第 1 章部分习题答案

习题1.1

一般而言，C++编译器要求待编译的程序保存在文件中。C++程序中一般涉及两类文件：头文件和源文件。大多数系统中，文件的名字由文件名和文件后缀（又称扩展名）组成。文件后缀通常表明文件的类型，如头文件的后缀可以是.h或.hpp等；源文件的后缀可以是.cc或.cpp等，具体的后缀与使用的编译器有关。通常可以通过编译器所提供的联机帮助文档了解其文件命名规范。

习题1.4

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The product of " << v1 << " and " << v2
              << " is " << v1 * v2 << std::endl;

    return 0;
}
```

习题1.7

由注释对嵌套导致的编译器错误信息通常令人迷惑。例如，在笔者所用的编译器中编译1.3节中给出的带有不正确嵌套注释的程序：

```
#include <iostream>
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

编译器会给出如下错误信息：

```
error C2143: syntax error : missing ';' before '<'
error C2501: 'include' : missing storage-class or type specifiers
```

```
warning C4138: '*/' found outside of comment    （第6行）
error C2143: syntax error : missing ';' before '{'    （第8行）
error C2447: '{' : missing function header (old-style formal list?)（第8行）
```

习题1.10

用for循环编写的程序如下：

```
#include <iostream>

int main()
{
    int sum = 0;
    for (int i = 50; i <= 100; ++i)
        sum += i;
    std::cout << "Sum of 50 to 100 inclusive is "
              << sum << std::endl;

    return 0;
}
```

用while循环编写的程序如下：

```
#include <iostream>

int main()
{
    int sum = 0, int i = 50;
    while (i <= 100) {
        sum += i;
        ++i;
    }
    std::cout << "Sum of 50 to 100 inclusive is "
              << sum << std::endl;
    return 0;
}
```

习题1.13

对于程序中出现的错误，编译器通常会给出简略的提示信息，包括错误出现的文件及代码行、错误代码、错误性质的描述。如果要获得关于该错误的详细信息，一般可以根据编译器给出的错误代码在其联机帮助文档中查找。

习题1.16

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入数据

    if (v1 >= v2)
        std::cout << "The bigger number is" << v1 << std::endl;
    else
        std::cout << "The bigger number is" << v2 << std::endl;

    return 0;
```

```
}
```

习题1.19

所有数的输出连在一起，不便于阅读。

程序修改如下：

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // 读入两个数

    // 用较小的数作为下界lower、较大的数作为上界upper
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }

    // 输出从lower到upper之间的值
    std::cout << "Values of " << lower << "to "
              << upper << "inclusive are: " << std::endl;
    for (int val = lower, count=1; val <= upper; ++val, ++count) {
        std::cout << val << " ";
        if (count % 10 == 0)                //每行输出10个值
            std::cout << std::endl;
    }

    return 0;
}
```

粗黑体部分为主要的修改：用变量count记录已输出的数的个数；若count的值为10的整数倍，则输出一个换行符。

习题1.22

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

int main()
{
    Sales_item trans1, trans2;

    // 读入交易
    std::cout << "Enter two transactions:" << std::endl;
    std::cin >> trans1 >> trans2;

    if (trans1.same_isbn(trans2))
        std::cout << "The total information: " << std::endl
                  << "ISBN, number of copies sold, "
                  << "total revenue, and average price are:"
                  << std::endl << trans1 + trans2;
    else
```

```
            std::cout << "The two transactions have different ISBN."
                      << std::endl;

        return 0;
    }
```

习题1.25

可从*C++ Primer*（第4版）的配套网站（http://www.awprofessional.com/cpp_primer）下载头文件Sales_item.h，然后使用该头文件编译并执行1.6节给出的书店程序。

第 2 章部分习题答案

习题2.1

它们的最小存储空间不同，分别为16位、32位和16位。一般而言，short类型为半个机器字（word）长，int类型为一个机器字长，而long类型为一个或两个机器字长（在32位机器中，int类型和long类型的字长通常是相同的）。因此，它们的表示范围不同。

习题2.4

34464。

100000超过了16位的unsigned short类型的表示范围，编译器对其二进制表示截取低16位，相当于对65536求余（求模，%），得34464。

习题2.7

(a) 'a',L'a',"a",L"a"

'a'为char型字面值，L'a'为wchar_t型字面值，"a"为字符串字面值，L"a"为宽字符串字面值。

(b) 10,10u,10L,10uL,012,0xC

10为int型字面值，10u为unsigned型字面值，10L为long型字面值，10uL为unsigned long型字面值，012为八进制表示的int型字面值，0xC为十六进制表示的int型字面值。

(c) 3.14,3.14f,3.14L

3.14为double型字面值，3.14f为float型字面值，3.14L为long double型字面值。

习题2.10

输出2M、然后换行的程序段：

```
// 输出"2M"和换行字符
std::cout << "2M" << '\n';
```

修改后的程序段：

```
// 输出'2', '\t', 'M'和换行字符
std::cout << '2' << '\t' << 'M' << '\n';
```

习题2.13

赋值运算符的左边（被赋值的对象）需要左值，见习题2.12。

习题2.16

(a) 非法：auto是关键字，不能用作变量名。使用另一变量名，如aut即可更正。

(c) 非法：>>运算符后面不能进行变量定义。改为：

```
int input_value;
std::cin >> input_value;
```

(d) 非法：同一定义语句中不同变量的初始化应分别进行。改为：

```
double salary = 9999.99, wage = 9999.99;
```

注意，(b)虽然语法上没有错误，但这个初始化没有实际意义，ival仍是未初始化的。

习题2.19

j的值是100。j的赋值所使用到的i应该是main函数中定义的局部变量i，因为局部变量的定义会屏蔽全局变量的定义。

习题2.22

问题主要在于使用了具体值100作为循环上界：100的意义在上下文中没有体现出来，导致程序的可读性差；若100这个值在程序中出现多次，则当程序的需求发生变化（如将100改变为200）时，对程序代码的修改复杂且易出错，导致程序的可维护性差。

改善方法：设置一个const变量（常量）取代100作为循环上界使用，并为该变量选择有意义的名字。

习题2.25

(d)非法。因为rval3是一个const引用，不能进行赋值。

合法赋值的作用：

(a)将一个double型字面值赋给int型变量ival，发生隐式类型转换，ival得到的值为3。

(b)将int值1赋给变量ival。

(c)将int值1赋给变量ival。

习题2.28

在笔者所用的编译器中编译上述程序，编译器会给出如下错误信息：

```
error C2628: 'Foo' followed by 'int' is illegal (did you forget a ';'?)      （第4行）
warning C4326: return type of 'main' should be 'int or void' instead of 'Foo' （第5行）
error C2440: 'return' : cannot convert from 'int' to 'Foo'                    （第6行）
```

也就是说，该编译器会对遗漏了类定义后面的分号给出提示。

习题2.31

(a)是定义，因为extern声明进行了初始化。

(b)是定义，变量定义的常规形式。

(c)是声明，extern声明的常规形式。

(d)是声明，声明了一个const引用。

第3章部分习题答案

习题3.1

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
     // 局部对象
     int base, exponent;
     long result=1;

     // 读入底数和指数
     cout << "Enter base and exponent:" << endl;
     cin >> base >> exponent;

     if (exponent < 0) {
          cout << "Exponent can't be smaller than 0" << endl;
          return -1;
     }

     if (exponent > 0) {
          // 计算底数的指数次方
          for (int cnt = 1; cnt <= exponent; ++cnt)
               result *= base;
     }

     cout << base
          << " raised to the power of "
          << exponent << ": "
          << result << endl;

     return 0;
}
```

习题3.4

s和s2的值均为空字符串。

习题3.7

测试两个string对象是否相等的程序：

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{

     string s1, s2;

     // 读入两个string对象
     cout << "Enter two strings:" << endl;
     cin >> s1 >> s2;

     // 测试两个string对象是否相等
     if (s1 == s2)
          cout << "They are equal." << endl;
     else if (s1 > s2)
          cout << "\"" << s1 << "\" is bigger than"
               << " \"" << s2 << "\"" << endl;
     else
          cout << "\"" << s2 << "\" is bigger than"
               << " \"" << s1 << "\"" << endl;

     return 0;
}
```

测试两个string对象的长度是否相等的程序：

```
#include <iostream>
#include <string>
using namespace std;

int main()
{

     string s1, s2;

     // 读入两个string对象
     cout << "Enter two strings:" << endl;
     cin >> s1 >> s2;

     // 比较两个string对象的长度
     string::size_type len1, len2;
     len1 = s1.size();
     len2 = s2.size();
     if (len1 == len2)
          cout << "They have same length." << endl;
     else if (len1 > len2)
          cout << "\"" << s1 << "\" is longer than"
               << " \"" << s2 << "\"" << endl;
     else
          cout << "\"" << s2 << "\" is longer than"
               << " \"" << s1 << "\"" << endl;

     return 0;
}
```

习题3.10

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
```

```
int main()
{
     string s, result_str;
     bool has_punct = false;//用于标记字符串中有无标点
     char ch;

     //输入字符串
     cout << "Enter a string:" << endl;
     getline(cin, s);

     //处理字符串：去掉其中的标点
     for (string::size_type index = 0; index != s.size(); ++index)
     {
          ch = s[index];
          if (ispunct(ch))
               has_punct = true;
          else
               result_str += ch;
     }

     if (has_punct)
          cout << "Result:" << endl << result_str <<endl;
     else {
          cout << "No punctuation character in the string?!" << endl;
          return -1;
     }

     return 0;
}
```

习题3.13

```
//读一组整数到vector对象，计算并输出每对相邻元素的和
#include <iostream>
#include <vector>
using namespace std;

int main()
{
     vector<int> ivec;
     int ival;

     // 读入数据到vector对象
     cout << "Enter numbers(Ctrl+Z to end):" << endl;
     while (cin>>ival)
          ivec.push_back(ival);

     // 计算相邻元素的和并输出
     if (ivec.size() == 0) {
          cout << "No element?!" << endl;
          return -1;
     }
     cout << "Sum of each pair of adjacent elements in the vector:"
          << endl;
     for (vector<int>::size_type ix = 0; ix < ivec.size()-1;
                                 ix = ix + 2) {
          cout << ivec[ix] + ivec[ix+1] << "\t";
```

```
        if ( (ix+1) % 6 == 0) // 每行输出6个和
            cout << endl;
    }

    if (ivec.size() % 2 != 0) // 提示最后一个元素没有求和
        cout << endl
             << "The last element is not been summed "
             << "and its value is "
             << ivec[ivec.size()-1] << endl;

    return 0;
}
```

修改后的程序：

```
//读一组整数到vector对象，计算首尾配对元素的和并输出
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    //读入数据到vector对象
    cout << "Enter numbers(Ctrl+Z to end):" << endl;
    while (cin>>ival)
        ivec.push_back(ival);

    //计算首尾配对元素的和并输出
    if (ivec.size() == 0) {
        cout << "No element?!" << endl;
        return -1;
    }
    cout << "Sum of each pair of counterpart elements in the vector:"
         << endl;
    vector<int>::size_type cnt = 0;
    vector<int>::size_type first, last;
    for (first = 0, last = ivec.size() - 1;
        first < last; ++first, --last) {
        cout << ivec[first] + ivec[last] << "\t";
        ++cnt;
        if ( cnt % 6 == 0) //每行输出6个和
            cout << endl;
    }

    if (first == last) //提示居中元素没有求和
        cout << endl
             << "The center element is not been summed "
             << "and its value is "
             << ivec[first] << endl;

    return 0;
}
```

习题3.16

方法一：

```
vector<int> ivec(10, 42);
```

方法二：

```
vector<int> ivec(10);
for (ix = 0; ix < 10; ++ix)
     ivec[ix] = 42;
```

方法三：

```
vector<int> ivec(10);
for (vector<int>::iterator iter = ivec.begin();
                           iter != ivec.end(); ++iter)
     *iter = 42;
```

方法四：

```
vector<int> ivec;
for (cnt = 1; cnt <= 10; ++cnt)
     ivec.push_back(42);
```

方法五：

```
vector<int> ivec;
vector<int>::iterator iter = ivec.end();
for (int i = 0; i != 10; ++i) {
     ivec.insert(iter, 42);
     iter = ivec.end();
}
```

各种方法都可达到目的，也许最后两种方法更好一些。它们使用标准库中定义的容器操作在容器中增添元素，无需在定义vector对象时指定容器的大小，比较灵活而且不容易出错。

习题3.19

```
//创建有10个元素的vector对象，
//然后使用迭代器将每个元素值改为当前值的2倍并输出
#include <iostream>
#include <vector>
using namespace std;

int main()
{
     vector<int> ivec(10, 20);//每个元素的值均为20

     //将每个元素值改为当前值的2倍并输出
     for (vector<int>::iterator iter = ivec.begin();
                                       iter != ivec.end(); ++iter) {
          *iter = (*iter)*2;
          cout << *iter << " ";
     }

     return 0;
}
```

习题3.22

将两个迭代器相加的操作是未定义的，因此用这种方法计算mid会出现编译错误。

第4章部分习题答案

习题4.1

(a)非法，buf_size是一个变量，不能用于定义数组的维数（维长度）。

(b)非法，get_size()是函数调用，不是常量表达式，不能用于定义数组的维数（维长度）。

(d)非法，存放字符串"fundamental"的数组必须有12个元素，st只有11个元素。

习题4.4

定义数组时可使用初始化列表（用花括号括住的一组以逗号分隔的元素初值）来初始化数组的部分或全部元素。如果是初始化全部元素，可以省略定义数组时方括号中给出的数组维数值。如果指定了数组维数，则初始化列表提供的元素个数不能超过维数值。如果数组维数大于列出的元素初值个数，则只初始化前面的数组元素，剩下的其他元素，若是内置类型则初始化为 0，若是类类型则调用该类的默认构造函数进行初始化。字符数组既可以用一组由花括号括起来、逗号隔开的字符字面值进行初始化，也可以用一个字符串字面值进行初始化。

习题4.7

将一个数组赋给另一个数组，就是将一个数组的元素逐个赋值给另一数组的对应元素，可用如下代码实现：

```
int main()
{
    const size_t array_size = 10;
    int ia1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int ia2[array_size];

    for (size_t ix = 0; ix != array_size; ++ix)
        ia2[ix] = ia1[ix];

    return 0;
}
```

将一个vector赋给另一个vector，也是将一个vector的元素逐个赋值给另一vector的对应元素，可用如下代码实现：

```
//将一个vector赋值给另一vector
//使用迭代器访问vector中的元素
#include <vector>
using namespace std;
```

```
int main()
{
      vector<int> ivec1(10, 20);//每个元素初始化为20
      vector<int> ivec2;

      for (vector<int>::iterator iter = ivec1.begin();
                              iter != ivec1.end();  ++iter)
          ivec2.push_back(*iter);

      return 0;
}
```

习题4.10

第一种形式强调了ip是一个指针，这种形式在阅读时不易引起误解，尤其是当一个语句中同时定义了多个变量时。

习题4.13

具有void*类型的指针可以保存任意类型对象的地址，因此p的初始化是合法的；而指向long型对象的指针不能用int型对象的地址来初始化，因此lp的初始化不合法。

习题4.16

该程序段使得i被赋值为42的平方，j被赋值为42与1024的乘积。

习题4.19

(a) 合法：定义了int型对象i。

(b) 非法：定义const对象时必须进行初始化，但ic没有初始化。

(c) 合法：定义了指向int型const对象的指针pic。

(d) 非法：因为cpi被定义为指向int型对象的const指针，但该指针没有初始化。

(e) 非法：因为cpic被定义为指向int型const对象的const指针，但该指针没有初始化。

习题4.22

两个while循环的差别为：前者的循环结束条件是cp为0值（即指针cp为0值）；后者的循环结束条件是cp所指向的字符为0值（即cp所指向的字符为字符串结束符null（即'\0'））。因此后者能正确地计算出字符串"hello"中有效字符的数目（放在cnt中），而前者的执行是不确定的。

注意，题目中的代码还有一个小问题，即cnt没有初始化为0值。

习题4.25

比较两个string类型的字符串的程序如下：

```
//比较两个string 类型的字符串
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
     string str1, str2;

     //输入两个字符串
     cout << "Enter two strings:" << endl;
     cin >> str1 >> str2;

     //比较两个字符串
     if (str1 > str2)
          cout << "\"" << str1 << "\"" << " is bigger than "
               << "\"" << str2 << "\"" << endl;
     else if (str1 < str2)
          cout << "\"" << str2 << "\"" << " is bigger than "
               << "\"" << str1 << "\"" << endl;
     else
          cout << "They are equal" << endl;

     return 0;
}
```

比较两个C风格字符串的程序如下：

```
//比较两个C风格字符串的值
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
     //char *str1 = "string1", *str2 = "string2";
     const int str_size = 80;
     char *str1, *str2;

     //为两个字符串分配内存
     str1 = new char[str_size];
     str2 = new char[str_size];
     if (str1 == NULL||str2 == NULL) {
          cout << "No enough memory!" << endl;
          return -1;
     }

     //输入两个字符串
     cout << "Enter two strings:" << endl;
     cin >> str1 >> str2;

     //比较两个字符串
     int result;
     result = strcmp(str1, str2);
     if (result > 0)
          cout << "\"" << str1 << "\"" << " is bigger than "
               << "\"" << str2 << "\"" << endl;
     else if (result < 0)
          cout << "\"" << str2 << "\"" << " is bigger than "
               << "\"" << str1 << "\"" << endl;
     else
          cout << "They are equal" << endl;

     //释放字符串所占用的内存
     delete [] str1 ;
     delete [] str2 ;
```

```
    return 0;
}
```

注意，此程序中使用了内存的动态分配与释放（见4.3.1节）。如果不用内存的动态分配与释放，可将主函数中第2、3两行代码、有关内存分配与释放的代码以及输入字符串的代码注释掉，再将主函数中第一行代码

```
//char *str1 = "string1", *str2 = "string2";
```

前的双斜线去掉即可。

习题4.28

```
// 从标准输入设备读入的元素数据建立一个int型vector对象，
// 然后动态创建一个与该vector对象大小一致的数组，
// 把vector对象的所有元素复制给新数组
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec;
    int ival;

    //读入元素数据并建立vector
    cout << "Enter numbers:(Ctrl+Z to end)" << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    //动态创建数组
    int *pia = new int[ivec.size()];

    //复制元素
    int *tp = pia;
    for (vector<int>::iterator iter = ivec.begin();
                        iter != ivec.end(); ++iter, ++tp)
        *tp = *iter;

    //释放动态数组的内存
    delete [] pia;

    return 0;
}
```

习题4.31

```
// 从标准输入设备读入字符串，并把该串存放在字符数组中
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main()
{
    string in_str;// 用于读入字符串的string对象
    const size_t str_size = 10;
    char result_str[str_size+1];
```

```
    // 读入字符串
    cout << "Enter a string(<=" << str_size
         << " characters):" << endl;
    cin >> in_str;

    // 计算需复制的字符的数目
    size_t len = strlen(in_str.c_str());
    if (len > str_size) {
        len = str_size;
        cout << "String is longer than " << str_size
             << " characters and is stored only "
             << str_size << " characters!" << endl;
    }

    // 复制len个字符至字符数组result_str
    strncpy(result_str, in_str.c_str(), len);

    // 在末尾加上一个空字符（null字符）
    result_str[len+1] = '\0';

    return 0;
}
```

为了接受可变长的输入，程序中用一个string对象存放读入的字符串，然后使用strncpy函数将该对象的适当内容复制到字符数组中。因为字符数组的长度是固定的，因此首先计算字符串的长度。若该长度小于或等于字符数组可容纳字符串的长度，则复制整个字符串至字符数组，否则，根据数组的长度，复制字符串中前面部分的字符，以防止溢出。

