个元素都与另一个区间中的相应元素匹配，则返回 true。匹配时可以使用等式或二元谓词 |
| is_permutation() | 如果可通过重新排列第二个区间，使得第一个区间和第二个区间对应的元素都匹配，则返回 true，否则返回 false。匹配可以是相等，也可以使用二元谓词进行判断。这是 C++11 新增的 |
| search() | 在序列中查找第一个与另一个序列的值匹配的值。匹配时可以使用等式或二元谓词 |
| search_n() | 查找第一个由 n 个元素组成的序列，其中每个元素都与给定值匹配。匹配时可以使用等式或二元谓词 |

下面更详细地讨论这些非修改型序列操作。对于每个函数，首先列出其原型，然后做简要地描述。和前面一样，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。有些函数接受两个区间，这两个区间的容器类型可以不同。例如，可以使用 equal()来对链表和矢量进行比较。作为参数传递的函数是函数对象，这些函数对象可以是指针（如函数名），也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。

1. all_of()（C++11）

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 true，或者该区间为空，则函数 all_of()返回 true；否则返回 false。

2. any_of()（C++11）

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 any_of()返回 false；否则返回 true。

3. none_of()（C++11）

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last,
             Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 all_of()返回 true；否则返回 false。

4. for_each()

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f);
```

for_each()函数将函数对象 f 用于[first, last]区间中的每个元素，它也返回 f。

5. find()

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
                   const T& value);
```

find()函数返回一个迭代器，该迭代器指向区间[first, last]中第一个值为 value 的元素；如果没有找到这样的元素，则返回 last。

6. find_if()

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if()函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 true 的元素；如果没有找到这样的元素，则返回 last。

7. find_if_not()

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if_not()函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 false 的元素；如果没有找到这样的元素，则返回 last。

8. find_end()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_end()函数返回一个迭代器，该迭代器指向[first1, last1] 区间中最后一个与[first2, last2] 区间的内容匹配的序列的第一个元素。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都返回 last1。

9. find_first_of()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
        class BinaryPredicate>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_first_of() 函数返回一个迭代器，该迭代器指向区间[first1, last1]中第一个与[first2, last2]区间中的任

何元素匹配的元素。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都将返回 last1。

10. adjacent_find()

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
         ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
         ForwardIterator last, BinaryPredicate pred);
```

adjacent_find()函数返回一个迭代器，该迭代器指向[first1, last1] 区间中第一个与其后面的元素匹配的元素。如果没有找到这样的元素，则返回 last。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

11. count()

```
template<class InputIterator, class T>
 typename iterator_traits<InputIterator>::difference_type
  count(InputIterator first, InputIterator last, const T& value);
```

count()函数返回[first, last)区间中与值 value 匹配的元素数目。对值进行比较时，将使用值类型的= =运算符。返回值类型为整型，它足以存储容器所能存储的最大元素数。

12. count_if()

```
template<class InputIterator, class Predicate>
 typename iterator_traits<InputIterator>::difference_type
  count_if(InputIterator first, InputIterator last, Predicate pred);
```

count if()函数返回[first, last]区间中这样的元素数目，即将其作为参数传递给函数对象 pred 时，后者的返回值为 true。

13. mismatch()

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
   mismatch(InputIterator1 first1,
            InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
                               class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate pred);
```

每个 mismatch()函数都在[first1, last1)区间中查找第一个与从 first2 开始的区间中相应元素不匹配的元素，并返回两个迭代器，它们指向不匹配的两个元素。如果没有发现不匹配的情况，则返回值为 pair<last1, first2 + (last1 - first1)>。第一个版本使用= =运算符来测试匹配情况；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 false，则 it1 和 it2 指向的元素不匹配。

14. equal()

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果[first1, last1)区间中每个元素都与以 first2 开始的序列中相应元素匹配，则 equal()函数返回 true，否则返回 false。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

15. is_permutation()（C++11）

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
```

```
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果通过对从 first2 开始的序列进行排列，可使其与区间[first1, last1]相应的元素匹配，则函数 is_permutation()返回 true，否则返回 false。第一个版本使用值类型的==运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素，也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

16. search()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
```

search()函数在[first1, last1]区间中搜索第一个与[first2, last2] 区间中相应的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

17. search_n()

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                          Size count, const T& value, BinaryPredicate pred);
```

search_n()函数在[first1, last1)区间中查找第一个与 count 个 value 组成的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

### G.5.2　修改式序列操作

表 G.14 对修改式序列操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

**表 G.14　　修改式序列操作**

| 函　数 | 描　述 |
|---|---|
| copy() | 将一个区间中的元素复制到迭代器指定的位置 |
| copy_n() | 从一个迭代器指定的地方复制 n 个元素到另一个迭代器指定的地方，这是 C++11 新增的 |
| copy_if() | 将一个区间中满足谓词测试的元素复制到迭代器指定的地方，这是 C++11 新增的 |
| copy_backward() | 将一个区间中的元素复制到迭代器指定的地方。复制时从区间结尾开始，由后向前进行 |
| move() | 将一个区间中的元素移到迭代器指定的地方，这是 C++11 新增的 |
| move_backward() | 将一个区间中的元素移到迭代器指定的地方；移动时从区间结尾开始，由后向前进行。这是 C++11 新增的 |
| swap() | 交换引用指定的位置中存储的值 |
| swap_ranges() | 对两个区间中对应的值进行交换 |
| iter_swap() | 交换迭代器指定的位置中存储的值 |
| transform() | 将函数对象用于区间中的每一个元素（或区间对中的每对元素），并将返回的值复制到另一个区间的相应位置 |
| replace() | 用另外一个值替换区间中某个值的每个实例 |
| replace_if() | 如果用于原始值的谓词函数对象返回 true，则使用另一个值来替换区间中某个值的所有实例 |
| replace_copy() | 将一个区间复制到另一个区间中，使用另外一个值替换指定值的每个实例 |
| replace_copy_if() | 将一个区间复制到另一个区间，使用指定值替换谓词函数对象为 true 的每个值 |

续表

| 函　数 | 描　述 |
|---|---|
| fill() | 将区间中的每一个值设置为指定的值 |
| fill_n() | 将 n 个连续元素设置为一个值 |
| generate() | 将区间中的每个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象 |
| generate_n() | 将区间中的前 n 个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象 |
| remove() | 删除区间中指定值的所有实例，并返回一个迭代器，该迭代器指向得到的区间的超尾 |
| remove_if() | 将谓词对象返回 true 的值从区间中删除，并返回一个迭代器，该迭代器指向得到的区间的超尾 |
| remove_copy() | 将一个区间中的元素复制到另一个区间中，复制时忽略与指定值相同的元素 |
| remove_copy_if() | 将一个区间中的元素复制到另一个区间中，复制时忽略谓词函数对象返回 true 的元素 |
| unique() | 将区间内两个或多个相同元素组成的序列压缩为一个元素 |
| unique_copy() | 将一个区间中的元素复制到另一个区间中，并将两个或多个相同元素组成的序列压缩为一个元素 |
| reverse() | 反转区间中的元素的排列顺序 |
| reverse_copy() | 按相反的顺序将一个区间中的元素复制到另一个区间中 |
| rotate() | 将区间中的元素循环排列，并将元素左转 |
| rotate_copy() | 以旋转顺序将区间中的元素复制到另一个区间中 |
| random_shuffle() | 随机重新排列区间中的元素 |
| shuffle() | 随机重新排列区间中的元素，使用的函数对象满足 C++11 对统一随机生成器的要求 |
| is_partitioned() | 如果区间根据指定的谓词进行了分区，则返回 true |
| partition() | 将满足谓词函数对象的所有元素都放在不满足谓词函数对象的元素之前 |
| stable_partition() | 将满足谓词函数对象的所有元素放置在不满足谓词函数对象的元素之前，每组中元素的相对顺序保持不变 |
| partition_copy() | 将满足谓词函数对象的所有元素都复制到一个输出区间中，并将其他元素都复制到另一个输出区间中，这是 C++11 新增的 |
| partition_point() | 对于根据指定谓词进行了分区的区间，返回一个迭代器，该迭代器指向第一个不满足该谓词的元素 |

下面详细地介绍这些修改型序列操作。对于每个函数，首先列出其原型，然后做简要的描述。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受两个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

copy()函数将[first, last)区间中的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

2. copy_n()（C++11）

```
template<class InputIterator, class Size class OutputIterator>
OutputIterator copy(InputIterator first, Size n,
                    OutputIterator result);
```

函数 copy_n()从位置 first 开始复制 n 个元素到区间[result, result + n] 中，并返回 result + n，即指向被复制到的最后一个位置后面的迭代器。该函数不要求目标和源不重叠。

3. copy_if()（C++11）

```
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);
```

函数 copy_if()将[first, last)区间中满足谓词 pred 的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)

区间中，也就是说，目标不能与源重叠。

4. copy_backward()

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 copy_backward()将[first, last)区间中的元素复制到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

5. move()（C++11）

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

函数 move()使用 std::move()将[first, last)区间中的元素移到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

6. move_backward()（C++11）

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 move_backward() std::move()将[first, last)区间中的元素移到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

7. swap()

```
template<class T> void swap(T& a, T& b);
```

swap()函数对引用指定的两个位置中存储的值进行交换（C++11 将这个函数移到了头文件 utility 中）。

8. swap_ranges()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2);
```

swap_ranges()函数将[first1, last1]区间中的值与从 first2 开始的区间中对应的值交换。这两个区间不能重叠。

9. iter_swap()

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

iter_swap()函数将迭代器指定的两个位置中存储的值进行交换。

10. transform()

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

第一个版本的 transform()将一元函数对象 op 应用到[first, last)区间中每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置为 op(*first)，依此类推。该函数返回 result + (last - first)，即目标区间的超尾值。

第二个版本的 transform()将二元函数对象 op 应用到[first1, last1)区间和[first2, last2)区间中的每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置成 op(*first1, *first2)，依此类推。该函数返回 result + (last – first)，即目标区间的超尾值。

11. replace()

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

replace()函数将[first, last]中的所有 old_value 替换为 new_value。

12. replace_if()

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

replace_if()函数使用 new_value 值替换[first, last]区间中 pred（old）为 true 的每个 old 值。

13. replace_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
     OutputIterator result,const T& old_ value, const T& new_ value);
```

replace_copy()函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替所有的 old_value。该函数返回 result + (last - first)，即目标区间的超尾值。

14. replace_copy_if()

