

# 目录

第一章 快速入门 .....	2
第二章 变量和基本类型 .....	7
第三章 标准库类型 .....	13
第四章 数组和指针 .....	21
第五章 表达式 .....	31
第六章 语句 .....	37
第七章 函数 .....	37
第八章 标准 IO 库 .....	37
第九章 顺序容器 .....	43
第十章 关联容器 .....	60
第十一章 泛型算法 .....	75
第十二章 类和数据抽象 .....	86
第十三章 复制控制 .....	94
第十四章 重载操作符与转换 .....	102
第十五章 面向对象编程 .....	116
第十六章 部分选做习题 .....	133
第十七章 用于大型程序的工具 .....	138
第十八章 特殊工具与技术 .....	138

# 第一章 快速入门

## 习题 1.1

查看所用的编译器文档，了解它所用的文件命名规范。编译并运行本节的 main 程序。

### 【解答】

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 1.2

修改程序使其返回 -1。返回值 -1 通常作为程序运行失败的指示器。然而，系统不同，如何（甚至是否）报告 main 函数运行失败也不同。重新编译并再次运行程序，看看你的系统如何处理 main 函数的运行失败指示器。

### 【解答】

笔者所使用的 Windows 操作系统并不报告 main 函数的运行失败，因此，程序返回 -1 或返回 0 在运行效果上没有什么区别。但是，如果在 DOS 命令提示符方式下运行程序，然后再键入 echo %ERRORLEVEL% 命令，则系统会显示返回值 -1。

## 习题 1.3

编一个程序，在标准输出上打印 "Hello, World"。

### 【解答】

```
#include <iostream>
int main()
{
    std::cout << "Hello, World" << std::endl;
    return 0;
}
```

## 习题 1.4

我们的程序利用内置的加法操作符 "+" 来产生两个数的和。编写程序，使用乘法操作符 "\*" 产生两个数的积。

### 【解答】

```
#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 1.5

我们的程序使用了一条较长的输出语句。重写程序，使用单独的语句打印每一个操作数。

### 【解答】

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The sum of ";
    std::cout << v1;
    std::cout << " and ";
    std::cout << v2;
    std::cout << " is ";
    std::cout << v1 + v2 ;
    std::cout << std::endl;
    return 0;
}
```

## 习题 1.6

解释下面的程序段：

```
std::cout << "The sum of " << v1;
<< " and " << v2;
<< " is " << v1 + v2
<< std::endl;
```

这段代码合法吗？如果合法，为什么？如果不合法，又为什么？

### 【解答】

这段代码不合法。

注意，第 1、2、4 行的末尾有分号，表示这段代码包含三条语句，即第 1、2 行各为一个语句，第 3、4 行构成一个

语句。“<<”为二元操作符，在第 2、3 两条语句中，第一个“<<”缺少左操作数，因此不合法。在第 2、3 行的开头加上“std::cout”，即可更正。

### 习题 1.7

编译有不正确嵌套注释的程序。

#### 【解答】

由注释对嵌套导致的梦芭莎优惠券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 1.8

指出下列输出语句哪些（如果有）是合法的。

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
```

预测结果，然后编译包含上述三条语句的程序，检查你的答案。纠正所遇到的错误。

#### 【解答】

第一条和第二条语句合法。

第三条语句中<<操作符之后至第二个双引号之前的部分被注释掉了，导致<<操作符的右操作数不是一个完整的字符串，所以不合法。在分号之前加上一个双引号即可更正。

### 习题 1.9

下列循环做什么？sum 的最终值是多少？

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

#### 【解答】

该循环求-100~100 之间所有整数的和（包括-100 和 100）。sum 的最终值是 0。

### 习题 1.10

用 for 循环编程，求从 50~100 的所有自然数的和。然后用 while 循环重写该程序。

#### 【解答】

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 1.11

用 while 循环编程，输出 10~0 递减的自然数。然后用 for 循环重写该程序。

#### 【解答】

用 while 循环编写的程序如下：

```
#include <iostream>
int main()
{
    int i = 10;
    while (i >= 0) {
        std::cout << i << " ";
        --i;
    }
    return 0;
}
```

用 for 循环编写的程序如下：

```
#include <iostream>
int main(www.bbooby.com)
{
    for (int i = 10; i >= 0; --i)
        std::cout << i << " ";
    return 0;
}
```

#### 习题 1.12

对比前面两个习题中所写的循环。两种形式各有何优缺点？

##### 【解答】

在 for 循环中，循环控制变量的初始化和修改都放在语句头部分，形式较简洁，且特别适用于循环次数已知情况。在 while 循环中，循环控制变量的初始化一般放在 while 语句之前，循环控制变量的修改一般放在循环体中，形式上不如 for 语句简洁，但它比较适用于循环次数不易预知情况（用某一条件控制循环）。两种形式各有优点，但它们在功能上是等价的，可以相互转换。

#### 习题 1.13

编译器不同，理解其诊断内容的难易程度也不同。编写一些程序，包含本小节“再谈编译”部分讨论的那些常见错误。研究编译器产生的信息，这样你在编译更复杂的程序遇到这些信息时不会陌生。

##### 【解答】

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 1.14

如果输入值相等，本节展示的程序将产生什么问题？

##### 【解答】

sum 的值即为输入值。因为输入的 v1 和 v2 值相等（假设为 x），所以 lower 和 upper 相等，均为 x。for 循环中的循环变量 val 初始化为 lower，从而 val<=upper 为真，循环体执行一次，sum 的值为 val（即输入值 x）；然后 val 加 1，val 的值就大于 upper，循环执行结束。

#### 习题 1.15

用两个相等的值作为输入编译并运行本节中的程序。将实际输出与你在习题 1.14 中所做的预测相比较，解释实际结果和你预计的结果间的不相符之处。

##### 【解答】

运行 1.4.3 节中给出的程序，输入两个相等的值（例如 3,3），则程序输出为：

Sum of 3 to 3 inclusive is 3

与习题 1.14 中给出的预测一致。

#### 习题 1.16

编写程序，输出用户输入的两个数中的较大者。

##### 【解答】

```
#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 1.17

编写程序，要求用户输入一组数。输出信息说明其中有多少个负数。

##### 【解答】

```
#include <iostream>
int main()
{
    int amount = 0, value;
    // 读入数据直到遇见文件结束符，计算所读入的负数的个数
```

```

while (std::cin >> value)
if (value <= 0)
++amount;
std::cout << "Amount of all negative values read is"
<< amount << std::endl;
return 0;
}

```

#### 习题 1. 18

编写程序，提示用户输入两个数并将这两个数范围内的每个数写到标准输出。

##### 【解答】

```

#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; val <= upper; ++val)
std::cout << val << " ";
return 0;
}

```

#### 习题 1. 19

如果上题给定数 1000 和 2000，程序将产生什么结果？修改程序，使每一行输出不超过 10 个数。

##### 【解答】

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. 20

编写程序，求用户指定范围内的数的和，省略设置上界和下界的 if 测试。假定输入数是 7 和 3，按照这个顺序，预测程序运行结果。然后按照给定的数是 7 和 3 运行程序，看结果是否与你预测的相符。如果不相符，反复研究关于 for 和 while 循环的讨论直到弄清楚其中的原因。

##### 【解答】

可编写程序如下：

```

// 1-20.cpp
// 省略设置上界和下界的 if 测试，求用户指定范围内的数的和
#include <iostream : www.bbooby.com >
int main()
{
std::cout << "Enter two numbers:" << std::endl;
int v1, v2;
std::cin >> v1 >> v2; // 读入数据
int sum = 0;

```

```
// 求和
for (int val = v1; val <= v2; ++val)
sum += val; // sum = sum + val
std::cout << "Sum of " << v1
<< " to " << v2
<< " inclusive is "
<< sum << std::endl;
return 0;
}
```

如果输入数据为 7 和 3, 则 v1 值为 7, v2 值为 3。for 语句头中将 val 的初始值设为 7 第一次测试表达式 val <= v2 时, 该表达式的值为 false, for 语句的循环体一次也不执行, 所以求和结果 sum 为 0。

#### 习题 1. 21

本书配套网站 ([http://www.awprofessional.com/cpp\\_primer](http://www.awprofessional.com/cpp_primer)) 的第 1 章的代码目录下有 Sales\_item.h 源文件。复制该文件到你的工作目录。编写程序, 循环遍历一组书的销售交易, 读入每笔交易并将交易写至标准输出。

【解答】

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // 读入 ISBN, 售出书的本数, 销售价格
    std::cout << "Enter transactions:" << std::endl;
    while (std::cin >> book)
    {
        // 输出 ISBN, 售出书的本数, 总收入, 平均价格
        std::cout << "ISBN, number of copies sold, "
        << "total revenue, and average price are:"
        << std::endl;
        std::cout << book << std::endl;
    }
    return 0;
}
```

#### 习题 1. 22

编写程序, 读入两个具有相同 ISBN 的 Sales\_item 对象并产生它们的和。

【解答】

```
#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. 23

编写程序, 读入几个具有相同 ISBN 的交易, 输出所有读入交易的和。

【解答】

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total, trans;
    // 读入交易
    std::cout << "Enter transactions:" << std::endl;
    if (std::cin >> total) {
        while (std::cin >> trans)
        if (total.same_isbn(trans)) // ISBN 相同
            total = total + trans;
        else { // ISBN 不同
            std::cout << "Different ISBN." << std::endl;
            return -1;
        }
        // 输出交易之和
        std::cout << "The total information: " << std::endl
        << "ISBN, number of copies sold, "
        << "total revenue, and average price are:"
        << std::endl << total;
    }
}
```

```

else {
std::cout << "No data?!" << std::endl;
return -1;
}
return 0;
}

```

### 习题 1.24

编写程序，读入几笔不同的交易。对于每笔新读入的交易，要确定它的 ISBN 是否和以前的交易的 ISBN 一样，并且记下每一个 ISBN 的交易的总数。通过给定多笔不同的交易来测试程序。这些交易必须代表多个不同的 ISBN，但是每个 ISBN 的记录应分在同一组。

【解答】

```

#include <iostream>
#include "Sales_item.h"
int main()
{
// 声明变量以保存交易记录以及具有相同 ISBN 的交易的数目
Sales_item trans1, trans2;
int amount;
// 读入交易
std::cout << "Enter transactions:" << std::endl;
std::cin >> trans1;
amount=1;
while (std::cin >> trans2)
if (trans1.same_isbn(trans2))// ISBN 相同
++amount;
else {
// ISBN 不同
std::cout << "Transaction amount of previous ISBN: "
<< amount << std::endl;
trans1 = trans2;
amount=1;
}
// 输出最后一个 ISBN 的交易数目
std::cout << "Transaction amount of the last ISBN: "
<< amount << std::endl;
return 0;
}

```

### 习题 1.25

使用源自本书配套网站的 Sales\_item.h 头文件，编译并执

行 1.6 节给出的书店程序。

【解答】

可从 C++ Primer (第 4 版) 的配套网站 ([http://www.awprofessional.com/cpp\\_primer](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。

### 习题 1.26

在书店程序中，我们使用了加法操作符而不是复合赋值操作符将 trans 加到 total 中，为什么我们不使用复合赋值操作符？

【解答】

因为在 1.5.1 节中提及的 Sales\_item 对象上的操作中只包含了 + 和 =，没有包含 += 操作。（但事实上，使用 Sales\_item.h 文件，已经可以用 += 操作符取代 = 和 + 操作符的复合使用。）

## 第二章 变量和基本类型

### 习题 2.1

int、long 和 short 类型之间有什么差别？

【解答】

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 2.2

unsigned 和 signed 类型有什么差别？

【解答】

前者为无符号类型，只能表示大于或等于 0 的数。后者为带符号类型，可以表示正数、负数和 0。

### 习题 2.3

如果在某机器上 short 类型占 16 位，那么可以赋给 short 类型的最大数是什么？unsigned short 类型的最大数又是什么？

【解答】

若在某机器上 short 类型占 16 位，那么可以赋给 short 类型的最大数是 215-1，即 32767；而 unsigned short 类型的最大数为 216-1，即 65535。

#### 习题 2.4

当给 16 位的 unsigned short 对象赋值 100000 时,赋的值是什么?

【解答】

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 2.5

float 类型 and double 类型有什么差别?

【解答】

二者的存储位数不同(一般而言, float 类型为 32 个二进制位, double 类型为 64 个二进制位),因而取值范围不同,精度也不同( float 类型只能保证 6 位有效数字,而 double 类型至少能保证 10 位有效数字)。

#### 习题 2.6

要计算抵押贷款的偿还金额,利率、本金和付款额应分别选用哪种类型?解释你选择的理由。

【解答】

利率可以选择 float 类型,因为利率通常为百分之几。一般只保留到小数点后两位,所以 6 位有效数字就足以表示了。本金可以选择 long 类型,因为本金通常为整数。long 类型可表示的最大整数一般为  $2^{31}-1$  (即 2147483647),应该足以表示了。付款额一般为实数,可以选择 double 类型,因为 float 类型的 6 位有效数字可能不足以表示。

#### 习题 2.7

解释下列字面值常量的不同之处。

- (a) 'a', L'a", "a", L"a"
- (b) 10, 10u, 10L, 10uL, 012, 0xC
- (c) 3.14, 3.14f, 3.14L

【解答】

- (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 2.8

确定下列字面值常量的类型:

- (a) -10 (b) -10u (c) -10. (d) -10e-2

【解答】

- (a) int 型
- (b) unsigned int 型
- (c) double 型
- (d) double 型

#### 习题 2.9

下列哪些(如果有)是非法的?

- (a) "Who goes with F\145rgus?\012"
- (b) 3.14e1L (c) "two" L"some"
- (d) 1024f (e) 3.14UL
- (f) "multiple line comment"

【解答】

(c) 非法。因为字符串字面值与宽字符串字面值的连接是未定义的。

(d) 非法。因为整数 1024 后面不能带后缀 f。

(e) 非法。因为浮点字面值不能带后缀 U。

(f) 非法。因为分两行书写的字符串字面值必须在第一行的末尾加上反斜线。

#### 习题 2.10

使用转义字符编写一段程序, 输出 2M, 然后换行。修改程序, 输出 2, 跟着一个制表符, 然后是 M, 最后是换行符。

【解答】

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 2.11

编写程序, 要求用户输入两个数——底数(base)和指数(exponent), 输出底数的指数次方的结果。

【解答】