注意，上述给出的是满足题目要求的一个解答，事实上，如果希望接受可变长的输入并完整地存放到字符数组中，可以采用动态创建数组来实现。

习题4.34

```
//4-34.cpp
//读入一组string类型的数据，并将它们存储在vector中。
//接着，把该vector对象复制给一个字符指针数组。
//为vector中的每个元素创建一个新的字符数组，
//并把该vector元素的数据复制到相应的字符数组中，
//最后把指向该数组的指针插入字符指针数组
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> svec;
    string str;

    // 输入vector元素
    cout << "Enter strings:(Ctrl+Z to end)" << endl;
    while (cin >> str)
        svec.push_back(str);

    // 创建字符指针数组
    char **parr = new char*[svec.size()];

    // 处理vector元素
    size_t ix = 0;
```

```
    for (vector<string>::iterator iter = svec.begin();
            iter != svec.end(); ++iter, ++ix) {
        // 创建字符数组
        char *p = new char[(*iter).size()+1];
        // 复制vector元素的数据到字符数组
        strcpy(p, (*iter).c_str());
        // 将指向该字符数组的指针插入到字符指针数组
        parr[ix] = p;
    }

    // 释放各个字符数组
    for (ix =0; ix != svec.size(); ++ix)
        delete [] parr[ix];
    // 释放字符指针数组
    delete [] parr;

    return 0;
}
```

第 5 章部分习题答案

习题5.1

加入如下所示的圆括号以标明该表达式的计算顺序：

```
(((12 / 3) * 4) + (5 * 15)) + ((24 % 4) / 2)
```

习题5.4

溢出：表达式的求值结果超出了其类型的表示范围。

如下表达式会导致溢出（假设int类型为16位）：

```
1000 * 1000
32766 + 5
3276 * 20
```

在这些表达式中，各操作数均为int类型，因此这些表达式的类型也是int，但它们的计算结果均超出了16位int型的表示范围（–32768~32767），导致溢出。

习题5.7

```
int val;
cin >> val;
while (val != 42)
```

或者，while循环条件也可以写成

```
while (cin >> ival && ival != 42)
```

习题5.10

```
bitset<30> bitset_quiz1;
bitset_quiz1[27] = 1;
bitset_quiz1[27] = 0;
```

习题5.13

该赋值语句不合法，因为该语句首先将0值赋给pi，然后将pi的值赋给ival，再将ival的值赋给dval。pi、ival和dval的类型各不相同，因此要完成赋值必须进行隐式类型转换，但系统无法将int型指针pi的值隐式转换为ival所需的int型值。

可改正如下：

```
double dval; int ival; int *pi;
```

```
dval = ival = 0;
pi = 0;
```

习题5.16

C++之名是 Rick Mascitti 在 1983 年夏天定名的（参见 *The C++ Programming Language(Special Edition)* 1.4 节），C 说明它本质上是从 C 语言演化而来的，“++”是 C 语言的自增操作符。C++语言是 C 语言的超集，是在 C 语言基础上进行的扩展（引入了 new、delete 等 C 语言中没有的操作符，增加了对面向对象程序设计的直接支持，等等），是先有 C 语言，再进行++。根据自增操作符前、后置形式的差别（参见习题 5.15 的解答），C++表示对 C 语言进行扩展之后，还可以使用 C 语言的内容；而写成++C 则表示无法再使用 C 的原始值了，也就是说 C++不能向下兼容 C 了，这与实际情况不符。

习题5.19

(a)、(d)、(f)合法。

这些表达式的执行结果如下：

(a)返回iter所指向的string对象，并使iter加1。

(d)调用iter所指向的string对象的成员函数empty。

(f)调用iter所指向的string对象的成员函数empty，并使iter加1。

习题5.22

```
//输出每种内置类型的长度
#include <iostream>
using namespace std;

int main()
{
    cout << "type\t\t\t" << "size" << endl
         << "bool\t\t\t" << sizeof(bool) << endl
         << "char\t\t\t" << sizeof(char) << endl
         << "signed char\t\t" << sizeof(signed char) << endl
         << "unsigned char\t\t" << sizeof(unsigned char) << endl
         << "wchar_t\t\t\t" << sizeof(wchar_t) << endl
         << "short\t\t\t" << sizeof(short) << endl
         << "signed short\t\t" << sizeof(signed short) << endl
         << "unsigned short\t\t" << sizeof(unsigned short) << endl
         << "int\t\t\t" << sizeof(int) << endl
         << "signed int\t\t" << sizeof(signed int) << endl
         << "unsigend int\t\t" << sizeof(unsigned int) << endl
         << "long\t\t\t" << sizeof(long) << endl
         << "sigend long\t\t" << sizeof(signed long) << endl
         << "unsigned long\t\t" << sizeof(unsigned long) << endl
         << "float\t\t\t" << sizeof(float) << endl
         << "double\t\t\t" << sizeof(double) << endl
         << "long double\t\t" << sizeof(long double) << endl;

    return 0;
}
```

习题5.25

添加圆括号说明其计算顺序如下：

(a) `((! ptr) == (ptr->next))`

(b) `(ch = ((buf[ (bp++) ]) != '\n'))`

习题5.28

这可以接受。

因为，操作数的求解次序通常对结果没什么影响。只有当二元操作符的两个操作数涉及同一对象，并改变该对象的值时，操作数的求解次序才会影响计算结果；后一种情况只会在部分（甚至是少数）程序中出现。在实际使用中，这种“潜在的缺陷”可以通过程序员的努力得到弥补，但“实现效率”的提高却能使所有使用该编译器的程序受益，因此利大于弊。

习题5.31

(a) 将`fval`的值从`float`类型转换为`bool`类型。

(b) 将`ival`的值从`int`类型转换为`float`类型，再将`fval + ival`的结果值转换为`double`类型，赋给`dval`。

(c) 将`ival`的值从`int`类型转换为`double`类型，`cval`的值首先提升为`int`类型，然后从`int`型转换为`double`型，与`dval + ival`的结果值相加。

第 6 章部分习题答案

习题6.1

空语句是由一个单独的分号构成的语句。例如，

```
int ival;
while (cin >> ival && ival != -1)
; // 空语句
```

该循环语句从输入流中读取数据，在遇到文件结束符或读到特定值（-1）前不做任何操作。

习题6.4

将会使得程序只计算并输出ISBN相同的第一组trans的总和，然后将total置为最后读入的那个trans。

习题6.7

```
// 统计读入的文本中大小写元音字母的个数
#include <iostream>
using namespace std;

int main()
{
     char ch;

     // 初始化每个元音的计数器
     int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;

     while (cin >> ch) {
          // 若ch是元音，将相应计数器加1
          switch (ch) {
               case 'a':
               case 'A':
                    ++aCnt;
                    break;
               case 'e':
               case 'E':
                    ++eCnt;
                    break;
               case 'i':
               case 'I':
                    ++iCnt;
                    break;
               case 'o':
               case 'O':
```

```
                ++oCnt;
                break;
            case 'u':
            case 'U':
                ++uCnt;
                break;
        }
    }

    // 输出结果
    cout << "Number of vowel a: \t" << aCnt << '\n'
         << "Number of vowel e: \t" << eCnt << '\n'
         << "Number of vowel i: \t" << iCnt << '\n'
         << "Number of vowel o: \t" << oCnt << '\n'
         << "Number of vowel u: \t" << uCnt << endl;

    return 0;
}
```

注意，初学者容易将诸如

```
case 'a':
case 'A':
```

这样的case分支（又称case标号）写成

```
case 'a','A':
```

这会出现编译错误，因为case标号中的值必须是常量表达式（不包括逗号表达式）。

习题6.10

(a)的错误在于：各个case标号对应的语句块中缺少必要的break语句，从而当ival值为'a'时，aCnt、eCnt和iouCnt都会加1；ival值为'e'时，eCnt和iouCnt都会加1。

修改为：

```
switch (ival) {
    case 'a':
        aCnt++;
        break;
    case 'e':
        eCnt++;
        break;
    default:
        iouCnt++;
        break;//此语句省略亦可
}
```

(b)的错误在于：在case 1标号之后、default标号之前定义了变量ix。因为，对于switch结构，只能在它的最后一个case标号或default标号后面定义变量，以避免出现代码跳过变量的定义和初始化的情况。

修改为：

```
int ix;
 switch (ival) {
    case 1:
```

```
        ix = get_value();
        ivec[ ix ] = ival;
        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = ival;
    }
```

(c)的错误在于：case标号中出现了多个值。因为一个case标号只能与一个值相关联。

修改为：

```
    switch (ival) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 9:
            oddcnt++;
            break;
        case 2:
        case 4:
        case 6:
        case 8:
        case 10:
            evencnt++;
            break;
    }
```

(d)的错误在于：case标号中不能使用ival、jval和kval。因为case标号中的值只能使用常量表达式，而ival、jval和kval都是变量。

将语句int ival=512, jval=1024, kval=4096;修改为：

```
const int ival=512, jval=1024, kval=4096;
```

习题6.13

执行过程如下：(1)指针dest加1；(2)指针source加1；(3)将source原来所指向的对象赋给dest原来所指向的对象。

习题6.16

```
// 6-16.cpp
// 给出两个int型的vector对象，判断一个对象是否是另一个对象的前缀
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> ivec1, ivec2;
    int ival;

    // 读入第一个vector对象的元素
    cout << "Enter elements for the first vector:(32767 to end)"
         << endl;
    cin >> ival;
    while (ival != 32767) {
```

```
        ivec1.push_back(ival);
        cin >> ival;
    }

    //读入第二个vector对象的元素
    cout << "Enter elements for the second vector:(32767 to end)"
         << endl;
    cin >> ival;
    while (ival != 32767) {
        ivec2.push_back(ival);
        cin >> ival;
    }

    // 比较两个vector对象
    vector<int>::size_type size1, size2;
    size1 = ivec1.size();
    size2 = ivec2.size();
    bool result = true;
    for (vector<int>::size_type ix = 0;
        ix != (size1 > size2 ? size2 : size1); ++ix)
        if (ivec1[ix] != ivec2[ix]) {
            result = false;
            break;
        }

    // 输出结果
    if (result)
        if (size1 < size2)
            cout << "The first vector is prefix of the second one."
                 << endl;
        else if (size1 == size2)
            cout << "Two vectors are equal." << endl;
        else
            cout << "The second vector is prefix of the first one."
                 << endl;
    else
        cout << "No vector is prefix of the other one.";

    return 0;
}
```

注意，在输入两个vector的元素时也可以不用特定值32767标记输入结束，而使用文件结束符（输入Ctrl+Z）来控制元素输入的结束。但是，使用后一种方法时，在第二个输入循环之前要记得将流cin恢复为有效状态（使用cin.clear()）。

习题6.19

重写循环如下：

```
while (iter != vec.end() && value != *iter++) {
}// end of while
```

习题6.22

重写代码如下：

```
do {
    int sz = get_size();
```

```
} while (sz <= 0);
```

习题6.25

```
// 6-25.cpp
// 从标准输入读入一系列string对象，
// 直到同一个单词连续出现两次，或者所有的单词都已读完，才结束读取。
// 输出重复出现的单词，或者输出没有任何单词连续重复出现的信息。
// 使用有条件的调试代码有条件地输出每一个读入的单词
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
     string currWord, preWord; // 记录当前单词及其前一单词
     cout << "Enter some words:(Ctrl+Z to end)" << endl;
     while (cin >> currWord) {
          // 若未定义NDEBUG，则输出读入的单词
          #ifndef NDEBUG
          cout << currWord << " ";
          #endif
          if (!isupper(currWord[0])) // 单词不是以大写字母开头
               continue;
          if (currWord == preWord )
               // 当前单词是重复出现且以大写字母开头
               break;
          else
               preWord = currWord;
     }

     //输出结果
     if (currWord == preWord && !currWord.empty())
          cout << "The repeated word: " << currWord << endl;
     else
          cout << "There is no repeated word that has initial capital."
               << endl;

     return 0;
}
```

在打开调试器的情况下编译和运行该程序，会输出所读入的每个单词；如果在关闭调试器的情况下编译和运行该程序，则不会输出所读入的每个单词。

第 7 章部分习题答案

习题7.1

形参在函数定义的形参表中进行定义，是一个变量，其作用域为整个函数。而实参出现在函数调用中，是一个表达式。进行函数调用时，用传递给函数的实参对形参进行初始化。

习题7.4

可编写如下abs函数，返回形参x的绝对值：

```
int abs(int x)
{
    return x >= 0 ? x : -x;
}
```

习题7.7

前者声明的是T类型的形参。在f中修改形参的值不会影响调用f时所传递的实参的值。

后者声明的是T类型的引用形参。在f中修改形参的值实际上相当于修改调用f时所传递的实参的值。

习题7.10

其局限在于：此处使用引用形参的唯一目的是避免复制实参，但没有将形参定义为const引用，从而导致不能使用字符串字面值来调用该函数（因为非const引用形参只能与完全同类型的非const对象关联）。

可更正为：

```
bool test(const string& s) { return s.empty(); }
```

习题7.13

```
// 7-13.cpp
// 计算数组元素之和。
// 三个求和函数以不同的方法处理数组边界
#include <iostream>
using namespace std;

// 传递指向数组第一个和最后一个元素的下一位置的指针
int sum1(const int *begin, const int *end)
{
    int sum = 0;
    while (begin != end) {
```

```
        sum += *begin++;
    }
    return sum;
}

// 传递数组大小
int sum2(const int ia[], size_t size)
{
    int sum = 0;
    for (size_t ix = 0; ix != size; ++ix) {
        sum += ia[ix];
    }
    return sum;
}

// 传递指向数组第一个元素的指针和数组大小
int sum3(int *begin, size_t size)
{
    int sum = 0;
    int *p = begin;
    while (p != begin + size) {
        sum += *p++;
    }
    return sum;
}

int main()
{
    int ia[] = { 1, 2, 3, 4};
    cout << "Summation from sum1(): " << sum1(ia, ia+4) << endl;
    cout << "Summation from sum2(): " << sum2(ia, 4) << endl;
    cout << "Summation from sum2(): " << sum3(ia, 4) << endl;

    return 0;
}
```

习题7.16

```
// 7-16.cpp
// 接受命令行选项，并输出传递给main的实参的值
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
    cout << "arguments passed to main(): " << endl;
    for (int i = 0; i != argc; ++i)
        cout << argv[i] << endl;

    return 0;
}
```

习题7.19

该程序段合法。其功能为：将数组ia的各元素赋值为0。

习题7.22

函数原型如下：

```
bool compare(matrix&, matrix&);
vector<int>::iterator change_val(int, vector<int>::iterator);
```

习题7.23

(b)、(c)、(d)合法。

(a)不合法。因为calc函数只有一个形参，调用该函数时却传递了两个实参。

习题7.25

(a)不合法。因为调用init函数时必须显式指定至少一个实参。

(c)合法，但可能不符合程序员的原意。因为这里是将char型实参'*'转换为int型再传递给形参wd。

习题7.28

```
// 7-28.cpp
// 编写函数，使其在第一次调用时返回0，
// 然后再次调用时按顺序产生正整数（即返回其当前的调用次数）
#include <iostream>
using namespace std;

size_t count_calls()
{
     static size_t ctr = -1; // ctr的生命期将跨越函数的多次调用
     return ++ctr;
}

int main()
{
     for (size_t i = 0; i != 10; ++i)
          cout << count_calls() << endl;

     return 0;
}
```

习题7.31

Sales_item类的头文件如下：

```
// Sales_item.hpp
// 自定义的Sales_item类的头文件
// 定义Sales_item类，
// 添加两个public成员input和output用于读和写Sales_item对象
#ifndef SALESITEM_H
#define SALESITEM_H
#include <iostream>
#include <string>

class Sales_item {
public:
     // Sales_item对象的操作
     std::istream& input(std::istream& in);
     std::ostream& output(std::ostream& out) const;
     double avg_price() const;
     bool same_isbn(const Sales_item &rhs) const
     {
```

```
        return isbn == rhs.isbn;
    }

    // 默认构造函数需要初始化内置类型的数据成员
    Sales_item::Sales_item(): units_sold(0), revenue(0.0) { }

private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

#endif
```

Sales_item 类的实现文件（源文件）如下：

```
// Sales_item.cpp
// 自定义的Sales_item类的实现文件（源文件）
#include "Sales_item.hpp"

std::istream& Sales_item::input(std::istream& in)
{
   double price;
   in >> isbn >> units_sold >> price;
   // 检验是否读入成功
   if (in)
      revenue = units_sold * price;
   else {// 读入失败：将对象复位为默认状态
      units_sold = 0;
      revenue = 0.0;
   }
   return in;
}

std::ostream& Sales_item::output(std::ostream& out) const
{
   out << isbn << "\t" << units_sold << "\t"
       << revenue << "\t" <<  avg_price();
   return out;
}

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

主程序如下：

```
// 7-31.cpp
// 利用自定义的Sales_item类读入并输出一组交易
#include "Sales_item.hpp"
#include <iostream>
using namespace std;

int main()
{
    Sales_item item;

    // 读入并输出一组交易
```

```
    cout << "Enter some transactions(Ctrl+Z to end):"
         << endl;
    while (item.input(cin)) {
        cout << "The transaction readed is:" << endl;
        item.output(cout);
        cout << endl;
    }

    return 0;
}
```

习题7.34

这组重载函数可定义如下：

```
void error(const string &s, int index, int upperBound)
{
    cout << s << "index is " << index
         << " and upperBound is " << upperBound << endl;
}

void error(const string &s)
{
     cout << s << endl;
}

void error(const string &s, char selectVal)
{
     cout << s << ": " << selectVal << endl;
}
```

习题7.37

(a) 可行函数是 `void f(int, int)`和 `void f(double, double = 3.14)`。该调用不合法，存在二义性：既可将 `2.56` 转换为 `int` 型而调用前者，亦可将 `42` 转换为 `double` 型而调用后者。

(b) 可行函数是 `void f(int)`和 `void f(double, double = 3.14)`。该调用合法，最佳匹配函数是 `void f(int)`。

(c) 可行函数是 `void f(int, int)`和 `void f(double, double = 3.14)`。该调用合法，最佳匹配函数是 `void f(int, int)`。

(d) 可行函数是 `void f(int, int)`和 `void f(double, double = 3.14)`。该调用合法，最佳匹配函数是 `void f(double, double = 3.14)`。

习题7.40

该函数调用不合法。因为函数的形参为枚举类型`Stat`，函数调用的实参为`int`类型。枚举类型对象只能用同一枚举类型的另一对象或一个枚举成员进行初始化，因此不能将`int`类型的实参值传递给枚举类型的形参。

第 8 章部分习题答案

习题8.1

如果os是一个ofstream对象，则os << "Goodbye!" << endl;将字符串“Goodbye!”及换行符写到os所关联的磁盘文件中。

如果os是一个ostringstream对象，则os << "Goodbye!" << endl;将字符串“Goodbye!”及换行符写到os所关联的字符串流中。

如果os是ifstream对象，则会出现一个编译错误，因为ifstream类中没有定义操作符“<<”。

习题8.4

将习题8.3编写的get函数的声明放在头文件get.hpp中，其定义放在实现文件get.cpp中，则可编写如下程序来调用该函数：

```
// 8-4.cpp
// 主函数以cin为实参调用get函数
#include "get.hpp"  // 引入上题定义的get函数
#include <iostream>
using namespace std;

int main()
{
    double dval;

    get(cin);
    cin >> dval;    // 重新使用恢复后的流
    cout << dval << endl;

    return 0;
}
```

习题8.7

在循环中，如果文件无法打开，则输出警告信息，清除文件流的状态，然后从vector中获取下一个文件名，再将break语句改成continue语句即可。如第二个循环可修改如下：

```
while (it != files.end()) {
    input.open(it->c_str());       // 打开文件
    if (!input) {                  // 打开文件失败
        cerr << "error: can not open file: "
             << *it << endl;
        input.clear();             // 清除文件流的状态
        ++it;                      // 获取下一文件
```

```
            continue;                    // 继续处理下一文件
        }
        // 若打开成功，则读入并处理文件流input
        while(input >> s)                // 处理文件
            process(s);
        input.close();                   // 关闭文件
        input.clear();                   // 清除文件流的状态
        ++it;                            // 获取下一文件
}
```

习题8.10

只需修改读文件的语句，使用文件流对象的>>操作符进行读即可。程序如下：

```
// 8-10.cpp
// 函数fileToVector打开文件用于输入，
// 将文件内容读入string类型的vector容器，
// 每个单词存储为该容器对象的一个元素。
// 主函数例示fileToVector函数的使用
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

int fileToVector(string fileName, vector<string>& svec)
{
    // 创建ifstream对象inFile并绑定到由形参fileName指定的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) // 打开文件失败
        return 1;