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
    OutputIterator result, Predicate pred, const T& new_ value);
```

replace_copy_if()函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替 pred(old)为 true 的所有 old 值。该函数返回 result + (last - first)，即目标区间的超尾值。

15. fill()

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

fill()函数将[first, last]区间中的每个元素都设置为 value。

16. fill_n()

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

fill_n()函数将从 first 位置开始的前 n 个元素都设置为 value。

17. generate()

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

generate()函数将[first, last)区间中的每个元素都设置为 gen()，其中 gen 是一个生成器函数对象，即不接受任何参数。例如，gen 可以是一个指向 rand()的指针。

18. generate_n()

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

generate_n()函数将从 first 开始的区间中前 n 个元素都设置为 gen()，其中，gen 是一个生成器函数对象，即不接受任何参数。例如，gen 可以是一个指向 rand()的指针。

19. remove()

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

remove()函数删除[first, last)区间中所有值为 value 的元素，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

**注意：**由于所有的 remove()和 unique()函数都不是成员函数，同时这些函数并非只能用于 STL 容器，因此它们不能重新设置容器的长度。相反，它们返回一个指示新超尾位置的迭代器。通常，被删除的元素只是被移到容器尾部。然而，对于 STL 容器，可以使用返回的迭代器和 erase()方法来重新设置 end()。

20. remove_if()

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

remove_if()函数将 pred(val)为 true 的所有 val 值从[first, last)区间删除，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

21. remove_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

remove_copy()函数将[first, last)区间中的值复制到从 result 开始的区间中，复制时将忽略 value。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有被删除的元素的顺序将保持不变。

22. remove_copy_if()

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

remove_copy_if()函数将[first, last)区间中的值复制到从 result 开始的区间，但复制时忽略 pred(val)为 true 的 val。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有删除的元素的顺序将保持不变。

23. unique()

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

unique()函数将[first, last)区间中由两个或更多相同元素构成的序列压缩为一个元素，并返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

24. unique_copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

unique_copy()函数将[first, last)区间中的元素复制到从 result 开始的区间中，并将由两个或更多个相同元素组成的序列压缩为一个元素。该函数返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符，对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

25. reverse()

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

reverse()函数通过调用 swap(first, last - 1)等来反转[first, last]区间中的元素。

26. reverse_copy

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

reverse_copy()函数按相反的顺序将[first, last)区间中的元素复制到从 result 开始的区间中。这两个区间不能重叠。

27. rotate()

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

rotate()函数将[first, last)区间中的元素左旋。middle 处的元素被移到 first 处，middle + 1 处的元素被移到 first + 1 处，依此类推。middle 前的元素绕回到容器尾部，以便 first 处的元素可以紧接着 last - 1 处的元素。

28. rotate_copy()

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

rotate_copy()函数使用为 rotate()函数描述的旋转序列，将[first, last)区间中的元素复制到从 result 开始的区间中。

29. random_shuffle()

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

这个版本的 random_shuffle()函数将[first, last)区间中的元素打乱。分布是一致的，即原始顺序的每种可能排列方式出现的概率相同。

30. random_shuffle()

```
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator&& random);
```

这个版本的 random_shuffle()函数将[first, last)区间中的元素打乱。函数对象 random 确定分布。假设有 n 个元素，表达式 random(n)将返回[0, n)区间中的一个值。在 C++98 中，参数 random 是一个左值引用，而在 C++11 中是一个右值引用。

31. shuffle()

```
template<class RandomAccessIterator, class Uniform RandomNumberGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    UniformRandomNumberGenerator&& rgen);
```

函数 shuffle()将[first, last)区间中的元素打乱。函数对象 rgen 确定分布，它应满足 C++11 指定的有关均匀随机数生成器的要求。假设有 n 个元素，表达式 rgen(n)将返回[0, n]区间中的一个值。

32. is_partitioned()（C++11）

```
template<class InputIterator, class Predicate>
bool is_partitioned(InputIterator first,
                    InputIterator last, Predicate pred);
```

如果区间为空或根据 pred 进行了分区（即满足谓词 pred 的元素都在不满足该谓词的元素前面），函数 is__partitioned()将返回 true，否则返回 false。

33. partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

函数 partition()将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

34. stable_partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

函数 stable_partition()将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前；在这两组中，元素的相对顺序保持不变。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

35. partition_copy()（C++11）

```
template<class InputIterator, class OutputIterator1,
         classs OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
     InputIterator first, InputIterator last,
     OutputIterator1 out_true, OutputIterator2 out_false
                                Predicate pred);
```

函数 partition_copy()将所有这样的元素都复制到从 out_true 开始的区间中，即其值 val 使得 pred(val)为 true；并将其他的元素都复制到从 out_false 开始的区间中。它返回一个 pair 对象，该对象包含两个迭代器，分别指向从 out_true 和 out_false 开始的区间的末尾。

36. partition_point()（C++11）

```
template<class ForwardIterator, class Predicate>
ForewardIterator partition_point(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

函数 partition_point()要求区间根据 pred 进行了分区。它返回一个迭代器，指向最后一个让谓词对象函数为 true 的值所在的位置。

### G.5.3　排序和相关操作

表 G.15 对排序和相关操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

**表 G.15　　排序和相关操作**

| 函　数 | 描　述 |
|---|---|
| sort() | 对区间进行排序 |
| stable_sort() | 对区间进行排序，并保留相同元素的相对顺序 |
| partial_sort() | 对区间进行部分排序，提供完整排序的前 n 个元素 |
| partial_sort_copy() | 将经过部分排序的区间复制到另一个区间中 |
| is_sorted() | 如果对区间进行了排序，则返回 true，这是 C++11 新增的 |
| is_sorted_until() | 返回一个迭代器，指向经过排序的区间末尾，这是 C++11 新增的 |
| nth_element() | 对于给定指向区间的迭代器，找到区间被排序时，相应位置将存储哪个元素，并将该元素放到这里 |
| lower_bound() | 对于给定的一个值，在一个排序后的区间中找到第一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序 |
| upper_bound() | 对于给定的一个值，在一个排序后的区间中找到最后一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序 |
| equal_range() | 对于给定的一个值，在一个排序后的区间中找到一个最大的区间，使得将这个值插入其中的任何位置，都不会破坏顺序 |
| binary_search() | 如果排序后的区间中包含了与给定的值相同的值，则返回 true，否则返回 false |
| merge() | 将两个排序后的区间合并为第三个区间 |
| inplace_merge() | 就地合并两个相邻的、排序后的区间 |
| includes() | 如果对于一个集合中的每个元素都可以在另外一个集合中找到，则返回 true |
| set_union() | 构造两个集合的并集，其中包含在任何一个集合中出现过的元素 |
| set_intersection() | 构造两个集合的交集，其中包含在两个集合中都出现过的元素 |
| set_difference() | 构造两个集合的差集，即包含第一个集合中且没有出现在第二个集合中的所有元素 |
| set_symmetric_difference() | 构造由只出现在其中一个集合中的元素组成的集合 |
| make_heap() | 将区间转换成堆 |
| push_heap() | 将一个元素添加到堆中 |
| pop_heap() | 删除堆中最大的元素 |
| sort_heap() | 对堆进行排序 |
| is_heap() | 如果区间是堆，则返回 true，这是 C++11 新增的 |
| is_heap_until() | 返回一个迭代器，指向属于堆的区间的末尾，这是 C++11 新增的 |
| min() | 返回两个值中较小的值，如果参数为 initializer_list，则返回最小的元素（这是 C++11 新增的） |
| max() | 返回两个值中较大的值，如果参数为 initializer_list，则返回最大的元素（这是 C++11 新增的） |
| minmax() | 返回一个 pair 对象，其中包含按递增顺序排列的两个参数；如果参数为 initializer_list，则返回 pair 对象包含最小和最大的元素。这是 C++11 新增的 |
| min_element() | 在区间找到最小值第一次出现的位置 |
| max_element() | 在区间找到最大值第一次出现的位置 |
| minmax_element() | 返回一个 pair 对象，其中包含两个迭代器，它们分别指向区间中最小值第一次出现的位置和区间中最大值最后一次出现的位置。这是 C++11 新增的 |
| lexicographic_compare() | 按字母顺序比较两个序列，如果第一个序列小于第二个序列，则返回 true，否则返回 false |
| next-permutation() | 生成序列的下一种排列方式 |
| previous_permutation() | 生成序列的前一种排列方式 |

本节中的函数使用为元素定义的<运算符或模板类型 Compare 指定的比较对象来确定两个元素的顺序。如果 comp 是一个 Compare 类型的对象，则 comp(a, b)就是 a<b 的统称，如果在排序机制中，a 在 b 之前，则返回 true。如果 a<b 返回 fasle，同时 b<a 也返回 false，则说明 a 和 b 相等。比较对象必须至少提供严格

弱排序功能（strict weak ordering）。这意味着：

- 表达式 comp(a, a)一定为 false，这是“值不能比其本身小”的统称（这是严格部分）。
- 如果 comp(a, b)为 true，且 comp(b, c)也为 true，则 comp(a, c)为 true（也就是说，比较是一种可传递的关系）。
- 如果 a 与 b 等价，且 b 与 c 也等价，则 a 与 c 等价（也就是说，等价也是一种可传递的关系）。

如果想将<运算符用于整数，则等价就意味着相等，但这一结论不能推而广之。例如，可以用几个描述邮件地址的成员来定义一个结构，同时定义一个根据邮政编码对结构进行排序的 comp 对象。则邮政编码相同的地址是等价的，但它们并不相等。

下面更详细地介绍排序及相关操作。对于每个函数，首先列出其原型，然后做简要的说明。我们将这一节分成几个小节。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last)区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. 排序

首先来看看排序算法。

（1）sort()

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

sort()函数将[first, last)区间按升序进行排序，排序时使用值类型的<运算符进行比较。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）stable_sort()

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

stable_sort()函数对[first, last)区间进行排序，并保持等价元素的相对顺序不变。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）partial_sort()

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
RandomAccessIterator last, Compare comp);
```

partial_sort()函数对[first, last)区间进行部分排序。将排序后的区间的前 middle - first 个元素置于[first, middle]区间内，其余元素没有排序。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）partial_sort_copy()

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                       InputIterator last,
                       RandomAccessIterator result_first,
                       RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

partial_sort_copy()函数将排序后的区间[first, last]中的前 n 个元素复制到区间[result_first, result_first + n]

中。n 的值是 last - first 和 result_last - result_first 中较小的一个。该函数返回 result - first + n。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（5）is_sorted（C++11）

```
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]是经过排序的，函数 is_sorted()将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（6）is_sorted_until（C++11）

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]包含的元素少于两个，函数 is_sorted_until 将返回 last；否则将返回迭代器 it，确保区间[first, it]是经过排序的。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（7）nth_element()

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

nth_element()函数找到将[first, last)区间排序后的第 n 个元素，并将该元素置于第 n 个位置。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

2. 二分法搜索

二分法搜索组中的算法假设区间是经过排序的。这些算法只要求正向迭代器，但使用随机迭代器时，效率最高。

（1）lower_bound()

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

lower_bound()函数在排序后的区间[first, last)中找到第一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）upper_bound()

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

upper_bound()函数在排序后的区间[first, last)中找到最后一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）equal_range()