```

#include <iostream>
int main()
{
// 局部对象
int base, exponent;
long result=1;
// 读入底数 ( base ) 和指数 ( exponent )
std::cout << "Enter base and exponent:" << std::endl;
std::cin >> base >> exponent;
if (exponent < 0) {
std::cout << "Exponent can't be smaller than 0" <<
std::endl;
return -1;
}
if (exponent > 0) {
// 计算底数的指数次方
for (int cnt = 1; cnt <= exponent; ++cnt)
result *= base;
}
std::cout << base
<< " raised to the power of "
<< exponent << ": "
<< result << std::endl;
return 0;
}

```

#### 习题 2.12

区分左值和右值，并举例说明。

【解答】

左值 ( lvalue ) 就是变量的地址，或者是一个代表“对象在内存中的位置”的表达式。右值 ( rvalue ) 就是变量的值，见 2.3.1 节。变量名出现在赋值运算符的左边，就是一个左值；而出现在赋值运算符右边的变量名或字面常量就是一个右值。

例如：

val1=val2/8

这里的 val1 是个左值，而 val2 和 8 都是右值。

#### 习题 2.13

举出一个需要左值的例子。

【解答】

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 2.14

下面哪些 ( 如果有 ) 名字是非法的？更正每个非法的标识符名字。

- (a) int double = 3.14159; (b) char \_;
- (c) bool catch-22; (d) char 1\_or\_2 ='1';
- (e) float Float = 3.14f;

【解答】

- (a) double 是 C++ 语言中的关键字，不能用作用户标识符，所以非法。此语句可改为：double dval = 3.14159;。
- (c) 名字 catch-22 中包含在字母、数字和下划线之外的字符“-”，所以非法。可将其改为：catch\_22;。
- (d) 名字 1\_or\_2 非法，因为标识符必须以字母或下划线开头，不能以数字开头。可将其改为：one\_or\_two;。

#### 习题 2.15

下面两个定义是否不同？有何不同？

```
int month = 9, day = 7;
```

```
int month =09, day = 07;
```

如果上述定义有错的话，那么应该怎样改正呢？

【解答】

这两个定义不同。前者定义了两个 int 型变量，初值分别为 9 和 7；后者也定义了两个 int 型变量，其中 day 被初始化为八进制值 7；而 month 的初始化有错：试图将 month 初始化为八进制值 09，但八进制数字范围为 0~7，所以出错。可将第二个定义改为：int month =011, day = 07;

#### 习题 2.16

假设 calc 是一个返回 double 对象的函数。下面哪些是非法定义？改正所有的非法定义。

- (a) int car = 1024, auto = 2048;
- (b) int ival = ival;
- (c) std::cin >> int input\_value;
- (d) double salary = wage = 9999.99;
- (e) double calc = calc();

【解答】

- (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 2.17

下列变量的初始值 ( 如果有 ) 是什么 ?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
    // ...
    return 0;
}
```

【解答】

global\_str 和 local\_str 的初始值均为空字符串 ,global\_int 的初始值为 0 , local\_int 没有初始值。

#### 习题 2. 18

解释下列例子中 name 的意义 :

```
extern std::string name;
std::string name("exercise 3.5a");
extern std::string name("exercise 3.5a");
```

【解答】

第一条语句是一个声明 , 说明 std::string 变量 name 在程序的其他地方定义。

第二条语句是一个定义 , 定义了 std::string 变量 name , 并将 name 初始化为 "exercise 3.5a"。

第三条语句也是一个定义 , 定义了 std::string 变量 name , 并将 name 初始化为 "exercise 3.5a" , 但这个语句只能出现在函数外部 ( 即 , name 是一个全局变量 )。

#### 习题 2. 19

下列程序中 j 的值是多少 ?

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
    // ...
}
```

【解答】

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. 20

下列程序段将会输出什么 ?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

【解答】

输出为 :

100 45

for 语句中定义的变量 i , 其作用域仅限于 for 语句内部。输出的 i 值是 for 语句之前所定义的变量 i 的值。

#### 习题 2. 21

下列程序合法吗 ?

```
int sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << "Sum from 0 to " << i
<< " is " << sum << std::endl;
```

【解答】

不合法。因为变量 i 具有语句作用域 , 只能在 for 语句中使用 , 输出语句中使用 i 属非法。

#### 习题 2. 22

下列程序段虽然合法 , 但是风格很糟糕。有什么问题呢 ? 怎样改善 ?

```
for (int i = 0; i < 100; ++i)
// process i
```

【解答】

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. 23

下列哪些语句合法 ? 对于那些不合法的使用 , 解释原因。

```
(a) const int buf;
(b) int cnt = 0;
const int sz = cnt;
(c) cnt++; sz++;
```

【解答】

(a) 不合法。因为定义 const 变量 ( 常量 ) 时必须进行初始化 , 而 buf 没有初始化。

- (b) 合法。  
(c) 不合法。因为修改了 const 变量 sz 的值。

#### 习题 2.24

下列哪些定义是非法的？为什么？如何改正？

- (a) int ival = 1.01; (b) int &rval1 = 1.01;  
(c) int &rval2 = ival; (d) const int &rval3 = 1;

【解答】

(b)非法。因为 rval1 是一个非 const 引用，非 const 引用不能绑定到右值，而 1.01 是一个右值。可改正为 int &rval1 = ival; (假设 ival 是一个已定义的 int 变量)。

#### 习题 2.25

在习题 2.24 给出的定义下，下列哪些赋值是非法的？如果赋值合法，解释赋值的作用。

- (a) rval2 = 3.14159; (b) rval2 = rval3;  
(c) ival = rval3; (d) rval3 = ival;

【解答】

(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 2.26

(a)中的定义和(b)中的赋值存在哪些不同？哪些是非法的？

- (a) int ival = 0; (b) ival = ri;  
const int &ri = 0; ri = ival;

【解答】

int ival = 0; 定义 ival 为 int 变量，并将其初始化为 0。  
const int &ri = 0; 定义 ri 为 const 引用，并将其绑定到右值 0。  
ival = ri; 将 0 值赋给 ival。  
ri = ival; 试图对 ri 赋值，这是非法的，因为 ri 是 const 引用，不能赋值。

#### 习题 2.27

下列代码输出什么？

```
int i, &ri = i;  
i = 5; ri = 10;  
std::cout << i << " " << ri << std::endl;
```

【解答】

输出：

10 10

ri 是 i 的引用，对 ri 进行赋值，实际上相当于对 i 进行赋值，所以输出 i 和 ri 的值均为 10。

#### 习题 2.28

编译以下程序，确定你的编译器是否会警告遗漏了类定义后面的分号。

```
class Foo {  
    // empty  
} // Note: no semicolon  
  
int main()  
{  
    return 0;  
}
```

如果编译器的诊断结果难以理解，记住这些信息以备后用。

【解答】

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 2.29

区分类中的 public 部分和 private 部分。

【解答】

类中 public 部分定义的成员在程序的任何部分都可以访问。通常在 public 部分放置操作，以便程序中的其他部分可以执行这些操作。类中 private 部分定义的成员只能被作为类的组成部分的代码（以及该类的友元）访问。通常在 private 部分放置数据，以对对象的内部数据进行隐藏。

#### 习题 2.30

定义表示下列类型的类的数据成员：

- (a) 电话号码 (b)地址  
(c) 员工或公司 (d)梦芭莎优惠券网里面的文章

【解答】

- (a) 电话号码

```
class Tel_number {  
public:
```

```
//...对象上的操作
private:
std::string country_number;
std::string city_number;
std::string phone_number;
};
```

(b) 地址

```
class Address {
public:
//...对象上的操作
private:
std::string country;
std::string city;
std::string street;
std::string number;
};
```

(c) 员工或公司

```
class Employee {
public:
// ...对象上的操作
private:
std::string ID;
std::string name;
char sex;
Address addr;
Tel_number tel;
};

class Company {
public:
// ...对象上的操作
private:
std::string name;
Address addr;
Tel_number tel;
};
```

(d) 某大学的学生

```
class Student {
public:
// ...对象上的操作
private:
std::string ID;
std::string name;
char sex;
std::string dept; // 所在系
```

```
std::string major;
Address home_addr;
Tel_number tel;
};
```

注意，在不同的具体应用中，类的设计会有所不同，这里给出的只是一般性的简单例子。

## 习题 2.31

判别下列语句哪些是声明，哪些是定义，请解释原因。

- (a) extern int ix = 1024;
- (b) int iy;
- (c) extern int iz;
- (d) extern const int &i;

【解答】

- (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。

## 习题 2.32

下列声明和定义哪些应该放在头文件中？哪些应该放在源文件中？请解释原因。

- (a) int var;
- (b) const double pi = 3.1416;
- (c) extern int total = 255;
- (d) const double sq2 = sqrt(2.0);

【解答】

- (a)、(c)、(d)应放在源文件中，因为(a)和(c)是变量定义，定义通常应放在源文件中。(d)中的 const 变量 sq2 不是用常量表达式初始化的，所以也应该放在源文件中。
- (b)中的 const 变量 pi 是用常量表达式初始化的，应该放在头文件中。参见 2.9.1 节。

## 习题 2.33

确定你的编译器提供了哪些提高警告级别的选项。使用这些选项重新编译以前选择的程序，查看是否会报告新的问题。

【解答】

在笔者所用的编译器 (Microsoft Visual C++ .NET 2003) 中，在 Project 菜单中选择 Properties 菜单项，在 Configuration Properties → C/C++ → General → Warning Level 中可以选择警告级别。

## 第三章 标准库类型

### 习题 3.1

用适当的 using 声明，而不用 std::前缀，访问标准库中的名字，重新编写 2.3 节的程序，计算一给定数的给定次幂的结果。

【解答】

```
#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 3.2

什么是默认构造函数？

【解答】

默认构造函数 ( default constructor ) 就是在没有显式提供初始化式时调用的构造函数。它由不带参数的构造函数，或者为所有形参提供默认实参的构造函数定义。如果定义某个类的变量时没有提供初始化式，就会使用默认构造函数。如果用户定义的类中没有显式定义任何构造函数，编译器就会自动为该类生成默认构造函数，称为合成的默认构造函数

( synthesized default constructor )。

### 习题 3.3

列举出三种初始化 string 对象的方法。

【解答】

- (1) 不带初始化式，使用默认构造函数初始化 string 对象。
- (2) 使用一个已存在的 string 对象作为初始化式，将新创建的 string 对象初始化为已存在对象的副本。
- (3) 使用字符串字面值作为初始化式，将新创建的 string 对象初始化为字符串字面值的副本。

### 习题 3.4

s 和 s2 的值分别是什么？

```
string s;
int main() {
    string s2;
}
```

【解答】

s 和 s2 的值均为空字符串。

### 习题 3.5

编写程序实现从标准输入每次读入一行文本。然后改写程序，每次读入一个单词。

【解答】

//从标准输入每次读入一行文本

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string line;
    // 一次读入一行，直至遇见文件结束符
    while (getline(cin, line))
        cout << line << endl; // 输出相应行以进行验证
    return 0;
}
```

修改后程序如下：

//从标准输入每次读入一个单词

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
```

```
string word;
// 一次读入一个单词，直至遇见文件结束符
while (cin >> word)
cout << word << endl; // 输出相应单词以进行验证
return 0;
}
```

注意，一般而言，应该尽量避免使用 `using` 指示而使用 `using` 声明（参见 17.2.4 节），因为如果应用程序中使用了多个库，使用 `using` 指示引入这些库中定义

的名字空间，容易导致名字冲突。但本书中的程序都只使用了标准库，没有使

用其他库。使用 `using` 指示引入名字空间 `std` 中定义的所有名字不会发生名字

冲突。因此为了使得代码更为简洁以节省篇幅，本书的许多代码中都使用了

`using` 指示 `using namespace std;` 来引入名字空间 `std`。另外，本题中并未要求

输出，加入输出是为了更清楚地表示读入的结果。本书后面部分有些地方与此

类似处理，不再赘述。

### 习题 3.6

解释 `string` 类型的输入操作符和 `getline` 函数分别如何处理空白字符。

#### 【解答】

`string` 类型的输入操作符对空白字符的处理：读取并忽略有效字符（非空白字

符）之前所有的空白字符，然后读取字符直至再次遇到空白字符，读取终止（该

空白字符仍留在输入流中）。

`getline` 函数对空白字符的处理：不忽略行开头的空白字符，读取字符直至遇到

换行符，读取终止并丢弃换行符（换行符从输入流中去掉但并不存储在 `string`

对象中）。

### 习题 3.7

编一个程序读入两个 `string` 对象，测试它们是否相等。若不相等，则指出两个

中哪个较大。接着，改写程序测试它们的长度是否相等，若不相等，则指出两

个中哪个较长。

#### 【解答】

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;
    C++ Primer (4 版) 习题解答
    41
    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;
    C++ Primer (4 版) 习题解答
    42
    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 3.8

编一个程序，从标准输入读取多个 string 对象，把它们连接起来存放到一个更大的 string 对象中，并输出连接后的 string 对象。接着，改写程序，将连接后相邻 string 对象以空格隔开。

【解答】

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string result_str, str;
    // 读入多个 string 对象并进行连接
    C++ Primer (4 版) 习题解答
    43
    cout << "Enter strings(Ctrl+Z to end):" << endl;
    while (cin >> str)
        result_str = result_str + str;
    // 输出连接后的 string 对象
    cout << "String equal to the concatenation of these
    strings is:"
    << endl << result_str << endl;
    return 0;
}
```

改写后的程序：

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string result_str, str;
    // 读入多个 string 对象并进行连接
    cout << "Enter strings(Ctrl+Z to end):" << endl;
    cin >> result_str; // 读入第一个 string 对象，放到结果对象中
    while (cin >> str)
        result_str = result_str + ' ' + str;
    // 输出连接后的 string 对象
    cout << "String equal to the concatenation of these
    strings is:"
```

```
<< endl << result_str << endl;
return 0;
```

C++ Primer (4 版) 习题解答  
44

```
}
```

### 习题 3.9

下列程序实现什么功能？实现合法吗？如果不合法，说明理由。

```
string s;
cout << s[0] << endl;
```

【解答】

该程序段输出 string 对象 s 所对应字符串的第一个字符。实现不合法。因为 s 是一个空字符串，其长度为 0，因此 s[0] 是无效的。

注意，在一些编译器（如 Microsoft Visual C++ .NET 2003）的实现中，该程序段并不出现编译错误。

### 习题 3.10

编一个程序，从 string 对象中去掉标点符号。要求输入到程序的字符串必须含有标点符号，输出结果则是去掉标点符号后的 string 对象。

【解答】

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
int main()
{
    string s, result_str;
    bool has_punct = false; // 用于标记字符串中是否有标点
    char ch;
    // 输入字符串
    C++ Primer (4 版) 习题解答
    45
    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 3.11

下面哪些 vector 定义不正确？

- (a) vector< vector<int> > ivec;
- (b) vector<string> svec = ivec;
- (c) vector<string> svec(10,"null");

C++ Primer (4 版) 习题解答

46

#### 【解答】

(b) 不正确。因为 svec 定义为保存 string 对象的 vector 对象，而 ivec 是

保存 vector<int> 对象的 vector 对象（即 ivec 是 vector 的 vector），二者

的元素类型不同，所以不能用 ivec 来初始化 svec。

### 习题 3.12

下列每个 vector 对象中元素个数是多少？各元素的值是什么？

- (a) vector<int> ivec1;
- (b) vector<int> ivec2(10);
- (c) vector<int> ivec3(10,42);
- (d) vector<string> svec1;
- (e) vector<string> svec2(10);
- (f) vector<string> svec3(10,"hello");

#### 【解答】

- (a) 元素个数为 0。
- (b) 元素个数为 10，各元素的值均为 0。
- (c) 元素个数为 10，各元素的值均为 42。
- (d) 元素个数为 0。
- (e) 元素个数为 10，各元素的值均为空字符串。
- (f) 元素个数为 10，各元素的值均为 "hello"。