    // 将文件内容读入到string类型的vector容器
    // 每个单词存储为该容器对象的一个元素
    string s;
    while (inFile >> s)   // 读入单词
        svec.push_back(s);
    inFile.close();       // 关闭文件
    if (inFile.eof())     // 遇到文件结束符
        return 4;
    if (inFile.bad())     // 发生系统级故障
        return 2;
    if (inFile.fail())    // 读入数据失败
        return 3;
}

int main()
{
    vector<string> svec;
    string fileName, s;

    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

    // 处理文件
    switch (fileToVector(fileName, svec)) {
        case 1:
```

```
                cout << "error: can not open file: "
                     << fileName << endl;
                return -1;
            case 2:
                cout << "error: system failure " << endl;
                return -1;
            case 3:
                cout << "error: read failure " << endl;
                return -1;
        }

        // 输出vector对象进行检验
        cout << "Vector:" << endl;
        for (vector<string>::iterator iter = svec.begin();
             iter != svec.end(); ++iter)
            cout << *iter << endl;

        return 0;
    }
```

习题8.13

可编写如下类似函数：

```
// 打开out绑定到给定文件
ofstream& open_file(ofstream &out, const string &file)
{
    out.close(); // 关闭以防它已经是打开的
    out.clear(); // 清除内部状态
    out.open(file.c_str()); // 打开给定文件
    return out;
}
```

习题8.16

程序如下：

```
// 8-16.cpp
// 将文件中的每一行存储在vector<string>容器对象中，
// 然后使用istringstream
// 从vector里以每次读一个单词的形式读取所存储的行
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

int fileToVector(string fileName, vector<string>& svec)
{
    // 创建ifstream对象inFile并绑定到由形参fileName指定的文件
    ifstream inFile(fileName.c_str());
    if (!inFile) // 打开文件失败
        return 1;
    // 将文件内容读入到string类型的vector容器
    // 每一行存储为该容器对象的一个元素
    string s;
    while (getline(inFile, s))
        svec.push_back(s);
    inFile.close();      // 关闭文件
```

```
    if (inFile.eof())    // 遇到文件结束符
        return 4;
    if (inFile.bad())    // 发生系统级故障
        return 2;
    if (inFile.fail())   // 读入数据失败
        return 3;
}

int main()
{
    vector<string> svec;
    string fileName, s;

    // 读入文件名
    cout << "Enter filename :" << endl;
    cin >> fileName;

    // 处理文件
    switch (fileToVector(fileName, svec)) {
        case 1:
            cout << "error: can not to open file: "
                 << fileName << endl;
            return -1;
        case 2:
            cout << "error: system failure " << endl;
            return -1;
        case 3:
            cout << "error: read failure " << endl;
            return -1;
    }

    // 使用istringstream从vector里以每次读一个单词的形式读取所存储的行
    string word;
    istringstream isstream;
    for (vector<string>::iterator iter = svec.begin();
        iter != svec.end(); ++iter) {
        // 将vector对象的当前元素复制给istringstream对象
        isstream.str(*iter);
        // 从istringstream对象中读取单词并输出
        while (isstream >> word) {
            cout << word << endl;
        }
        isstream.clear(); // 将istringstream流置为有效状态
    }

    return 0;
}
```

第 9 章部分习题答案

习题9.1

(c)和(d)是错误的。前者错在用于初始化ivec的一对指针中，ia+8超出了数组的上界，因此会导致运行时出现数组访问越界错误，正确的指针应为ia+7；后者错在用于初始化slist的一对指针的顺序错了，应该是sa在前，sa+6在后，因为它们分别用于标记要复制的第一个元素和停止复制的条件（第二个指针标记要复制的最后一个元素的下一位置）。

习题9.4

```
list< deque<int> > lst;
```

注意，必须用空格隔开两个相邻的>符号，以示这是两个分开的符号，否则，系统会认为>>是单个符号，为右移操作符，从而导致编译时错误。

习题9.7

错误在于while循环的条件表达式iter1 < iter2中使用了< 操作符，因为list容器的迭代器不支持关系操作。可更正为：

```
while (iter1 != iter2) /* ... */
```

习题9.10

(a)是错误的。因为返回的迭代器的类型为const vector<int>，不能用来对类型为vector<int>的迭代器it进行初始化。

(b)是错误的。因为list容器的迭代器不支持算术运算（+）。

(d)是错误的。因为循环条件中迭代器it与0值进行比较，导致运行时内存访问非法的错误，应该将it!=0改为it!=svec.end()。

习题9.13

程序如下：

```
// 9-13.cpp
// findInt函数在形参迭代器标记的范围内寻找给定的int型数值，
```

```
// 返回指向所找到元素的迭代器。
// 主函数例示findInt函数的使用
#include <iostream>
#include <vector>
using namespace std;

// 在迭代器标记的范围内寻找给定的int型数值，返回指向所找到元素的迭代器
vector<int>::iterator findInt(vector<int>::iterator beg,
                              vector<int>::iterator end, int ival)
{
    while (beg != end)
        if (*beg == ival) // 找到，则结束循环
            break;
        else
            ++beg;

    return beg;           // 若找到，则beg指向所找到的元素；
                          // 否则，beg与end相等
}

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6};
    vector< int > ivec(ia, ia+7);

    // 读入要找的数据
    cout << "Enter a integer:" << endl;
    int ival;
    cin >> ival;

    // 调用findInt函数查找ival
    vector<int>::iterator it;
    it = findInt(ivec.begin(), ivec.end(), ival);
    if (it != ivec.end()) // 找到（函数返回的迭代器与调用函数的第二个实参不相等）
        cout << ival << " is a element of the vector." << endl;
    else
        cout << ival << " isn't a element of the vector." << endl;

    return 0;
}
```

注意，为了提高程序的通用性，使得对于不同类型的迭代器及元素类型，都能用同一查找函数进行查找，习题 9.12 和习题 9.13 中的findInt函数可以实现为函数模板（见第 16 章）。例如，可用如下函数模板findValue实现习题 9.13 中的查找函数findInt的功能：

```
// 函数模板findValue:
// 在迭代器beg和end标记的范围内查找给定的值，返回指向所找到元素的迭代器
template<typename T1, typename T2>
T1 findValue(T1 beg, T1 end, T2 val)
{
    while (beg != end)
        if (*beg == val) // 找到，则结束循环
            break;
        else
            ++beg;

    return beg; // 若找到，则beg指向所找到的元素；否则，beg与end相等
}
```

习题9.16

int型的vector容器应该使用的索引类型为vector<int>::size_type。

习题9.19

存在的错误包括：

- 对vector进行插入后会导致迭代器失效，而循环中使用了先前保存的迭代器mid。
- 循环中使用的迭代器iter没有初始化。
- if语句中的条件应该是比较iter所指向的元素（即*iter）与some_val是否相等。

可更正为：

```
vector<int>::iterator iter = iv.begin();
while ( iter != iv.begin() + iv.size()/2 ) {
     if (*iter == some_val) {
          iter = iv.insert(iter, 2 * some_val);
          iter += 2;    // 使iter指向下一个要处理的原始元素
     }
     else
          ++iter;       // 使iter指向下一个要处理的原始元素
}
```

注意，insert函数在迭代器所指元素的前面插入新元素，返回值为指向新插入元素的迭代器，所以将iter加上2才能指向下一个要处理的原始元素。

习题9.22

vec.resize(100)操作使容器vec中包含100个元素：前25个元素保持原值，后75个元素采用值初始化（见3.3.1节）。

若再做操作vec.resize(10)，则使容器vec中包含10个元素，只保留原来的前10个元素，后面的元素被删除。

习题9.25

如果val1与val2相等，则不会删除任何元素。

如果val1和val2中的一个不存在，或两个都不存在，则会发生运行时错误。

习题9.28

程序如下：

```
// 9-28.cpp
// 将一个list容器的所有元素赋值给一个vector容器，
// 其中list容器中存储的是指向C风格字符串的char*指针，
// 而vector容器的元素则是string类型
#include <iostream>
#include <list>
#include <vector>
#include <string>
using namespace std;
```

```
int main()
{
    char* sa[] = {"Mary", "Tom", "Bob", "Alice"};
    list<char*> slst(sa, sa+4);
    vector<string> svec;
    string str;

    // 将list对象的所有元素赋值给vector对象
    svec.assign(slst.begin(), slst.end());

    // 输出list对象中的元素
    for (list<char*>::iterator lit = slst.begin();
         lit != slst.end(); ++lit) {
        cout << *lit << " ";
    }
    cout << endl;

    // 输出vector对象中的元素
    for (vector<string>::iterator vit = svec.begin();
         vit != svec.end(); ++vit) {
        cout << *vit << " ";
    }
    cout << endl;

    return 0;
}
```

习题9.31

容器的容量不能比其长度小。

一般而言，在初始时或插入元素后，容量不会恰好等于所需要的长度（往往是容量大于长度）。因为系统会根据一定的分配策略预留一些额外的存储空间以备容器的增长，从而避免额外的重新分配内存、复制元素、释放内存等操作，提高性能。

习题9.34

程序如下：

```
// 9-34.cpp
// 使用迭代器将string对象中的字符都改为大写字母
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    string str = "This is a example";
    for (string::iterator iter = str.begin();
         iter != str.end(); ++iter)
        *iter = toupper(*iter);  // 将字符转换为对应的大写字母
    return 0;
}
```

习题9.37

因为string对象中的字符是连续存储的，所以为了提高性能应事先将对象的容量指定为至少100

个字符大小，以避免多次进行内存的重新分配。可使用reserve函数达到这一目的（见9.4节）。

习题9.40

程序如下：

```
// 9-40.cpp
// 接收下列两个string对象：
// string q1("When lilacs last in the dooryard bloom'd");
// string q2("The child is father of the man");
// 然后使用assign和append操作，创建string对象：
// string sentence("The child is in the dooryard");
#include <iostream>
#include <string>
using namespace std;

int main()
{
     string q1("When lilacs last in the dooryard bloom'd");
     string q2("The child is father of the man");
     string sentence;

     // 将sentence赋值为"The child is "
     sentence.assign(q2.begin(), q2.begin() + 13);

     // 在sentence末尾添加"in the dooryard"
     sentence.append(q1.substr(q1.find("in"), 15));

     // 输出sentence
     cout << sentence << endl;

     return 0;
}
```

习题9.43

程序如下：

```
// 9-43.cpp
// 使用stack对象处理带圆括号的表达式：
// 遇到左圆括号时，将其标记下来；然后在遇到右圆括号时，
// 弹出stack对象中这两边括号之间的相关元素（包括左圆括号），
// 接着在stack对象中压入一个值'@'，
// 用以表明这个用一对圆括号括起来的表达式已经被替换
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main()
{
     stack<char> sexp; // 处理表达式的stack对象
     string exp;       //存储表达式的string对象

     // 读入表达式
     cout << "Enter a expression:" << endl;
     cin >> exp;

     // 处理表达式
     string::iterator iter = exp.begin();
```

```
    while (iter != exp.end()) {
        if (*iter != ')') //读到的字符不是右圆括号
            sexp.push(*iter);
        else {
             // 读到的字符是右圆括号，弹出元素，直到栈顶为左圆括号或栈为空
            while (sexp.top() != '(' && !sexp.empty()) {
                sexp.pop();
            }

            if (sexp.empty()) // 栈为空
                cout << "parentheses are not matched" << endl;
            else {            // 栈顶为左圆括号
                sexp.pop();
                sexp.push('@');
            }
        }
        ++iter;
    }

    return 0;
}
```

第 10 章部分习题答案

习题10.1

```
// 10-1.cpp
// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vecotr容器里
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
{
    pair<string, int> sipr;
    string str;
    int ival;
    vector< pair<string, int> > pvec;  // 存储pair对象的vector对象

    // 读入一系列string和int型数据
    cout << "Enter a string and an integer(Ctrl+Z to end):"
         << endl;
    while (cin >> str >> ival) {
        sipr = make_pair(str, ival);   // 生成pair对象
        pvec.push_back(sipr);          // 将pair对象存储在vector容器
    }

    return 0;
}
```

注意，存储pair对象的vector对象，其类型为vector< pair<string,int> >，两个>符号之间必须有空格，否则，系统会将其误认为提取操作符>>，出现编译错误。

习题10.2

方法一：在定义pair对象时提供初始化式从而创建pair对象，程序如下：

```
// 10-2(1).cpp
// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vecotr容器里。
// 采用提供初始化式的方法创建pair对象
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
```

```
{
     string str;
     int ival;
     vector< pair<string, int> > pvec;

     // 读入一系列string和int型数据
     cout << "Enter a string and an integer(Ctrl+Z to end):"
          << endl;
     while (cin >> str >> ival) {
          pair<string, int> sipr(str, ival);// 创建pair对象
          pvec.push_back(sipr); // 将pair对象存储在vector容器
     }

     return 0;
}
```

方法二：直接访问pair对象的数据成员从而生成pair对象，程序如下：

```
// 10-2(2).cpp
// 读入一系列string和int型数据，将每一组存储在一个pair对象中，
// 然后将这些pair对象存储在vecotr容器里。
// 采用直接访问数据成员的方法生成pair对象。
#include <iostream>
#include <utility> // 使用其中的pair类型
#include <vector>
#include <string>
using namespace std;

int main()
{
     pair<string, int> sipr;
     string str;
     int ival;
     vector< pair<string, int> > pvec;

     // 读入一系列string和int型数据
     cout << "Enter a string and an integer(Ctrl+Z to end):"
          << endl;
     while (cin >> str >> ival) {
          // 生成pair对象
          sipr.first = str;
          sipr.second = ival;
          pvec.push_back(sipr); // 将pair对象存储在vector容器
     }

     return 0;
}
```

方法三：使用make_pair函数生成pair对象，见习题10.1解答。

这三种方法在程序编写和理解上没有太大差别。不过笔者个人更喜欢第三种方法，因为使用make_pair函数似乎可以更明确地表明生成pair对象这一行为。

习题10.3

关联容器和顺序容器的本质差别在于：关联容器通过键（key）存储和读取元素，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

习题10.4

list类型适用于需要在容器的中间位置插入和删除元素的情况。例如，以无序方式读入一系列学

生数据，并按学号顺序存储。

vector类型适用于需要随机访问元素的情况。例如，在序号为1..n的一系列人员当中，访问第x个人的信息。

deque类型适用于需要在容器的尾部或首部插入和删除元素的情况。例如，对服务窗口进行管理，先来的顾客先得到服务。

map类型适用于需要键-值对的集合的情况。例如，字典、电话簿的建立和使用。

set类型适用于需要使用键集合的情况。例如，黑名单的建立和使用。

习题10.5

可定义如下的map对象wordLines：

```
map < string, list<int> > wordLines;
```

习题10.6

可以定义map对象以vector<int>::iterator和pair<int, string>为键关联int型对象。

不能定义map对象以list<int>::iterator为键关联int型对象。因为键类型必须支持 < 操作，而list容器的迭代器类型不支持<操作。

习题10.7

mapped_type、key_type和value_type分别是：vector<int>、int和pair< const int, vector<int> >。

习题10.8

假设map的迭代器为iter，要赋给元素的值为val，则可以用iter->second = val;语句给map的元素赋值。（注意，键是不能修改的，所以只能给值成员赋值。）

习题10.9

可以建立一个map对象，保存所读入的单词及其出现次数（以单词为键，对应的值为单词的出现次数）。

对于map容器，如果下标所表示的键在容器中不存在，则添加新元素，利用这一特性可编写程序如下：

```
// 10-9.cpp
// 通过建立map对象保存所读入的单词及其出现次数,
// 统计并输出所读入的单词出现的次数
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    map<string, int> wordCount;
    string word;

    // 读入单词并统计其出现次数
```

```
    cout << "Enter some words(Ctrl+Z to end):"
         << endl;
    while (cin >> word)
        ++wordCount[word];  // word的出现次数加1

    // 输出结果
    cout << "word\t\t" << "times" << endl;
    for (map<string, int>::iterator iter = wordCount.begin();
        iter != wordCount.end(); ++iter)
        cout << (*iter).first << "\t\t"
             << (*iter).second << endl;
    return 0;
}
```

习题10.10

程序段

```
map<int, int> m;
m[0] = 1;
```

的功能是：首先创建一个空的map容器m，然后在m中增加一个键为0的元素，并将其赋值为1。而程序段

```
vector<int> v;
v[0] = 1;
```

将出现运行时错误。因为vector容器v为空，其中下标为0的元素不存在。对于vector容器而言，不能对尚不存在的元素直接赋值，只能使用push_back、insert等函数增加元素。

习题10.11

可用作map容器对象的下标的类型必须是支持<操作的类型。

下标操作符返回的类型为map容器中定义的mapped_type类型。例如，对于如下定义的对象：

```
map<string, int> wordCount;
```

可用作其下标的类型为string以及C风格字符串类型（包括字面值、数组名和指针），下标操作符返回的类型为int。

习题10.12

使用insert函数对map对象进行插入操作时，如果试图插入的元素所对应的键已经在容器中，则insert将不做任何操作。而且，带一个键-值pair形参的insert函数将返回一个pair对象，该对象包含一个迭代器和一个bool值，其中迭代器指向map中具有相应键的元素，而bool值则表示是否插入了该元素。

利用上述特点，可编写程序如下：

```
// 10-12.cpp
// 通过建立map对象保存所读入的单词及其出现次数,
// 统计并输出所读入的单词出现的次数。
// 其中使用insert函数代替下标操作
#include <iostream>
#include <map>
#include <utility>
#include <string>
```

```
using namespace std;

int main()
{
     map<string, int> wordCount;
     string word;

     // 读入单词并统计其出现次数
     cout << "Enter some words(Ctrl+Z to end):"
          << endl;
     while (cin >> word) {
          // 插入元素<word, 1>
          pair<map<string, int>::iterator, bool> ret =
                    wordCount.insert(make_pair(word, 1));
          if (!ret.second) // 该单词已在容器中存在
               ++ret.first->second; // 将该单词的出现次数加1
     }

     // 输出结果
     cout << "word\t\t" << "times" << endl;
     for (map<string, int>::iterator iter = wordCount.begin();
     iter != wordCount.end(); ++iter)
          cout << (*iter).first << "\t\t"
               << (*iter).second << endl;

     return 0;
}
```

使用下标操作的程序更简洁，更容易编写和阅读，而insert函数的返回值的使用比较复杂。但使用insert函数可以避免使用下标操作所带来的副作用，即避免对新插入元素的不必要的值初始化，而且可以显式表示元素的插入（下标操作是隐式表示元素的插入），有其优点。

习题10.13

参数类型为pair<const string, vector<int> >。

返回值类型为pair<map<string, vector<int> >::iterator, bool>。

习题10.14

前者返回map容器中给定键k的出现次数，其返回值只能是0或1；后者在map容器中存在按给定键k索引的元素的情况下，返回指向该元素的迭代器，否则返回超出末端迭代器。

习题10.15

count适合用于解决判断map容器中某键是否存在的问题，而find适合用于解决在map容器中查找指定键对应的元素的问题。

习题10.16

假设map对象为xmap，要查找的键为k，则可以定义并初始化如下变量iter：

```
map<string, vector<int> >::iterator iter = xmap.find(k);
```

习题10.17

该程序中使用find函数，是为了当文本中出现的单词word是要转换的单词（即以该单词为键的元素存在于map容器trans_map中）时获取该元素的引用，以便获取对应的转换后的单词。

如果使用下标操作符，则必须先通过使用count函数来判断元素是否存在，否则，当元素不存在时，会创建新的元素并插入到容器中，从而导致单词转换结果出错。

习题10.18

程序如下：

```
// 10-18.cpp
// 定义一个map对象，其元素的键是家族姓氏，
// 而值则是存储该家族孩子名字的vector对象。
// 进行基于家族姓氏的查询，输出该家族所有孩子的名字
#include <iostream>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main()
{
    map< string, vector<string> > children;
    string surname, childName;

    // 读入条目（家族姓氏及其所有孩子的名字）
    do {
        cout << "Enter surname(Ctrl+Z to end):" << endl;
        cin >> surname;
        if (!cin)  // 读入结束
            break;