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
```

```
    ForwardIterator first, ForwardIterator last, const T& value,
    Compare comp);
```

equal_range()函数在排序后的区间[first, last)区间中找到这样一个最大的子区间[it1, it2)，即将 value 插入到该区间的任何位置都不会破坏顺序。该函数返回一个由 it1 和 it2 组成的 pair。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（4）binary_search()

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

如果在排序后的区间[first, last]中找到与 value 等价的值，则 binary_search()函数返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

**注意：**前面说过，使用<进行排序时，如果 a<b 和 b<a 都为 false，则 a 和 b 等价。对于常规数字来说，等价意味着相等；但对于只根据一个成员进行排序的结构来说，情况并非如此。因此，在确保顺序不被破坏的情况下，可插入新值的位置可能不止一个。同样，如果使用比较对象 comp 进行排序，等价意味着 comp(a, b)和 comp（b，a）都为 false（这是“如果 a 不小于 b，b 也不小于 a，则 a 与 b 等价”的统称）。

3. 合并

合并函数假设区间是经过排序的。

（1）merge()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

merge()函数将排序后的区间[first1, last1] 中的元素与排序后的区间[first2, last2] 中的元素进行合并，并将结果放到从 result 开始的区间中。目标区间不能与被合并的任何一个区间重叠。在两个区间中发现了等价元素时，第一个区间中的元素将位于第二个区间中的元素前面。返回值是合并的区间的超尾迭代器。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（2）inplace_merge()

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last,
     Compare comp);
```

inplace_merge()函数将两个连续的、排序后的区间——[first, middle] 和[middle, last] ——合并为一个经过排序的序列，并将其存储在[first, last] 区间中。第一个区间中的元素将位于第二个区间中的等价元素之前。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

4. 使用集合

集合操作可用于所有排序后的序列，包括集合（set）和多集合（multiset）。对于存储一个值的多个实例的容器（如 multiset）来说，定义是广义的。对于两个多集合的并集，将包含较大数目的元素实例，而交集将包含较小数目的元素实例。例如，假设多集合 A 包含了字符串“apple”7 次，多集合 B 包含该字符串 4 次。则 A 和 B 的并集将包含 7 个“apple”实例，它们的交集将包含 4 个实例。

（1）includes()

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2, Compare comp);
```

如果[first2, last2)区间中的每一个元素在[first1, last1)区间中都可以找到，则 includes()函数返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）set_union()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

set_union()函数构造一个由[first1, last1]区间和[first2, last2] 区间组合而成的集合，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。并集包含在任何一个集合（或两个集合）中出现的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）set_intersection()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1,InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_intersection()函数构造[first1, last1)区间和[first2, last2)区间的交集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。交集包含两个集合中共有的元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）set_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_difference()函数构造[first1, last1)区间与[first2, last2)区间的差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。差集包含出现在第一个集合中，但不出现在第二个集合中的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（5）set_symmetric_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp);
```

set_symmetric_difference()函数构造[first1, last1)区间和[first2, last2)区间的对称（symmetric）差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。对称差集包含出现在第一个集合中，但不出现在第二个集合中，或者出现在第二个集合中，但不出现在第一个集合中的所有元素。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

5. 使用堆

堆（heap）是一种常用的数据形式，具有这样特征：第一个元素是最大的。每当第一个元素被删除或添加了新元素时，堆都可能需要重新排列，以确保这一特征。设计堆是为了提高这两种操作的效率。

（1）make_heap()

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

make_heap()函数将[first, last)区间构造成一个堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）push_heap()

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

push_heap()函数假设[first, last – 1)区间是一个有效的堆，并将 last - 1 位置（即被假设为有效堆的区间后面的一个位置）上的值添加到堆中，使[first, last)区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）pop_heap()

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

pop_heap()函数假设[first, last)区间是一个有效堆，并将位置 last - 1 处的值与 first 处的值进行交换，使[first, last – 1]区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）sort_heap()

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

sort_heap()函数假设[first, last)区间是一个有效堆，并对其进行排序。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）is_heap()（C++11）

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last
               Compare comp);
```

如果区间[first, last]是一个有效的堆，函数 is_heap()将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）is_heap_until()（C++11）

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
            RandomAccessIterator first, RandomAccessIterator last
            Compare comp);
```

如果区间[first, last)包含的元素少于两个，则返回 last；否则返回迭代器 it，而区间[first, it)是一个有效的堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

6. 查找最小和最大值

最小函数和最大函数返回两个值或值序列中的最小值和最大值。

（1）min()

```
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

这些版本的 min()函数返回两个值中较小一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T min(initializer_list<T> t);

template<class T, class Compare>
T min(initializer_list<T> t), Compare comp);
```

这些版本的 min()函数是 C++11 新增的，它返回初始化列表 t 中最小的值。如果有多个相等的值且最小，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）max()

```
template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

这些版本的 max() 函数返回这两个值中较大的一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T max(initializer_list<T> t);

template<class T, class Compare>
T max(initializer_list<T> t), Compare comp);
```

这些版本的 max()函数是 C++11 新增的，它返回初始化列表 t 中最大的值。如果有多个相等的值且最大，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）minmax()（C++11）

```
template<class T>
  pair<const T&,const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
  pair<const T&,const T&> minmax(const T& a, const T& b,
                                 Compare comp);
```

如果 b 小于 a，这些版本的 minmax()函数返回 pair(b, a)，否则返回 pair(a, b)。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> pair<T,T> minmax(initializer_list<T> t);

template<class T, class Compare>
pair<T,T> minmax(initializer_list<T> t), Compare comp);
```

这些版本的 minmax()函数返回初始化列表 t 中最小元素和最大元素的拷贝。如果有多个最小的元素，则返回其中的第一个；如果有多个最大的元素，则返回其中的最后一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）min_element()

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

min_element()函数返回这样一个迭代器，该迭代器指向[first, last)区间中第一个最小的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）max_element()

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

max_element()函数返回这样一个迭代器，该迭代器指向[first, last] 区间中第一个最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）minmax_element()（C++11）

```
template<class ForwardIterator>
  pair<ForwardIterator,ForwardIterator>
     minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  pair<ForwardIterator,ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last,
                Compare comp);
```

函数 minmax_element()返回一个 pair 对象，其中包含两个迭代器，分别指向区间[first, last)中最小和最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（7）lexicographical_compare()

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     Compare comp);
```

如果[first1, last1]区间中的元素序列按字典顺序小于[first2, last2]区间中的元素序列，则 lexicographical_compare()函数返回 true，否则返回 false。字典比较将两个序列的第一个元素进行比较，即对*first1 和*first2 进行比较。如果*first1 小于*first2，则该函数返回 true；如果*first2 小于*first1，则返回 fasle；如果相等，则继续比较两个序列中的下一个元素。直到对应的元素不相等或到达了某个序列的结尾，比较才停止。如果在到达某个序列的结尾时，这两个序列仍然是等价的，则较短的序列较小。如果两个序列等价，且长度相等，则任何一个序列都不小于另一个序列，因此函数将返回 false。第一个版本使用<来比较元素，而第二个版本使用 comp 比较对象。字典比较是按字母顺序比较的统称。

7. 排列组合

序列的排列组合是对元素重新排序。例如，由 3 个元素组成的序列有 6 种可能的排列方式，因为对于第一个位置，有 3 种选择；给第一个位置选定元素后，第二个位置有两种选择；第三个位置有 1 种选择。例如，数字 1、2 和 3 的 6 种排列如下：

```
123 132 213 232 312 321
```

通常，由 n 个元素组成的序列有 n*(n−1)*...*1 或 n!种排列。

排列函数假设按字典顺序排列各种可能的排列组合，就像前一个例子中的 6 种排列那样。这意味着，通常，在每个排列之前和之后都有一个特定的排列。例如，213 在 231 之前，312 在 231 之后。然而，第一个排列（如示例中的 123）前面没有其他排列，而最后一个排列（321）后面没有其他排列。

（1）next_permutation()

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

next_permutation()函数将[first, last]区间中的序列转换为字典顺序的下一个排列。如果下一个排列存在，则该函数返回 true；如果下一个排列不存在（即区间中包含的是字典顺序的最后一个排列），则该函数返回 false，并将区间转换为字典顺序的第一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp

比较对象。

（2）prev_permutation()

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

previous_permutation()函数将[first, last] 区间中的序列转换为字典顺序的前一个序列。如果前一个排列存在，则该函数返回 true；如果前一个序列不存在（即区间包含的是字典顺序的第一个排列），则该函数返回 false，并将该区间转换为字典顺序的最后一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp 比较对象。

### G.5.4　数值运算

表 G.16 对数值运算进行了总结，这些操作是由头文件 numeric 描述的。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能；如果对某个函数非常感兴趣，则可以了解其细节。

**表 G.16　　数值运算**

| 函　　数 | 描　　述 |
|---|---|
| accumulate() | 计算区间中的值的总和 |
| inner_product() | 计算 2 个区间的内部乘积 |
| partial_sum() | 将使用一个区间计算得到的小计复制到另一个区间中 |
| adjacent_difference() | 将使用一个区间的元素计算得到的相邻差集复制到另一个区间中 |
| iota() | 将使用运算符++生成的一系列相邻的值赋给一个区间中的元素，这是 C++11 新增的 |

1. accumulate()

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

accumulate()函数将 acc 的值初始化为 init，然后按顺序对[first, last]区间中的每一个迭代器 i 执行 acc = acc + * i（第一个版本）或 acc = binary_op(acc, *i)（第二个版本）。然后返回 acc 的值。

2. inner_product()

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

inner_product()函数将 acc 的值初始化为 init，然后按顺序对[first1, last1]区间中的每一个迭代器 i 和[first2, first2 + (last1−first1)]区间中对应的迭代器 j 执行 acc = * i * * j（第一个版本）或 acc = binary_op(*i, *j)（第二个版本）。也就是说，该函数对每个序列的第一个元素进行计算，得到一个值，然后对每个序列中的第二个元素进行计算，得到另一个值，依此类推，直到到达第一个序列的结尾（因此，第二个序列至少应同第一个序列一样长）。然后，函数返回 acc 的值。

3. partial_sum()

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
```

```
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result,
                           BinaryOperation binary_op);
```

partial_sum()函数将*first 赋给*result，将*first + *(first + 1)赋给*(result + 1)（第一个版本），或者将binary_op(*first, *(first + 1))赋给*(result + 1)（第二个版本），依此类推。也就是说，从 result 开始的序列的第 n 个元素将包含从 first 开始的序列的前 n 个元素的总和（或 binary_op 的等价物）。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许使用结果覆盖原来的序列。

4. adjacent_difference()

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   BinaryOperation binary_op);
```

adjacent_difference()函数将*first 赋给 result(*result = *first)。目标区间中随后的位置将被赋值为源区间中相邻位置的差集（或 binary_op 等价物）。也就是说，目标区间的下一个位置(result +1)将被赋值为*(first + 1) - * first（第一个版本）或 binary_op(*(first + 1), * first)（第二个版本），依此类推。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许结果覆盖原来的序列。