### 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。

如果读入元素个数

为奇数，则提示用户最后一个元素没有求和，并输出其值。

然后修改程序：头

尾元素两两配对（第一个和最后一个，第二个和倒数第二个，以此类推），计

算每对元素的和，并输出。

#### 【解答】

// 读一组整数到 vector 对象，计算并输出每对相邻元素的和

C++ Primer (4 版) 习题解答

47

```
#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;
```

```
}
```

C++ Primer (4 版) 习题解答

48

```
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:" << 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;
    }
    C++ Primer ( 4 版 ) 习题解答
    49
    cout << "Sum of each pair of counterpart elements in
    the vector:"
    << endl;
    vector<int>::size_type cnt = 0;
    for (vector<int>::size_type 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 3.14

读入一段文本到 vector 对象，每个单词存储为 vector 中的一个元素。把 vector 对象中每个单词转化为大写字母。输出 vector 对象中转化后的元素，每 8 个单

词为一行输出。

### 【解答】

//读入一段文本到 vector 对象，每个单词存储为 vector 中的一个元素。

C++ Primer ( 4 版 ) 习题解答

50

//把 vector 对象中每个单词转化为大写字母。

//输出 vector 对象中转化后的元素，每 8 个单词为一行输出

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
    vector<string> svec;
    string str;
    // 读入文本到 vector 对象
    cout << "Enter text(Ctrl+Z to end):" << endl;
    while (cin>>str)
        svec.push_back(str);
    //将 vector 对象中每个单词转化为大写字母,并输出
    if (svec.size() == 0) {
        cout << "No string?!" << endl;
        return -1;
    }
    cout << "Transformed elements from the vector:"
    << endl;
    for (vector<string>::size_type ix = 0; ix != svec.size();
    ++ix) {
        C++ Primer ( 4 版 ) 习题解答
        51
        for (string::size_type index = 0; index != svec[ix].size();
        ++index)
            if (islower(svec[ix][index]))
                //单词中下标为 index 的字符为小写字母
                svec[ix][index] = toupper(svec[ix][index]);
        cout << svec[ix] << " ";
        if ((ix + 1) % 8 == 0) //每 8 个单词为一行输出
            cout << endl;
    }
    return 0;
}
```

### 习题 3.15

下面程序合法吗？如果不合法，如何更正？

```
vector<int> ivec;  
ivec[0] = 42;
```

【解答】

不合法。因为 ivec 是空的 vector 对象 其中不含任何元素，而下标操作只

能用于获取已存在的元素。

更正：将赋值语句改为语句 ivec.push\_back(42);。

### 习题 3.16

列出三种定义 vector 对象的方法，给定 10 个元素，每个元素值为 42。指出是

否还有更好的实现方法，并说明为什么。

【解答】

方法一：

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

C++ Primer (4 版) 习题解答

52

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 3.17

### C++ Primer (4 版) 习题解答

53

重做 3.3.2 节的习题，用迭代器而不是下标操作来访问 vector 中的元素。

【解答】

重做习题 3.13 如下：

//读一组整数到 vector 对象，计算并输出每对相邻元素的和

//使用迭代器访问 vector 中的元素

```
#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;
```

```
    vector<int>::size_type cnt = 0;
```

C++ Primer (4 版) 习题解答

54

```
    for (vector<int>::iterator iter = ivec.begin();
```

```
        iter < ivec.end()-1;
```

```
        iter = iter + 2) {
```

```
        cout << *iter + *(iter+1) << "\t";
```

```
        ++cnt;
```

```
        if (cnt % 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.end()-1) << endl;
```

```

return 0;
}
//读一组整数到 vector 对象, 计算首尾配对元素的和并输出
//使用迭代器访问 vector 中的元素
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int> ivec;
int ival;
C++ Primer ( 4 版 ) 习题解答
55
//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;
for (vector<int>::iterator first = ivec.begin(),
last = ivec.end() - 1;
first < last;
++first, --last) {
cout << *first + *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 "
C++ Primer ( 4 版 ) 习题解答
56
<< "and its value is "
<< *first << endl;
return 0;

```

```

}
重做习题 3.14 如下 :
//读入一段文本到 vector 对象,每个单词存储为 vector 中
的一个元素。
//把 vector 对象中每个单词转化为大写字母。
//输出 vector 对象中转化后的元素, 每 8 个单词为一行输出。
//使用迭代器访问 vector 中的元素
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
vector<string> svec;
string str;
//读入文本到 vector 对象
cout << "Enter text(Ctrl+Z to end):" << endl;
while (cin>>str)
svec.push_back(str);
//将 vector 对象中每个单词转化为大写字母,并输出
if (svec.size() == 0) {
C++ Primer ( 4 版 ) 习题解答
57
cout << "No string?!" << endl;
return -1;
}
cout << "Transformed elements from the vector:"
<< endl;
vector<string>::size_type cnt = 0;
for (vector<string>::iterator iter = svec.begin();
iter != svec.end(); ++iter) {
for (string::size_type index = 0; index != (*iter).size();
++index)
if (islower((*iter)[index]))
//单词中下标为 index 的字符为小写字母
(*iter)[index] = toupper((*iter)[index]);
cout << *iter << " ";
++cnt;
if (cnt % 8 == 0) //每 8 个单词为一行输出
cout << endl;
}
return 0;

```

```
}
```

### 习题 3.18

编写程序来创建有 10 个元素的 vector 对象。用迭代器把每个元素值改为当前值的 2 倍。

【解答】

C++ Primer (4 版) 习题解答

58

```
//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
#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
    for (vector<int>::iterator iter = ivec.begin();
        iter != ivec.end(); ++iter)
        *iter = (*iter)*2;
    return 0;
}
```

### 习题 3.19

验证习题 3.18 的程序，输出 vector 的所有元素。

【解答】

//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()
```

```
{
```

C++ Primer (4 版) 习题解答

59

```
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 3.20

解释一下在上几个习题的程序实现中你用了哪种迭代器，并说明原因。

【解答】

上述几个习题的程序实现中使用了类型分别为 `vector<int>::iterator` 和 `vector<string>::iterator` 的迭代器，通过这些迭代器分别访问元素类型为 `int` 和 `string` 的 vector 对象中的元素。

### 习题 3.21

何时使用 `const` 迭代器？又在何时使用 `const_iterator`？解释两者的区别。

【解答】

`const` 迭代器是迭代器常量，该迭代器本身的值不能修改，即该迭代器在定义时

需要初始化，而且初始化之后，不能再指向其他元素。若需要指向固定元素的

迭代器，则可以使用 `const` 迭代器。

`const_iterator` 是一种迭代器类型，对这种类型的迭代器解引用会得到一个指

向 `const` 对象的引用，即通过这种迭代器访问到的对象是常量。该对象不能修

改，因此，`const_iterator` 类型只能用于读取容器内的元素，不能修改元素的

值。若只需遍历容器中的元素而无需修改它们，则可以使用 `const_iterator`。

### 习题 3.22

如果采用下面的方法来计算 `mid` 会产生什么结果？

C++ Primer (4 版) 习题解答

60

```
vector<int>::iterator mid = (vi.begin() + vi.end())/2;
```

【解答】

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。

### 习题 3.23

解释下面每个 `bitset` 对象包含的位模式：

(a) `bitset<64> bitvec(32);`

(b) `bitset<32> bv(1010101);`

(c) `string bstr; cin >> bstr; bitset<8> bv(bstr);`

【解答】

(a) `bitvec` 有 64 个二进制位，(位编号从 0 开始) 第 5 位置为 1，其余位置均

为 0。

(b) `bv` 有 32 个二进制位，(位编号从 0 开始) 第 0、2、4、

5、7、8、11、13、  
14、16、17、18、19 位置为 1，其余位置均为 0。因为十进制数 1010101 对应的  
二进制数为 000000000000011110110100110110101。  
(c) bv 有 8 个二进制位，( 位编号从 0 开始 ) 用读入的字符串的从右至左的 8  
个字符对 bv 的 0~7 位进行初始化。

#### 习题 3.24

考虑这样的序列 1,2,3,5,8,13,21，并初始化一个将该序列数字所对应的位置设

置为 1 的 `bitset<32>` 对象。然后换个方法，给定一个空的 `bitset` 对象，编写一

小段程序把相应的数位设置为 1。

#### 【解答】

`bitset<32>` 对象的初始化：

```
bitset<32> bv(0x20212e)
```

方法二：

```
bitset<32> bv;
```

```
int x = 0, y = 1, z;
```

C++ Primer ( 4 版 ) 习题解答

61

```
z = x + y;
```

```
while (z <= 21) {
```

```
    bv.set(z);
```

```
    x = y;
```

```
    y = z;
```

```
    z = x + y;
```

```
}
```

注意，设置为 1 的数位的位编号符合斐波那契数列的规律。

## 第四章 数组和指针

#### 习题 4.1

假设 `get_size` 是一个没有参数并返回 `int` 值的函数，下列哪些定义是非法的？

为什么？

```
unsigned buf_size = 1024
```

(a) `int ia[buf_size];`

(b) `int ia[get_size()];`

(c) `int ia[4*7-14];`

(d) `char st[11] = "fundamental";`

#### 【解答】

(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 4.2

下列数组的值是什么？

```
string sa[10];
```

C++ Primer ( 4 版 ) 习题解答

62

```
int ia[10];
```

```
int main(){
```

```
    string sa2[10];
```

```
    int ia2[10];
```

```
}
```

#### 【解答】

`sa` 和 `sa2` 为元素类型为 `string` 的数组，自动调用 `string` 类的默认构造函数将

各元素初始化为空字符串；`ia` 为在函数体外定义的内置数组，各元素初始化为

0；`ia2` 为在函数体内定义的内置数组，各元素未初始化，其值不确定。

#### 习题 4.3

下列哪些定义是错误的？

(a) `int ia[7] = {0, 1, 1, 2, 3, 5, 8};`

(b) `vector<int> ivec = {0, 1, 1, 2, 3, 5, 8};`

(c) `int ia2[] = ia;`

(d) `int ia3[] = ivec;`

#### 【解答】

(b)错误。`vector` 对象不能用这种方式进行初始化。

(c)错误。不能用一个数组来初始化另一个数组。

(d)错误。不能用 `vector` 对象来初始化数组。

#### 习题 4.4

如何初始化数组的一部分或全部元素？

#### 【解答】

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

C++ Primer (4 版) 习题解答

63

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

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，也可以用一串字符串字面值进行初

始化。

习题 4.5

列出使用数组而不是 vector 的缺点。

【解答】

与 vector 类型相比，数组具有如下缺点：数组的长度是固定的，而且数组

不提供获取其容量大小的 size 操作，也不提供自动添加元素的 push\_back 操作。

因此，程序员无法在程序运行时知道一个给定数组的长度，而且如果需要更改

数组的长度，程序员只能创建一个更大的新数组，然后把原数组的所有元素复

制到新数组的存储空间中去。与使用 vector 类型的程序相比，使用内置数组的

程序更容易出错且难以调试。

习题 4.6

下面的程序段企图将下标值赋给数组的每个元素，其中在下标操作上有一些错

误，请指出这些错误。

```
const size_t array_size = 10;
```

```
int ia[array_size];
```

```
for (size_t ix = 1; ix <= array_size; ++ix)
```

```
ia[ix] = ix;
```

【解答】

该程序段的错误是：数组下标使用越界。

根据数组 ia 的定义，该数组的下标值应该是 0~9(即 array\_size-1)，而不是从

1 到 array\_size，因此其中的 for 语句出错，可更正如下：

```
for (size_t ix = 0; ix < array_size; ++ix)
```

```
ia[ix] = ix;
```

习题 4.7

编写必要的代码将一个数组赋给另一个数组，然后把这段代码改用 vector 实现。

考虑如何将一个 vector 赋给另一个 vector。

C++ Primer (4 版) 习题解答

64

【解答】

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)
```

```
C++ Primer (4 版) 习题解答
```

65

```
ivec2.push_back(*iter);
```

```
return 0;
```

```
}
```

习题 4.8

编写程序判断两个数组是否相等，然后编写一段类似的程序比较两个 vector。

【解答】

判断两个数组是否相等，可用如下程序：

```
//判断两个数组是否相等
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
const int arr_size = 6;
```

```
int ia1[arr_size], ia2[arr_size];
```

```

size_t ix;
//读入两个数组的元素值
cout << "Enter " << arr_size
<< " numbers for array1:" << endl;
for (ix = 0; ix != arr_size; ++ix)
cin >> ia1[ix];
cout << "Enter " << arr_size
<< " numbers for array2:" << endl;
for (ix = 0; ix != arr_size; ++ix)
cin >> ia2[ix];
C++ Primer ( 4 版 ) 习题解答
66
//判断两个数组是否相等
for (ix = 0; ix != arr_size; ++ix)
if (ia1[ix] != ia2[ix]) {
cout << "Array1 is not equal to array2." << endl;
return 0;
}
cout << "Array1 is equal to array2." << endl;
return 0;
}
判断两个 vector 是否相等，可用如下程序：
//判断两个 vector 是否相等
//使用迭代器访问 vector 中的元素
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int> ivec1, ivec2;
int ival;
//读入两个 vector 的元素值
cout << "Enter numbers for vector1(-1 to end):" <<
endl;
cin >> ival;
while (ival != -1) {
ivec1.push_back(ival);
C++ Primer ( 4 版 ) 习题解答
67
cin >> ival;
}
cout << "Enter numbers for vector2(-1 to end):" <<
endl;
cin >> ival;

```

```

while (ival != -1) {
ivec2.push_back(ival);
cin >> ival;
}
//判断两个 vector 是否相等
if (ivec1.size() != ivec2.size()) //长度不等的 vector 不相
等
cout << "Vector1 is not equal to vector2." << endl;
else if (ivec1.size() == 0) //长度都为 0 的 vector 相等
cout << "Vector1 is equal to vector2." << endl;
else //两个 vector 长度相等且不为 0
vector<int>::iterator iter1, iter2;
iter1 = ivec1.begin();
iter2 = ivec2.begin();
while (*iter1 == *iter2 && iter1 != ivec1.end()
&& iter2 != ivec2.end()) {
++iter1;
++iter2;
}
if (iter1 == ivec1.end())//所有元素都相等
cout << "Vector1 is equal to vector2." << endl;
C++ Primer ( 4 版 ) 习题解答
68
else
cout << "Vector1 is not equal to vector2." << endl;
}
return 0;
}

```

#### 习题 4.9

编写程序定义一个有 10 个 int 型元素的数组，并以元素在数组中的位置作为各元素的初值。

#### 【解答】

//定义一个有 10 个 int 型元素的数组，  
//并以元素在数组中的位置（1~10）作为各元素的初值

```

int main()
{

```

```

const int array_size = 10;
int ia[array_size];
for (size_t ix = 0; ix != array_size; ++ix)
ia[ix] = ix+1;
return 0;
}

```

#### 习题 4.10



下面提供了两种指针声明的形式，解释宁愿使用第一种形式的原因：