        // 插入新条目
        vector<string> chd;
        pair<map<string, vector<string> >::iterator, bool> ret =
                children.insert(make_pair(surname, chd));

        if (!ret.second) {// 该家族姓氏已在map容器中存在
            cout << "repeated surname: " << surname << endl;
            continue;
        }

        cout << "Enter children's name(Ctrl+Z to end):" << endl;
        while (cin >> childName) // 读入该家族所有孩子的名字
            ret.first->second.push_back(childName);
        cin.clear(); // 使输入流重新有效
    } while (cin);

    cin.clear(); // 使输入流重新有效

    // 读入要查询的家族
    cout << "Enter a surname to search:" << endl;
    cin >> surname;

    // 根据读入的家族姓氏进行查找
    map< string, vector<string> >::iterator iter =
                  children.find(surname);
```

```
    // 输出查询结果
    if ( iter == children.end()) // 找不到该家族姓氏
        cout << "no this surname: " << surname << endl;
    else {// 找到了该家族姓氏
        cout << "children: " << endl;
        // 输出该家族中所有孩子的名字
        vector<string>::iterator it = iter->second.begin();
        while (it != iter->second.end())
            cout << *it++ << endl;
    }

    return 0;
}
```

习题10.19

map对象的类型修改为map< string, vector< pair<string, string> > >。

程序如下：

```
// 10-19.cpp
// 定义一个map对象，其元素的键是家族姓氏，
// 而值则是vector对象，该vector对象存储pair类型的对象，
// pair对象记录每个孩子的名字和生日。
// 进行基于家族姓氏的查询，输出该家族所有孩子的名字和生日
#include <iostream>
#include <utility>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main()
{
    map< string, vector< pair<string, string> > > children;
    string surname, childName, birthday;

    // 读入条目（家族姓氏及其所有孩子的名字和生日）
    do {
        cout << "Enter surname(Ctrl+Z to end):" << endl;
        cin >> surname;

        if (!cin)  // 读入结束
            break;

        // 插入新条目
        vector< pair<string, string> > chd;
        pair<map<string, vector< pair<string, string> > >::iterator, bool> ret =
        children.insert(make_pair(surname, chd));
        if (!ret.second) {// 该家族姓氏已在map容器中存在
            cout << "repeated surname: " << surname << endl;
            continue;
        }

        cout << "Enter children's name and birthday(Ctrl+Z to end):"
             << endl;
        // 读入该家族所有孩子的名字和生日
        while (cin >> childName >> birthday) {
            ret.first->second.push_back(make_pair(childName, birthday));
        }
```

```
        cin.clear(); // 使输入流重新有效
    } while (cin);

    cin.clear(); // 使输入流重新有效

    // 读入要查询的家族
    cout << "Enter a surname to search:" << endl;
    cin >> surname;

    // 根据读入的家族姓氏进行查找
    map< string, vector< pair<string, string> > >::iterator iter;
    iter = children.find(surname);

    // 输出查询结果
    if ( iter == children.end()) // 找不到该家族姓氏
        cout << "no this surname: " << surname << endl;
    else {// 找到了该家族姓氏
        cout << "children\t\tbirthday" << endl;
        // 输出该家族中所有孩子的名字和生日
        vector< pair<string, string> >::iterator it =
                                iter->second.begin();
        while (it != iter->second.end()) {
            cout << it->first << "\t\t" << it->second << endl;
            it++;
        }
    }

    return 0;
}
```

习题10.20

可以使用map类型的应用如下：

- 字典：map对象定义为map<string, string> dictionary。
- 电话簿：map对象定义为map<string, string> telBook。
- 商品价目表：map对象定义为map<string, float> priceList。

可以使用下标操作符或insert函数插入元素，使用find函数读取元素。

习题10.21

map容器和set容器的差别在于：map容器是键-值对的集合，set而容器只是键的集合；map类型适用于需要了解键与值的对应的情况，例如，字典（需要了解单词（键）与其解释（值）的对应情况），而set类型适用于只需判断某值是否存在的情况，例如，判断某人的名字是否在黑名单中。

习题10.22

set容器和list容器的主要差别在于：set容器中的元素不能修改，而list容器中的元素无此限制；set容器适用于保存元素值不变的集合，而list容器适用于保存会发生变化的元素。

习题10.23

程序如下：

```
// 10-23.cpp
// 函数restricted_wc:
```

```
// 根据形参所指定文件建立单词排除集，
// 将被排除的单词存储在vector对象中，
// 并从标准输入设备读入文本，对不在排除集中的单词进行出现次数统计。
// 主函数例示restricted_wc函数的使用
#include <iostream>
#include <fstream>
#include <map>
#include <vector>
#include <string>
using namespace std;

void restricted_wc(ifstream &removeFile,
                   map<string, int> &wordCount)
{
    vector<string> excluded; // 保存被排除的单词
    string removeWord;
    while (removeFile >> removeWord)
        excluded.push_back(removeWord);

    // 读入文本并对不在排除集中的单词进行出现次数统计
    cout << "Enter text(Ctrl+Z to end):" << endl;
    string word;
    while (cin >> word) {
        bool find = false;

        // 确定该单词是否在排除集中
        vector<string>::iterator iter = excluded.begin();
        while (iter != excluded.end()) {
            if (*iter == word) {
                find = true;
                break;
            }
            ++iter;
        }

        // 如果单词不在排除集中，则进行计数
        if (!find)
            ++wordCount[word];
    }
}

int main()
{
    map<string, int> wordCount;
    string fileName;

    // 读入包含单词排除集的文件的名字并打开相应文件
    cout << "Enter filename:" << endl;
    cin >> fileName;
    ifstream exFile(fileName.c_str());
    if (!exFile) {// 打开文件失败
        cout << "error: can not open file:" << fileName << endl;
        return -1;
    }

    // 调用restricted_wc函数，
    // 对输入文本中不在排除集中的单词进行出现次数统计
    restricted_wc(exFile, wordCount);

    // 输出统计结果
    cout << "word\t\t" << "times" << endl;
```

```
    map<string, int>::iterator iter = wordCount.begin();
    while (iter != wordCount.end()) {
        cout << iter->first << "\t\t" << iter->second << endl;
        iter++;
    }

    return 0;
}
```

使用set的好处是：可以简单地使用count函数来检查单词是否出现在排除集中（而使用vector则需用循环比较来完成）。

注意，也可以使用标准库中提供的泛型算法find来检查单词是否出现在排除集中，将restricted_wc函数中的第二个while循环修改如下：

```
while (cin >> word) {
if (find(excluded.begin(), excluded.end(), word) == excluded.end())
        // 该单词不在排除集中
        ++wordCount[word];  // 进行计数
}
```

习题10.24

```
// 10-24.cpp
// 建立一个单词排除集，
// 用于识别以's' 结尾、但这个结尾的's' 又不能删除的单词。
// 使用这个排除集删除输入单词尾部的's' 生成该单词的非复数版本。
// 如果输入的是排除集中的单词，则保持该单词不变
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
    set<string> excluded;

    // 建立单词排除集
    excluded.insert("success");
    excluded.insert("class");
    //...// 可以在此处继续插入其他以s结尾的单词

    string word;
    cout << "Enter a word(Ctrl+Z to end)" << endl;
    // 读入单词并根据排除集生成该单词的非复数版本
    while (cin >> word) {
        if (!excluded.count(word)) // 该单词未在排除集合中出现
            word.resize(word.size()-1);// 去掉单词末尾的's'
        cout << "non-plural version: " << word << endl;
        cout << "Enter a word(Ctrl+Z to end)" << endl;
    }

    return 0;
}
```

习题10.25

可通过用户的输入数据来模拟读书及时间的流逝：如果用户希望选择一本书来阅读，则从vector

容器中随机选择一本书提供给用户，从vector容器中删除该书并将该书放入set中；如果用户的输入表明最后没有读这本书，则从set中删除该书并将该书重新放入到vector容器中。

程序如下：

```
// 10-25.cpp
// 定义一个vector容器，存储在未来6个月里要阅读的书的名字，
// 定义一个set，用于记录已经看过的书名。
// 本程序支持从vector中选择一本没有读过而现在要读的书，
// 并将该书名放入记录已读书目的set中，
// 并且支持从已读书目的set中删除该书的记录。
// 在虚拟的6个月后，输出已读书目和还没有读的书目
#include <iostream>
#include <set>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    vector<string> books;
    set<string> readedBooks;
    string name;

    // 建立保存未来6个月要阅读的书名的vector对象
    cout << "Enter names for books you'd like to read\(Ctrl+Z to end):"
         << endl;
    while (cin >> name)
        books.push_back(name);

    cin.clear(); // 使流对象重新有效

    bool timeOver = false;
    string answer, bookName;
    // 用当前系统时间设置随机数发生器种子
    srand( (unsigned)time( NULL ) );

    // 模拟随时间的流逝而改变读书记录
    while (!timeOver && !books.empty()) {
        // 时间未到6个月且还有书没有读过
        cout << "Would you like to read a book?(Yes/No)" << endl;
        cin >> answer;

        if (answer[0] == 'y' || answer[0] == 'Y') {
            // 在vector中随机选择一本书
            int i = rand() % books.size(); // 产生一个伪随机数
            bookName = books[i];
            cout << "You can read this book: "
                 << bookName << endl;
            readedBooks.insert(bookName);   // 将该书放入已读集合
            books.erase(books.begin() + i); // 从vector对象中删除该书

            cout << "Did you read it?(Yes/No)" << endl;
            cin >> answer;
            if (answer[0] == 'n' || answer[0] == 'N') {
                // 没有读这本书
                readedBooks.erase(bookName);   // 从已读集合中删除该书
                books.push_back(bookName);     // 将该书重新放入vector中
            }
```

```
        }

        cout << "Time over?(Yes/No)" << endl;
        cin >> answer;
        if (answer[0] == 'y' || answer[0] == 'Y') {
            // 虚拟的6个月结束了
            timeOver = true;
        }
    }
    if (timeOver) {// 虚拟的6个月结束了
        // 输出已读书目
        cout << "books read:" << endl;
        for (set<string>::iterator sit = readedBooks.begin();
              sit != readedBooks.end(); ++sit)
            cout << *sit << endl;
        // 输出还没有读的书目
        cout << "books not read:" << endl;
        for (vector<string>::iterator vit = books.begin();
              vit != books.end(); ++vit)
            cout << *vit << endl;
    }
    else
        cout << "Congratulations! You have read all these books."
             << endl;

    return 0;
}
```

习题10.26

程序如下：

```
// 10-26.cpp
// 建立作者及其作品的multimap容器。
// 使用find 函数在multimap中查找元素，并调用erase将其删除。
// 当所寻找的元素不存在时，程序依然能正确执行
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    multimap<string, string>  authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        while (cin >> work)
            authors.insert(make_pair(author, work));
        cin.clear();// 读入了一位作者的所有作品后使流对象重新有效
    }while (cin);
    cin.clear(); // 使流对象重新有效
```

```
    // 读入要找的作者
    cout << "Who is the author that you want erase:" << endl;
    cin >> searchItem;

    // 找到该作者对应的第一个元素
    multimap<string,string>::iterator iter =
                                authors.find(searchItem);
    if (iter != authors.end())
        // 删除该作者的所有作品
        authors.erase(searchItem);
    else
        cout << "Can not find this author!" << endl;

    // 输出multimap对象
    cout << "author\t\twork:" << endl;
        for (iter = authors.begin(); iter != authors.end(); ++iter)
            cout << iter->first << "\t\t" << iter->second << endl;

    return 0;
}
```

对find函数所返回的迭代器进行判断，当该迭代器指向authors中的有效元素时才进行erase操作，从而保证当所寻找的元素不存在时，程序依然能正确执行。

习题10.27

程序如下：

```
// 10-27.cpp
// 建立作者及其作品的multimap容器。
// 使用equal_range函数获取迭代器，然后删除一段范围内的元素。
// 当所寻找的元素不存在时，程序依然能正确执行
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    multimap<string, string>  authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        while (cin >> work)
            authors.insert(make_pair(author, work));

        cin.clear();// 读入了一位作者的所有作品后使流对象重新有效
    }while (cin);

    cin.clear(); // 使流对象重新有效
```

```
    // 读入要找的作者
    cout << "Who is the author that you want erase:" << endl;
    cin >> searchItem;

    // 确定该作者对应的multimap元素的范围
    typedef multimap<string, string>::iterator itType;
    pair<itType, itType> pos = authors.equal_range(searchItem);

    if (pos.first != pos.second)
        // 删除该作者的所有作品
        authors.erase(pos.first, pos.second);
    else
        cout << "Can not find this author!" << endl;

    // 输出multimap对象
    cout << "author\t\twork:" << endl;
        for (itType iter = authors.begin();
              iter != authors.end(); ++iter)
            cout << iter->first << "\t\t" <<iter->second << endl;

    return 0;
}
```

注意，equal_range函数返回存储一对迭代器的pair对象。如果multimap中存在与参数匹配的元素，则pair对象中的两个迭代器分别指向该键关联的第一个实例和最后一个实例的下一位置；如果找不到与参数匹配的元素，则对象中的两个迭代器都指向此键应插入的位置。

习题10.28

```
// 10-28.cpp
// 建立作者及其作品的multimap容器。
// 以下面的格式按姓名首字母的顺序输出作者及其作品:
// Author Names Beginning with 'A':
// Author, book, book, ...
// ...
// Author Names Beginning with 'B':
// ...
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
    multimap<string, string>  authors;
    string author, work, searchItem;

    // 建立作者及其作品的multimap容器
    do {
        cout << "Enter author name(Ctrl+Z to end):" << endl;
        cin >> author;

        if (!cin)
            break;

        cout << "Enter author's works(Ctrl+Z to end):" << endl;
        while (cin >> work)
            authors.insert(make_pair(author, work));

        cin.clear();// 读入了一位作者的所有作品后使流对象重新有效
    }while (cin);
```

```
    // 输出multimap对象
    typedef multimap<string, string>::iterator itType;
    itType iter = authors.begin();
    if (iter == authors.end()) { // multimap容器为空
        cout << "empty multimap!" << endl;
        return 0;
    }
    string currAuthor, preAuthor;// 记录当前作者及其前一作者
    do {
        currAuthor = iter->first;

        if (preAuthor.empty() || currAuthor[0] != preAuthor[0])
            // 如果出现了首字母不同的作者，则输出该首字母
            cout << "Author Names Beginning with '"
                 << iter->first[0] << "':" << endl;

        // 输出作者
        cout << currAuthor;

        // 输出该作者的所有作品
        pair<itType, itType> pos = authors.equal_range(iter->first);
        while (pos.first != pos.second) {
            cout << ", " << pos.first->second;
            ++pos.first;
        }
        cout << endl; // 输出了一个作者的所有作品后，换行

        iter = pos.second; // iter指向下一作者
        preAuthor = currAuthor; // 将当前作者设为前一作者
    }while (iter != authors.end());

    return 0;
}
```

习题10.29

其含义为：迭代器对pos当中第一个迭代器所指向的muiltimap元素的值。此处的pos是一个pair对象，pos中存储由函数equal_range返回的一对迭代器，其中第一个迭代器指向键 search_item所关联的第一个实例，第二个迭代器指向键search_item所关联的最后一个实例的下一位置。pos.first访问这对迭代器中的第一个迭代器，而pos.first->second则访问该迭代器所指向的muiltimap元素的值（即键search_item所关联的值，程序中search_item的值为“Alain de Botton”，所以pos.first->second就是作者“Alain de Botton”写的某本书的书名）。

习题10.30

TextQuery类的成员函数已在10.6.4节中给出，此处不再赘述。

习题10.31

（假设要查询的单词为xstr）如果没有找到该单词，则main函数输出：

```
xstr occurs 0 times
```

习题10.32

主程序如下：

```
// 10-32.cpp
// 文本查询主程序
// 使用以vector容器存储行号的TextQuery类
#include "TextQuery.hpp"

string make_plural(size_t, const string&, const string&);
ifstream& open_file(ifstream&, const string&);
void print_results(const vector<TextQuery::line_no>& locs,
                   const string& sought, const TextQuery &file)
{
    // 如果找到单词sought，则输出该单词出现的行数
    typedef vector<TextQuery::line_no> line_nums;
    line_nums::size_type size = locs.size();
    cout << "\n" << sought << " occurs "
         << size << " "
         << make_plural(size, "time", "s") << endl;

    // 输出出现该单词的每一行
    line_nums::const_iterator it = locs.begin();
    for ( ; it != locs.end(); ++it) {
        cout << "\t(line "
             // 行号从1开始
             << (*it) + 1 << ") "
             << file.text_line(*it) << endl;
    }
}

// main函数接受文件名为参数
int main(int argc, char **argv)
{
    // open the file from which user will query words
    ifstream infile;
    if (argc < 2 || !open_file(infile, argv[1])) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }

    TextQuery tq;
    tq.read_file(infile);  // 建立map容器

    // 循环接受用户的查询要求并输出结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        cin >> s;
        //将s变为小写
        string ret;
        for (string::const_iterator it = s.begin();
                                    it != s.end(); ++it) {
             ret += tolower(*it);
        }
        s = ret;

        // 如果用户输入文件结束符或字符'q'及'Q'，则结束循环
        if (!cin || s == "q" || s == "Q") break;
```

```
        // 获取出现所查询单词所有行的行号
        vector<TextQuery::line_no> locs = tq.run_query(s);

        // 输出出现次数及所有相关文本行
        print_results(locs, s, tq);
     }
    return 0;
}
```

TextQuery类的头文件如下：

```
// TextQuery.hpp(for 10-32)
// TextQuery类的头文件
// 使用vector容器存储行号
#ifndef TEXTQUERY_H
#define TEXTQUERY_H
#include <string>
#include <vector>
#include <map>
#include <cctype>
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class TextQuery {
public:
    // 类型别名
    typedef string::size_type str_size;
    typedef vector<string>::size_type line_no;

    // 接口：
    // read_file建立给定文件的内部数据结构
    void read_file(ifstream &is)
         { store_file(is); build_map(); }

    // run_query查询给定单词并返回该单词所在行的行号集合
    vector<line_no> run_query(const string&) const;

    // text_line返回输入文件中指定行号对应的行
    string text_line(line_no) const;

private:
    //read_file所用的辅助函数
    void store_file(ifstream&); // 存储输入文件
    void build_map(); // 将每个单词与一个行号集合相关联

    //保存输入文件
    vector<string> lines_of_text;

    //将单词与出现该单词的行的行号集合相关联
    map< string, vector<line_no> > word_map;

    //去掉标点并将字母变成小写
    static std::string cleanup_str(const std::string&);
};

#endif
```

TextQuery类的实现文件（源文件）如下：

```
// TextQuery.cpp(for 10-32)
// TextQuery类的实现文件（源文件）
// 使用vector容器存储行号
#include "TextQuery.hpp"
#include <sstream>

string TextQuery::text_line(line_no line) const
{
    if (line < lines_of_text.size())
        return lines_of_text[line];
    throw out_of_range("line number out of range");
}

// 读输入文件，将每行存储为lines_of_text的一个元素
void TextQuery::store_file(ifstream &is)
{
    string textline;
    while (getline(is, textline))
       lines_of_text.push_back(textline);
}

// 在输入vector中找以空白为间隔的单词
// 将单词以及出现该单词的行的行号一起放入word_map
void TextQuery::build_map()
{
    // 处理输入vector中的每一行
    for (line_no line_num = 0;
                line_num != lines_of_text.size();
                ++line_num)
    {
        // 一次读一个单词
        istringstream line(lines_of_text[line_num]);
        string word;
        while (line >> word) {
               // 去掉标点
               word = cleanup_str(word);
               // 将行号加入到vector容器中
               if (word_map.count(word) == 0) //单词不在map容器中
                   //下标操作将加入该单词
                   word_map[word].push_back(line_num);
               else {//单词已在map容器中
                    if (line_num != word_map[word].back())
                         //行号与vector容器中最后一个元素不相等
                         word_map[word].push_back(line_num);
               }
          }
    }
}

vector<TextQuery::line_no>
TextQuery::run_query(const string &query_word) const
{
    // 注意，为了避免在word_map中加入单词，使用find函数而不用下标操作
    map<string, vector<line_no> >::const_iterator
                        loc = word_map.find(query_word);
    if (loc == word_map.end())
        return vector<line_no>();  // 找不到，返回空的vector对象
    else
        // 获取并返回与该单词关联的行号vector对象
        return loc->second;
}
```