5. iota()（C++11）

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last,T value);
```

函数 iota()将 value 赋给*first，再将 value 递增（就像执行运算++value），并将结果赋给下一个元素，依次类推，直到最后一个元素。

# 附录 H　精选读物和网上资源

有很多有关 C++和编程的优秀图书和网上资源。下述清单只是其中的一些代表作而不是全部，因此还有很多优秀的图书和网站这里没有列出；然后该清单确实具有广泛的代表性。

## H.1　精选读物

- Becker, Pete. The C++ Standard Library Extensions. Upper Saddle River, NJ：Addison-Wesley，2007。

本书讨论第一个 TR1（Technical Report）库。这是一个可选的 C++98 库，但 C++11 包含其大部分元素。涉及的主题包括无序集合模板、智能指针、正则表达式库、随机数库和元组。

- Booch, Grady, Robert A. Maksimchuk, Michael W. Engel, and Bobbi J.Young. Object-Oriented Analysis and Design, Third Edition. Upper Saddle River, NJ：Addison-Wesley, 2007。

本书介绍了 OOP 概念，讨论了 OOP 方法，并提供一些示例应用程序，示例是使用 C++编写的。

- Cline, Marshall, Greg Lomow and Mike Girou. *C++FAQ，Second Edition*（C++常见问题解答，第二版）。Reading，MA：Addison-Wesley，1998。

本书解答了多个经常问到的有关 C++语言的问题。

- Josuttis, Nicolai M. The C++ Standard Library: A Tutorial and Reference（C++标准库教程和参考手册）。Reading. MA：Addison-Wesley，1999。

本书介绍了标准模板库（STL）以及其他 C++库特性，如复数支持、区域和输入/输出流。

- Karlsson, Björn. Beyond the C++ Standard Library:An Introduction to Boost. Upper Saddle River，NJ：Addison-Wesley，2006。

顾名思义，本书探讨多个 Boost 库。

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*。Upper Saddle River，NJ：Addison-Wesley，2005。

本书针对的是了解 C++的程序员，提供了 55 条规定和指南。其中一些是技术性的，如解释何时应定义复制构造函数和赋值运算符；其他一些更为通用，如对 is-a 和 has-a 关系的讨论。

- Meyers, Scott. *Effctive STL: 50 Specific Ways to Improve Your Use of the Stadard Template Library*。Reading，MA：Addison-Wesley，2001。

本书提供了选择容器和算法的指南，并讨论了使用 STL 的其他方面。

- Meyers, Scott. *More Effecitve C++: 35 New Ways to Improve Your Programs and Designs*。Reading，MA：Addison-Wesley，1996。

本书秉承了《*Effecitve C++*》的传统，对语言中的一些较模糊的问题进行了解释，介绍了实现各种目标的方法，如设计智能指针，并反映了 C++程序员在过去几年中获得的其他一些经验。

- Musser, David R, Gillmer J. Derge, and Atul Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition. Reading，MA：Addison-Wesley，2001。

要介绍并演示 STL 的功能，需要一整本书，该书可满足您的需求。

- Stroustrup, Bjarne. The C++ Programming Language. Third Edition。Reading，MA：Addison-Wesley，1997。

Stroustrup 创建了 C++，因此这是一部权威作品。不过，如果对 C++有一定的了解，将可以很容易地掌握它。它不仅介绍了语言，而且提供了多个如何使用该语言的示例，同时讨论了 OOP 方法。随着语言的发展，这本书已有多个版本，该版本增加了对标准库元素的讨论，如 STL 和字符串。

- Stroustrup, Bjarne. *The Design and Evolution of C++*。Reading，MA：Addison-Wesley，1994。

如果想了解 C++的演进过程及其为何以这种方式演进，请阅读该书。

- Vandevoorde, David and Nocoli M. Jpsittos C++Templates: The Complete Guide。Reading，MA：Addison-Wesley，2003。

有关模板的内容很多，该参考手册做了详细介绍。

## H.2 网上资源

- ISO/ANSI 2011 C++标准（ISO/IEC 14882:2011）。美国国家标准化组织（ANSI）和国际标准化组织（ISO）都提供。

ANSI 提供 PDF 格式的电子版下载（售价 381 美元），这通过下述网址订购：

http://webstore.ansi.org

ISO 通过下述网址提供该文档的 PDF 文件下载和光盘，售价都是 352 瑞士法郎：

www.iso.org

价格可能有变。

- C++FAQ Lite 站点可以回答常见问题（英语、汉语、法语、俄语和葡萄牙语），它是 Cline 等编著的一本图书的删节版本。当前的网址如下：

http://www.parashift.com/C++-faq-lite

- 在下面的新闻组，可以找到有关 C++问题的比较中肯的讨论：

group:comp.lang.C++.moderated

- 使用 Google、Bing 和其他搜索引擎可找到有关特定 C++主题的信息。

# 附录 I　转换为 ISO 标准 C++

您可能想将一些用 C 或老式 C++版本开发的程序转换为标准 C++，本附录提供了这方面的一些指南。其中的一些内容是关于从 C 转换为 C++的，另一些是关于从老式 C++转换为标准 C++的。

## I.1　使用一些预处理器编译指令的替代品

C/C++预处理器提供了一系列的编译指令。通常，C++惯例是使用这些编译指令来管理编译过程，而避免用编译指令替换代码。例如，#include 编译指令是管理程序文件的重要组件。其他编译指令（如# ifndef 和# endif）使得能够控制是否对特定的代码块进行编译。# pragma 编译指令使得能够控制编译器特定的编译选项。这些都是非常有帮助（有时是必不可少）的工具。但使用# define 编译指令时应谨慎。

### I.1.1　使用 const 而不是#define 来定义常量

符号常量可提高代码的可读性和可维护性。常量名指出了其含义，如果要修改它的值，只需定义修改一次，然后重新编译即可。C 使用预处理器来创建常量的符号名称。

```
#define MAX_LENGTH 100
```

这样，预处理器将在编译之前对源代码执行文本置换，即用 100 替换所有的 MAX_LENGTH。

而 C++则在变量声明使用限定符 const：

```
const int MAX_LENGTH = 100;
```

这样 MAX_LENGTH 将被视为一个只读的 int 变量。

使用 const 的方法有很多优越性。首先，声明显式指明了类型。使用# define 时，必须在数字后加上各种后缀来指出除 char、int 或 double 之外的类型。例如，使用 100L 来表明 long 类型，使用 3.14F 来表明 float 类型。更重要的是，const 方法可以很方便地用于复合类型，如下例所示：

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

最后，const 标识符遵循变量的作用域规则，因此，可以创建作用域为全局、名称空间或数据块的常量。在特定函数中定义常量时，不必担心其定义会与程序的其他地方使用的全局常量冲突。例如，对于下面的代码：

```
#define n 5
const int dz = 12;
...
void fizzle()
{
    int n;
    int dz;
    ...
}
```

预处理器将把：

```
int n;
```

替换为：

```
int 5;
```

从而导致编译错误。而 fizzle()中定义的 dz 是本地变量。另外，必要时，fizzle()可以使用作用域解析运算符（::），以::dz 的方式访问该常量。

虽然 C++借鉴了 C 语言中的关键字 const，但 C++版本更有用。例如，对于外部 const 值，C++版本有内部链接，而不是变量和 C 中 const 所使用的默认外部链接。这意味着使用 const 的程序中的每个文件都必须定义该 const。这好像增加了工作量，但实际上，它使工作更简单。使用内部链接时，可以将 const 定义

放在工程中的各种文件使用的头文件中。对于外部链接，这将导致编译错误，但对于内部链接，情况并非如此。另外，由于 const 必须在使用它的文件中定义（在该文件使用的头文件中定义也满足这样的要求），因此可以将 const 值用作数组长度参数：

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
    loads[i] = 50;
```

这在 C 语言中是行不通的，因为定义 MAX_LENGTH 的声明可能位于一个独立的文件中，在编译时，该文件可能不可用。坦白地说，在 C 语言中，可以使用 static 限定符来创建内部链接常量。也就是说，C++通过默认使用 static，让您可以少记住一件事。

顺便说一句，修订后的 C 标准（C99）允许将 const 用作数组长度，但必须将数组作为一种新式数组——变量数组，而这不是 C++标准的一部分。

在控制何时编译头文件方面，# define 编译指令仍然很有帮助：

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// code goes here
#endif
```

但对于符号常量，习惯上还是使用 const，而不是#define。另一个好方法——尤其是在有一组相关的整型常量时——是使用 enum：

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

### I.1.2 使用 inline 而不是# define 来定义小型函数

在创建类似于内联函数的东西时，传统的 C 语言方式是使用一个#define 宏定义：

```
#define Cube(X) X*X*X
```

这将导致预处理器进行文本置换，将 X 替换为 Cube()的参数：

```
y = Cube(x);       // replaced with y = x*x*x;
y = Cube(x + z++); // replaced with x + z++*x + z++*x + z++;
```

由于预处理器使用文本置换，而不是真正地传递参数，因此使用这种宏可能导致意外的、错误的结果。要避免这种错误，可以在宏中使用大量的圆括号来确保正确的运算顺序：

```
#define Cube(X) ((X)*(X)*(X))
```

但即使这样做，也无法处理使用诸如 Z++等值的情况。

C++方法是使用关键字 inline 来标识内联函数，这种方法更可靠，因为它采用的是真正的参数传递。另外，C++内联函数可以是常规函数，也可以是类方法：

```
class dormant
{
private:
    int period;
    ...
public:
    int Period() const { return period; } // automatically inline
    ...
};
```

#define 宏的一个优点是，它是无类型的，因此将其用于任何类型，运算都是有意义的。在 C++中，可以创建内联模板来使函数独立于类型，同时传递参数。

总之，请使用 C++内联技术，而不是 C 语言中的#define 宏。

## I.2 使用函数原型

实际上，您没有选择的余地。虽然在 C 语言中，原型是可选的，但在 C++中，它确实是必不可少的。请注意，在使用之前定义的函数（如内联函数）是其原型。

应尽可能在函数原型和函数头中使用 const。具体地说，对于表示不可修改的数据的指针参数和引用参数，应使用 const。这不仅使编译器能够捕获修改数据的错误，也使函数更为通用。也就是说，接受 const 指针或引用的函数能够同时处理 const 数据和非 const 数据，而不使用 const 指针或引用的函数只能处理非

const 数据。

## I.3　使用类型转换

Stroustrup 对 C 语言的抱怨之一是其无规律可循的类型转换运算符。确实，类型转换通常是必需的，但标准类型转换太不严格。例如，对于下面的代码：