```
int *ip; // good practice
int* ip; // legal but misleading
```

C++ Primer (4 版) 习题解答

69

【解答】

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 4.11

解释下列声明语句，并指出哪些是非法的，为什么？

- (a) `int* ip;`
- (b) `string s, *sp = 0;`
- (c) `int i; double* dp = &i;`
- (d) `int* ip, ip2;`
- (e) `const int i = 0, *p = i;`
- (f) `string *p = NULL;`

【解答】

(a)合法。定义了一个指向 `int` 型对象的指针 `ip`。

(b)合法。定义了 `string` 对象 `s` 和指向 `string` 型对象的指针 `sp`，`sp` 初始化为 0

值。

(c)非法。`dp` 为指向 `double` 型对象的指针，不能用 `int` 型对象 `i` 的地址进行初

始化。

(d)合法。定义了 `int` 对象 `ip2` 和指向 `int` 型对象的指针 `ip`。

(e)合法。定义了 `const int` 型对象 `i` 和指向 `const int` 型对象的指针 `p`，`i` 初

始化为 0，`p` 初始化为 0。

(f)合法。定义了指向 `string` 型对象的指针 `p`，并将其初始化为 0 值。

习题 4.12

已知一指针 `p`，你可以确定该指针是否指向一个有效的对象吗？如果可以，如何

确定？如果不可以，请说明原因。

【解答】

C++ Primer (4 版) 习题解答

70

无法确定某指针是否指向一个有效对象。因为，在 C++ 语言中，无法检测指

针是否未被初始化，也无法区分一个地址是有效地址，还是由指针所分配的存

储空间中存放的不确定值的二进制位形成的地址。

习题 4.13

下列代码中，为什么第一个指针的初始化是合法的，而第二个则不合法？

```
int i = 42;
void *p = &i;
long *lp = &i;
```

【解答】

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 4.14

编写代码修改指针的值；然后再编写代码修改指针所指对象的值。

【解答】

下列代码修改指针的值：

```
int *ip;
int ival1, ival2;
ip = &ival1;
ip = &ival2;
```

下列代码修改指针所指对象的值：

```
int ival = 0;
int *ip = &ival;
*ip = 8;
```

习题 4.15

C++ Primer (4 版) 习题解答

71

解释指针和引用的主要区别。

【解答】

使用引用 (reference) 和指针 (pointer) 都可间接访问另一个值，但它们之

间存在两个重要区别：(1) 引用总是指向某个确定对象 (事实上，引用就是该对

象的别名)，定义引用时没有进行初始化会出现编译错误；(2) 赋值行为上存

在差异：给引用赋值修改的是该引用所关联的对象的值，而不是使该引用与另

一个对象关联。引用一经初始化，就始终指向同一个特定对象。给指针赋值修

改的是指针对象本身，也就是使该指针指向另一对象，指针在不同时刻可指向

不同的对象 (只要保证类型匹配)。

习题 4.16



下列程序段实现什么功能？

```
int i = 42, j = 1024;
int *p1 = &i, *p2 = &j;
*p2 = *p1 * *p2;
*p1 *= *p1;
```

【解答】

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 4.17

已知 p1 和 p2 指向同一个数组中的元素, 下面语句实现什么功能？

```
p1 += p2 - p1;
```

当 p1 和 p2 具有什么值时这个语句是非法的？

【解答】

此语句使得 p1 也指向 p2 原来所指向的元素。原则上说, 只要 p1 和 p2 的类型相同, 则该语句始终是合法的。只有当 p1 和 p2 不是同类型指针时, 该语句才不合法 (不能进行-操作)。

但是, 如果 p1 和 p2 不是指向同一个数组中的元素, 则这个语句的执行结果可能是错误的。因为-操作的结果类型 ptrdiff\_t 只能保证足以存放同一数组中两个指针之间的差距。如果 p1 和 p2 不是指向同一个数组中的元素, 则-操作的结

C++ Primer (4 版) 习题解答

72

果有可能超出 ptrdiff\_t 类型的表示范围而产生溢出, 从而该语句的执行结果不能保证 p1 指向 p2 原来所指向的元素 (甚至不能保证 p1 为有效指针)。

#### 习题 4.18

编写程序, 使用指针把一个 int 型数组的所有元素设置为 0。

【解答】

```
// 使用指针把一个 int 型数组的所有元素设置为 0
int main()
{
    const size_t arr_size = 8;
    int int_arr[arr_size] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    // pbegin 指向第一个元素, pend 指向最后一个元素的下一内存位置
    for (int *pbegin = int_arr, *pend = int_arr + arr_size;
         pbegin != pend; ++pbegin)
        *pbegin = 0; // 当前元素置 0
```

```
    return 0;
}
```

#### 习题 4.19

解释下列 5 个定义的含义, 指出其中哪些定义是非法的：

- (a) int i;
- (b) const int ic;
- (c) const int \*pic;
- (d) int \*const cpi;
- (e) const int \*const cpic;

【解答】

C++ Primer (4 版) 习题解答

73

- (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 4.20

下列哪些初始化是合法的？为什么？

- (a) int i = -1;
- (b) const int ic = i;
- (c) const int \*pic = &ic;
- (d) int \*const cpi = &ic;
- (e) const int \*const cpic = &ic;

【解答】

- (a) 合法：定义了一个 int 型对象 i, 并用 int 型字面值-1 对其进行初始化。
- (b) 合法：定义了一个 int 型 const 对象 ic, 并用 int 型对象对其进行初始化。
- (c) 合法：定义了一个指向 int 型 const 对象的指针 pic, 并用 ic 的地址对其进行初始化。
- (d) 不合法：cpi 是一个指向 int 型对象的 const 指针, 不能用 const int 型对象 ic 的地址对其进行初始化。
- (e) 合法：定义了一个指向 int 型 const 对象的 const 指针 cpic, 并用 ic 的地址对其进行初始化。

#### 习题 4.21

根据上述定义，下列哪些赋值运算是合法的？为什么？

C++ Primer (4 版) 习题解答

74

- (a) `i = ic;` (b) `pic = &ic;`  
(c) `cpi = pic;` (d) `pic = cpic;`  
(e) `cpic = &ic;` (f) `ic = *cpic;`

【解答】

(a)、(b)、(d)合法。

(c)、(e)、(f)均不合法，因为 `cpi`、`cpic` 和 `ic` 都是 `const` 变量（常量），常量不能被赋值。

#### 习题 4.22

解释下列两个 `while` 循环的差别：

```
const char *cp = "hello";
int cnt;
while (cp) { ++cnt; ++cp; }
while (*cp) { ++cnt; ++cp; }
```

【解答】

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 4.23

下列程序实现什么功能？

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
}
++cp;
```

C++ Primer (4 版) 习题解答

75

【解答】

该程序段从数组 `ca` 的起始地址（即字符 `'h'` 的存储地址）开始，输出一段内

存中存放的字符，每行输出一个字符，直至存放 0 值（`null`）的字节为止。（注

意，输出的内容一般来说要多于 5 个字符，因为字符数组

`ca` 中没有 `null` 结束符。）

#### 习题 4.24

解释 `strcpy` 和 `strncpy` 的差别在哪里，各自的优缺点是什么？

【解答】

`strcpy` 和 `strncpy` 的差别在于：前者复制整个指定的字符串，后者只复制指定字符串中指定数目的字符。

`strcpy` 比较简单，而使用 `strncpy` 可以适当地控制复制字符的数目，因此比

`strcpy` 更为安全。

#### 习题 4.25

编写程序比较两个 `string` 类型的字符串，然后编写另一个程序比较两个 C 风格字符串的值。

【解答】

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;
```

C++ Primer (4 版) 习题解答

76

//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;
C++ Primer ( 4 版 ) 习题解答
77
//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;
C++ Primer ( 4 版 ) 习题解答
78
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 4.26

编写程序从标准输入设备读入一个 string 类型的字符串。考虑如何编程实现从标准输入设备读入一个 C 风格字符串。

【解答】

从标准输入设备读入一个 string 类型字符串的程序段：

```
string str;
```

```
cin >> str;
```

从标准输入设备读入一个 C 风格字符串可如下实现：

```
const int str_size = 80;
```

```
char str[str_size];
```

```
cin >> str;
```

#### 习题 4.27

假设有下面的 new 表达式, 请问如何释放 pa ?

```
int *pa = new int[10];
```

【解答】

用语句 delete [] pa; 释放 pa 所指向的数组空间。

#### 习题 4.28

C++ Primer ( 4 版 ) 习题解答

79

编写程序由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。

【解答】

// 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)
C++ Primer ( 4 版 ) 习题解答
80
*tp = *iter;
//释放动态数组的内存
delete [] pia;
return 0;
}
```

#### 习题 4.29

对本节第 5 条框中的两段程序：

- (a) 解释这两段程序实现的功能。  
 (b) 平均来说，使用 string 类型的程序执行速度要比用 C 风格字符串的快很多，  
 在我们用了 5 年的 PC 机上其平均执行速度分别是：  
 user 0.47 # string class  
 user 2.55 # C-style character string  
 你预计的也一样吗？请说明原因。

#### 【解答】

(a) 这两段程序的功能是：执行一个循环次数为 1000000 的循环，在该循环的  
 循环体中：创建一个新字符串，将一个已存在的字符串复制给新字符串，然后  
 比较两个字符串，最后释放新字符串。

(b) 使用 C 风格字符串的程序需要自己管理内存的分配和释放，而使用 string  
 类型的程序由系统自动进行内存的分配和释放，因此比使用 C 风格字符串的程  
 序要简短，执行速度也要快一些。

#### 习题 4.30

编写程序连接两个 C 风格字符串字面值，把结果存储在一个 C 风格字符串中。  
 然后再编写程序连接两个 string 类型字符串 这两个 string 类型字符串与前  
 面的 C 风格字符串字面值具有相同的内容。

#### 【解答】

连接两个 C 风格字符串字面值的程序如下：

// 连接两个 C 风格字符串字面值，

C++ Primer ( 4 版 ) 习题解答

81

// 把结果存储在一个 C 风格字符串中

#include <cstring>

int main()

```
{
const char *cp1 = "Mary and Linda ";
const char *cp2 = "are firends.";
size_t len = strlen(cp1) + strlen(cp2);
char *result_str = new char[len+1];
strcpy(result_str, cp1);
strcat(result_str, cp2);
delete [] result_str;
return 0;
}
```

相应的连接两个 string 类型字符串的程序如下：

// 连接两个 string 类型字符串

#include <string>

using namespace std;

int main()

```
{
const string str1("Mary and Linda ");
const string str2("are firends.");
```

string result\_str;

result\_str = str1;

result\_str += str2;

C++ Primer ( 4 版 ) 习题解答

82

return 0;

}

#### 习题 4.31

编写程序从标准输入设备读入字符串，并把该串存放在字符数组中。描述你的  
 程序如何处理可变长的输入。提供比你分配的数组长度长的字符串数据测试你  
 的程序。

#### 【解答】

// 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) {
C++ Primer ( 4 版 ) 习题解答
83
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 4.32

编写程序用 int 型数组初始化 vector 对象。

【解答】

```

// 用 int 型数组初始化 vector 对象
#include <iostream>
#include <vector>
using namespace std;
int main()
C++ Primer ( 4 版 ) 习题解答
84
{
const size_t arr_size = 8;
int int_arr[arr_size];

```

```

// 输入数组元素
cout << "Enter " << arr_size << " numbers:" << endl;
for (size_t ix = 0; ix != arr_size; ++ix)
cin >> int_arr[ix];
// 用 int 型数组初始化 vector 对象
vector<int> ivec(int_arr, int_arr + arr_size);
return 0;
}

```

#### 习题 4.33

编写程序把 int 型 vector 复制给 int 型数组。

【解答】

```

// 把 int 型 vector 复制给 int 型数组
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int> ivec;
int ival;
// 输入 vector 元素
cout << "Enter numbers: (Ctrl+Z to end)" << endl;
C++ Primer ( 4 版 ) 习题解答
85
while (cin >> ival)
ivec.push_back(ival);
// 创建数组
int *parr = new int[ivec.size()];
// 复制元素
size_t ix = 0;
for (vector<int>::iterator iter = ivec.begin();
iter != ivec.end(); ++iter, ++ix)
parr[ix] = *iter;
// 释放数组
delete [] parr;
return 0;
}

```

#### 习题 4.34

编写程序读入一组 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。

【解答】

//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

C++ Primer (4 版) 习题解答

86

```
#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
```

C++ Primer (4 版) 习题解答

87

```
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;
```

```
}
```

习题 4.35

输出习题 4.34 中建立的 vector 对象和数组的内容。输出数组后,记得释放字符数组。

【解答】

//4-35.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,

//然后把指向该数组的指针插入字符指针数组。

//输出建立的 vector 对象和数组的内容

```
#include <iostream>
```

```
#include <vector>
```

```
#include <string>
```

```
using namespace std;
```

C++ Primer (4 版) 习题解答

88

```
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;
```

```
}
```

```
// 输出 vector 对象的内容
```

```
cout << "Content of vector:" << endl;
```

```
for (vector<string>::iterator iter2 = svec.begin();
```

C++ Primer (4 版) 习题解答

89

```
iter2 != svec.end(); ++iter2)
```

```

cout << *iter2 << endl;
// 输出字符数组的内容
cout << "Content of character arrays:" << endl;
for (ix =0; ix != svec.size(); ++ix)
cout << parr[ix] << endl;
// 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;
}

```

#### 习题 4.36

重写程序输出 ia 数组的内容，要求在外层循环中不能使用 typedef 定义的类型。

【解答】

```

//4-36.cpp
//重写程序输出 ia 数组的内容
//在外层循环中不使用 typedef 定义的类型
#include <iostream>
using namespace std;
int main()
{
C++ Primer (4 版) 习题解答
90
int ia[3][4] = { // 3 个元素,每个元素是一个有 4 个 int 元
素的数
组
{0, 1, 2, 3}, // 0 行的初始化列表
{4, 5, 6, 7}, // 1 行的初始化列表
{8, 9, 10, 11} // 2 行的初始化列表
};
int (*p)[4];
for (p = ia; p != ia + 3; ++p)
for (int *q = *p; q != *p + 4; ++q)
cout << *q << endl;
return 0;
}

```

## 第五章 表达式

#### 习题 5.1

在下列表达式中，加入适当的圆括号以标明其计算顺序。编

译该表达式并输出

其值，从而检查你的回答是否正确。

$12 / 3 * 4 + 5 * 15 + 24 \% 4 / 2$

【解答】

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 5.2

计算下列表达式的值，并指出哪些结果值依赖于机器？

$-30 * 3 + 21 / 5$

$-30 + 3 * 21 / 5$

$30 / 3 * 21 \% 5$

$-30 / 3 * 21 \% 4$

C++ Primer (4 版) 习题解答

91

【解答】

各表达式的值分别为-86、-18、0、-2。其中，最后一个表达式的结果值依赖于机器，因为

该表达式中除操作只有一个操作数为负数。

#### 习题 5.3

编写一个表达式判断一个 int 型数值是偶数还是奇数。

【解答】

如下表达式可以判断一个 int 型数值（假设为 ival）是偶数还是奇数：

$ival \% 2 == 0$

若 ival 是偶数，则该表达式的值为真（true），否则为假（false）。

#### 习题 5.4

定义术语“溢出”的含义，并给出导致溢出的三个表达式。

【解答】

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 5.5

解释逻辑与操作符、逻辑或操作符以及相等操作符的操作数在什么时候计算。

【解答】

逻辑与、逻辑或操作符采用称为“短路求值”（short-circuit evaluation）的求值策略，即先



计算左操作数，再计算右操作数，且只有当仅靠左操作数的值无法确定该逻辑运算的结果

时，才会计算右操作数。

C++ Primer (4 版) 习题解答

92

相等操作符的左右操作数均需进行计算。

#### 习题 5.6

解释下列 while 循环条件的行为：

```
char *cp = "Hello World";
```

```
while (cp && *cp)
```