```
// 去掉标点并将字母变成小写
string TextQuery::cleanup_str(const string &word)
{
    string ret;
    for (string::const_iterator it = word.begin();
                                it != word.end(); ++it) {
        if (!ispunct(*it))
            ret += tolower(*it);
    }
    return ret;
}
```

定义函数make_plural和open_file的源文件如下：

```
// functions.cpp
// 定义函数make_plural和open_file
#include <fstream>
#include <string>
using namespace std;

// 如果ctr不为1，返回word的复数版本
string make_plural(size_t ctr, const string &word,
                           const string &ending)
{
     return (ctr == 1) ? word : word + ending;
}

// 打开输入文件流in并绑定到给定的文件
ifstream& open_file(ifstream &in, const string &file)
{
    in.close();     // close in case it was already open
    in.clear();     // clear any existing errors
    // if the open fails, the stream will be in an invalid state
    in.open(file.c_str()); // open the file we were given
    return in; // condition state is good if open succeeded
}
```

使用vector容器存储行号，在进行行号的插入时，为了不存储重复的行号，需先判断该行号是否已在容器中，再决定是否插入；而使用set容器存储行号，则可以利用set容器所提供的insert函数的特点（如果元素已经存在，insert函数不进行操作），无需判断行号是否已存在，直接利用insert函数插入元素即可，性能相对较高。

从设计上看，vector容器的特点是特别适用于需随机访问元素的情况。而此处对于存储的行号，不需要进行随机访问（只需顺序访问），因而未能充分体现vector容器的优点。而set容器因其插入操作的简单，使得设计更为简洁。因此，此处使用set容器更好。

习题10.33

因为print_results函数调用text_line函数时，传递由run_query函数所获取的set对象中的元素为实参，而由build_map函数生成的map容器里，set对象中出现的元素（即行号）不可能为负数。

第 11 章部分习题答案

习题11.1

```
// 11-1.cpp
// 读取一系列int型数据，并将它们存储到vector对象中，
// 然后使用algorithm头文件中定义的名为count的函数，
// 统计某个指定的值出现了多少次
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int ival, searchValue;
    vector<int> ivec;

    // 读入int型数据并存储到vector对象中，直至遇到文件结束符
    cout << "Enter some integers(Ctrl+Z to end):" << endl;
    while (cin >> ival)
        ivec.push_back(ival);

    cin.clear();// 使输入流重新有效

    // 读入欲统计其出现次数的int值
    cout << "Enter an integer you want to search:" << endl;
    cin >> searchValue;

    // 使用count函数统计该值出现的次数并输出结果
    cout << count(ivec.begin(), ivec.end(), searchValue)
         << " elements in the vector have value "
         << searchValue << endl;

    return 0;
}
```

习题11.4

从函数调用上看没有错误。

调用accumulate函数必须满足的条件包括：容器内的元素类型必须与第三个实参的类型匹配，或者可转换为第三个实参的类型。上述调用中的第三个实参为int类型，而vector对象中的元素的类型为double类型，可以转换为int类型。

但计算的结果不准确。因为将double类型转换为int类型会截去小数部分，得到的求和结果是各元素的整数部分的和，是一个int类型的值，与实际的元素值总和相比会有比较大的误差。

习题11.7

(a) 有错。错误在于 vec 是一个空的 vector 容器，而 copy 函数试图将 list 容器 lst 中的元素复制到 vec。更正为：

```
copy(lst.begin(), lst.end(), back_inserter(vec));
```

(b) 有错。错误在于虽然为 vector 对象 vec 分配了内存，但该对象仍是一个空的 vector 对象，而在空的容器上调用 fill_n 函数是错误的。更正为：

```
vector<int> vec;
vec.resize(10);
fill_n(vec.begin(), 10, 0);
```

习题11.10

该程序段可改写如下：

```
// 使用find_if函数统计长度大6的单词的数目
vector<string>::iterator iter = words.begin();
vector<string>::size_type wc = 0;
while ((iter = find_if(iter, words.end(), GT6)) != words.end()) {
     ++wc; // 单词计数加1
     // 找到一个匹配的单词后，iter加1，继续在输入序列的剩余部分中查找
     ++iter;
}
```

习题11.13

三种插入迭代器的区别在于插入元素的位置不同：

- back_inserter，使用 push_back 实现在容器末端插入。
- front_inserter，使用 push_front 实现在容器前端插入。
- inserter，使用 insert 实现在容器中指定位置插入。

因此，除了所关联的容器外，inserter还带有第二个实参——指向插入起始位置的迭代器。

习题11.16

程序如下：

```
// 11-16.cpp
// 使用copy算法将一个文件的内容写到标准输出中
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <string>
#include <iterator>
#include <algorithm>
using namespace std;

int main()
{
     string fileName;

     // 输入文件名
```

```
    cout << "Enter input file name: " << endl;
    cin >> fileName;

    // 打开文件
    ifstream inFile(fileName.c_str());
    if (!inFile) {
        cout << "Can not open file: " << fileName << endl;
        return EXIT_FAILURE;
    }

    // 使用copy算法将文件的内容写到标准输出中
    ostream_iterator<string> outIter(cout, " ");// 以空格分隔数据
    istream_iterator<string> inIter(inFile), eof;
    copy(inIter, eof, outIter);

    // 关闭文件
    inFile.close();

    return 0;
}
```

习题11.19

```
// 11-19.cpp
// 使用reverse_iterator对象以逆序输出vector容器对象的内容
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int ia[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> ivec(ia, ia+10);
    vector<int>::reverse_iterator r_iter; // 反向迭代器

    // 逆序输出vector容器对象的元素
    for (r_iter = ivec.rbegin(); r_iter != ivec.rend(); ++r_iter)
        cout << *r_iter << endl;
    return 0;
}
```

注意，reverse_iterator对象的自增操作使得迭代器指向容器中的前一元素。

习题11.22

给copy算法传递一对反向迭代器实参，即可对指定范围的元素进行逆序复制。需要注意的是第二个反向迭代器实参应指向要复制的第一个元素的前一元素（即指向第2个位置上的元素）。

程序如下：

```
// 11-22.cpp
// 对于一个存储了10个元素的vector对象，
// 将其中第3～7个位置上的元素以逆序复制给list对象
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
```

```
int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> ivec(ia, ia+10);
    list<int> ilst;
    vector<int>::reverse_iterator rstart, rend; // 反向迭代器

    rstart = ivec.rbegin(); // 获得指向尾元素的反向迭代器
    // 获得指向第7个元素的反向迭代器
    for (int cnt1 = 1; cnt1 != 4; ++cnt1)
        ++rstart;

    // 获得指向第2个元素的反向迭代器
    rend = rstart;
    for (int cnt2 = 1; cnt2 != 6; ++cnt2)
        ++rend;

    // 逆序复制元素
    copy(rstart, rend, inserter(ilst, ilst.begin()));

    // 输出list对象，以检验是否为7..3
    for (list<int>::iterator iter = ilst.begin();
                             iter != ilst.end(); ++iter)
        cout << *iter << endl;

    return 0;
}
```

习题11.25

copy算法至少需要使用输入迭代器和输出迭代器；reverse算法至少需要使用双向迭代器；unique算法至少需要使用前向迭代器。

习题11.28

该程序段试图将容器lst中的元素以逆序复制到vector容器vec1中。

其中存在错误：vec1是一个空的vector容器，尚未分配存储空间，所以该程序段运行时会产生内存访问非法的错误。可将定义vec1的语句更正为：

```
vector<int> vec1(100);
```

创建包含100个元素的vector容器vec1，其中的每个元素采用值初始化（初始化为0）。这样一来，就不会存在运行时错误了。

第 12 章部分习题答案

习题12.1

```
class Person{
private:
     std::string name;
     std::string address;
};
```

习题12.4

将Person类的数据成员name、address声明为private以实现信息隐藏，将成员函数getName、getAddress声明为public以提供外界使用的接口，外界通过该接口访问Person类的数据成员。而构造函数通常声明为public以便创建Person类的对象。

习题12.7

封装是一种将低层次的元素组合起来形成新的、高层次实体的技术。例如，函数是封装的一种形式：函数所执行的细节行为被封装在函数本身这个更大的实体中；类也是一个封装的实体：它代表若干成员的聚集，大多数（良好设计的）类类型隐藏了实现该类型的成员（见12.1.2节）。

封装隐藏了内部元素的实现细节（例如，可以调用一个函数但不能访问它所执行的语句），其主要优点在于：避免类内部出现无意的、可能破坏对象状态的用户级错误；使得在修改类的实现时只要保持类的接口不变，就无需改变用户级代码。因此，封装是有用的。

习题12.10

size是Record类中定义的局部类型名字，三个Record函数是Record类的重载构造函数，分别接受不同的参数，byte_count和name是Record类的数据成员，以上成员的访问标号为默认的private，只能在Record类中使用；get_count和get_name是Record类的成员函数，其访问标号为public，可以由程序的任意部分使用。

习题12.13

最简单的一种解决方式是参照12.1.3节至12.2节的内容，给出如下程序：

```
// 12-13.cpp
// 扩展Screen类以包含move、set和display操作。
// 通过执行如下表达式来测试类：
// myScreen.move(4,0).set('#').display(cout)
```

```
#include <iostream>
#include <string>
using namespace std;

class Screen {
public:
     typedef string::size_type index;
     char get() const { return contents[cursor]; }
     inline char get(index ht, index wd) const;
     index get_cursor() const;
     Screen(index hght, index wdth, const string &cntnts);

     // 增加三个成员函数
     Screen& move(index r, index c);
     Screen& set(char);
     Screen& display(ostream &os);

private:
     std::string contents;
     index cursor;
     index height, width;
};

Screen::Screen(index hght, index wdth, const string &cntnts):
         contents(cntnts), cursor(0), height(hght), width(wdth)
{
}

char Screen::get(index r, index c) const
{
     index row = r * width;
     return contents[row + c];
}

inline Screen::index Screen::get_cursor() const
{
     return cursor;
}

// 增加的三个成员函数的定义
Screen& Screen::set(char c)
{
     contents[cursor] = c;
     return *this;
}

Screen& Screen::move(index r, index c)
{
     index row = r * width;
     cursor = row + c;
     return *this;
}

Screen& Screen::display(ostream &os)
{
     os << contents;
     return *this;
}

int main()
{
     // 根据屏幕的高度、宽度和内容的值来创建Screen
     Screen myScreen(5, 6, "aaaaa\naaaaa\naaaaa\naaaaa\naaaaa\n");
```

```
    // 将光标移至指定位置，设置字符并显示屏幕内容
    myScreen.move(4,0).set('#').display(cout);

    return 0;
}
```

这个解决方法已满足了题目提出的要求，但存在一些缺陷：

(1) 创建Screen对象时必须给出表示整个屏幕内容的字符串，即使有些位置上没有内容。

(2) 显示的屏幕内容没有恰当地分行，而是连续显示，因此（4,0）位置上的'#'，在实际显示时不一定正好在屏幕的（4,0）位置，显示效果较差。

(3) 如果创建的Screen对象是一个const对象，则不能使用display函数进行显示（因为const对象只能使用const成员）。

(4) 如果move操作的目的位置超出了屏幕的边界，会出现运行时错误。

要解决第一个缺陷，可以如下修改构造函数：

```
Screen::Screen(index hght, index wdth, const string &cntnts = ""):
        cursor(0), height(hght), width(wdth)
{
    // 将整个屏幕内容置为空格
    contents.assign(hght*wdth, ' ');
    // 用形参string对象的内容设置屏幕的相应字符
    if (cntnts.size() != 0)
        contents.replace(0, cntnts.size(), cntnts);
}
```

要解决第二个缺陷，可以如下修改display函数：

```
Screen& Screen::display(ostream &os)
{
    string::size_type index = 0;
    while (index != contents.size()) {
        os << contents[index];
        if ((index+1) % width == 0) {
            os << '\n';
        }
        ++index;
    }
    return *this;
}
```

要解决第三个缺陷，可以在Screen类定义体中增加如下函数声明：

```
const Screen& display(ostream &os) const;
```

声明display函数的一个重载版本，供const对象使用。并在Screen类定义体外增加该函数的定义如下：

```
const Screen& Screen::display(ostream &os) const
{
    string::size_type index = 0;
    while (index != contents.size()) {
        os << contents[index];
        if ((index+1) % width == 0) {
            os << '\n';
        }
        ++index;
    }
    return *this;
```

```
}
```

注意，两个重载的display函数的函数体完全一样，因此，为了减少重复代码，可以定义一个do_display函数来完成实际的显示工作，而两个重载的display函数都调用该函数。这个起辅助作用的do_display函数可以定义在Screen类的private部分。读者可以自己完成这一工作。

要解决第四个缺陷，可以如下修改move函数：

```
Screen& Screen::move(index r, index c)
{
     // 行、列号均从0开始
     if (r >= height || c >= width) {
          cerr << "invalid row or column" << endl;
          throw EXIT_FAILURE;
     }
     index row = r * width;
     cursor = row + c;

     return *this;
}
```

修改后的move函数对目的行和列的标号进行检查以避免操作越界。

习题12.16

会出现编译错误。因为，成员函数返回类型index是在Screen类的作用域之外的，而在Screen类之外没有定义index类型。要使用Screen类中定义的index类型，必须在index之前加上Screen::。

习题12.19

允许类用户不指定数据成员的初始值的构造函数应该是不带参数的构造函数，而允许类用户指定所有数据成员的初始值的构造函数应该带有三个形参（类型分别与三个数据成员的类型匹配），因此需要提供两个构造函数。

方法一：在NoName类的定义体中public部分增加下面两个构造函数的定义。

```
NoName()
{
}
NoName(std::string *pstr, int iv, double dv)
{
     pstring = pstr;
     ival = iv;
     dval = dv;
}
```

方法二：在NoName类的定义体中public部分增加下面两个构造函数的声明。

```
NoName();
NoName(std::string *pstr, int iv, double dv);
```

然后在 NoName 类的定义体外给出它们的定义：

```
NoName::NoName()
{
}
NoName::NoName(std::string *pstr, int iv, double dv)
{
```

```
        pstring = pstr;
        ival = iv;
        dval = dv;
    }
```

注意，方法一中定义的构造函数默认为inline函数，方法二中也可以将两个构造函数显式指定为inline函数。

习题12.22

根据类X中数据成员的定义次序，应该首先初始化rem，然后再初始化base，上述初始化列表的效果是用尚未初始化的base值来初始化rem，所以出错。

可更正为：

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

或者：

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

习题12.25

```
Sales_item(std::istream &is = std::cin);
```

习题12.28

接受一个string的Sales_item构造函数应该为explicit。因为如果不将其声明为explicit，则编译器可以使用它进行隐式类型转换（将一个string对象转换为Sales_item对象），而这种行为是容易发生语义错误的。

将构造函数设置为explicit的好处是可以避免因隐式类型转换而带来的错误；缺点是当用户的确需要进行相应的类型转换时，不能依靠隐式类型转换，必须显式地创建临时对象。

习题12.31

因为pair类定义了构造函数，所以尽管其数据成员为public，也不能采用这种显式初始化方式。只有没有定义构造函数且其全体数据成员均为public的类，才可以采用与初始化数组元素相同的方式初始化其成员。

可更正为：

```
pair<int, int> p2(0, 42);
```

习题12.34

```
Sales_item add(const Sales_item &obj1, const Sales_item &obj2)
{
    if (!obj1.same_isbn(obj2))
        return obj1;
    Sales_item temp;
    temp.isbn = obj1.isbn;
    temp.units_sold = obj1.units_sold + obj2.units_sold;
    temp.revenue = obj1.revenue + obj2.revenue;
    return temp;
}
```

注意，需要将该函数指定为Sales_item类的友元。

相应的Sales_item类可定义如下：

```
class Sales_item {
public:
    // 将add函数指定为Sales_item类的友元
    friend Sales_item add(const Sales_item&, const Sales_item&);
    bool same_isbn(const Sales_item &rhs) const
    { return isbn == rhs.isbn; }

    // 构造函数
    Sales_item(const std::string &book = ""):
           isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is)
    {
        cin >> isbn >> units_sold >> revenue;
    }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

习题12.37

可如下定义Account类：

```
class Account {
public:
    // 构造函数
    Account(std::string own, double amnt)
    {
        owner = own;
        amount = amnt;
    }
    // 计算余额
    void applyint()
    {
        amount += amount * interestRate;
    }
    // 返回当前利率
    static double rate()
    {
        return interestRate;
```

```
    }
    // 设置新的利率
    static void rate(double newRate)
    {
        interestRate = newRate;
    }
    // 存款
    double deposit(double amnt)
    {
        amount += amnt;
        return amount;
    }
    // 取款
    bool withdraw(double amnt)
    {
        if (amount < amnt) // 余额不足
            return false;
        else {
            amount -= amnt;
            return true;
        }
    }
    // 查询当前余额
    double getBalance()
    {
        return amount;
    }
private:
    std::string owner;
    double amount;
    static double interestRate;
};
double Account::interestRate = 2.5;
```

注意，static数据成员必须在类定义体外部定义，且只定义一次。一般可将它放在类的实现文件（源文件）中。

习题12.40

Bar类可定义如下：

```
class Bar {
public:
    Foo FooVal()
    {
        callsFooVal++;
        return fval;
    }
private:
    static int ival;
    static Foo fval;
    static int callsFooVal;
};
```

注意，应在Bar类的定义体外部对其static数据成员进行初始化：

```
int Bar::ival = 20;
Foo Bar::fval(0);
int Bar::callsFooVal = 0;
```

第 13 章部分习题答案

习题13.1

复制构造函数是具有如下特点的构造函数：只有单个形参，且形参是对本类类型对象的引用（常用const修饰）。

复制构造函数在下列情况下使用：

- 根据另一个同类型的对象显式或隐式初始化一个对象。
- 复制一个对象，将它作为实参传给一个函数。
- 从函数返回时复制一个对象。
- 初始化顺序容器中的元素。
- 根据元素初始化式列表初始化数组元素。

习题13.4

复制构造函数可编写如下：

```
NoName::NoName(const NoName& other)
{
     pstring = new std::string;
     *pstring = *(other.pstring);
     i = other.i;
     d = other.d;
}
```

也可以采用初始化列表的形式编写如下：

```
NoName::NoName(const NoName& other):
pstring(new std::string(*(other.pstring))), i(other.i), d(other.d) { }
```

习题13.7

一般而言，如果一个类需要定义复制构造函数，则该类也需要定义赋值操作符。具体而言，如果一个类中包含指针型数据成员，或者在进行赋值操作时有一些特定工作要做，则该类通常需要定义赋值操作符。

习题13.10

可以定义一个static数据成员作为计数器，用于为每个雇员产生唯一的雇员标识。因为每个雇员的雇员标识是唯一的，所以该类需要复制构造函数及赋值操作符，以便在“根据已有雇员创建新的雇

员”，或者“将一个雇员赋值给另一个雇员”时提供唯一的雇员标识。

Employee类可定义如下：

```
class Employee {
public:
    // 构造函数
    Employee(): name("NoName"), id(counter)
    {
        ++counter;
    }

    Employee(std::string nm): name(nm), id(counter)
    {
        ++counter;
    }

    Employee(const Employee& other): name(other.name), id(counter)
    {
        ++counter;
    }

    // 赋值操作符
    Employee& operator = (const Employee& rhe)
    {
        name = rhe.name;
        return *this;
    }
private:
    std::string name;
    int id;
    static int counter;
};
```

假设雇员标识为从1开始的整数，在Employee类定义体外部对其static成员进行初始化：

```
int Employee::counter = 1;
```

此处给出的代码中，雇员标识为简单的整数，也可以利用setId成员生成string型的雇员标识，只需进行相应修改即可。例如，如果雇员标识为形如“Empl-0001”的字符串，则可以将Employee类定义如下：

```
class Employee {
public:
    // 构造函数
    Employee(): name("NoName")
    {
        setId();
    }

    Employee(std::string nm): name(nm)
    {
        setId();
    }

    Employee(const Employee& other): name(other.name)
    {
        setId();
    }

    // 赋值操作符
    Employee& operator = (const Employee& rhe)
```

```
    {
        name = rhe.name;
        return *this;
    }
private:
    std::string name;
    std::string id;
    static int counter;
    // 设置雇员ID
    void setId()
    {
        id = "Empl-";
        if (counter < 10)
            id += "000";
        else if(counter < 100)
            id += "00";
        else if(counter < 1000)
            id += "0";
        else {
            std::cerr << "no valid employee id!" << std::endl;
        }

        char buffer[5];
        _itoa(counter, buffer, 10);  // 使用库函数将整数转换为字符串
        id += buffer;
        ++counter; // 计数器加1
    }
};
```