```
struct Doof
{
    double feeb;
    double steeb;
    char sgif[10];
};
Doof leam;
short * ps = (short *) & leam; // old syntax
int * pi = int * (&leam); // new syntax
```

C 语言不能防止将一种类型的指针转换为另一种完全不相关的类型的指针。

从某种意义上看，这种情况与 goto 语句相似。goto 语句的问题太灵活了，导致代码混乱。解决方法是提供更严格的、结构化程度更高的 goto 版本，来处理需要使用 goto 语句的常见任务，诸如 for 循环、while 循环和 if else 语句等语言元素应运而生。对于类型转换不严格的问题，标准 C++提供了类似的解决方案，即用严格的类型转换来处理最常见的、需要进行类型转换的情况。下面是第 15 章介绍的类型转换运算符：

- dynamic_cast；
- static_cast；
- const_cast；
- reinterpret_cast。

因此，在执行涉及指针的类型转换时，应尽可能使用上述运算符之一。这样做不但可以指出类型转换的目的，并可以检查类型转换是否是按预期那样使用的。

## I.4　熟悉 C++特性

如果使用的是 malloc()和 free()，请改用 new 和 delete；如果是使用 setjmp()和 longjmp()处理错误，则请改用 try、throw 和 catch。另外，对于表示 true 和 false 的值，应将其类型声明为 bool。

## I.5　使用新的头文件

C++标准指定了头文件的新名称，请参见第 2 章。如果使用的是老式头文件，则应当改用新名称。这样做不仅仅是形式上的改变，因为新版本有时新增了特性。例如，头文件 ostream 提供了对宽字符输入和输出的支持，还提供了新的控制符，如 boolalpha 和 fixed（请参见第 17 章）。对于众多格式化选项的设置来说，这些控制符提供的接口比使用 setf()或 iomanip 函数更简单。如果确实使用的是 setf()，则在指定常量时，请使用 ios_base 而不是 ios，即使用 ios_base::fixed 而不是 ios::fixed。另外，新的头文件包含名称空间。

## I.6　使用名称空间

名称空间有助于组织程序中使用的标识符，避免名称冲突。由于标准库是使用新的头文件组织实现的，它将名称放在 std 名称空间中，因此使用这些头文件需要处理名称空间。

出于简化的目的，本书的示例通常使用编译指令 using 来使 std 名称空间中的名称可用：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;  // a using-directive
```

然而，不管需要与否，都导出名称空间中的所有名称，是与名称空间的初衷背道而驰的。

稍微要好些的方法是，在函数中使用 using 编译指令，这将使名称在该函数中可用。

更好也是推荐的方法是，使用 using 声明或作用域解析运算符（::），只使程序需要的名称可用。例如，下面的代码使 cin、cout 和 end1 可用于文件的剩余部分：

```
#include <iostream>
using std::cin;      // a using-declaration
using std::cout;
using std::endl;
```

但使用作用域解析运算符只能使名称在使用该运算符的表达式中可用：

```
cout << std::fixed << x << endl;  //using the scope resolution operator
```

这样做可能很麻烦，但可以将通用的 using 声明放在一个头文件中：

```
// mynames – a header file
using std::cin;           // a using-declaration
using std::cout;
using std::endl;
```

还可以将通用的 using 声明放在一个名称空间中：

```
// mynames – a header file
#include <iostream>

namespace io
{
    using std::cin;
    using std::cout;
    using std::endl;
}

namespace formats
{
    using std::fixed;
    using std::scientific;
    using std:boolalpha;
}
```

这样，程序可以包含该文件，并使用所需的名称空间：

```
#include "mynames"
using namespace io;
```

## I.7 使用智能指针

每个 new 都应与 delete 配对使用。如果使用 new 的函数由于引发异常而提前结束，将导致问题。正如第 15 章介绍的，使用 autoptr 对象跟踪 new 创建的对象将自动完成 delete 操作。C++11 新增的 unique_ptr 和 shared_ptr 提供了更佳的替代方案。

## I.8 使用 string 类

传统的 C 风格字符串深受不是真正的类型之苦。可以将字符串存储在字符数组中，也可以将字符数组初始化为字符串。但不能使用赋值运算符将字符串赋给字符数组，而必须使用 strcpy()或 strncpy()。不能使用关系运算符来比较 C 风格字符串，而必须使用 strcmp()（如果忘记了这一点，使用了>运算符，将不会出现语法错误，程序将比较字符串的地址，而不是字符串的内容）。

而 string 类（参见第 16 章和附录 F）使得能够使用对象来表示字符串，并定义了赋值运算符、关系运算符和加法运算符（用于拼接）。另外，string 类还提供了自动内存管理功能，因此通常不用担心字符串被保存前，有人可能会跨越数组边界或将字符串截短。

String 类提供了许多方便的方法。例如，可以将一个 string 对象追加到另一个对象的后面，也可以将 C 风格的字符串，甚至 char 值追加到 string 对象的后面。对于接受 C 风格字符串参数的函数，可以使用 c_str()方法来返回一个适当的 char 指针。

string 类不仅提供了一组设计良好的方法来处理与字符串相关的工作（如查找子字符串），而且与 STL 兼容，因此，可以将 STL 算法用于 string 对象。

## I.9　使用 STL

标准模板库（请参见第 16 章和附录 G）为许多编程需要提供了现成的解决方案，应使用它。例如，与其声明一个 double 或 string 对象数组，不如创建 vector<double>对象或 vector<string>对象。这样做的好处与使用 string 对象（而不是 C 风格字符串）相似。赋值运算符已被定义，因此可以使用赋值运算符将一个 vector 对象赋给另一个 vector 对象。可以按引用传递 vector 对象，接收这种对象的函数可以使用 size()方法来确定 vector 对象中元素数目。内置的内存管理功能使得当使用 pushback()方法在 vector 对象中添加元素时，其大小将自动调整。当然，还可以根据实际需要来使用其他有用的类方法和通用算法。在 C++11 中，如果长度固定的数组是更好的解决方案，可使用 array<double>或 array<string>。

如果需要链表、双端队列（或队列）、栈、常规队列、集合或映射，应使用 STL，它提供了有用的容器模板。算法库使得可以将矢量的内容轻松地复制到链表中，或将集合的内容同矢量进行比较。这种设计使得 STL 成为一个工具箱，它提供了基本部件，可以根据自己的需要进行装配。

在设计内容广泛的算法库时，效率是一个主要的设计目标，因此只需要完成少量的编程工作，便可以得到最好的结果。另外，实现算法时使用了迭代器的概念，这意味着这些算法不仅可用于 STL 容器。具体地说，它们也可用于传统数组。

# 附录 J　复习题答案

## 第 2 章复习题答案

1. 它们叫作函数。

2. 这将导致在最终的编译之前，使用 iostream 文件的内容替换该编译指令。

3. 它使得程序可以使用 std 名称空间中的定义。

4. `cout << "Hello, world\n";`

或

```
cout << "Hello, world" << endl;
```

5. `int cheeses;`

6. `cheeses = 32;`

7. `cin >> cheeses;`

8. `cout << "We have " << cheeses << " varieties of cheese\n";`

9. 调用函数 froop()时，应提供一个参数，该参数的类型为 double，而该函数将返回一个 int 值。例如，可以像下面这样使用它：

```
int gval = froop(3.14159);
```

函数 rattle()接受一个 int 参数且没有返回值。例如，可以这样使用它：

```
rattle(37);
```

函数 prune()不接受任何参数且返回一个 int 值。例如，可以这样使用它：

```
int residue = prune();
```

10. 当函数的返回类型为 void 时，不用在函数中使用 return。然而，如果不提供返回值，则可以使用它：

```
return;
```

## 第 3 章复习题答案

1. 有多种整型类型，可以根据特定需求选择最适合的类型。例如，可以使用 short 来存储空格，使用 long 来确保存储容量，也可以寻找可提高特定计算的速度的类型。

2.
```
short rbis = 80;        // or short int rbis = 80;
unsigned int q = 42110; // or unsigned q = 42110;
unsigned long ants = 3000000000;
// or long long ants = 3000000000;
```

注意：不要指望 int 变量能够存储 3000000000；另外，如果系统支持通用的列表初始化，可使用它：

```
short rbis = {80};          // = is optional
unsigned int q {42110};  // could use = {42110}
long long ants {3000000000};
```

3. C++没有提供自动防止超出整型限制的功能，可以使用头文件 climits 来确定限制情况。

4. 常量 33L 的类型为 long，常量 33 的类型为 int。

5. 这两条语句并不真正等价，虽然对于某些系统来说，它们是等效的。最重要的是，只有在使用 ASCII 码的系统上，第一条语句才将得分设置为字母 A，而第二条语句还可用于使用其他编码的系统。其次，65 是一个 int 常量，而‘A’是一个 char 常量。

6. 下面是 4 种方式：

```
char c = 88;
cout << c << endl;          // char type prints as character

cout.put(char(88));         // put() prints char as character
```

```
cout << char(88) << endl;  // new-style type cast value to char

cout << (char)88 << endl;  // old-style type cast value to char
```

7. 这个问题的答案取决于这两个类型的长度。如果 long 为 4 个字节，则没有损失。因为最大的 long 值将是 20 亿，即有 10 位数。由于 double 提供了至少 13 位有效数字，因而不需要进行任何舍入。long long 类型可提供 19 位有效数字，超过了 double 保证的 13 位有效数字。

8.
```
a. 8 * 9 + 2 is 72 + 2 is 74
b. 6 * 3 / 4 is 18 / 4 is 4
c. 3 / 4 * 6 is 0 * 6 is 0
d. 6.0 * 3 / 4 is 18.0 / 4 is 4.5
e. 15 % 4 is 3
```

9. 下面的代码都可用于完成第一项任务：

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

要将它们作为 double 类型相加，再进行转换，可采取下述方式之一：

```
int pos = (int) (x1 + x2);
int pos = int(x1 + x2);
```

10.
```
a. int
b. float
c. char
d. char32_t
e. double
```

### 第 4 章复习题答案

1.
```
a. char actors[30];
b. short betsie[100];
c. float chuck[13];
d. long double dipsea[64];
```

2.
```
a. array<char,30> actors;
b. array<short, 100> betsie;
c. array<float, 13> chuck;
d. array<long double, 64> dipsea;
```

3.
```
int oddly[5] = {1, 3, 5, 7, 9};
```

4.
```
int even = oddly[0] + oddly[4];
```

5.
```
cout << ideas[1] << "\n"; // or << endl;
```

6.
```
char lunch[13] = "cheeseburger";// number of characters + 1
```

或者

```
char lunch[] = "cheeseburger"; // let the compiler count elements
```

7.
```
string lunch = "Waldorf Salad";
```

如果没有 using 编译指令，则为：

```
std::string lunch = "Waldorf Salad";
```

8.
```
struct fish {
    char kind[20];
    int weight;
    float length;
};
```

9.
```
fish petes =
{
    "trout",
    12,
    26.25
};
```

10.
```
enum Response {No, Yes, Maybe};
```

11.
```
double * pd = &ted;
cout << *pd << "\n";
```

12.
```
float * pf = treacle; // or = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
            // or use *pf and *(pf + 9)
```