【解答】

该 while 循环的条件为：当指针 cp 为非空指针并且 cp 所指向的字符不为空字符 null ( '\0' )

时执行循环体。即该循环可以对字符串 "Hello World" 中的字符进行逐个处理。

#### 习题 5.7

编写 while 循环条件从标准输入设备读入整型(int)数据，当读入值为 42 时循环结束。

【解答】

```
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 5.8

编写表达式判断 4 个值 a、b、c 和 d 是否满足 a 大于 b、b 大于 c 而且 c 大于 d 的条件。

【解答】

表达式如下：

```
a > b && b > c && c > d
```

#### 习题 5.9

假设有下面两个定义：

C++ Primer (4 版) 习题解答

93

```
unsigned long ul1 = 3, ul2 = 7;
```

下列表达式的结果是什么？

(a) ul1 & ul2 (b) ul1 && ul2

(c) ul1 | ul2 (d) ul1 || ul2

【解答】

各表达式的结果分别为 3、true、7、true。

#### 习题 5.10

重写 bitset 表达式：使用下标操作符对测验结果进行置位

(置 1) 和复位(置 0)。

【解答】

```
bitset<30> bitset_quiz1;
```

```
bitset_quiz1[27] = 1;
```

```
bitset_quiz1[27] = 0;
```

#### 习题 5.11

请问每次赋值操作完成后，i 和 d 的值分别是多少？

```
int i; double d;
```

```
d = i = 3.5;
```

```
i = d = 3.5;
```

【解答】

赋值语句 d=i=3.5;完成后，i 和 d 的值均为 3。因为赋值操作具有右结合性，所以首先

将 3.5 赋给 i (此时发生隐式类型转换，将 double 型字面值 3.5 转换为 int 型值 3，赋给 i)，

然后将表达式 i=3.5 的值 (即赋值后 i 所具有的值 3) 赋给 d。

赋值语句 i=d=3.5;完成后，d 的值为 3.5，i 的值为 3。因为先将字面值 3.5 赋给 d，然后

将表达式 d=3.5 的值 (即赋值后 d 所具有的值 3.5) 赋给 i (这时也同样发生隐式类型转换)。

#### 习题 5.12

解释每个 if 条件判断产生什么结果？

C++ Primer (4 版) 习题解答

94

```
if (42 = i) // ...
```

```
if (i = 42) // ...
```

【解答】

前者发生语法错误，因为其条件表达式 42=i 是一个赋值表达式，赋值操作符的左操作数必

须为一个左值，而字面值 42 不能作为左值使用。

后者代码合法，但其条件表达式 i=42 是一个永真式 (即其逻辑值在任何情况下都为 true)，

因为该赋值表达式的值为赋值操作完成后的 i 值 (42)，而 42 为非零值，解释为逻辑值 true。

#### 习题 5.13

下列赋值操作是不合法的，为什么？怎样改正？

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

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

【解答】

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 5.14

虽然下列表达式都是合法的，但并不是程序员期望的操作，为什么？怎样修改这些表达式以使其能反映程序员的意图？

(a) if ( ptr = retrieve\_pointer() != 0 )

(b) if ( ival = 1024 )

(c) ival += ival + 1;

【解答】

对于表达式(a)，程序员的意图应该是将 retrieve\_pointer() 的值赋给 ptr，然后判断 ptr

C++ Primer ( 4 版 ) 习题解答

95

的值是否为 0，但因为操作符“=”的优先级比“!=”低，所以该表达式实际上是将

retrieve\_pointer()是否为 0 的判断结果 true 或 false 赋给 ptr，因此不是程序员期望的操作。

对于表达式(b)，程序员的意图应该是判断 ival 的值是否与 1024 相等，但误用了赋值操作符。

对于表达式(c)，程序员的意图应该是使 ival 的值增加 1，但误用了操作符“+=”。

各表达式可修改如下：

(a) if ( ( ptr = retrieve\_pointer() ) != 0 )

(b) if ( ival == 1024 )

(c) ival += 1; 或 ival++; 或 ++ival;

#### 习题 5.15

解释前自增操作和后自增操作的差别。

【解答】

前自增操作和后自增操作都使其操作数加 1，二者的差别在于：前自增操作将修改后操作数

的值作为表达式的结果值；而后自增操作将操作数原来的、未修改的值作为表达式的结果值。

#### 习题 5.16

你认为为什么 C++ 不叫作++C？

【解答】

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 5.17

如果输出 vector 内容的 while 循环使用前自增操作符，那会怎么样？

【解答】

将导致错误的结果：ivec 的第一个元素没有输出，并企图对一个多余的元素进行解引用。

C++ Primer ( 4 版 ) 习题解答

96

#### 习题 5.18

编写程序定义一个 vector 对象，其每个元素都是指向 string 类型的指针，读

取该 vector 对象，输出每个 string 的内容及其相应的长度。

【解答】

//定义一个 vector 对象，其每个元素都是指向 string 类型的指针，

//读取该 vector 对象，输出每个 string 的内容及其相应的长度

```
#include <iostream>
```

```
#include <string>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
vector<string*> spvec;
```

```
//读取 vector 对象
```

```
string str;
```

```
cout << "Enter some strings(Ctrl+Z to end)" << endl;
```

```
while (cin >> str) {
```

```
string *pstr = new string; //指向 string 对象的指针
```

```
*pstr = str;
```

```

spvec.push_back(pstr);
}
//输出每个 string 的内容及其相应的长度
vector<string*>::iterator iter = spvec.begin();
while (iter != spvec.end()) {
C++ Primer ( 4 版 ) 习题解答
97
cout << **iter << (**iter).size() << endl;
iter++;
}
//释放各个动态分配的 string 对象
iter = spvec.begin();
while (iter != spvec.end()) {
delete *iter;
iter++;
}
return 0;
}

```

#### 习题 5.19

假设 `iter` 为 `vector<string>::iterator` 类型的变量,指出下面哪些表达式是

合法的,并解释这些合法表达式的行为。

- (a) `*iter++`; (b) `(*iter)++`;  
(c) `*iter.empty()`; (d) `iter->empty()`;  
(e) `++*iter`; (f) `iter++->empty()`;

【解答】

(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 5.20

C++ Primer ( 4 版 ) 习题解答

98

编写程序提示用户输入两个数,然后报告哪个数比较小。

【解答】

可编写程序如下：

```

//提示用户输入两个数,然后报告哪个数比较小
#include <iostream>
using namespace std;
int main()
{
int val1, val2;

```

```

//提示用户输入两个数并接受输入
cout << "Enter two integers:" << endl;
cin >> val1 >> val2;
//报告哪个数比较小
cout << "The smaller one is"
<< (val1 < val2 ? val1 : val2) << endl;
return 0;
}

```

#### 习题 5.21

编写程序处理 `vector<int>` 对象的元素：将每个奇数值元素用该值的两倍替换。

【解答】

//处理 `vector<int>` 对象的元素：

//将每个奇数值元素用该值的两倍替换

```
#include <iostream>
```

```
#include <vector>
```

C++ Primer ( 4 版 ) 习题解答

99

```
using namespace std;
```

```
int main()
```

```
{
```

```
vector<int> ivec(20,1);//ivec 包含 20 个值为 1 的元素
```

//将每个奇数值元素用该值的两倍替换

```
for (vector<int>::iterator iter = ivec.begin();
```

```
iter != ivec.end(); ++iter)
```

```
*iter = (*iter % 2 == 0 ? *iter : *iter * 2);
```

```
return 0;
```

```
}
```

#### 习题 5.22

编写程序输出每种内置类型的长度。

【解答】

//输出每种内置类型的长度

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
cout << "type\t\t" << "size" << endl
```

```
<< "bool\t\t" << sizeof(bool) << endl
```

```
<< "char\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" << sizeof(wchar_t) << endl
```

C++ Primer ( 4 版 ) 习题解答

100

```
<< "short\t\t" << sizeof(short) << endl
<< "signed short\t\t" << sizeof(signed short) << endl
<< "unsigned short\t\t" << sizeof(unsigned short) <<
endl
<< "int\t\t" << sizeof(int) << endl
<< "signed int\t\t" << sizeof(signed int) << endl
<< "unsigend int\t\t" << sizeof(unsigned int) << endl
<< "long\t\t" << sizeof(long) << endl
<< "sigend long\t\t" << sizeof(signed long) << endl
<< "unsigned long\t\t" << sizeof(unsigned long) <<
endl
<< "float\t\t" << sizeof(float) << endl
<< "double\t\t" << sizeof(double) << endl
<< "long double\t\t" << sizeof(long double) << endl;
return 0;
}
```

#### 习题 5.23

预测下列程序的输出，并解释你的理由。然后运行该程序，输出的结果和你预

测的一样吗？如果不一样，为什么？

```
int x[10]; int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

#### 【解答】

在表达式 `sizeof(x)` 中，`x` 是数组名，该表达式的结果为数组 `x` 所占据的存储空

间的字节数，为 10 个 `int` 型元素所占据的字节数。

表达式 `sizeof(*x)` 的结果是指针常量 `x` 所指向的对象（数组中第一个 `int` 型元

素）所占据的存储空间的字节数。

C++ Primer (4 版) 习题解答

101

表达式 `sizeof(p)` 的结果是指针变量 `p` 所占据的存储空间的字节数。

表达式 `sizeof(*p)` 的结果是指针变量 `p` 所指向的对象（一个 `int` 型数据）所占

据的存储空间的字节数。

各种数据类型在不同的系统中所占据的字节数不一定相同，因此在不同的系统

中运行上述程序段得到的结果不一定相同。在 Microsoft Visual C++ .NET 2003

系统中，一个 `int` 型数据占据 4 个字节，一个指针型数据也占据 4 个字节，因

此运行上述程序得到的输出结果为：

10

1

#### 习题 5.24

本节的程序与 5.5 节在 `vector` 对象中添加元素的程序类似。两段程序都使用递

减的计数器生成元素的值。本程序中，我们使用了前自减操作，而 5.5 节的程

序则使用了后自减操作。解释为什么一段程序中使用前自减操作而在另一段程

序中使用后自减操作。

#### 【解答】

5.5 节的程序中必须使用后自减操作。如果使用前自减操作，则是用减 1 后的

`cnt` 值创建 `ivec` 的新元素，添加到 `ivec` 中的元素将不是 10~1，而是 9~0。

本节的程序中使用后自减操作或前自减操作均可，因为对 `cnt` 的自减操作和对

`cnt` 值的使用不是出现在同一表达式中，`cnt` 自减操作的前置或后置形式不影响

对 `cnt` 值的使用。

#### 习题 5.25

根据表 5-4 的内容，在下列表达式中添加圆括号说明其操作数分组的顺序（即

计算顺序）：

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

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

#### 【解答】

添加圆括号说明其计算顺序如下：

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

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

C++ Primer (4 版) 习题解答

102

#### 习题 5.26

习题 5.25 中的表达式的计算次序与你的意图不同，给它们加上圆括号使其以你

所希望的操作次序求解。

#### 【解答】

添加圆括号获得与上题不同的操作次序如下：

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

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

#### 习题 5.27

由于操作符优先级的问题，下列表达式编译失败。请参照表 5-4 解释原因，应

该如何改正？

```
string s = "word";  
// add an 's' to the end, if the word doesn't already end  
in 's'  
string pl = s + s[s.size() - 1] == 's' ? "" : "s";
```

【解答】

由表 5-4 可知,在语句 `string pl = s + s[s.size() - 1] == 's' ? "" : "s";`

中,赋值、加法、条件操作符三者的操作次序为:先执行“+”操作,再用表达式

式 `s + s[s.size() - 1]` 的结果参与条件操作,最后将条件操作的结果赋给 `pl`。

但表达式 `s + s[s.size() - 1]` 的结果是一个 `string` 对象,不能与字符 `'s'` 进行

相等比较,所以编译失败。

改正为: `string pl = s + (s[s.size() - 1] == 's' ? "" : "s");`。

#### 习题 5.28

除了逻辑与和逻辑或外,C++没有明确定义二元操作符的求解次序,编译器可自

由地提供最佳的实现方式。只能在“实现效率”和程序语言使用中“潜在的缺

陷”之间寻求平衡。你认为这可以接受吗?说出你的理由。

【解答】

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

C++ Primer (4 版) 习题解答

103

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 5.29

假设 `ptr` 指向类类型对象,该类拥有一个名为 `ival` 的 `int` 型数据成员, `vec` 是

保存 `int` 型元素的 `vector` 对象,而 `ival`、`jval` 和 `kval` 都是 `int` 型变量。请解

释下列表达式的行为,并指出哪些(如果有的话)可能是不正确的,为什么?

如何改正?

(a) `ptr->ival != 0` (b) `ival != jval < kval`

(c) `ptr != 0 && *ptr++` (d) `ival++ && ival`

(e) `vec[ival++] <= vec[ival]`

【解答】

表达式的行为如下:

(a) 判断 `ptr` 所指向的对象的 `ival` 成员是否不等于 0。

(b) 判断 `ival` 是否不等于“`jval` 是否小于 `kval`”的判断结果,即判断 `ival` 是

否不等于 `true` (1) 或 `false` (0)。

(c) 判断 `ptr` 是否不等于 0。如果 `ptr` 不等于 0,则求解 `&&` 操作的右操作数,即,

`ptr` 加 1,且判断 `ptr` 原来所指向的对象是否为 0。

(d) 判断 `ival` 及 `ival+1` 是否为 `true` (非 0 值)(注意,如果 `ival` 为 `false`,

则无需继续判断 `ival+1`)。

(e) 判断 `vec[ival]` 是否小于或等于 `vec[ival+1]`。

其中,(d)和(e)可能不正确,因为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。

可改正如下:

(d) `ival && ival + 1`

(e) `vec[ival] <= vec[ival + 1]`

#### 习题 5.30

下列语句哪些(如果有的话)是非法的或错误的?

C++ Primer (4 版) 习题解答

104

(a) `vector<string> svec(10);`

(b) `vector<string> *pvec1 = new vector<string>(10);`

(c) `vector<string> **pvec2 = new vector<string>[10];`

(d) `vector<string> *pv1 = &svec;`

(e) `vector<string> *pv2 = pvec1;`

(f) `delete svec;`

(g) `delete pvec1;`

(h) `delete [] pvec2;`

(i) `delete pv1;`

(j) `delete pv2;`

【解答】

错误的有(c)和(f)。

(c)的错误在于:`pvec2` 是指向元素类型为 `string` 的 `vector` 对象的指针的指

针(即 `pvec2` 的类型为 `vector<string> **`),而 `new` 操作返回的是一个指向元

素类型为 `string` 的 `vector` 对象的指针,不能用于初始化 `pvec2`。

(f)的错误在于:`svec` 是一个 `vector` 对象,不是指针,不能

对它进行 delete 操作。

### 习题 5.31

根据 5.12.2 节的变量定义，解释在计算下列表达式的过程中发生了什么类型转换？

- (a) if (fval)
- (b) dval = fval + ival;
- (c) dval + ival + cval;

记住，你可能需要考虑操作符的结合性，以便在表达式含有多个操作符的情况

下确定答案。

【解答】

C++ Primer (4 版) 习题解答  
105

- (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。

### 习题 5.32

给定下列定义：