同样，需在Employee类定义体外部对其static成员进行初始化：

```
int Employee::counter = 1;
```

习题13.13

因为该Employee类不需要在析构函数中释放资源，也没有特定的操作要做，所以该类不需要显式定义析构函数，使用编译器合成的析构函数即可满足需要。

习题13.16

习题 13.19 的解答中给出了本节中描述的 Message 类的定义，此处不再赘述。

习题13.19

Message类的save和remove操作可定义如下：

```
void Message::save(Folder& fldr)
{
    addFldr(&fldr);
    fldr.addMsg(this);// 更新相应的目录
}

void Message::remove(Folder& fldr)
{
    remFldr(&fldr);
    fldr.remMsg(this);// 更新相应的目录
}
```

下面给出满足习题 13.16 至习题 13.19 要求的 Message 类和 Folder 类的完整定义：

```
#include <set>
#include <string>
class Message;
class Folder {
public:
     Folder() { }
     // 复制控制成员
     Folder(const Folder&);
     Folder& operator=(const Folder&);
     ~Folder();

     // 在指定Message的目录集中增加/删除该目录
     void save (Message&);
     void remove(Message&);

     // 在该目录的消息集中增加/删除指定Message
     void addMsg(Message*);
     void remMsg(Message*);
private:
     std::set<Message*> messages; // 该目录中的消息集

     // 复制控制成员所使用的实用函数：
     // 将目录加到形参所指的消息集中
     void put_Fldr_in_Messages(const std::set<Message*>&);

     // 从目录所指的所有消息中删除该目录
     void remove_Fldr_from_Messages();
};

class Message {
public:
     // folders自动初始化为空集
     Message(const std::string &str = ""):
     contents (str) { }

     // 复制控制成员
     Message(const Message&);
     Message& operator=(const Message&);
     ~Message();

     // 在指定Folder的消息集中增加/删除该消息
     void save (Folder&);
     void remove(Folder&);

     // 在包含该消息的目录集中增加/删除指定Folder
     void addFldr(Folder*);
     void remFldr(Folder*);
private:
     std::string contents;          // 实际消息文本
     std::set<Folder*> folders;     // 包含该消息的目录

     // 复制构造函数、赋值、析构函数所使用的实用函数：
     // 将消息加到形参所指的目录集中
     void put_Msg_in_Folders(const std::set<Folder*>&);

     // 从消息所在的所有目录中删除该消息
     void remove_Msg_from_Folders();
};
```

```
Folder::Folder(const Folder &f):
messages(f.messages)
{
     // 将该目录加到f所指向的每个消息中
     put_Fldr_in_Messages(messages);
}

// 将该目录加到rhs所指的消息集中
void Folder::put_Fldr_in_Messages(const std::set<Message*> &rhs)
{
     for(std::set<Message*>::const_iterator beg = rhs.begin();
                                        beg != rhs.end(); ++beg)
     (*beg)->addFldr(this); // *beg指向一个消息
}

Folder& Folder::operator=(const Folder &rhs)
{
     if (&rhs != this) {
          remove_Fldr_from_Messages();   // 更新现有消息
          messages = rhs.messages;       // 从rhs复制消息指针集
          // 将该目录加到rhs中的每个消息中
          put_Fldr_in_Messages(rhs.messages);
     }
     return *this;
}

// 从对应消息中删除该目录
void Folder::remove_Fldr_from_Messages()
{
     // 从对应消息中删除该目录
     for(std::set<Message*>::const_iterator beg =
     messages.begin (); beg != messages.end (); ++beg)
     (*beg)->remFldr(this); // *beg指向一个消息
}

Folder::~Folder()
{
     remove_Fldr_from_Messages();
}

void Folder::save(Message& msg)
{
     addMsg(&msg);
     msg.addFldr(this);// 更新相应的消息
}

void Folder::remove(Message& msg)
{
     remMsg(&msg);
     msg.remFldr(this);// 更新相应的消息
}

void Folder::addMsg(Message* msg)
{
     messages.insert(msg);
}

void Folder::remMsg(Message* msg)
{
     messages.erase(msg);
}
```

```
Message::Message(const Message &m):
contents(m.contents), folders(m.folders)
{
     // 将该消息加到指向m的每个目录中
     put_Msg_in_Folders(folders);
}

// 将该消息加到rhs所指的目录集中
void Message::put_Msg_in_Folders(const std::set<Folder*> &rhs)
{
     for(std::set<Folder*>::const_iterator beg = rhs.begin();
                                          beg != rhs.end(); ++beg)
     (*beg)->addMsg(this); // *beg指向一个目录
}

Message& Message::operator=(const Message &rhs)
{
     if (&rhs != this) {
          remove_Msg_from_Folders(); // 更新现有目录
          contents = rhs.contents;   // 从rhs复制消息内容
          folders = rhs.folders;     // 从rhs复制目录指针集
          // 将该消息加到中的每个目录中
          put_Msg_in_Folders(rhs.folders);
     }
     return *this;
}

// 从对应目录中删除该消息
void Message::remove_Msg_from_Folders()
{
     // 从对应目录中删除该消息
     for(std::set<Folder*>::const_iterator beg =
     folders.begin (); beg != folders.end (); ++beg)
     (*beg)->remMsg(this); // *beg指向一个目录
}

Message::~Message()
{
     remove_Msg_from_Folders();
}

void Message::save(Folder& fldr)
{
     addFldr(&fldr);
     fldr.addMsg(this);// 更新相应的目录
}

void Message::remove(Folder& fldr)
{
     remFldr(&fldr);
     fldr.remMsg(this);// 更新相应的目录
}

void Message::addFldr(Folder* fldr)
{
     folders.insert(fldr);
}

void Message::remFldr(Folder* fldr)
{
     folders.erase(fldr);
```

```
}
```

习题13.22

使用计数（use count）是复制控制成员中使用的编程技术。将一个计数器与类指向的对象相关联，用于跟踪该类有多少个对象共享同一指针。创建一个单独类指向共享对象并管理使用计数。由构造函数设置共享对象的状态并将使用计数置为 1。每当由复制构造函数或赋值操作符生成一个新副本时，使用计数加 1。由析构函数撤销对象或作为赋值操作符的左操作数撤销对象时，使用计数减 1。赋值操作符和析构函数检查使用计数是否已减至 0，如果是，则撤销对象。

习题13.25

所谓值型类，是指具有值语义的类，其特征为：对该类对象进行复制时，会得到一个不同的新副本，对副本所做的改变不会影响原有对象。

习题13.28

(a) 对于 `TreeNode` 类，默认构造函数和必要的复制控制成员如下：

```
TreeNode::TreeNode() :
          count(0), left(0), right(0)
{ }

TreeNode::TreeNode(const TreeNode &orig) :
          value(orig.value)
{
    count = orig.count;
    if (orig.left)
        left = new TreeNode(*orig.left);
    else
        left = 0;
    if (orig.right)
        right = new TreeNode(*orig.right);
    else
        right = 0;
}

TreeNode::~TreeNode()
{
    if (left)
        delete left;
    if (right)
        delete right;
}
```

(b) 对于 `BinStrTree` 类，默认构造函数和必要的复制控制成员如下：

```
BinStrTree::BinStrTree() : root(0)
{ }

BinStrTree::BinStrTree(const BinStrTree &orig)
{
    if (orig.root)
        root = new TreeNode(*orig.root);
    else
        root = 0;
```

```
}

BinStrTree::~BinStrTree()
{
    if (root)
        delete root;
}
```

第 14 章部分习题答案

习题14.1

重载操作符与内置操作符的不同之处在于：重载操作符必须具有至少一个类类型或枚举类型的操作数；重载操作符不保证操作数的求值顺序，例如，&&和||的重载版本失去了“短路求值”特性，两个操作数都要进行求值，而且不规定操作数的求值顺序。

重载操作符与内置操作符的相同之处在于：操作符的优先级、结合性及操作数数目均相同。

习题14.4

表达式"cobble" == "stone"中应用了C++语言内置的==版本。

表达式svec1[0] == svec2[0]中应用了string类所定义的==版本。

表达式svec1 == svec2中应用了vector类所定义的==版本。

习题14.7

```
ostream& operator << (ostream& out, const CheckoutRecord& c)
{
    out << c.book_id << "\t" << c.title << endl
        << "date borrowed: " << c.date_borrowed << endl
        << "date due: " << c.date_due << endl
        << "borrower: " << c.borrower.first << ", "
                        << c.borrower.second << endl;
    out << "wait list: " << endl;
    for (vector< pair<string, string>* >::const_iterator
           iter = c.wait_list.begin(); iter != c.wait_list.end(); ++iter)
        out << "\t" << (*iter)->first << ", "
            << (*iter)->second << endl;
    return out;
}
```

注意，Date为用户自定义类，该类一般可以提供重载的<<操作符，所以直接使用该操作符即可。pair和vector为标准库模板类，没有提供重载的<<操作符，所以在CheckoutRecord类的输出操作符中需要逐个输出相应元素。另外，注意输出操作符<<应指定为CheckoutRecord类的友元，为此，需要在CheckoutRecord类的定义体内增加如下声明：

```
friend ostream& operator<<(ostream&, CheckoutRecord&);
```

习题14.10

该Sales_item输入操作符的错误在于：没有对输入失败的情况进行处理，从而导致对象s可能处

于不一致的状态。例如，有可能在成功地读入了一个新的isbn之后遇到了流错误，因此对象s原有的units_sold和revenue成员没变，结果会将另一个isbn与对象s原有的units_sold和revenue数据相关联。

如果将上题中的数据作为输入，则：(a) 的结果与上题相同；(b) 的结果为——将形参 Sales_item 对象的 isbn 成员设置为 10，units_sold 成员和 revenue 成员保持原值不变。

习题14.13

从Sales_item类的应用需求来看，也许该类还可以支持算术操作符-，相应地可以定义一个-=操作符，以支持相应的复合赋值并支持-操作符的实现。

其中复合赋值操作符-=应定义为Sales_item类的成员，而算术操作符-应定义为非成员操作符，它们均用于对ISBN相同的Sales_item对象进行操作，定义如下：

```
Sales_item& Sales_item::operator-=(const Sales_item& rhs)
{
    units_sold -= rhs.units_sold;
    revenue -= rhs.revenue;
    return *this;
}

Sales_item operator-(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);
    ret -= rhs;
    return ret;
}
```

习题14.16

从应用角度考虑，CheckoutRecord类还可以定义其他赋值操作符。例如，当读者办理续借手续的时候，可以通过赋值操作修改相应CheckoutRecord对象的date_due成员，该赋值操作使用Date类型的操作数；当对于某本书有新的读者预约的时候，可以通过赋值操作在相应CheckoutRecord对象的wait_list中增加该读者，该赋值操作使用pair<string, string>类型的操作数。

这两个赋值操作符可定义如下：

```
// 设置新的date_due
CheckoutRecord& CheckoutRecord::operator=(const Date& new_due)
{
    date_due = new_due;
    return *this;
}

// 增加一个等待者
CheckoutRecord& CheckoutRecord::operator=(const
                                  std::pair<string, string>& awaiter)
{
    pair<string, string> *ppa = new pair<string, string>;
    *ppa = awaiter; // 复制形参pair对象
    wait_list.push_back(ppa);
    return *this;
}
```

习题14.19

可将这个操作定义为普通的成员函数，例如，pair<string, string>& get_awaiter (const size_t index)及const pair<string, string>& get_awaiter (const size_t index) const。

习题14.22

ScreenPtr类的相等操作符和不等操作符可定义如下：

```
inline bool
operator==(const ScreenPtr &lhs, const ScreenPtr &rhs)
{
    return lhs.ptr == rhs.ptr;
}

inline bool
operator!=(const ScreenPtr &lhs, const ScreenPtr &rhs)
{
        return !(lhs == rhs); // 根据==操作符定义!=
}
```

注意，关系操作符一般定义为普通非成员函数；此处定义的==操作符应指定为ScreenPtr类的友元；通常==和!=操作符应相互联系起来定义（此处定义==操作符完成实际的对象比较工作，而!=操作符则调用==操作符，亦可定义!=操作符完成实际的对象比较工作，而==操作符则调用!=操作符）。

习题14.25

可为CheckedPtr类定义==、!=、>、>=、<、<=等操作符如下：

```
// 相等操作符
bool operator==(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
             && lhs.curr == rhs.curr;
}

bool operator!=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs == rhs);
}

// 关系操作符
bool operator<(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
             && lhs.curr < rhs.curr;
}

bool operator<=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs > rhs);
}

bool operator>(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return lhs.beg == rhs.beg && lhs.end == rhs.end
```

```
            && lhs.curr > rhs.curr;
}

bool operator>=(const CheckedPtr& lhs, const CheckedPtr& rhs)
{
    return !(lhs < rhs);
}
```

注意，这些操作符一般应定义为非成员函数，如果操作需要访问CheckedPtr类的数据成员，则应指定为CheckedPtr类的友元；对CheckedPtr对象的比较仅在两个对象指向同一数组时才有意义；这些操作符可以相互联系起来定义。

习题14.28

因为对const对象不能使用自增和自减操作符。

习题14.31

可定义如下函数对象：

```
class ifThenElse {
public:
    int operator() (int val1, int val2, int val3)
    {
        return val1 ? val2 : val3;
    }
};
```

该函数对象接受三个int型形参，并测试第一个形参是否为true(非0)值。如果测试成功，就返回第二个形参；否则，就返回第三个形参。

如果希望该函数对象能接受任意类型的形参，可将其定义为类模板（见16.1.2节）：

```
template<typename T>
class ifThenElse {
public:
    T operator() (T val1, T val2, T val3)
    {
        return val1 ? val2 : val3;
    }
};
```

习题14.34

如果使用vector存放int型序列，则可编写程序如下：

```
// 14-34.cpp
// 定义函数对象EQ_cls，测试两个值是否相等；
// 使用标准库算法replace_if和EQ_cls类，
// 替换vector对象中给定值的所有实例
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class EQ_cls {
public:
```

```
    EQ_cls(int val = 0): spValue(val) { }
    bool operator() (const int &ival)
    {
        return ival == spValue;
    }
private:
    int spValue;
};

int main()
{
    vector<int> ivec;
    int ival;
    cout << "Enter numbers(Ctrl+Z to end): " << endl;
    // 读入vector元素
    while (cin >> ival)
        ivec.push_back(ival);

    cin.clear(); // 使流重新有效

    int replacedVal, newVal;
    // 读入需替换的指定值
    cout << "Enter a value that will be replaced: " << endl;
    cin >> replacedVal;

    // 读入用于替换的新值
    cout << "Enter a new value: " << endl;
    cin >> newVal;

    // 替换等于指定值的vector元素
    replace_if(ivec.begin(), ivec.end(), EQ_cls(replacedVal),
               newVal);

    // 输出替换后的vector对象以进行对比
    cout << "new sequence: " << endl;
    for (vector<int>::iterator iter = ivec.begin();
                               iter != ivec.end(); ++iter)
        cout << *iter << " ";

    return 0;
}
```

注意，为了提高 `EQ_cls` 类的通用性，可以将其定义为模板类（见习题 14.31 的解答）。

习题14.37

可分别定义如下函数对象：

(a) `bind2nd(greater<int>(), 1024)`

(b) `bind2nd(not_equal_to<string>(), "pooh")`

(c) `bind2nd(multiplies<int>(), 2)`

可如下编写程序来使用这些函数对象以完成所需功能：

```
// 14-37.cpp
// 使用标准库函数对象和函数适配器，定义三个对象分别用于：
// (a) 查找大于1024的所有值。
// (b) 查找不等于"pooh"的所有字符串。
// (c) 将所有值乘以2。
// 使用vector容器存放序列
#include <iostream>
#include <functional>
```

```
#include <algorithm>
#include <string>
#include <vector>
using namespace std;

int main()
{
     const int ARR_SIZE = 7;
     int ia[ARR_SIZE] = {1, 1025, 2, 1026, 1030, 3, 1048};
     vector<int> ivec(ia, ia+ARR_SIZE);
     string sa[ARR_SIZE] = {"many", "mach", "that", "pooh", "this", "pooh", "happy"};
     vector<string> svec(sa, sa+ARR_SIZE);

     // (a) 查找大于1024的所有值
     cout << "all values that are greater than 1024: " << endl;
     vector<int>::iterator iter = ivec.begin();
     // 使用bind2nd函数适配器将greater对象的右操作数绑定为1024
     while ((iter = find_if(iter, ivec.end(),
                            bind2nd(greater<int>(), 1024)))
          != ivec.end()) { // 找到了下一个大于1024的元素
          // 输出元素
          cout << *iter << ' ';
          // iter加1以便在剩余元素中进行查找
          ++iter;
     }

     // (b) 查找不等于"pooh"的所有字符串
     cout << endl << "all strings that are not equal to pooh: "
          << endl;
     vector<string>::iterator it = svec.begin();
     // 使用bind2nd函数适配器将not_equal_to对象的右操作数绑定为"pooh"
     while ((it = find_if(it, svec.end(),bind2nd(not_equal_to<string>(), "pooh")))
          != svec.end()) { // 找到了下一个不等于"pooh"的元素
          // 输出元素
          cout << *it << ' ';
          // it加1以便在剩余元素中进行查找
          ++it;
     }

     // (c) 将所有值乘以2
     // 使用bind2nd函数适配器将multiplies对象的右操作数绑定为2
     transform(ivec.begin(), ivec.end(), ivec.begin(),
                                         bind2nd(multiplies<int>(), 2));
     // 输出元素
     cout << endl << "all values multiplied by 2: " << endl;
     for (vector<int>::iterator it2 = ivec.begin();
                                it2 != ivec.end(); ++it2)
          cout << *it2 << ' ';

     return 0;
}
```

习题14.40

将Sales_item对象转换为string类型和double类型的操作符的代码如下：

```
Sales_item::operator string() const
{
     return isbn;
}

Sales_item::operator double() const
{
```

```
    return revenue;
}
```

定义这些操作符并不是个好办法，因为一般不必在需要string类型和double类型对象的地方使用Sales_item对象，这种用法没有太大的实际意义。

习题14.43

上述定义的转换操作符将等待列表为空的CheckoutRecord对象转换为bool真值（true），将等待列表不为空的CheckoutRecord对象转换为bool假值（false）。

这并不是这个CheckoutRecord类型转换唯一可能的含义，也可以将bool转换操作符定义为其他含义，如返回当前日期是否已超过date_due的判断结果（可用于判断某本借出的书是否已过期）。

上述定义的转换操作符可用于判断是否有读者在等待借阅这本书。在应用中，当某本书被归还时，可以用相应的CheckoutRecord对象作为判断条件以确定是否通知该书的等待者。因此这个转换可算是一种转换操作的良好使用。

习题14.46

候选函数包括：

(1) 内置的+操作符。

(2) `LongDouble operator+(LongDouble&, int);`（LongDouble 类的友元）。

(3) `LongDouble LongDouble::operator+(const Complex &);`。

(4) `LongDouble operator+(const LongDouble &, double);`（全局函数）。

调用(1)所需的类型转换为：将ld从LongDouble类型转换为double类型（可使用LongDouble类中定义的转换操作，为类类型转换）。

调用(2)所需的类型转换为：将15.05从double类型转换为int类型（可使用标准转换）。

调用(3)所需的类型转换为：将15.05从double类型转换为Complex类型（可使用Complex类的构造函数，为类类型转换）。

调用(4)无需进行类型转换。

因此，4 个候选函数均为可行函数。因为调用(4)无需进行类型转换，所以该函数为最佳可行函数。

第 15 章部分习题答案

习题15.1

所谓“虚成员”就是其声明中在返回类型前带有关键字“virtual”的类成员函数。C++中基类通过将成员函数指定为虚成员来指出希望派生类重定义的那些函数。除了构造函数外，任意非static成员函数都可以为虚成员。