13. 这里假设已经包含了头文件 iostream 和 vector，并有一条 using 编译指令：

```
unsigned int size;
cout << "Enter a positive integer: ";
```

```
cin >> size;
int * dyn = new int [size];
vector<int> dv(size);
```

14. 是的，它是有效的。表达式“home of the jolly bytes”是一个字符串常量，因此，它将判定为字符串开始的地址。cout 对象将 char 地址解释为打印字符串，但类型转换(int *)将地址转换为 int 指针，然后作为地址被打印。总之，该语句打印字符串的地址，只要 int 类型足够宽，能够存储该地址。

15.
```
struct fish
{
    char kind[20];
    int weight;
    float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

16. 使用 cin >>address 将使得程序跳过空白，直到找到非空白字符为止。然后它将读取字符，直到再次遇到空白为止。因此，它将跳过数字输入后的换行符，从而避免这种问题。另一方面，它只读取一个单词，而不是整行。

17.
```
#include <string>
#include <vector>
#include <array>
const int Str_num {10}; // or = 10
...
std::vector<std::string> vstr(Str_num);
std::array<std::string, Str_num> astr;
```

### 第 5 章复习题答案

1. 输入条件循环在进入输入循环体之前将评估测试表达式。如果条件最初为 false，则循环不会执行其循环体。退出条件循环在处理循环体之后评估测试表达式。因此，即使测试表达式最初为 false，循环也将执行一次。for 和 while 循环都是输入条件循环，而 do while 循环是退出条件循环。

2. 它将打印下面的内容：
```
01234
```
注意，cout<< endl;不是循环体的组成部分，因为没有大括号。

3. 它将打印下面的内容：
```
0369
12
```

4. 它将打印下面的内容：
```
6
8
```

5. 它将打印下面的内容：
```
k = 8
```

6. 使用*= 运算符最简单：
```
for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
```

7. 将语句放在一对大括号中将形成一个复合语句或代码块。

8. 当然，第一条语句是有效的。表达式 1，024 由两个表达式组成——1 和 024，用逗号运算符连接。值为右侧表达式的值。这是 024，八进制为 20，因此该声明将值 20 赋给 X。第二条语句也是有效的。然而，运算符优先级将导致它被判定成这样：
```
(y = 1), 024;
```
也就是说，左侧表达式将 y 设置成 1，整个表达式的值（没有使用）为 024 或 20（八进制）。

9. cin>> ch 将跳过空格、换行符和制表符，其他两种格式将读取这些字符。

### 第 6 章复习题答案

1. 这两个版本将给出相同的答案，但 if else 版本的效率更高。例如，考虑当 ch 为空格时的情况。版本 1 对空格加 1，然后看它是否为换行符。这将浪费时间，因为程序已经知道 ch 为空格，因此它不是换行符。在这种情况下，版本 2 将不会查看字符是否为换行符。

2. ++ch 和 ch + 1 得到的数值相同。但++ch 的类型为 char，将作为字符打印，而 ch + 1 是 int 类型（因为将 char 和 int 相加），将作为数字打印。

3. 由于程序使用 ch = '$'，而不是 ch = ='$'，因此输入和输出将如下：

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

在第二次打印前，每个字符都被转换为$字符。另外，表达式 ch=$的值为$字符的编码，因此它是非 0 值，因而为 true;所以每次 ct2 将被加 1。

4.
```
a. weight >= 115 && weight < 125
b. ch == 'q' || ch == 'Q'
c. x % 2 == 0 && x != 26
d. x % 2 == 0 && !(x % 26 == 0)
e. donation >= 1000 && donation <= 2000 || guest == 1
f. (ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')
```

5. 不一定。例如，如果 x 为 10，则!x 为 0，!!x 为 1。然而，如果 x 为 bool 变量，则!!x 为 x。

6.
```
(x < 0)? -x : x
```

或

```
(x >= 0)? x : -x;
```

7.
```
switch (ch)
   {
      case 'A': a_grade++;
                break;
      case 'B': b_grade++;
                break;
      case 'C': c_grade++;
                break;
      case 'D': d_grade++;
                break;
      default: f_grade++;
                break;
}
```

8. 如果使用整数标签，且用户输入了非整数（如 q），则程序将因为整数输入不能处理字符而挂起。但是，如果使用字符标签，而用户输入了整数（如 5），则字符输入将 5 作为字符处理。然后，switch 语句的 default 部分将提示输入另一个字符。

9. 下面是一个版本：

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
    if (ch == '\n')
        line++;
}
```

### 第 7 章复习题答案

1. 这 3 个步骤是定义函数、提供原型、调用函数。

2.
```
a. void igor(void);   // or void igor()
b. float tofu(int n); // or float tofu(int);
c. double mpg(double miles, double gallons);
d. long summation(long harray[], int size);
e. double doctor(const char * str);
f. void ofcourse(boss dude);
g. char * plot(map *pmap);
```

3.
```
void set_array(int arr[], int size, int value)
{
   for (int i = 0; i < size; i++)
       arr[i] = value;
}
```

4.
```
void set_array(int * begin, int * end, int value)
{
   for (int * pt = begin; pt != end; pt++)
       pt* = value;
}
```

5.
```
double biggest (const double foot[], int size)
{
    double max;
    if (size < 1)
    {
       cout << "Invalid array size of " << size << endl;
       cout << "Returning a value of 0\n";
       return 0;
}
else     // not necessary because return terminates program
{
       max = foot[0];
       for (int i = 1; i < size; i++)
          if (foot[i] > max)
              max = foot[i];
       return max;
   }
}
```

6. 将 const 限定符用于指针，以防止指向的原始数据被修改。程序传递基本类型（如 int 或 double）时，它将按值传递，以便函数使用副本。这样，原始数据将得到保护。

7. 字符串可以存储在 char 数组中，可以用带双引号的字符串来表示，也可以用指向字符串第一个字符的指针来表示。

8.
```
int replace(char * str, char c1, char c2)
{
    int count = 0;
    while (*str) // while not at end of string
    {
        if (*str == c1)
        {
            *str = c2;
            count++;
        }
        str++; // advance to next character
    }
    return count;
}
```

9. 由于 C++将“pizza”解释为其第一个元素的地址，因此使用*运算符将得到第一个元素的值，即字符 p。由于 C++将“taco”解释为第一个元素的地址，因此它将“taco”[2]解释为第二个元素的值，即字符 c。换句话来说，字符串常量的行为与数组名相同。

10. 要按值传递它，只要传递结构名 glitz 即可。要传递它的地址，请使用地址运算符&glitz。按值传递将自动保护原始数据，但这是以时间和内存为代价的。按地址传递可节省时间和内存，但不能保护原始数据，除非对函数参数使用了 const 限定符。另外，按值传递意味着可以使用常规的结构成员表示法，但传递指针则必须使用间接成员运算符。

11.
```
int judge (int (*pf)(const char *));
```

12. a. 注意，如果 ap 是一个 applicant 结构，则 ap.credit_ratings 就是一个数组名，而 ap.credit_ratings[i]是一个数组元素：
```
 void display(applicant ap)
{
    cout << ap.name << endl;
    for (int i = 0; i < 3; i++)
        cout << ap.credit_ratings[i] << endl;
}
```

b. 注意，如果 pa 是一个指向 applicant 结构的指针，则 pa->credit_ratings 就是一个数组名，而 pa->credit_ratings[i]是一个数组元素：
```
void show(const applicant * pa)
{
   cout << pa->name << endl;
   for (int i = 0; i < 3; i++)
       cout << pa->credit_ratings[i] << endl;
}
```

13.
```
typedef void (*p_f1)(applicant *);
 p_f1 p1 = f1;
 typedef const char * (*p_f2)(const applicant *, const applicant *);
```

```
p_f2 p2 = f2;
p_f1 ap[5];
p_f2 (*pa)[10];
```

### 第 8 章复习题答案

1. 只有一行代码的小型、非递归函数适合作为内联函数。
2. a. void song(const char * name，int times = 1);
   b. 没有。只有原型包含默认值的信息。
   c. 是的，如果保留 times 的默认值：

```
void song(char * name = "O, My Papa", int times = 1);
```

3. 可以使用字符串"\""或字符'"'来打印引号，下面的函数演示了这两种方法。

```
#include <iostream.h>
void iquote(int n)
{
    cout << "\"" << n << "\"";
}

void iquote(double x)
{
    cout << '"' << x << '"';
}

void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4. a. 该函数不应修改结构成员，所以使用 const 限定符。

```
 void show_box(const box & container)
{
    cout << "Made by " << container. maker << endl;
    cout << "Height = " << container.height << endl;
    cout << "Width = " << container.width << endl;
    cout << "Length = " << container.length << endl;
    cout << "Volume = " << container.volume << endl;
}
b. void set_volume(box & crate)
   {
      crate.volume = crate.height * crate.width * crate.length;
   }
```

5. 首先，将原型修改成下面这样：

```
// function to modify array object
void fill(std::array<double, Seasons> & pa);
// function that uses array object without modifying it
void show(const std::array<double, Seasons> & da);
```

注意，show()应使用 const，以禁止修改对象。

接下来，在 main()中，将 fill()调用改为下面这样：

```
fill(expenses);
```

函数 show()的调用不需要修改。

接下来，新的 fill()应类似于下面这样：

```
void fill(std::array<double, Seasons> & pa) // changed
{
    using namespace std;
    for (int i = 0; i < Seasons; i++)
    {
        cout << "Enter " << Snames[i] << " expenses: ";
        cin >> pa[i]; // changed
    }
}
```

注意到(*pa)[i]变成了更简单的 pa[i]。

最后，修改 show()的函数头：

```
void show(std::array<double, Seasons> & da)
```

6. a. 通过为第二个参数提供默认值：

```
double mass(double d, double v = 1.0);
```

也可以通过函数重载：

```
double mass(double d, double v);
double mass(double d);
```

b. 不能为重复的值使用默认值，因为必须从右到左提供默认值。可以使用重载：

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

c. 可以使用函数重载：