```
char cval; int ival; unsigned int ui;
float fval; double dval;
```

指出可能发生的（如果有的话）隐式类型转换：

- (a) cval = 'a' + 3; (b) fval = ui - ival \* 1.0;
- (c) dval = ui \* fval; (d) cval = ival + fval + dval;

【解答】

- (a) 'a' 首先提升为 int 类型，再将 'a' + 3 的结果值转换为 char 型，赋给 cval。
- (b) ival 转换为 double 型与 1.0 相乘，ui 转换为 double 型再减去 ival \* 1.0 的结果值，减操作的结果转换为 float 型，赋给 fval。
- (c) ui 转换为 float 型与 fval 相乘，结果转换为 double 型，赋给 dval。
- (d) ival 转换为 float 型与 fval 相加，结果转换为 double 型，再与 dval 相加，结果转换为 char 型，赋给 cval。

### 习题 5.33

给定下列定义：

```
int ival; double dval;
```

```
const string *ps; char *pc; void *pv;
```

用命名的强制类型转换符号重写下列语句：

(a) pv = (void\*)ps; (b) ival = int(\*pc);

C++ Primer (4 版) 习题解答

106

(c) pv = &dval; (d) pc = (char\*) pv;

【解答】

- (a) pv = static\_cast<void\*> (const\_cast<string\*> (ps));
- (b) ival = static\_cast<int> (\*pc);
- (c) pv = static\_cast<void\*> (&dval);
- (d) pc = static\_cast<char\*> (pv);\_

## 第六章 语句

## 第七章 函数

## 第八章 标准 IO 库

8.1 假设 os 是一个 ofstream 对象，下面程序做了什么？

```
os << "Goodbye!" << endl;
```

如果 os 是 ostream 对象呢？或者，os 是 ifstream 对象呢？

答：第一个，向文件中写入“Goodbye”，第二个向 string 对象中写入“Goodbye”，第三个，如果 os 是一个 ifstream 对象，则错误，因为 ifstream 类中没有定义操作符 <<。

8.2 下面的声明是错误的，指出其错误并改正之：  
ostream print(ostream os);

答：标准库类型不允许做复制或赋值操作。形参或返回类型不能为流类型，所以上句代码错误，因为它把流类型的对象当做了形参。应改为传递指向该对象的指针或引用：

```
ostream &print(ostream &os);
```

8.3 编写一个函数，其唯一的形参和返回值都是 istream& 类型。该函数应一直读取流直到到达文件的结束符为止，还应将读到的内容输出到标准输出中。最后，重设流使其有效，并返回该流。

答：

```
// 定义控制台应用程序的入口点。
```





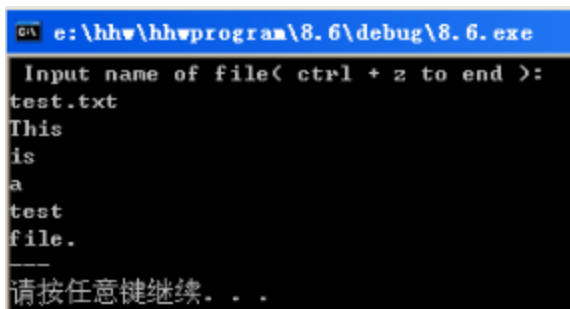
```

        if ( is.fail() )           // bad input
        {
            cerr << " bad data, try again.";
            is.clear();           // reset the
stream
            is.setstate(istream::eofbit);
// 结 束?死 循环;
            continue;
        }
        // process input
        cout << ival << endl;
    }
    is.clear(); // 将?in中D的?所 有?D状?态-?
值? 都?设 为a有?D效?i状?态-?
    return is;
}

int _tmain(int argc, _TCHAR* argv[])
{
    string fName;
    cout << " Input name of file( ctrl + z to
end ): \n";
    cin >> fName;
    ifstream readfile;
    readfile.open(fName.c_str()); // open the file
    if (!readfile)
    {
        cerr << " error: cannot open the input
file:" << fName << endl;
        return -1;
    }
    f(readfile);

    system("pause");
    return 0;
}

```



8.7 本节所编写的两个程序，在打开 vector 容器中

存放的任何文件失败时，使用 break 跳出 while 循环。重写这两个循环，如果文件无法打开，则输出警告信息，然后从 vector 中获取下一个文件名继续。

```

ifstream input;
vector<string>::const_iterator it = files.begin();
while ( it != files.end() )
{
    input.open( it->c_str() );
    if ( !input )
    {
        cerr << "iã error: can not open input file:
“iã << *it << endl;
        input.clear();
        ++it; // 获?取 下?一 个?文?件t
        continue; // 继 续?处ã理 下?一 个?文?件t
    }
    while ( input >> s )
        process(s);
    input.close();
    input.clear();
    ++it;
}

```

8.8 上一个习题的程序可以不用 continue 语句实现。分别使用或不使用 continue 语句编写该程序。

**不使用 continue 语句:**

```

ifstream input;
vector<string>::const_iterator it = files.begin();
while ( it != files.end() )
{
    input.open( it->c_str() );
    if ( !input )
    {
        cerr << "iã error: can not open input file:
“iã << *it << endl;
        input.clear();
        ++it; // 获?取 下?一 个?文?件t
    }
    else
    {
        while ( input >> s )
            process(s);
        input.close();
    }
}

```

```

        input.clear();
        ++it;
    }
}

```

8.9 编写函数打开文件用于输入，将文件内容读入 string 类型的 vector 容器，每一行存储为该容器对象的一个元素。

// 8.9.cpp : 定义控制台应用程序的入口点  
//

```
#include "stdafx.h"
```

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    string s;
    ifstream input("test.txt"); // open the file
    //cout << " Input files name:\n\t";
    if ( !input ) // fail to open
    {
        cerr << " error: can not open input file:
" << endl;
        return -1;
    }

```

// 打开后，将文件内容读入string类型的vector容器，每一行存储为  
// 该容器对象的一个元素。

```

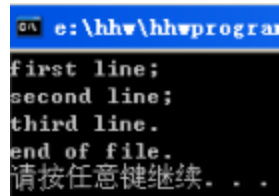
vector<string> fileWord;
while ( getline(input, s) )
{
    fileWord.push_back(s);
    //input.clear();
}
input.close(); // close the file
// 输出出来
vector<string>::const_iterator it =
fileWord.begin();

```

```

while ( it != fileWord.end() )
{
    cout << *it << endl;
    ++it; // 获取下一个文件
}
system("pause");
return 0;
}

```



8.10 重写上面的程序，把文件中的每个单词存储为容器的一个元素。

```

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    string s;
    ifstream input("test.txt"); // open the file
    //cout << " Input files name:\n\t";
    if ( !input ) // fail to open
    {
        cerr << " error: can not open input file:
" << endl;
        return -1;
    }

```

// 打开后，将文件内容读入string类型的vector容器，每一行存储为

// 该容器对象的一个元素。

```

vector<string> fileWord;
while ( input >> s )
{
    fileWord.push_back(s);
    //input.clear();
}

```

// 输出出来

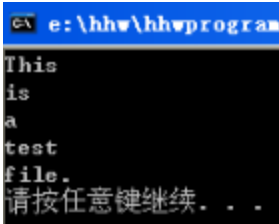
```
vector<string>::const_iterator it =
```



```

fileWord.begin();
while ( it != fileWord.end() )
{
    cout << *it << endl;
    ++it; // 获取下一个文件
}
system("pause");
return 0;
}

```



8.11 对于 `open_file` 函数,请解释为什么在调用 `open` 前先调用 `clear` 函数。如果忽略这个函数调用,会出现什么问题?如果在 `open` 后面调用 `clear` 函数,又会怎样?

此时不清楚通过形参 `in` 传进来的流对象的当前状态,所以在调用 `open` 前,先调用 `clear` 函数,将 `in` 置为有效状态。如果忽略,则 `in` 的原来的状态不会被清除,当用它打开另一个文件后,有可能导致对另一个文件的操作出现问题。

在 `open` 后调用 `clear` 函数, `in` 都会被置为有效状态,从而为后续的文件操作带来问题。

8.12 对于 `open_file` 函数,请解释如果程序执行 `close` 函数失败,会产生什么结果?

会不能打开给定文件,因为 `open` 函数会检查流是否已经打开,如果打开,则设置内部状态指示发生了错误,接下来使用 `in` 的任何尝试都会失败。

8.13 编写类似 `open_file` 的程序打开文件用于输出。

```

// 8.13. cpp : 定义控制台应用程序的入口点
//
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

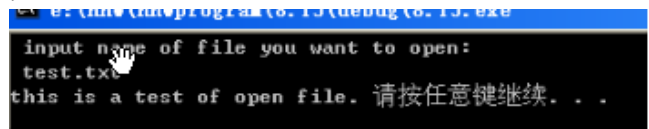
int _tmain(int argc, _TCHAR* argv[])
{
    cout << "input name of file you want to open:\n

```

```

";
    string fName;
    cin >> fName;
    fstream fOpen;
    fOpen.clear();
    fOpen.open(fName.c_str()); // open the file
    if ( !fOpen )
    {
        cerr << " cannot open the file given." <<
endl;
        return -1;
    }
    string s;
    while ( fOpen >> s )
    {
        cout << s << " ";
    }
    fOpen.close();
    system("pause");
    return 0;
}

```



8.14 使用 `open_file` 函数以及 8.2 节第一个习题编写的程序,打开给定文件并读取内容。

// 8.14. cpp : 定义控制台应用程序的入口点  
//

```

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
istream & f( istream & in )
{
    string ival;
    while
        ( in >> ival, !in.eof()) // 遇到文件结束
符之前一直读入数据
    {
        if(in.bad()) // input stream is corrupted;
bail out, 流是否已被破坏
        throw
runtime_error(

```

```

        "IO stream corrupted"
    );
    if ( in.fail() ) // bad input
    {
        cerr <<
            " bad date, try again:";
        in.clear(); // reset the stream
        in.setstate(istream::eofbit); // 结束
死循环
        continue;
    }
    // process input
    cout << ival << endl;
}
in.clear();
// 将in中的所有状态值都设为有效状态
return in;
}
int openFile( string fName )
{
    fstream fOpen;
    fOpen.clear();
    fOpen.open(fName.c_str()); // open the file

    if ( !fOpen )
    {
        cerr << " cannot open the file given." <<
endl;
        return -1;
    }
    string s;
    while ( fOpen >> s )
    {
        f( fOpen ); // 调用f()函数检测从文件中
读入数据
        cout << s << " ";
    }
    fOpen.close();
    return 1;
}

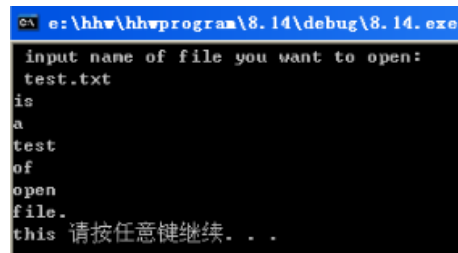
int _tmain(int argc, _TCHAR* argv[])
{
    cout << " input name of file you want to open:\n

```

```

";
    string fName;
    cin >> fName;
    openFile( fName );
    system("pause");
    return 0;
}

```



8.15 使用 8.2 节第一个习题编写的函数输出 istream 对象的内容。

// 8.15.cpp : 定义控制台应用程序的入口点

```

//
#include "stdafx.h"
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
istream & f( istream & in )
{
    string val;
    while
        ( in >> val, !in.eof() )
        // 遇到文件结束符之前一直读入数据
    {
        if(in.bad() // input stream is
corrupted; bail out, 流是否已被破坏
            throw runtime_error("IO stream
corrupted");
        if
            ( in.fail() ) // bad input
        {
            cerr << " bad date, try again:";
            in.clear(); // reset the stream
            in.setstate(istream::eofbit); //
结束死循环
            continue;
        }
        // process input

```

```

        cout << val << " ";
    }
    in.clear(); //将in中的所有状态值都设为有效状态
    return in;
}
int _tmain(int argc, _TCHAR* argv[])
{
    int val1 = 512, val2 = 1024;
    ostringstream format_message;
    format_message << " val1: " << val1 << " val2: " << val2 << "\n";
    istringstream
    input_istring( format_message.str() );
    f( input_istring );

    system("pause");
    return 0;
}

```



8.16 编写程序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。

// 8.16.cpp : 定义控制台应用程序的入口点

```

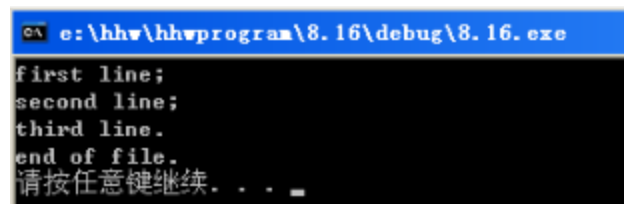
//
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    string line, word;
    ifstream input("test.txt"); // open the file
    if ( !input ) // fail to open
    {
        cerr << " error: can not open input file: " << endl;
        return -1;
    }
}

```

```

    }
    // 打开后，将文件内容读入string类型的vector容器，每一行存储为
    // 该容器对象的一个元素。
    vector<string> fileWord;
    while
        ( getline(input, line) )
    {
        fileWord.push_back(line);
        //input.clear();
    }
    input.close(); // close the file 存储完毕
    // 使用istringstream从vector里以每次读一个单词的形式读取所存储的行
    vector<string>::const_iterator it =
    fileWord.begin();
    while
        ( it != fileWord.end() )
    {
        istringstream divWord(*it);
        // 将每行分解为每个单词输出出来
        while
            ( divWord >> word )
        {
            cout << word << " ";
        }
        cout << endl;
        ++it;
        // 获取下一行
    }
    system("pause");
    return 0;
}

```



## 第九章 顺序容器

1.解释下列初始化，指出哪些是错误的，为什么？

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
```

```
string sa[6] = {
    "Fort Sunter", "Manassas", "Perryville",
    "Vicksburg", "Meridian", "Chancellorsville" };
```

(a) `vector<string> svec( sa, sa+6 );`

(b) `list<int> ilist( ia + 4, ia + 6 );`

(c) `vector<int> ivec( ia, ia + 8 );`

(d) `list<string> slist ( sa + 6, sa );`

(c) 错误，初始化迭代器的终止指针访问数组越界。

(d) 错误，初始化容器的迭代器起始点和终止点指定错误，顺序反了。

2. 创建和初始化一个 `vector` 对象有 4 种方式，为每种方式提供一个例子，并解释每个例子生成的 `vector` 对象包含什么值。

```
int ia[3] = { 1, 2, 3 };
```

(1) `vector<int> ivec1( 3 );` // 默认初始化，内容为 3 个 0

(2) `vector<int> ivec2( ia, ia+3 );` // 把数组 `ia` 里的值复制到 `ivec2` 中

(3) `vector<int> ivec3 ( ivec2 );` // 用 `ivec2` 来初始化 `ivec3`

(4) `vector<int> ivec4 ( 3, 6 );` // 将 `ivec4` 初始化为 3 个 6

3. 解释复制容器对象的构造函数和使用两个迭代器的构造函数之间的差别。

前者的构造函数将一个对象的全部元素复制到另一个容器对象里，而且要求两个对象的类型和元素的类型都相同；

后者可以将一个容器初始化为另一个容器的子序列，而且不要求两个容器的类型是同类型的。

4. 定义一个 `list` 对象来存储 `deque` 对象，该对象存放 `int` 类型的元素。