习题15.4

因为不同种类的借阅资料有不同的登记、检查和过期规则，所以check_out、check_in、is_late和apply_fine等函数应定义为虚函数。print也应定义为虚函数，因为不同种类的借阅资料，需打印的项目可能不同。

due_date、date_borrowed、title和member等函数可能是公共的，因为应还日期、借出日期、标题和读者等应该是所有借阅资料的公共性质，相应的操作也应该是公共的。

习题15.7

可定义如下Lds_item类：

```
// 有限折扣类
class Lds_item : public Item_base {
public:
     // 构造函数
     Lds_item(const std::string& book = "",
              double sales_price = 0.0,
              size_t qty = 0, double disc_rate = 0.0):
                       Item_base(book, sales_price),
                       max_qty(qty), discount(disc_rate) { }

     // 重定义基类版本以实现有限折扣策略
     // 对低于上限的购书量使用折扣价格
     double Lds_item::net_price(size_t cnt) const
     {
          if (cnt <= max_qty)
               return cnt * (1 - discount) * price;
          else
               return cnt * price - max_qty * discount * price;
     }
private:
     size_t max_qty;        // 可打折的购书量上限
     double discount;       // 折扣率
};
```

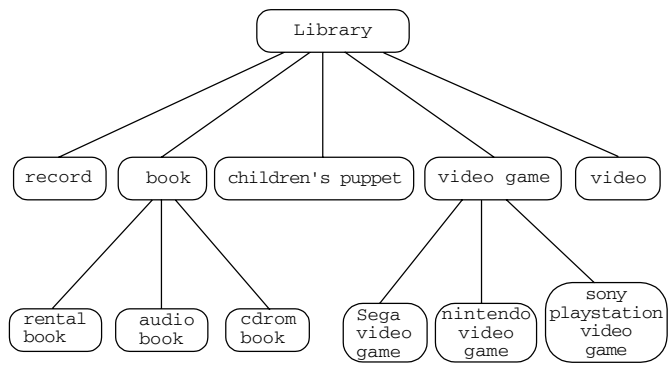
注意，构造函数的定义见15.4节。

习题15.10

根据这些项目之间的“是一种（Is A）”关系，可组织成如下继承层次：

- 类 book、record、video、children's puppet、video game 继承基类 Library。
- 类 audio book、cdrom book、rental book 继承类 book。
- 类 sega video game、nintendo video game、sony playstation video game 继承类 video game。

如下图所示。



注意，此处采用了与*C++ Primer*（第4版）一致的表示方式。如果采用UML类图表示，则有所不同。

习题15.13

C1中的成员函数访问ConcreteBase的static成员的方式如下：

(1) 使用ConcreteBase::成员名（即基类名::成员名）。

(2) 使用C1::成员名（即派生类名::成员名）。

(3) 通过C1类对象或对象引用，使用点（.）操作符。

(4) 通过C1类对象的指针，使用箭头（->）操作符。

(5) 直接使用成员名。

C2类型的对象不能访问obj_count成员，因为该成员是受保护成员，不能通过对象访问。

C2类型的对象访问object_count的方式（假设obj为C2类型的对象）如下：

```
obj.object_count();
obj.ConcreteBase::object_count();
obj.C2::object_count();
```

习题15.16

(a) 没有在初始化列表中向基类构造函数传递实参。

(b) 初始化列表中出现了非直接基类Base。

(c) 初始化列表中出现了非直接基类Base而没有出现直接基类C1。

(d) 初始化列表中使用了未定义的变量id。

(e) 缺少初始化列表：Base类没有默认构造函数，其派生类必须用初始化列表的方式向Base类的构造函数传递实参。

习题15.19

可能错在该类没有提供虚析构函数，因为该类有可能作为基类使用。

习题15.22

```
// 批量购买折扣类
class Bulk_item : public Disc_item {
public:
     Bulk_item(const std::string& book = "",
               double sales_price = 0.0,
               size_t qty = 0, double disc_rate = 0.0):
                    Disc_item(book, sales_price, qty, disc_rate) { }

     // 重定义基类版本以实现批量购买折扣策略：
     // 若购书量高于下限，则使用折扣价格
     double net_price(size_t cnt) const
     {
          if (cnt >= quantity)
               return cnt * (1 - discount) * price;
          else
               return cnt * price;
     }
};

// 有限折扣类
class Lds_item : public Disc_item {
public:
     // 构造函数
     Lds_item(const std::string& book = "",
                   double sales_price = 0.0,
               size_t qty = 0, double disc_rate = 0.0):
                        Disc_item(book, sales_price, qty, disc_rate) { }

     // 重定义基类版本以实现有限折扣策略：
     // 对低于上限的购书量使用折扣价格
     double net_price(size_t cnt) const
     {
          if (cnt <= quantity)
               return cnt * (1 - discount) * price;
          else
               return cnt * price - quantity * discount * price;
     }
};
```

习题15.25

错误的有：

(a) 单纯从语法上看并没有错误，但Derived中声明的copy是一个非虚函数，而不是对Base中声明的虚函数copy的重定义，因为派生类中重定义的虚函数必须具有与基类中虚函数相同的原型（唯一的例外是返回类型可以稍有不同，见15.8.2节clone函数的定义）。而且Derived中定义的copy函数还屏

蔽了基类Base的copy函数。

习题15.28

将习题 15.21 及习题 15.22 的解答中定义的 Item_base 类层次放在头文件 Item.hpp 中。可编写程序如下：

```
// 15-28.cpp
// 定义一个vector保存Item_base类型的对象，
// 将一些Bulk_item类型对象复制到vector中。
// 遍历vector并根据容器中元素计算net_price总和
#include "Item.hpp"  // 引入Item_base类层次的定义
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

int main()
{
    vector<Item_base> itemVec;
    string isbn;
    double price, qty, discount;

    // 读入Bulk_item对象并复制到vector中
    cout << "Enter some Bulk_item objects(Ctrl+Z to end): " << endl;
    while (cin >> isbn >> price >> qty >> discount) {
        itemVec.push_back(Bulk_item(isbn, price, qty, discount));
    }

    // 遍历vector并根据容器中元素计算购买100本书的net_price总和
    double sum = 0.0;
    for (vector<Item_base>::iterator iter = itemVec.begin();
                                      iter != itemVec.end(); ++iter)
        sum += iter->net_price(100);

    // 输出结果
    cout << "summation of net price: " << sum << endl;

    return 0;
}
```

习题15.31

```
// 有限折扣类
class Lds_item : public Item_base {
public:
    Lds_item* clone() const
    {
        return new Lds_item(*this);
    }

    // 其他成员同习题15.7中的定义，此处不再赘述
};
```

习题15.34

(a) 通过定义debug函数的形参。

在debug函数中增加一个形参ctrl，用于控制调试的打开或关闭。

将各类中的debug函数修改如下：

```
// 不使用折扣策略的基类
class Item_base {
    // ...其余成员（略）
public:
    // 参数ctrl置0将关闭调试
    virtual void debug(int ctrl = 1, ostream& os = cout) const
    {
        if (ctrl == 0)
            return;

        os << isbn << "\t" << price;
    }
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
    // ...其余成员（略）
public:
    // 参数ctrl置0将关闭调试
    virtual void debug(int ctrl = 1, ostream& os = cout) const
    {
        if (ctrl == 0)
            return;

        Item_base::debug(ctrl, os);
        os << "\t" << quantity << "\t"
           << discount;
    }
};
```

(b) 通过定义类数据成员。该成员允许个体对象打开或关闭调试信息的显示。

可在Item_base类中定义一个受保护数据成员is_debug，用于控制调试信息的显示与否，相应地需要修改类层次中各个类的构造函数以初始化该成员。Item_base类和Disc_item类中的debug函数根据is_debug控制是否显示调试信息，而且可以在Item_base类中提供一个公有的set_debug函数，以允许个体对象打开或关闭调试信息的显示。

修改后的类定义代码如下：

```
// 不使用折扣策略的基类
class Item_base {
    // ...其余成员（略）
public:
    Item_base(const std::string &book = "",
                double sales_price = 0.0, bool dbg = false):
                isbn(book), price(sales_price), is_debug(dbg){ }
    // 根据is_debug成员决定是否显示其他数据成员
    virtual void debug(ostream& os = cout) const
    {
        if (!is_debug)
            return;

        os << isbn << "\t" << price;
    }
```

```
    // 设置is_debug数据成员
    void set_debug(bool dbg)
    {
        is_debug = dbg;
    }
protected:
    bool is_debug;
};

// 保存折扣率和可实行折扣策略的数量
// 派生类将使用这些数据实现定价策略
class Disc_item : public Item_base {
    // ...其余成员（略）
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0, size_t qty = 0,
                  double disc_rate = 0.0, bool dbg = false):
                   Item_base(book, sales_price, dbg),
                    quantity(qty), discount(disc_rate) { }

    // 根据debug成员决定是否显示其他数据成员
    virtual void debug(ostream& os = cout) const
    {
        if (!is_debug)
            return;

        Item_base::debug(os);
        os << "\t" << quantity << "\t"
           << discount;
    }
};

// 批量购买折扣类
class Bulk_item : public Disc_item {
    // ...其余成员（略）
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0, size_t qty = 0,
              double disc_rate = 0.0, bool dbg = false):
              Disc_item(book, sales_price, qty, disc_rate, dbg) { }
};

// 有限折扣类
class Lds_item : public Disc_item {
    // ...其余成员（略）
public:
    // 构造函数
    Lds_item(const std::string& book = "",
                  double sales_price = 0.0, size_t qty = 0,
                  double disc_rate = 0.0, bool dbg = false):
                  Disc_item(book, sales_price, qty, disc_rate, dbg) { }
};
```

注意，各个类中省略的成员定义见习题 15.21 及习题 15.22 的解答。

习题15.37

因为该类型别名仅在Basket类中使用。

习题15.40

(a) 处理表达式`Query("fiery") & Query("bird") | Query("wind")`所执行的构造函数如下：

- `Query(const std::string&)`（执行 3 次）。
- `WordQuery(std::string&)`（执行 3 次）。
- `AndQuery(Query, Query)`（执行 1 次）。
- `BinaryQuery(Query, Query, std::string)`（执行 2 次）。
- `Query_base()`（执行 5 次）。
- `OrQuery(Query, Query)`（执行 1 次）。
- `Query(Query_base*)`（执行 2 次）。

注意，如果要创建`Query`对象`q`，还需要执行`Query(const Query&)`1次。

(b) 执行`cout << q`所调用的`display`函数和重载的`<<`操作符如下：

- `Query` 类的 `operator<<`。
- `Query` 类的 `display`。
- `BinaryQuery` 类的 `display`。
- `WordQuery` 类的 `display`。

(c) 计算`q.eval`时所调用的`eval`函数如下：

- `Query` 类的 `eval`。
- `OrQuery` 类的 `eval`。
- `AndQuery` 类的 `eval`。
- `WordQuery` 类的 `eval`。

第 16 章部分习题答案

习题16.1

```
// 16-1.cpp
// 编写一个模板函数返回形参的绝对值。
// 用三种不同类型的值调用该模板
#include <iostream>
using namespace std;

template <typename T>
T absVal(T val)
{
    return val > 0 ? val : -val;
}

int main()
{
    double dval = 0.88;
    float fval = -12.3;

    cout << absVal(-3) << endl;
    cout << absVal(dval) << endl;
    cout << absVal(fval) << endl;

    return 0;
}
```

习题16.4

函数模板（function template）是可用于不同类型的函数的定义。函数模板用template关键字后接用尖括号（<>）括住、以逗号分隔的一个或多个模板形参的列表来定义。

类模板（class template）是一个类定义，可以用来定义一组特定类型的类。类模板用template关键字后接用尖括号（<>）括住、以逗号分隔的一个或多个模板形参的列表来定义。

习题16.7

(a) 非法。模板类型形参前必须带有关键字typename（或class），模板非类型形参前必须带有类型名，而这里的U作为f1的形参类型使用，应该是一个类型形参，所以应在模板形参表中U的前面加上class或typename。

(b) 非法。如果单纯从模板函数的定义语法来看，该定义是合法的。但是，模板形参T没有作为类型在模板函数的形参表中出现，因此将无法对其进行模板实参推断，所以，该模板函数的定义是错误的。

(c) 非法。inline不能放在关键字template之前，应放在模板形参表之后、函数返回类型之前。

(d) 在标准C++中非法：没有指定函数f4的返回类型。（早期的C++版本可以接受：将返回类型隐式定义为int。）

(e) 合法。定义了一个模板函数f5，该函数的返回类型与形参类型相同，均可绑定到任意类型（而不一定是char类型）。

习题16.10

在标准C++中，声明为typename的类型形参与声明为class的类型形参没有区别。但是，标准C++之前的系统有可能只支持使用关键字class来声明模板类型形参。

习题16.13

```
// 16-13.cpp
// 编写一个函数，接受一个容器的引用并打印该容器的元素。
// 使用容器的size_type 和size成员控制打印元素的循环
#include <iostream>
#include <string>
#include <vector>
using namespace std;

template <typename Parm>
void print(const Parm& c)
{
     typename Parm::size_type index = 0;
     while (index != c.size()) {
          cout << c[index] << ' ';
          ++index;
     }
}

int main()
{
     int ia[] = {1, 2, 1, 4, 1, 6, 1};
     string sa[] ={"this", "is", "Mary", "test", "example"};
     vector<int> ivec(ia, ia+7);
     vector<string> svec(sa, sa+5);

     print(ivec);
     cout << endl;
     print(svec);

    return 0;
}
```

注意，对不支持下标操作的容器，不能使用上述print函数。

习题16.16

```
template <typename T, std::size_t N>
void printValues(T (&arr)[N])
{
     for (std::size_t i =0; i != N; ++i)
          std::cout<< arr[i] << std::endl;
}
```

习题16.19

所谓“实例化”，指的是产生模板的特定类型实例的过程。模板不能直接使用，必须在使用时进行实例化。类模板的实例化在引用实际模板类类型时进行，函数模板的实例化在调用它或用它对函数指针进行初始化或赋值时进行。

习题16.22

错误的有：

(a) 实参cobj的类型为char，但是，不能使用函数模板calc产生第一个形参为非指针类型的函数实例。

(b) 实参dobj的类型为double，但是，不能使用函数模板calc产生第一个形参为非指针类型的函数实例。

(c) 函数模板fcn中两个形参的类型必须是相同的，而函数调用fcn(ai, cobj)中给出的两个实参类型不同，不能进行实例化。

习题16.25

只需按如下方式使用显式模板实参：

```
compare<std::string>("mary", "mac")
```

亦可采用如下强制类型转换方式：

```
compare(static_cast<std::string>("mary"),
        static_cast<std::string>("mac"))
```

或

```
compare(std::string("mary"), std::string("mac"))
```

习题16.28

如果所用的编译器支持分别编译模型，则类模板的内联成员函数应该与类模板的定义一起放在头文件中，而非内联成员函数和static数据成员的定义应该放在实现文件中。因为内联函数的定义必须在函数被扩展时能为编译器所见，而非内联成员函数一般而言并不希望让用户看见，所以将其放在实现文件中（同时，在类的实现文件中应该导出（export）类模板，以便让编译器了解要记住该模板定义，并自动跟踪相关的模板定义）。

另外，如果不想导出整个类模板，而只是导出个别成员，则非导出成员的定义应放在头文件中，并且实现文件中不在类模板本身指定export，而是在被导出的特定成员定义上指定export。

习题16.31

错误在于：在类模板List的定义体中使用类模板ListItem时未指定模板实参。将ListItem改为ListItem<elemType>即可。

习题16.34

参照本节给出的 Queue 类定义，对习题 16.6 的解答中给出的 List 类进行补充，可得到如下 List 类定义：

```
template <class Type> class List;
template <class Type> class ListItem {
     friend class List<Type>;
     // 私有类：没有public部分
     ListItem(const Type &t): item(t), next(0) { }
     Type item;             // 元素中存储的数据
     ListItem *next;        // 指向下一元素的指针
};

template <class Type> class List {
public:
     // 默认构造函数
     List() : front(0), end(0) { }

     // 复制控制成员
     List(const List& l) : front(0), end(0)
     {
          copy_elems(l)
     }

     List& operator=(const List&);

     ~List()
     {
          destroy();
     }

     // 其他操作
     void insert(ListItem<Type> *ptr, const Type& value);
     void del(ListItem<Type> *ptr);
     ListItem<Type> *find(const Type& value);
     ListItem<Type> *first()
     {
          return front;
     }

     ListItem<Type> *last()
     {
          return end;
     }

     bool empty() const // 判断List是否为空
     {
          return front == 0;
     }

     Type& getElem(ListItem<Type> *ptr)
     {
          // 不检查ptr
          return ptr -> item;
     }
private:
     ListItem<Type> *front, *end; // 指向List中头尾元素的指针
```

```
    void destroy();
    void copy_elems(const List&);
};

// 删除List中所有元素
template <class Type>
void List<Type>::destroy()
{
    while (!empty())
        del(front);
}

// 删除ptr所指向的元素
template <class Type>
void List<Type>::del(ListItem<Type> *ptr)
{
    // 不检查ptr
    ListItem<Type>* p = front;

    // 获取ptr所指元素的前一元素的指针p
    while (p != ptr && p != 0 && p -> next != ptr)
        p = p -> next;

    if (p != 0)  {// 找到这样的指针p：说明ptr指向List中的元素
        if (p == ptr) { // 要删除的是第一个元素
            front = ptr -> next;
        }
        else {
            p -> next = ptr -> next;
        }
        if (ptr == end)
            end = p -> next;
        delete ptr;
    }
    else
        throw out_of_range("no such element");
}

// 在ptr所指元素的后面插入元素
template <class Type>
void List<Type>::insert(ListItem<Type> *ptr, const Type& val)
{
    // 不检查ptr
    // 创建ListItem对象
    ListItem<Type> *pt = new ListItem<Type>(val);

    // 将ListItem插入List
    if (empty()) // 原List为空
        front = pt; // List现在只有一个元素
    else { // 将新元素插入到ptr所指元素的后面
        pt -> next = ptr -> next;
        ptr -> next = pt;
    }

    if (ptr == end)
        end = pt; // 修改List的end指针
}

// 将orig中的元素复制到这个List中
template <class Type>
void List<Type>::copy_elems(const List &orig)
```

```
{
     // 当pt == 0（即到达orig.end时）循环结束
     for (ListItem<Type> *pt = orig.front; pt; pt = pt->next) {
          insert(end, pt -> item);
     }
}

// 赋值操作符：复制所有元素
template <class Type>
List<Type>& List<Type>::operator = (const List &orig)
{
     front = end = 0;
     copy_elems(orig);
     return *this;
}

// 查找值为value的元素，返回其指针
template <class Type>
ListItem<Type>* List<Type>::find(const Type& value)
{
     ListItem<Type>* pt = front;
     while (pt && pt -> item != value)
          pt = pt -> next;
     return pt;
}
```

注意，这里给出的是一个简单的例子List类，其中的操作为简单起见未对形参指针进行检查，更为完善的List类可参照标准库中给出的list容器类进行定义。

习题16.37

有效的模板实例化包括(a)和(b)。

(c)之所以无效，是因为非类型模板实参必须是编译时常量表达式，不能用变量asize作模板实参。

(d)之所以无效，是因为db是double型常量，而该模板实例化所需要的非类型模板实参为int型常量。

习题16.40

因为输入和输出操作符需要访问Screen类的私有数据成员，所以要使输入和输出操作符能够工作，Screen类需要将这两个操作符设为友元，可在Screen类的定义体内增加如下友元声明：

```
friend std::ostream&
operator << <hi, wid> (std::ostream&, const Screen<hi, wid>&);

friend std::istream&
operator >> <hi, wid> (std::istream&, Screen<hi, wid>&);
```

习题16.43

可对习题 16.34 解答中给出的 List 类进行如下修改：

(1) 在类定义体中public部分增加如下成员：

```
template <class Iter>
List(Iter first, Iter last) : front(first), end(last)
{
```

```
    copy_elems(beg, last)
}

template <class Iter>
void assign(Iter, Iter);
```