```
int average(int a, int b);
double average(double x, double y);
```

d. 不能这样做，因为两个版本的特征标将相同。

7.
```
template<class T>
T max(T t1, T t2) // or T max(const T & t1, const T & t2)
{
    return t1 > t2? t1 : t2;
}
```

8.
```
template<> box max(box b1, box b2)
{
    return b1.volume > b2.volume? b1 : b2;
}
```

9. v1 的类型为 float，v2 的类型为 float &，v3 的类型为 float &，v4 的类型为 int，v5 的类型为 double。字面值 2.0 的类型为 double，因此表达式 2.0 * m 的类型为 double。

### 第 9 章复习题答案

1. a. homer 将自动成为自动变量。

b. 应该在一个文件中将 secret 定义为外部变量，并在第二个文件中使用 extern 来声明它。

c. 可以在外部定义前加上关键字 static，将 topsecret 定义为一个有内部链接的静态变量。也可在一个未命名的名称空间中进行定义。

d. 应在函数中的声明前加上关键字 static，将 beencalled 定义为一个本地静态变量。

2. using 声明使得名称空间中的单个名称可用，其作用域与 using 所在的声明区域相同。using 编译指令使名称空间中的所有名称可用。使用 using 编译指令时，就像在一个包含 using 声明和名称空间本身的最小声明区域中声明了这些名称一样。

3.
```
#include <iostream>
int main()
{
    double x;
    std::cout << "Enter value: ";
    while (! (std::cin >> x) )
    {
        std::cout << "Bad input. Please enter a number: ";
        std::cin.clear();
       while (std::cin.get() != '\n')
           continue;
    }
    std::cout << "Value = " << x << std::endl;
    return 0;
}
```

4. 下面是修改后的代码：

```
#include <iostream>
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
    double x;
    cout << "Enter value: ";
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get() != '\n')
            continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5. 可以在每个文件中包含单独的静态函数定义。或者每个文件都在未命名的名称空间中定义一个合适

的 average()函数。

6. 
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7. 
```
1
4, 1, 2
2
2
4, 1, 2
2
```

**第 10 章复习题答案**

1. 类是用户定义的类型的定义。类声明指定了数据将如何存储，同时指定了用来访问和操纵这些数据的方法（类成员函数）。

2. 类表示人们可以类方法的公有接口对类对象执行的操作，这是抽象。类的数据成员可以是私有的（默认值），这意味着只能通过成员函数来访问这些数据，这是数据隐藏。实现的具体细节（如数据表示和方法的代码）都是隐藏的，这是封装。

3. 类定义了一种类型，包括如何使用它。对象是一个变量或其他数据对象（如由 new 生成的），并根据类定义被创建和使用。类和对象之间的关系同标准类型与其变量之间的关系相同。

4. 如果创建给定类的多个对象，则每个对象都有其自己的数据内存空间;但所有的对象都使用同一组成员函数（通常，方法是公有的，而数据是私有的，但这只是策略方面的问题，而不是对类的要求）。

5. 这个示例使用 char 数组来存储字符数据，但可以使用 string 类对象。

```
// #include <cstring>

// class definition
class BankAccount
{
private:
     char name[40];    // or std::string name;
     char acctnum[25]; // or std::string acctnum;
     double balance;
public:
     BankAccount(const char * client, const char * num, double bal = 0.0);
//or BankAccount(const std::string & client,
//               const std::string & num, double bal = 0.0);
     void show(void) const;
     void deposit(double cash);
     void withdraw(double cash);
};
```

6. 在创建类对象或显式调用构造函数时，类的构造函数都将被调用。当对象过期时，类的析构函数将被调用。

7. 有两种可能的解决方案（要使用 strncpy()，必须包含头文件 cstring 或 string.h;要使用 string 类，必须包含头文件 string）：

```
BankAccount::BankAccount(const char * client, const char * num, double bal)
{
    strncpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

或者：

```
BankAccount::BankAccount(const std::string & client,
                         const std::string & num, double bal)
{
    name = client;
    acctnum = num;
    balance = bal;
}
```

请记住，默认参数位于原型中，而不是函数定义中。

8. 默认构造函数是没有参数或所有参数都有默认值的构造函数。拥有默认构造函数后，可以声明对象，而不初始化它，即使已经定义了初始化构造函数。它还使得能够声明数组。

9.
```
// stock30.h
#ifndef STOCK30_H_
#define STOCK30_H_

class Stock
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();    // default constructor
    Stock(const std::string & co, long n, double pr);
    ~Stock() {} // do-nothing destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    const string & co_name() const { return company; }
};
```

10. this 指针是类方法可以使用的指针，它指向用于调用方法的对象。因此，this 是对象的地址，*this 是对象本身。

### 第 11 章复习题答案

1. 下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
Stonewt operator*(double mult);

// definition – let constructor do the work
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2. 成员函数是类定义的一部分，通过特定的对象来调用。成员函数可以隐式访问调用对象的成员，而无需使用成员运算符。友元函数不是类的组成部分，因此被称为直接函数调用。友元函数不能隐式访问类成员，而必须将成员运算符用于作为参数传递的对象。请比较复习题 1 和复习题 4 的答案。

3. 要访问私有成员，它必须是友元，但要访问公有成员，可以不是友元。

4. 下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
friend Stonewt operator*(double mult, const Stonewt & s);

// definition – let constructor do the work
Stonewt operator*(double mult, const Stonewt & s)
{
    return Stonewt(mult * s.pounds);
}
```

5. 下面的 5 个运算符不能重载：

- sizeof。
- .。
- .*。
- ::。
- ?:。

6. 这些运算符必须使用成员函数来定义。

7. 下面是一个可能的原型和定义：

```
// prototype and inline definition
operator double () {return mag;}
```

但请注意，使用 magval()方法比定义该转换函数更符合逻辑。

### 第 12 章复习题答案

1. a. 语法是正确的，但该构造函数没有将 str 指针初始化。该构造函数应将指针设置成 NULL 或使用 new [ ]来初始化它。

b. 该构造函数没有创建新的字符串，而只是复制了原有字符串的地址。它应当使用 new [ ]和 strcpy()。

c. 它复制了字符串，但没有给它分配存储空间，应使用 new char[len + 1]来分配适当数量的内存。

2. 首先，当这种类型的对象过期时，对象的成员指针指向的数据仍将保留在内存中，这将占用空间，同时不可访问，因为指针已经丢失。可以让类析构函数删除构造函数中 new 分配的内存，来解决这种问题。其次，析构函数释放这种内存后，如果程序将这样的对象初始化为另一个对象，则析构函数将试图释放这些内存两次。这是因为将一个对象初始化为另一个对象的默认初始化，将复制指针值，但不复制指向的数据，这将使两个指针指向相同的数据。解决方法是，定义一个复制构造函数，使初始化复制指向的数据。第三，将一个对象赋给另一个对象也将导致两个指针指向相同的数据。解决方法是重载赋值运算符，使之复制数据，而不是指针。

3. C++自动提供下面的成员函数：

- 如果没有定义构造函数，将提供默认构造函数。
- 如果没有定义复制构造函数，将提供复制构造函数。
- 如果没有定义赋值运算符，将提供赋值运算符。
- 如果没有定义析构函数，将提供默认析构函数。
- 如果没有定义地址运算符，将提供地址运算符。

默认构造函数不完成任何工作，但使得能够声明数组和未初始化的对象。默认复制构造函数和默认赋值运算符使用成员赋值。默认析构函数也不完成任何工作。隐式地址运算符返回调用对象的地址（即 this 指针的值）。

4. 应将 personality 成员声明为字符数组或 car 指针，或者将其声明为 String 对象或 string 对象。该声明没有将方法设置为公有的，因此会有几个小错误。下面是一种可能的解决方法，修改的地方以粗体显示：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char personality[40]; // provide array size
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    friend ostream & operator<<(ostream & os, const nifty & n);
};  // note closing semicolon

nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}

nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
```

```
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

下面是另一种解决方案：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char * personality; // create a pointer
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    nifty(const nifty & n);
    ~nifty() { delete [] personality; }
    nifty & operator=(const nifty & n) const;
    friend ostream & operator<<(ostream & os, const nifty & n);
};   // note closing semicolon

nifty::nifty()
{
    personality = NULL;
    talents = 0;
}

nifty::nifty(const char * s)
{
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

5. a.

```
Golfer nancy; // default constructor
Golfer lulu("Little Lulu"); // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer; // default constructor
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker"; // Golfer(const char * name, int g)
*par = nancy; // default assignment operator
nancy = "Nancy Putter";// Golfer(const char * name, int g), then
                  // the default assignment operator
```

注意，对于语句 5 和 6，有些编译器还将调用默认的赋值运算符。

b. 类应定义一个复制数据（而不是地址）的赋值运算符。

### 第 13 章复习题答案

1. 基类的公有成员成为派生类的公有成员。基类的保护成员成为派生类的保护成员。基类的私有成员被继承，但不能直接访问。复习题 2 的答案提供了这些通用规定的特例。

2. 不能继承构造函数、析构函数、赋值运算符和友元。

3. 如果返回的类型为 void，仍可以使用单个赋值，但不能使用连锁赋值：

```
baseDMA magazine("Pandering to Glitz", 1);
baseDMA gift1, gift2, gift3;
gift1 = magazine;       // ok
gift 2 = gift3 = gift1; // no longer valid
```

如果方法返回一个对象，而不是引用，则该方法的执行速度将有所减慢，这是因为返回语句需要复制对象。

4. 按派生的顺序调用构造函数，最早的构造函数最先调用。调用析构函数的顺序正好相反。

5. 是的，每个类都必须有自己的构造函数。如果派生类没有添加新成员，则构造函数可以为空，但必须存在。

6. 只调用派生类方法。它取代基类定义。仅当派生类没有重新定义方法或使用作用域解析运算符时，才会调用基类方法。然而，应把将所有要重新定义的函数声明为虚函数。

7. 如果派生类构造函数使用 new 或 new[ ]运算符来初始化类的指针成员，则应定义一个赋值运算符。更普遍地说，如果对于派生类成员来说，默认赋值不正确，则应定义赋值运算符。

8. 当然，可以将派生类对象的地址赋给基类指针;但只有通过显式类型转换，才可以将基类对象的地址赋给派生类指针（向下转换），而使用这样的指针不一定安全。

9. 是的，可以将派生类对象赋给基类对象。对于派生类中新增的数据成员都不会传递给基类对象。然而，程序将使用基类的赋值运算符。仅当派生类定义了转换运算符（即包含将基类引用作为唯一参数的构造函数）或使用基类为参数的赋值运算符时，相反方向的赋值才是可能的。

10. 它可以这样做，因为 C++允许基类引用指向从该基类派生而来的任何类型。

11. 按值传递对象将调用复制构造函数。由于形参是基类对象，因此将调用基类的复制构造函数。复制构造函数以基类引用为参数，该引用可以指向作为参数传递的派生对象。最终结果是，将生成一个新的基类对象，其成员对应于派生对象的基类部分。

12. 按引用（而不是按值）传递对象，这样可以确保函数从虚函数受益。另外，按引用（而不是按值）传递对象可以节省内存和时间，尤其对于大型对象。按值传递对象的主要优点在于可以保护原始数据，但可以通过将引用作为 const 类型传递，来达到同样的目的。

13. 如果 head()是一个常规方法，则 ph->head()将调用 Corporation::head();如果 head()是一个虚函数，则 ph->head()将调用 PublicCorporation::head()。