```
list< deque<int> > ilist;
```

5. 为什么我们不可以使用容器来存储 `iostream` 对象？

因为 `iostream` 对象不支持复制和赋值操作。

6. 假设有一个名为 `Foo` 的类，这个类没有定义默认构造函数，但提供了需要一个 `int` 型参数的构造函数

数。定义一个存放 `Foo` 的 `list` 对象，该对象有 10 个元素。

```
list<Foo> flist( 10, 1 );
```

7. 下面的程序错在哪里？如何改正？

```
list<int> lst1;
```

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

```
while ( iter1 < iter2 ) /* ... */
```

错在 `while` 循环里的条件表达式中使用了 `<` 操作符，因为 `list` 容器的迭代器不支持关系操作符，可改为 `iter1 != iter2`

8. 假设 `vec_iter` 绑定到 `vector` 对象的一个元素，该 `vector` 对象存放 `string` 类型的元素，请问下面的语句实现什么功能？

```
it ( vec_iter->empty() ) //...
```

判断 `vec_iter` 所指向的 `vector` 元素是否为空字符串。

9. 编写一个循环将 `list` 容器的元素逆序输出。

```
list<int> ilist( 10, 1 );
```

```
list<int>::iterator it = ilist. end();
```

```
--it;
```

```
for ( ; it != ilist.begin(); --it )
```

```
    cout << *it << endl;
```

10. 下列迭代器的用法哪些是错误的？

```
const vector< int > ivec ( 10 );
```

```
vector< string > svec ( 10 );
```

```
list< int > ilist( 10 );
```

(a) `vector<int>::iterator it = ivec.begin();`

(b) `list<int>::iterator it = ilist.begin() + 2;`

(c) `vector<string>::iterator it = &svec[0];`

(d) `for ( vector<string>::iterator it = svec.begin(); it != 0; ++it ) //...`

(b) 错误，`list` 的迭代器不支持算术运算。

(a) 错，`ivec.begin()` 返回的是 `const vector<int>` 的迭代器，不能用来初始化 `vector<int>` 的迭代器。

- (c) 错误，迭代器不支持用&操作符来初始化。  
 (d) 错误，it 与 0 进行比较会产生运行时错误。

11. 要标记出有效的迭代器范围，迭代器需满足什么约束？

first 和 last 须指向同一个容器中的元素或超出末端的下一位置。last 不能位于 first 之前。

12. 编写一个函数，其形参是一对迭代器和一个 int 型数值，实现在迭代器标记的范围内寻找该 int 型数值的功能，并返回一个 bool 结果，以指明是否找到指定数据。

// 11.15\_9.12\_iterator\_Function.cpp：定义控制台应用程序的入口点。

//

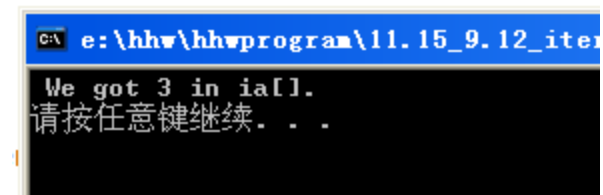
```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;
```

```
typedef vector<int>::iterator iter;
bool findTheInt( iter first, iter last, int x )
{
    assert( first <= last );
    if ( first > last )
        cout << " iterator error." << endl;
    for ( iter it = first; it != last; ++it )
    {
        if ( *it == x )
            return true;
    }
    return false;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 0, 1, 2, 3, 4, 5 };
    vector<int> ivec( ia, ia+5 );
    bool find3 = false;
    find3 = findTheInt( ivec.begin(), ivec.end(), 3 );
```

```
if ( find3 )
{

    cout << " We got 3 in ia[]." << endl;
}
else
    cout << " There is not 3 in ia[]." << endl;
system("pause");
return 0;
}
```



13. 重写程序，查找元素的值，并返回指向找到的元素的迭代器。确保程序在要寻找的元素不存在时也能正确工作。

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;
```

```
typedef vector<int>::iterator iter;
iter getTheIterOfX( iter first, iter last, int x )
{
    assert ( first <= last );

    for ( iter it = first; it != last; ++it )
    {
        if ( *it == x )
        {
            return it;
        }
    }
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 0, 1, 2, 3, 4, 5 };
```

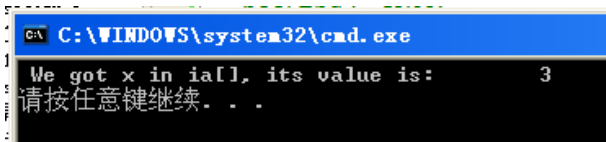
```

vector<int> ivec( ia, ia+5 );

iter it = getTheIterOfX( ivec.begin(), ivec.end(),
3 );
if ( (*it) )
{
    cout << " We got x in ia[], its value is:\t" <<
*it << endl;
}

system("pause");
return 0;
}

```



14.使用迭代器编写程序，从标准输入设备读入若干string 对象，并将它们存储在一个vector 对象中，然后输出该vector 对象中的所有元素。

// 11.15\_9.14\_vector\_string.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some strings ( ctrl+z to end ):" <<
endl;
    vector<string> strVec;
    string str;
    while ( cin >> str )
    {
        strVec.push_back( str );
    }

    cout << " All the member of the vector<string>
strVec are:\n";

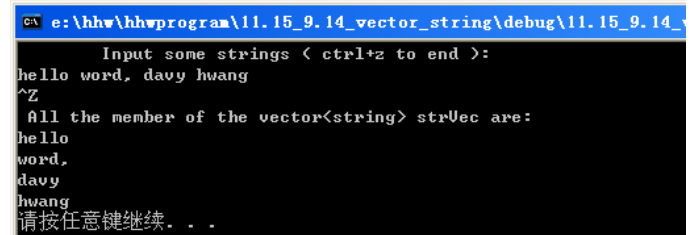
```

```

for ( vector<string>::iterator it = strVec.begin();
it != strVec.end(); ++it )
{
    cout << *it << endl;
}

system("pause");
return 0;
}

```



15.用 list 容器类型重写习题 9.14 得到的程序，列出改变了容器类型后要做的修改。

// 11.15\_9.15\_list\_string.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <list>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some strings ( ctrl+z to end ):" <<
endl;
    list<string> strLst;
    string str;
    while ( cin >> str )
    {
        strLst.push_back( str );
    }

    cout << " All the member of the List<string> strLst
are:\n";
    for ( list<string>::iterator it = strLst.begin(); it !=
strLst.end(); ++it )
    {
        cout << *it << endl;
    }
}

```

```

    }

    system("pause");
    return 0;
}

```

16.int 型的 vector 容器应该使用什么类型的索引?  
使用 `vector<int>::size_type` 的索引。

17.读取存放 string 对象的 list 容器时, 应该使用什么类型?

使用 `list<string>::iterator it`, 进行索引, 然后解引用 `*it`.

或者使用 `list<string>::const_iterator` 类型的迭代器。

若想实现逆序读取, 则可使用

`list<string>::reverse_iterator` 和 `list<string>::const_reverse_iterator` 迭代器。

18.编写程序将int型的list容器的所有元素复制到两个 deque 容器中。list 容器的元素如果为偶数, 则复制到一个 deque 容器中; 如果为奇数, 则复制到另一个 deque 容器里。

// 11.15\_9.18\_list\_to\_deque.cpp : 定义控制台应用程序的入口点。

//

```

#include "stdafx.h"
#include <iostream>
#include <list>
#include <deque>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{

```

```

    cout << "\tInput some int numbers ( ctrl+z to end ):" << endl;

```

```

    list<int> iLst;
    int iVal;
    while ( cin >> iVal )
    {
        iLst.push_back( iVal );
    }

```

```

    deque<int> iOddDeque, iEvenDeque;

```

```

    for ( list<int>::iterator it = iLst.begin(); it != iLst.end(); ++it )
    {

```

```

        if ( *it % 2 == 0 )
        {
            iEvenDeque.push_back( *it );
        }
        else if ( *it % 2 == 1 )
        {
            iOddDeque.push_back( *it );
        }
    }

```

```

    cout << " All the numbers inputed, the odd numbers are:" << endl;

```

```

    for ( deque<int>::iterator it = iOddDeque.begin(); it != iOddDeque.end(); ++it )
    {

```

```

        cout << *it << " ";
    }

```

```

    cout << "\n All the numbers inputed, the even numbers are:" << endl;

```

```

    for ( deque<int>::iterator it = iEvenDeque.begin(); it != iEvenDeque.end(); ++it )
    {

```

```

        cout << *it << " ";
    }

```

```

    system("pause");
    return 0;
}

```



```

e:\hhw\hhwprogram\11.15.9.18_list_to_deque\debu
Input some int numbers < ctrl+z to end >:
1 2 3 4 5 6 7 8 9
^Z
All the numbers inputed, the odd numbers are:
1 3 5 7 9
All the numbers inputed, the even numbers are:
2 4 6 8 请按任意键继续. . .

```

19.假设 iv 是一个 int 型的 vector 容器，下列程序存在什么错误？如何改正之。

```

vector<int>::iterator mid = iv.begin() + iv.end()/2;
while ( vector<int>::iterator iter != mid )
    if ( iter == some_val )

```

(1) 当执行 `it.insert` 操作之后，迭代器 `mid` 就失效了，是因为 `iv.end()` 失效了。

(2) 迭代器 `iter` 没有初始化；

(3) `if` 语句中错误，因为 `if ( *iter == some_val )`

可改为：

```

vector<int>::iterator iter = iv.begin();
while ( iter != iv.begin() + iv.end() / 2 )
{
    if ( *iter == some_val )
    {
        it.insert ( iter, 2 * some_val );
        iter += 2; // important
    }
    else
        ++iter;
}

```

20.编写程序判断一个 `vector<int>` 容器所包含的元素是否与一个 `list<int>` 容器的完全相同。

// 11.15\_9.20\_compare\_list\_and\_vector.cpp : 定义控制台应用程序的入口点。

//

```

#include "stdafx.h"
#include <iostream>
#include <list>
#include <vector>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{

```

```

    cout << "\tInput some int numbers to a list ( 999 to end ):" << endl;

```

```

    list<int> iLst;
    int iVal;
    while ( cin >> iVal )
    {
        if ( iVal == 999 ) break;
        iLst.push_back( iVal );
    }

```

```

    cout << "\nInput some int numbers to a vector( 888 to end ):" << endl;

```

```

    vector<int> iVec;
    int i;
    while ( cin >> i )
    {
        if ( i == 888 ) break;
        iVec.push_back( i );
    }

```

```

    bool bSame = true;

```

```

    if ( iLst.size() != iVec.size() )
    {
        bSame = false;
    }
    else
    {
        list<int>::iterator it_l = iLst.begin();
        vector<int>::iterator it_v = iVec.begin();
        while ( it_l != iLst.end() )
        {
            if ( *it_l != *it_v )
            {
                bSame = false;
                break;
            }
            it_l++;
            it_v++;
        }
    }

```

```

    if ( bSame )
    {

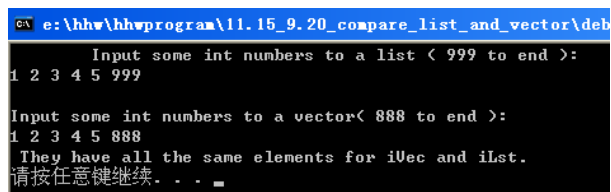
```

```

        cout << " They have all the same elements
for iVec and iLst. " << endl;
    }
    else
        cout << " They are not same of iVec and iLst.
" << endl;

    system("pause");
    return 0;
}

```



21. 假设 c1 和 c2 都是容器，下列用法给 c1 和 c2 的元素类型带来什么约束？

if ( c1 < c2 )

（如果有的话）对 c1 和 c2 的约束又是什么？

c1 和 c2 的元素类型的约束为：类型必须相同，且都必须支持 < 操作符。

22. 已知容器 vec 存放了 25 个元素，那么 vec.resize(100) 操作实现了什么功能？若再做操作 vec.resize(10)，实现的功能又是什么？

将 vec 的大小改为 100，且把新添加的元素值初始化。

又将 vec 的后 90 个元素删除，将 vec 的大小改为 10

23. 使用只带有一个长度参数的 resize 操作对元素类型有什么要求？

对容器里存放的数据类型必须支持值默认初始化。

24. 编写程序获取 vector 容器的第一个元素。分别使用下标操作符，front 函数以及 begin 函数实现该功能，并提供空的 vector 容器测试你的程序。

// 11.15\_9.24\_vector\_getTheFirstElement.cpp：定义控制台应用程序的入口点。

//

```

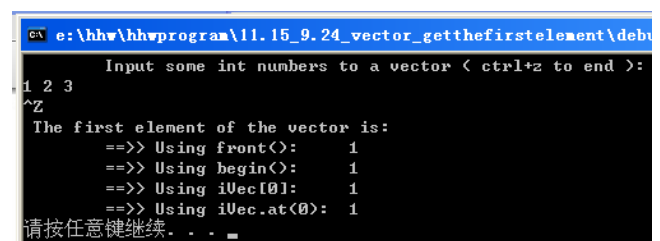
#include "stdafx.h"
#include <iostream>
#include <vector>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some int numbers to a vector
( ctrl+z to end ):" << endl;
    vector<int> iVec;
    int i;
    while ( cin >> i )
    {
        iVec.push_back( i );
    }

    if ( !iVec.empty() )
    {
        cout << " The first element of the vector is:";
        cout << "\n\t==>> Using front():\t" <<
iVec.front();
        cout << "\n\t==>> Using begin():\t" <<
*iVec.begin();
        cout << "\n\t==>> Using iVec[0]:\t" <<
iVec[0];
        cout << "\n\t==>> Using iVec.at(0):\t" <<
iVec.at(0) << endl;
    }

    system("pause");
    return 0;
}

```



当用空的 vector 时，会产生运行时错误，或者抛出异常(使用 vec.at(0) 时)。

25.需要删除一段元素时,如果 val1 和 val2 相等,那么程序会发生什么事情?如果 val1 和 val2 中的一个不存在,或两个都不存在,程序又会怎么样?