(2) 在List类定义体中private部分增加如下成员：

```
template <class Iter> void copy_elems(Iter, Iter);
```

assign 和 copy_elems 函数可在类定义体外实现如下：

```
template <class T> template <class Iter>
void List<T>::assign(Iter first, Iter last)
{
   destroy();
   copy_elems(first, last);
}

template <class T> template <class Iter>
void List<T>::copy_elems(Iter first, Iter last)
{
    while(first != last) {
        insert(end, first -> item);
        first = first -> next;
    }
    // 插入最后一个元素
    insert(end,first -> item);
}
```

习题16.46

如果是使用复制构造函数进行复制（包括：根据已存在的Handle对象创建新的Handle对象，用Handle对象作函数参数，函数返回Handle对象等），则复制Handle对象时将复制两个指针：指向基础对象的指针和指向使用计数的指针，并将使用计数加1；如果是使用赋值操作符进行复制，则首先将右操作数的使用计数加1，然后将左操作数的使用计数减1（如果使用计数减至0，则删除相应基础对象），再将右操作数的两个指针复制给左操作数。

复制之后，两个Handle对象将引用同一基础对象。

习题16.49

```
// Sales_item.hpp(for 16-49)
// 定义Sales_item句柄类
#ifndef SALESITEM_H
#define SALESITEM_H

#include "Handle.hpp"   // Handle.hpp定义泛型句柄类，见习题16.45解答
#include "Item.hpp"     // Item.hpp定义Item_base类层次，见习题15.35解答

// 用于Item_base层次的使用计数式句柄类
class Sales_item {
public:
    // 默认构造函数：创建未绑定的句柄
    Sales_item(): h() { }

    // 将创建绑定到Item_base对象副本的句柄
```

```
    Sales_item(const Item_base &item) : h(item.clone()) { }

    // 成员访问操作符
    const Item_base* operator->() const
    {
        return h.operator -> ();
    }

    const Item_base& operator*() const
    {
        return *h;
    }

private:
    Handle<Item_base> h; // 使用计数式句柄
};

#endif
```

习题16.52

下面定义的函数模板count计算sought中的元素在vec中的出现次数：

```
// count.hpp(for 16-52)
// 定义函数模板count计算一个vector中某些值的出现次数
#ifndef COUNT_H
#define COUNT_H

#include <vector>
#include <algorithm>
using std::vector;
using std::size_t;

template <typename Type>
size_t count(const vector<Type> &vec, const vector<Type> &sought)
{
    size_t result = 0;
    for (typename vector<Type>::const_iterator
            iter = sought.begin(); iter != sought.end(); ++iter)
        result += std::count(vec.begin(), vec.end(), *iter);

    return result;
}

#endif
```

注意，利用标准库中提供的泛型算法count计算指定值在vector中的出现次数。sought也是一个vector，用于存放要查找的那些值。

习题16.55

因为Queue针对const char*的特化版本中只有一个数据成员，该数据成员的类型为类类型。这个类类型在被复制、被赋值以及被撤销时可以完成正确的工作。

习题16.58

针对char*的Queue类特化与针对const char*的Queue类特化类似，只需将所有const char*改为

char*即可，代码如下：

```
// 针对char*的特化
template<> class Queue<char*> {
public:
    void push(char*);
    void pop()
    {
        real_queue.pop();
    }

    bool empty() const
    {
        return real_queue.empty();
    }

    // 注意，返回类型与模板形参类型不匹配
    std::string front()
    {
        return real_queue.front();
    }

    const std::string &front() const
    {
        return real_queue.front();
    }

private:
    Queue<std::string> real_queue;
};

void Queue<char*>::push(char* val)
{
    return real_queue.push(val);
}
```

针对char*的push和pop函数的特化与针对const char*的push和pop函数的特化类似，只需将所有const char*改为char*即可，代码如下：

```
template <>
void Queue<char*>::push(char *const &val)
{
    // 分配新的字符数组并从val复制字符
    char* new_item = new char[strlen(val) + 1];
    strncpy(new_item, val, strlen(val) + 1);

    // 新分配一个元素并进行初始化
    QueueItem<char*> *pt =
            new QueueItem<char*>(new_item);

    // 在队列中增加元素
    if (empty())
        head = tail = pt; // 队列中只有一个元素
    else {
        tail->next = pt;  // 将新元素加到队尾
        tail = pt;
    }
}

template <>
```

```
void Queue<char*>::pop()
{
     // 保存头指针以进行元素删除
     QueueItem<char*> *p = head;
     delete head->item;     // 删除push操作中分配的字符数组
     head = head->next;     // 修改头指针
     delete p;              // 删除原来的队头元素
}
```

习题16.61

```
// 比较两个对象
template <typename T> int compare(const T& v1, const T& v2)
{
     cout << "compares two objects" << endl;
     if (v1 < v2)
          return -1;
     if (v2 < v1)
          return 1;
     return 0;
}

// 比较两个序列中的元素
template <class U, class V> int compare(U v1, U v2, V beg)
{
     cout << "compares elements in two sequences" << endl;
     return 0;
}

// 处理C风格字符串的普通函数
int compare(const char* p1, const char* p2)
{
     cout << "plain function to handle C-style character strings"
          << endl;
     return strcmp(p1, p2);
}
```

第 17 章部分习题答案

习题17.1

(a)异常对象 r 的类型是 range_error。

(b)被抛出的异常对象是对指针 p 解引用的结果，其类型与 p 的静态类型相匹配，为 exception。

习题17.4

```
int main()
{
     try {
          // use of the C++ standard library
     }
     catch(const exception &e) {
          cerr << e.what() << endl;
          abort();
     }
     return 0;
}
```

习题17.7

一种方法是将有可能发生异常的代码放在try块中，以便在异常发生时捕获异常：

```
void exercise(int *b, int *e)
{
     vector<int> v(b, e);
     int *p = new int[v.size()];
     try {
          ifstream in("ints");
          // 此处发生异常
     }
     catch {
          delete p;  // 释放数组
          //...进行其他处理
     }
     // ...
}
```

另一种方法是定义一个类来封装数组的分配和释放，以保证正确释放资源：

```
class Resource {
public:
     Resource(size_t sz) : r(new int[sz]) { }
     ~Resource() { if (r) delete r; }
     //...其他操作
```

```
private:
    int *r;
};
```

函数exercise相应修改为：

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    Resource res(v.size());  // 使用Resource对象
    ifstream in("ints");
    // exception occurs here
    // ...
}
```

注意，此处给出的Resource类非常简略，要达到实用，还需定义其他操作，包括复制构造函数、赋值操作、解引用操作、箭头操作、下标操作等，以支持内置指针及数组的使用方式并保证自动删除Resource对象所引用的数组。另外，可将该Resource类定义为类模板，以支持多种数组元素类型。

习题17.10

如果函数有形如throw()的异常说明，则该函数不抛出任何异常。

如果函数没有异常说明，则该函数可以抛出任意类型的异常。

习题17.13

```
namespace Bookstore {
    class out_of_stock: public std::runtime_error {
    public:
        explicit out_of_stock(const std::string &s):
                    std::runtime_error(s) { }
    };
    class isbn_mismatch: public std::logic_error {
    public:
        explicit isbn_mismatch(const std::string &s):
                        std::logic_error(s) { }
        isbn_mismatch(const std::string &s,
                const std::string &lhs, const std::string &rhs):
                std::logic_error(s), left(lhs), right(rhs) { }
        const std::string left, right;
        virtual ~isbn_mismatch() throw() { }
    };
}
```

习题17.16

将Query类以及Query_base类层次定义为命名空间chapterrefinheritance的成员，将TextQuery类定义为命名空间chapterrefalgs的成员，并相应修改主函数中的代码（使用限定名引用这些类，或者使用相关的using声明）。

代码略。

习题17.19

假定有下面的operator*的声明，operator*是嵌套命名空间cplusplus_primer:: MatrixLib的成员：

```
namespace cplusplus_primer {
    namespace MatrixLib {
            class matrix { /* ... */ };
            matrix operator*
                (const matrix &, const matrix &);
        // ...
    }
}
```

怎样在全局作用域中定义这个操作符？只需给出操作符定义的原型。

【解答】

将函数返回类型及函数名加上命名空间名字限定即可：

```
cplusplus_primer::MatrixLib::matrix cplusplus_primer::MatrixLib::operator*
(const matrix &, const matrix &)
{ /*...*/ }
```

习题17.22

全局作用域中声明的函数void compute(int)与compute函数的调用匹配。

候选函数：命名空间primerLib中声明的两个compute函数（因using声明使得它们在全局作用域中可见），以及全局作用域中声明的三个compute函数。

可行函数：因函数调用中给出的实参0为int类型，所以可行函数为以下4个函数。

- void compute(int)。
- void compute(double, double = 3.4)。
- void compute(char*, char* = 0)。
- primerLib 中声明的 void compute(const void*)。

其中，第一个为完全匹配，第二个需要将实参隐式转换为double类型，第三个需要将实参隐式转换为char*类型，第四个需要将实参隐式转换为void*类型方可匹配，所以第一个为最佳匹配。

如果将using声明放在main中的compute调用之前，则primerLib中声明的void compute(const void*)与compute函数的调用匹配。

候选函数：命名空间primerLib中声明的两个compute函数（因using声明使得它们在main函数的函数体作用域中可见）。

可行函数：因函数调用中给出的实参0为int类型，所以可行函数为primerLib中声明的void compute(const void*)。需要将实参隐式转换为void*类型方可匹配。

习题17.25

(c)和(b)是不允许的。

因为C对B的继承是私有继承，使得在D中B的默认构造函数成为不可访问的（见15.2.5节），所以尽管存在从“D*”到“B*”以及从“D*”到“A*”的转换，但这些转换不可访问。

习题17.28

该类层次的概略定义如下：

```
class ZooAnimal {
```

```
    //...成员略
};
class Bear : public ZooAnimal {
    //...成员略
}
class Endangered {
    //...成员略
}
class Panda : public Bear, public Endangered {
    //...成员略
}
```

习题17.31

错误的有：

- `dval = 3.14159;` `MI` 的基类 `Derived` 和 `Base1` 中都定义了成员 `dval`，此处无法确定使用哪个 `dval`。
- `id = 1;` 使用的是 `Base1` 中定义的成员 `id`，但该成员为 `private` 成员，不能在 `MI` 中使用，并且，对指针 `id` 赋以 `int` 型值 `2` 也是错误的。

习题17.34

派生类不必为虚基类提供初始化式的情况是：虚基类具有默认构造函数（显式提供或由编译器合成）。

第 18 章部分习题答案

习题18.1

Vector类的头文件如下：

```
// Vector.hpp
// Vector类的头文件
// 实现自己的Vector类的版本，包括vector成员reserve、resize、size、
// capacity以及const和非const下标操作符
#ifndef MYVECTOR_H
#define MYVECTOR_H
#include <memory>
#include <cstddef>
using namespace std;

template <class T> class Vector {
public:
     Vector(): elements(0), first_free(0), end(0) { }
     void push_back(const T&);
     void reserve(const size_t capa);

     // 调整Vector大小，使其能容纳n个元素：
     // 如果n小于Vector当前大小，则删除多余元素；
     // 否则，添加采用值初始化的新元素
     void resize(const size_t n);

     // 调整Vector大小，使其能容纳n个元素：所有新添加的元素值都为t
     void resize(const size_t n, const T& t);

     // 下标操作符
     T& operator[] (const size_t);
     const T& operator[] (const size_t) const;

     // 返回Vector大小
     size_t size()
     {    return first_free - elements; }

     // 返回Vector容量
     size_t capacity()
     {    return end - elements;  }
private:
     static std::allocator<T> alloc;     // 用于获取未构造内存的对象
     void reallocate();                  // 获取更多空间并复制现有元素
     T* elements;                        // 指向第一个元素的指针
     T* first_free;                      // 指向第一个自由元素的指针
     T* end;                             // 指向数组末端的下一元素位置的指针
};
```

```
#include "Vector.cpp"      // 引入Vector类的实现文件
#endif
```

Vector 类的实现文件如下：

```
// Vector.cpp
// Vector类的实现文件（源文件）
template <class T> allocator<T> Vector<T>::alloc;

template <class T> void Vector<T>::push_back(const T& t)
{
     if (first_free == end)
          // 已用完所分配的空间
          reallocate(); // 获取更多空间并复制现有元素至新分配空间
     alloc.construct(first_free, t);
     ++first_free;
}

template <class T> void Vector<T>::reallocate()
{
     // 计算当前大小并分配两倍于当前元素数的空间
     ptrdiff_t size = first_free - elements;
     ptrdiff_t newcapacity = 2 * max(size, 1);

     // 分配空间以保存newcapacity个T类型的元素
     T* newelements = alloc.allocate(newcapacity);

     // 在新空间中构造现有元素的副本
     uninitialized_copy(elements, first_free, newelements);

     // 逆序撤销旧元素
     for (T *p = first_free; p != elements; /* empty */ )
               alloc.destroy(--p);

     // 不能用0值指针调用deallocate
     if (elements)
          // 释放保存元素的内存
          alloc.deallocate(elements, end - elements);

     // 使数据结构指向新元素
     elements = newelements;
     first_free = elements + size;
     end = elements + newcapacity;
}

template <class T>
void Vector<T>::reserve(const size_t capa)
{
     // 计算当前数组的大小
     size_t size = first_free - elements;

     // 分配可保存capa个T类型元素的空间
     T* newelements = alloc.allocate(capa);

     // 在新分配的空间中构造现有元素的副本
     if (size <= capa)
          uninitialized_copy(elements, first_free, newelements);
     else
          // 如果capa小于原来数组的size，则去掉多余的元素
          uninitialized_copy(elements, elements + capa, newelements);

     // 逆序撤销旧元素
```

```
    for (T *p = first_free; p != elements; /* 表达式为空 */ )
        alloc.destroy(--p);

    if (elements)
        // 释放旧元素占用的内存
        alloc.deallocate(elements, end - elements);

    // 使数据结构指向新元素
    elements = newelements;
    first_free = elements + min(size,capa);
    end = elements + capa;
}

template <class T>
void Vector<T>::resize(const size_t n)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    } else if (n > size)
        // 添加采用值初始化的新元素
        uninitialized_fill(elements + size, elements + n, T());
    else
        // 逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */ )
            alloc.destroy(--p);

    // 使数据结构指向新元素
    first_free = elements + n;
}

template <class T>
void Vector<T>::resize(const size_t n, const T& t)
{
    // 计算当前大小及容量
    size_t size = first_free - elements;
    size_t capacity = end - elements;

    if (n > capacity) {
        reallocate(); // 获取更多空间并复制现有元素
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    } else if (n > size)
        // 添加值为t的新元素
        uninitialized_fill(elements + size, elements + n, t);
    else
        // 逆序撤销多余元素
        for (T *p = first_free; p != elements + n; /* 表达式为空 */ )
            alloc.destroy(--p);

    // 使数据结构指向新元素
    first_free = elements + n;
}

template <class T>
T& Vector<T>::operator[] (const size_t index)
{
    return elements[index];
```

```
}

template <class T>
const T& Vector<T>::operator[] (const size_t index) const
{
    return elements[index];
}
```

注意，Microsoft Visual C++ .NET 2003支持模板的包含编译模型，但不要将模板类的实现文件显式地加入到project中，否则会引起编译错误。

习题18.4

因为allocator类提供的是可感知类型的内存分配，限制construct函数只能使用元素类型的复制构造函数，可以获得更高的类型安全性。

习题18.7

因为 Vector 的新版本中类的接口保持不变，所以，使用 Vector 的新版本，习题 18.3 解答中给出的示例程序可不加修改地运行。

习题18.10

(a) new表达式创建一个iStack对象，其top成员初始化为0，stack成员初始化为包含20个0值int型元素的vector，并返回指向该iStack对象的指针。用new表达式的返回值对指针ps进行初始化（使ps指向该iStack对象）。

(b) new表达式创建一个const iStack对象，其top成员初始化为0，stack成员初始化为包含15个0值int型元素的vector，返回指向该iStack对象的指针，并试图用new表达式的返回值对指针ps2进行初始化。其中有错误：new表达式返回一个const iStack型指针，不能用该指针对iStack型指针ps2进行初始化。

(c) 试图使用new表达式分配一个包含100个iStack对象的数组，并用该数组的首地址对指针ps3进行初始化。其中有错误：使用new表达式动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数实现初始化（见4.3.1节），因此分配iStack对象数组的new表达式需要使用iStack类的默认构造函数对数组元素进行初始化，但iStack类没有提供默认构造函数，因而出错。

习题18.13

使用dynamic_cast操作符时，如果运行时实际绑定到引用或指针的对象不是目标类型的对象（或其派生类的对象），则dynamic_cast失败。

(b) dynamic_cast失败。因为目标类型为C，但pb实际指向的不是C类对象，而是一个B类（C的基类）对象。

注意，要使用RTTI，一般需要在编译器中设置相应编译选项。例如，在Microsoft Visual C++ .NET 2003中，在project菜单中选择properties菜单项，在configuration properties -> C/C++ -> Language中打开RTTI选项。

习题18.16

如果我们需要在派生类中增加新的成员函数（假设为函数f），但又无法取得基类的源代码，因而无法在基类中增加相应的虚函数，这时，可以在派生类中增加非虚成员函数。但这样一来，就无法用基类指针来调用f。如果在程序中需要通过基类指针（如使用该继承层次的某个类中所包含的指向基类对象的指针数据成员p）来调用f，则必须使用dynamic_cast将p转换为指向派生类的指针，才能调用f。也就是说，如果无法为基类增加虚函数，就可以使用dynamic_cast代替虚函数。

习题18.19

假设指针qb1和qb2的类型为Query_base*，则判断两个Query_base指针是否指向相同的类型的typeid表达式如下：

```
typeid(*qb1) == typeid(*qb2)
```

判断该类型是否为AndQuery的typeid表达式如下：

```
typeid(*qb1) == typeid(AndQuery)
```

习题18.22

Sales_item类的isbn成员是一个数据成员，其类型为std::string。可以表示Sales_item类的isbn成员的指针的类型为：

```
std::string Sales_item::*
```

习题18.25

Screen类的成员cursor的类型是Screen::index，即std::string::size_type。

习题18.28

```
class Screen {
public:
     // 成员函数指针的类型别名
     typedef Screen& (Screen::*PmfType) ();

     typedef std::string::size_type index;

     char get() const;
     char get(index ht, index wd) const;
     // 光标移动函数
     Screen& home();
     Screen& forward();
     Screen& back();
     Screen& up();
     Screen& down();
private:
     std::string contents;
     index cursor;
     index height, width;
public:
```

```
    // 成员函数指针
    PmfType pmf;
};
```

习题18.31

提供成员函数setPmf来设置成员函数指针成员：

```
void setPmf(PmfType p)
{
    pmf = p;
}
```

Screen类的完整定义体如下：

```
class Screen {
public:
    // 成员函数指针的类型别名
    typedef Screen& (Screen::*PmfType) ();

    typedef std::string::size_type index;

    // 接受函数指针形参的构造函数
    Screen(PmfType p = &Screen::home) : pmf(p) { }

    // 设置成员函数指针的函数
    void setPmf(PmfType p)
    {
        pmf = p;
    }

    char get() const;
    char get(index ht, index wd) const;
    // 光标移动函数
    Screen& home();
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
private:
    std::string contents;
    index cursor;
    index height, width;
public:
    // 成员函数指针
    PmfType pmf;
};
// ...其他成员函数的实现（略）
```

注意，如果将成员函数指针定义为public数据成员，则可按如下方式使用该指针：

```
Screen myScreen;
myScreen.setPmf(&Screen::down);
(myScreen.*(myScreen.pmf))();  // 等价于myScreen.down();
```

如果将成员函数指针定义为private数据成员，则应该提供一个public成员函数（如getPmf）来获取该指针，因此可按如下方式使用该指针：

```
Screen myScreen;
myScreen.setPmf(&Screen::down);
(myScreen.*(myScreen.getPmf()))();  // 等价于myScreen.down();
```

getPmf函数可定义如下：

```
PmfType getPmf()
{
    return pmf;
}
```

习题18.34

第一个声明指出：compute是一个用C语言编写的函数，该函数接受一个int*类型及一个int类型的形参，返回int型值。

第二个声明指出：compute是一个用C语言编写的函数，该函数接受一个double*类型及一个double类型的形参，返回double型值。

如果这两个声明单独出现，则是合法的；如果二者同时出现，则是不合法的，因为这两个 compute 函数构成了函数重载，而 C 语言是不支持函数重载的。