14. 首先，这种情况不符合 is-a 模型，因此公有继承不适用。其次，House 中的 area()定义隐藏了 area()的 Kitchen 版本，因为这两个方法的特征标不同。

### 第 14 章复习题答案

1.

| Class Bear | class PolarBear | 公有，北极熊是一种熊 |
|---|---|---|
| class Kitchen | class Home | 私有，家里有厨房 |
| class Person | class Programmer | 公有，程序员是一种人 |
| class Person | class HorseAndJockey | 私有，马和驯马师的组合中包含一个人 |
| class Person<br>class Automobile | class Driver | 人是公有的，因为司机是一个人；汽车是私有的，因为司机有一辆汽车 |

2.
```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    fb.tell();
    cout << glip << endl;
}
```

3.
```
Gloam::Gloam(int g, const char * s)
            : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
            : glip(g), Frabjous(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    Frabjous::tell();
    cout << glip << endl;
}
```

4.
```
class Stack<Worker *>
{
private:
    enum {MAX = 10};     // constant specific to class
    Worker * items[MAX]; // holds stack items
    int top;             // index for top stack item
```

```
    public:
        Stack();
        Boolean isempty();
        Boolean isfull();
        Boolean push(const Worker * & item); // add item to stack
        Boolean pop(Worker * & item);        // pop top into item
    };
```

5. 
```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

程序清单 14.18 生成 4 个模板：ArrayTP<int, 10>、ArrayTP<double, 10>、ArrayTP<int, 5>和 Array<ArrayTP<int, 5>, 10>。

6. 如果两条继承路线有相同的祖先，则类中将包含祖先成员的两个拷贝。将祖先类作为虚基类可以解决这种问题。

### 第 15 章复习题答案

1. a. 友元声明如下：

```
friend class clasp;
```

b. 这需要一个前向声明，以便编译器能够解释 void snip（muff&）：

```
    snip(muff &):
    class muff; // forward declaration
    class cuff {
    public:
        void snip(muff &) { ... }
        ...
    };
    class muff {
        friend void cuff::snip(muff &);
        ...
};
```

c. 首先，cuff 类声明应在 muff 类之前，以便编译器可以理解 cuff::snip()。其次，编译器需要 muff 的一个前向声明，以便可以理解 snip(muff &)。

```
    class muff; // forward declaration
    class cuff {
    public:
        void snip(muff &) { ... }
        ...
    };
    class muff {
        friend void cuff::snip(muff &);
        ...
};
```

2. 不。为使类 A 拥有一个本身为类 B 的成员函数的友元，B 的声明必须位于 A 的声明之前。一个前向声明是不够的，因为这种声明可以告诉 A：B 是一个类;但它不能指出类成员的名称。同样，如果 B 拥有一个本身是 A 的成员函数的友元，则 A 的这个声明必须位于 B 的声明之前。这两个要求是互斥的。

3. 访问类的唯一方法是通过其有接口，这意味着对于 Sauce 对象，只能调用构造函数来创建一个。其他成员（soy 和 sugar）在默认情况下是私有的。

4. 假设函数 f1()调用函数 f2()。f2()中的返回语句导致程序执行在函数 f1()中调用函数 f2()后面的一条语句。throw 语句导致程序沿函数调用的当前序列回溯，直到找到直接或间接包含对 f2()的调用的 try 语句块为止。它可能在 f1()中、调用 f1()的函数中或其他函数中。找到这样的 try 语句块后，将执行下一个匹配的 catch 语句块，而不是函数调用后的语句。

5. 应按从子孙到祖先的顺序排列 catch 语句块。

6. 对于示例#1，如果 pg 指向一个 Superb 对象或从 Superb 派生而来的任何类的对象，则 if 条件为 true。具体地说，如果 pg 指向 Magnificent 对象，则 if 条件也为 true。对于示例#2，仅当指向 Superb 对象时，if 条件才为 true，如果指向的是从 Superb 派生出来的对象，则 if 条件不为 true。

7. Dynamic_cast 运算符只允许沿类层次结构向上转换，而 static_cast 运算符允许向上转换和向下转换。static_cast 运算符还允许枚举类型和整型之间以及数值类型之间的转换。

### 第 16 章复习题答案

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
    string st; // a string object
public:
    RQ1() : st("") {}
    RQ1(const char * s) : st(s) {}
    ~RQ1() {};
// more stuff
};
```

不再需要显式复制构造函数、析构程序和赋值运算符，因为 string 对象提供了自己的内存管理功能。

2. 可以将一个 string 对象赋给另一个。string 对象提供了自己的内存管理功能，所以一般不需要担心字符串超出存储容量。

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia= new int[20]; // wrong, use with new, not new[]
auto_ptr<string>(new string);   // wrong, no name for pointer
int rigue = 7;
auto_ptr<int>(&rigue);           // wrong, memory not allocated by new
auto_ptr dbl (new double);       // wrong, omits <double>
```

5. 栈的 LIFO 特征意味着可能必须在到达所需要的球棍（club）之前删除很多球棍。

6. 集合将只存储每个值的一个拷贝，因此，5 个 5 分将被存储为 1 个 5 分。

7. 使用迭代器使得能够使用接口类似于指针的对象遍历不以数组方式组织的数据，如双向链表中的数据。

8. STL 方法使得可以将 STL 函数用于指向常规数组的常规指针以及指向 STL 容器类的迭代器，因此提高了通用性。

9. 可以将一个 vector 对象赋给另一个。vector 管理自己的内存，因此可以将元素插入到矢量中，并让它自动调整长度。使用 at()方法，可以自动检查边界。

10. 这两个 vector 函数和 random_shuffle()函数要求随机访问迭代器，而 list 对象只有双向迭代器。可以使用 list 模板类的 sort()成员函数（参见附录 G），而不是通用函数来排序，但没有与 random_shuffle()等效的成员函数。然而，可以将链表复制到矢量中，然后打乱矢量，并将结果重新复制到链表中。

### 第 17 章复习题答案

1. iostream 文件定义了用于管理输入和输出的类、常量和操纵符，这些对象管理用于处理 I/O 的流和缓冲区。该文件还创建了一些标准对象（cin、cout、cerr 和 clog 以及对应的宽字符对象），用于处理与每个程序相连的标准输入和输出流。

2. 键盘输入生成一系列字符。输入 121 将生成 3 个字符，每个字符都由一个 1 字节的二进制码表示。要将这个值存储为 int 类型，则必须将这 3 个字符转换为 121 值的二进制表示。

3. 在默认情况下，标准输出和标准错误都将输出发送给标准输出设备（通常为显示器）。然而，如果要求操作系统将输出重定向到文件，则标准输出将与文件（而不是显示器）相连，但标准错误仍与显示器相连。

4. ostream 类为每种 C++基本类型定义了一个 operator <<()函数的版本。编译器将下面的表达式：
```
cout << spot
```
解释为：
```
cout.operator<<(spot)
```

这样，它便能够将该方法调用与具有相同参数类型的函数原型匹配。

5. 可以将返回 ostream &类型的输出方法拼接。这样，通过一个对象调用方法时，将返回该对象。然后，返回对象将可以调用序列中的下一个方法。

6. //rq17-6.cpp

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << "Enter an integer: ";
    int n;
    cin >> n;
    cout << setw(15) << "base ten" << setw(15)
           << "base sixteen" << setw(15) << "base eight" << "\n";
    cout.setf(ios::showbase); // or cout << showbase;
    cout << setw(15) << n << hex << setw(15) << n
           << oct << setw(15) << n << "\n";
    return 0;
}
```

7. //rq17-7.cpp

```
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
// or cout << showpoint << fixed << right;
    cout << "First format:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    return 0;
}
```

8. 下面是输出：

```
ct1 = 5; ct2 = 9
```

该程序的前半部分忽略空格和换行符，而后半部分没有。注意，程序的后半部分从第一个 q 后面的换行符开始读取，将换行符计算在内。

9. 如果输入行超过 80 个字符，ignore()将不能正常工作。在这种情况下，它将跳过前 80 个字符。

### 第 18 章复习题答案

1.

```
class Z200
{
private:
    int j;
    char ch;
    double z;
public:
```

```
    Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
...
};

double x {8.8}; // or = {8.8}
std::string s {"What a bracing effect!"};
int k{99};
Z200 zip{200,'Z',0.67});
std::vector<int> ai {3, 9, 4, 7, 1};
```

2. r1(w)合法，形参 rx 指向 w。

r1(w+1)合法，形参 rx 指向一个临时变量，这个变量被初始化为 w+1。

r1(up(w))合法，形参 rx 指向一个临时变量，这个变量被初始化为 up(w)的返回值。

一般而言，将左值传递给 const 左值引用参数时，参数将被初始化为左值。将右值传递给函数时，const 左值引用参数将指向右值的临时拷贝。

r2(w)合法，形参 rx 指向 w。

r2(w+1)非法，因为 w+1 是一个右值。

r2(up(w))非法，因为 up(w)的返回值是一个右值。

一般而言，将左值传递给非 const 左值引用参数时，参数将被初始化为左值；但非 const 左值形参不能接受右值实参。

r3(w)非法，因为右值引用不能指向左值（如 w）。

r3(w+1)合法，rx 指向表达式 w+1 的临时拷贝。

r3(up(w))合法，rx 指向 up(w)的临时返回值。

3. a.
```
double & rx
const double & rx
const double & rx
```

非 const 左值引用与左值实参 w 匹配。其他两个实参为右值，const 左值引用可指向它们的拷贝。

b.
```
double & rx
double && rx
double && rx
```

左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

c.
```
const double & rx
double && rx
double && rx
```

const 左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

总之，非 const 左值形参与左值实参匹配，非 const 右值形参与右值实参匹配；const 左值形参可与左值或右值形参匹配，但编译器优先选择前两种方式（如果可供选择的话）。

4. 它们是默认构造函数、复制构造函数、移动构造函数、析构函数、复制赋值运算符和移动赋值运算符。这些函数之所以特殊，是因为编译器将根据情况自动提供它们的默认版本。

5. 在转让数据所有权（而不是复制数据）可行时，可使用移动构造函数，但对于标准数组，没有转让其所有权的机制。如果 Fizzle 使用指针和动态内存分配，则可将数据的地址赋给新指针，以转让其所有权。

6.
```
#include <iostream>
#include <algorithm>
template<typename T>
  void show2(double x, T fp) {std::cout << x << " -> " << fp(x) << '\n';}
int main()
{
    show2(18.0, [](double x){return 1.8*x + 32;});
    return 0;
}
```

7.
```
#include <iostream>
#include <array>
#include <algorithm>
const int Size = 5;
template<typename T>
  void sum(std::array<double,Size> a, T& fp);
int main()
```

```
{
    double total = 0.0;
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c, [&total](double w){total += w;});
    std::cout << "total: " << total << '\n';
    std::cin.get();
    return 0;
}
template<typename T>
  void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
           fp(*pt);
    }
}
```