相等,则不会删除任何元素;

若有一个不存在,则会发生运行时错误;

26.假设有如下 ia 的定义,将 ia 复制到一个 vector 容器和一个 list 容器中。使用单个迭代器参数版本的 erase 函数将 list 容器中的奇数值元素删除掉,然后将 vector 容器中的偶数值元素删除掉。

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

// 11.15\_9.26\_ia\_To\_vector\_and\_list\_Erase.cpp: 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
```

```
#include <iostream>
```

```
#include <list>
```

```
#include <vector>
```

```
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
```

```
{
```

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

```
    vector<int> iVec( ia, ia+11 );
```

```
    list<int> iLst( ia, ia+11 );
```

```
    cout << " Before erase, the elements of iVec are:" << endl;
```

```
    for ( vector<int>::iterator it = iVec.begin(); it != iVec.end(); ++it )
```

```
        cout << *it << " ";
```

```
    cout << "\n Before erase, the elements of iLst are:" << endl;
```

```
    for ( list<int>::iterator it = iLst.begin(); it != iLst.end(); ++it )
```

```
        cout << *it << " ";
```

```
    // 2 erase
```

```
    for ( vector<int>::iterator iter = iVec.begin(); iter != iVec.end(); ++iter )
```

```
{
    if ( *iter % 2 == 0 )
    {
        iter = iVec.erase( iter );
    }
}

for ( list<int>::iterator Lit = iLst.begin(); Lit != iLst.end(); ++Lit )
{
    if ( *Lit % 2 == 1 )
    {
        Lit = iLst.erase( Lit );
        --Lit;
    }
}
```

```
// show
```

```
cout << "\n After erase, the elements of iVec are:" << endl;
```

```
for ( vector<int>::iterator it = iVec.begin(); it != iVec.end(); ++it )
```

```
    cout << *it << " ";
```

```
cout << "\n After erase, the elements of iLst are:" << endl;
```

```
for ( list<int>::iterator it = iLst.begin(); it != iLst.end(); ++it )
```

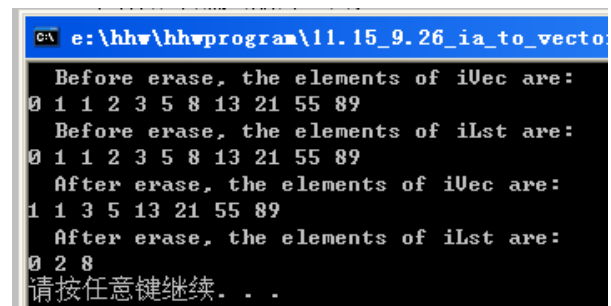
```
    cout << *it << " ";
```

```
cout << endl;
```

```
system("pause");
```

```
return 0;
```

```
}
```



```
e:\hhw\hhwprogram\11.15_9.26_ia_to_vector
Before erase, the elements of iVec are:
0 1 1 2 3 5 8 13 21 55 89
Before erase, the elements of iLst are:
0 1 1 2 3 5 8 13 21 55 89
After erase, the elements of iVec are:
1 1 3 5 13 21 55 89
After erase, the elements of iLst are:
0 2 8
请按任意键继续. . .
```

27.编写程序处理一个 string 类型的 list 容器。在该

容器中寻找一个特殊值，如果找到，则将它删除掉。  
用 deque 容器重写上述程序。

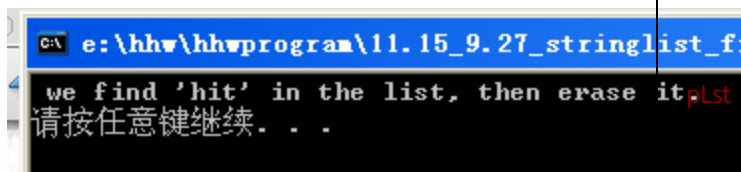
// 11.15\_9.27\_StringList\_Find\_Earse.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <list>
#include <string>
#include <algorithm>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string strSerVal("hit");
    string strArr[] = { "Hello", "world", "hit" };
    list<string> strLst( strArr, strArr + 3 );
    list<string>::iterator Lit = find( strLst.begin(),
strLst.end(), strSerVal );
    if ( Lit != strLst.end() )
    {
        cout << " we find 'hit' in the list, then erase
it." << endl;
        strLst.erase( Lit );
    }

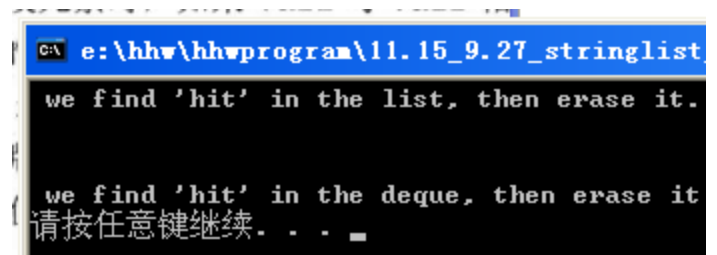
    system("pause");
    return 0;
}
```



用 deque 重写以上程序:

```
string strSerVal("hit");
string strArr[] = { "Hello", "world", "hit" };
deque<string> strDeq( strArr, strArr + 3 );
deque<string>::iterator Dit = find( strDeq.begin(),
strDeq.end(), strSerVal );
if ( Dit != strDeq.end() )
{
    cout << "\n\n we find 'hit' in the deque,
```

```
then erase it." << endl;
    strDeq.erase( Dit );
}
```



28.编写一个程序将一个 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。

// 11.15\_9.28\_assignListToVector.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <list>
#include <vector>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    char *c_arr[] = { "one", "two", "three" };
    list<char*> pLst;
    pLst.assign( c_arr, c_arr+3 );
    cout << "\tAll the members in the list<char*>
pLst are:\n";
    for ( list<char*>::iterator it = pLst.begin(); it !=
pLst.end(); ++it )
    {
        cout << *it << " ";
    }

    vector<string> strVec;
    strVec.assign( pLst.begin(), pLst.end() );
    cout << "\n\tAfter assignment from list, the
vector<string> strVec are:\n";
    for ( vector<string>::iterator it = strVec.begin();
```

```

it != strVec.end(); ++it )
{
    cout << *it << " ";
}
cout << endl;

system("pause");
return 0;
}

```

```

C:\e:\hhw\hhwprogram\11.15_9.28_assignlisttovector\debug\11.15_9.28
All the members in the list<char*> pList are:
one two three
After assignment from list, the vector<string> strVec are:
one two three
请按任意键继续. . .

```

29.解释 `vector` 的容量和长度之间的区别。为什么在连续存储元素的容器中需要支持“容量”的概念？而非连续的容器，如 `list`，则不需要。

容量 `capacity`，是指容器在必须分配新的存储空间之前可以存储的元素的总数。而长度是指容器当前拥有的元素的个数。

对于连续存储的容器来说，容器中的元素是连续存储的，当往容器中添加一个元素时，如果容器中已经没有空间容纳新的元素，则为了保持元素的连续存储，必须重新分配存储空间，用来存放原来的元素以及新添加的元素：首先将存放在旧存储空间中的元素复制到新的存储空间里，然后插入新的元素，最后撤销旧的存储空间。如果在每次添加新元素时，都要这样分配和撤销内存空间，其性能将会慢得让人无法接受。为了提高性能，连续存储元素的容器实际分配的容量要比当前所需的空间多一些，预留了一些额外的存储区，用于存放新添加的元素，使得不必为每个新的元素重新分配容器。所以，在连续存储元素的容器中需要支持“容量”的概念。而对于不连续存储的容器，不存在这样的内存分配问题。例如当在 `list` 容器中添加一个元素，标准库只需创建一个新元素，然后将该新元素连接到已经存在的链表中，不需要重新分配存储空间，也不必复制任何已存在的元素。所以这类容器不需要支持“容量”的概念。

30.编写程序研究标准库为 `vector` 对象提供的内存分配策略。

// 11.15\_9.30\_vector\_capacity\_and\_reserve.cpp：定义控制台应用程序的入口点。

```

//

#include "stdafx.h"
#include <iostream>
#include <vector>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    vector<int> ivec;
    cout << " ivec.size() : " << ivec.size() << endl
        << " ivec.capacity() : " << ivec.capacity() <<
endl;

    // add new elements:
    for ( size_t ix = 1; ix != 11; ++ix )
    {
        ivec.push_back( ix );
        cout << " ivec.size() : " << ivec.size() << endl
            << " ivec.capacity() : " << ivec.capacity() <<
endl;
    }

    // use up the current capacity of the vector
    while ( ivec.size() != ivec.capacity() )
    {
        ivec.push_back( 0 );
    }

    // add one more
    ivec.push_back(0);
    // show the current capacity of the vector
    cout << " ivec.size() : " << ivec.size() << endl
        << " ivec.capacity() : " << ivec.capacity() <<
endl;

    // resize the capacity of the vector
    ivec.reserve( 100 );
    // show the current capacity of the vector
    cout << " ivec.size() : " << ivec.size() << endl
        << " ivec.capacity() : " << ivec.capacity() <<
endl;

    // use up the current capacity of the vector

```

```

while ( ivec.size() != ivec.capacity() )
{
    ivec.push_back( 0 );
}

// add one more
ivec.push_back(0);
// show the current capacity of the vector
cout << " ivec.size() : " << ivec.size() << endl
    << " ivec.capacity() : " << ivec.capacity() <<
endl;

system("pause");
return 0;
}

```

```

e:\hhw\hhwprogram\11.1
ivec.size() : 0
ivec.capacity() : 0
ivec.size() : 1
ivec.capacity() : 1
ivec.size() : 2
ivec.capacity() : 2
ivec.size() : 3
ivec.capacity() : 3
ivec.size() : 4
ivec.capacity() : 4
ivec.size() : 5
ivec.capacity() : 6
ivec.size() : 6
ivec.capacity() : 6
ivec.size() : 7
ivec.capacity() : 9
ivec.size() : 8
ivec.capacity() : 9
ivec.size() : 9
ivec.capacity() : 9
ivec.size() : 10
ivec.capacity() : 13
ivec.size() : 14
ivec.capacity() : 19
ivec.size() : 14
ivec.capacity() : 100
ivec.size() : 101
ivec.capacity() : 150
请按任意键继续

```

31. 容器的容量可以比其长度小吗？在初始时或插

入元素后，容量是否恰好等于所需要的长度？为什么？

不能。

在初始时，或插入元素后，容量不会恰好等于所需要的长度，一般会大于长度，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，从而避免了重新分配内存、复制元素、释放内存等操作，提高性能。

32. 解释下面程序实现的功能：

```

vector<string> svec;
svec.reserve( 1024 );
string text_word;
while ( cin >> text_word )
    svec.push_back( text_word );
svec.resize( svec.size() + svec.size()/2 );

```

如果该程序读入了 256 个单词，在调整大小后，该容器的容量可能是多少？如果读入 512，或 1000，或 1048 个单词呢？

功能：将 `svec` 的 `size` 设定为 1024，然后从标准输入设备读入一系列单词，最后将该 `vector` 对象的大小调整为所读入单词个数的 1.5 倍。

当读入 256 或 512 时，容器的容量不变，仍为 1024，因为容器大小没有超出已分配内存容量。调整大小只改变容器中有效元素的个数，不会改变容量。

当读入 1000 时，最后调整容量时，需要 1500 个元素的存储空间，超过了已分配的容量 1024，所以可能重新分配为 1536 的容量。

当读入 1048 时，在读入 1025 个时，容器的容量可能会增长为 1536，最后输入完 1048 个单词后，在调整大小后，需要 1572 个元素的存储空间，超过了已分配的 1536，因此再次重分配，容量再增长 0.5 倍，变为 2304。

33. 对于下列程序任务，采用哪种容器实现最合适？解释选择的理由。如果无法说明采用某种容器比另一种容器更好的原因，请解释为什么？

(a) 从一个文件中读入未知数目的单词，以生成英文句子。

(b) 读入固定数目的单词，在输入时将它们按字母顺序插入到容器中，下一章将介绍何时处理此类问题的关联容器。

(c) 读入未知数目的单词，总是在容器尾部插入新单词，从容器首部删除下一个值。

(d) 从一个文件中读入未知数目的整数。对这些整数排序，然后把它们输出到标准输出设备。

(a) 以给确定的顺序随机处理这些单词，采用 `vector` 最合适。

(b) `list` 何时，因为需要在容器的任意位置插入元素。

(c) `deque` 合适，因为总是在尾部和首部进行插入和删除的操作。

(d) 如果一边输入一边排序，则采用 `list` 合适，如果先读入所有的整数，然后排序，则用 `vector` 合适，因为进行排序最好有随机访问的能力。

34.使用迭代器将 `string` 对象中的字符都改为大写字母。

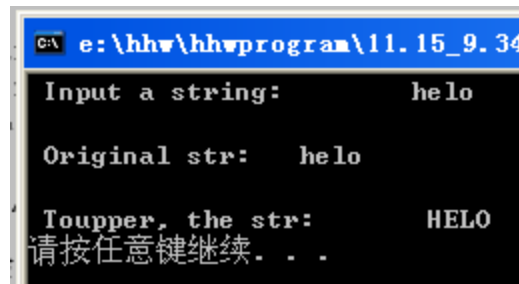
// 11.15\_9.34\_string\_toupper.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    string str;
    cout << "Input a string:\t";
    cin >> str;
    cout << "\n Original str:\t " << str << endl;
    for ( string::iterator it = str.begin(); it != str.end();
    ++it )
    {
        *it = toupper( *it );
    }

    cout << "\n Toupper, the str:\t " << str << endl;
    system("pause");
    return 0;
}
```

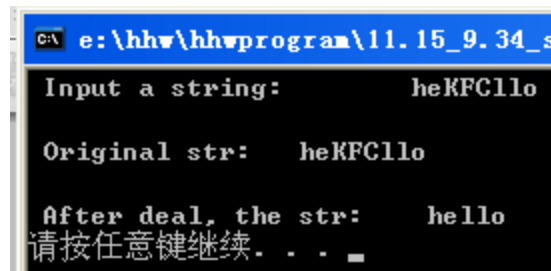


35.使用迭代器寻找和删除 `string` 对象中的所有的大写字符。

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    string str;
    cout << "Input a string:\t";
    cin >> str;
    cout << "\n Original str:\t " << str << endl;
    for ( string::iterator it = str.begin(); it != str.end();
    ++it )
    {
        if ( isupper( *it ) )
        {
            str.erase( it );
            it--;
        }
    }

    cout << "\n After deal, the str:\t " << str << endl;
    system("pause");
    return 0;
}
```



36.编写程序用 `vector<char>` 容器初始化 `string` 对象。



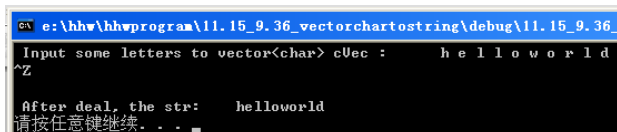
// 11.15\_9.36\_vectorCharToString.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    vector<char> cVec;
    char cVal;
    cout << "Input some letters to vector<char>
cVec :\t";
    while ( cin >> cVal )
        cVec.push_back( cVal );

    string str( cVec.begin(), cVec.end() );
    cout << "\n After deal, the str:\t " << str << endl;
    system("pause");
    return 0;
}
```



37.假设希望一次读取一个字符并写入 string 对象，而且已知需要读入至少 100 个字符，考虑应该如何提高程序的性能？

string 对象中的字符是连续存储的，为了提高性能，事先应该将对象的容量指定为至少 100 个字符的容量，以避免多次进行内存的重新分配。可使用 reserve 函数实现。

38.已知有如下 string 对象：

“ab2c3d7R4E6”

编写程序寻找该字符串中所有的数字字符，然后再寻找所有的字母字符。以两种版本编写该程序：第一个版本使用 find\_first\_of 函数，而第二个版本则使用 find\_first\_not\_of 函数。

第一个版本：

// 11.15\_9.38\_find\_fist\_of.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string strSearchVal( "ab2c4d7R4E6" );
    string numerics("0123456789");
    string::size_type pos = 0;
    // find num
    while ( ( pos = strSearchVal.find_first_of( numerics,
pos )) != string::npos )
    {
        cout << "\n Found number at index: " <<
pos << " element is: " << strSearchVal[pos] << endl;
        ++pos;
    }

    // find letters
    string
letters("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
NOPQRSTUVWXYZ");
    pos = 0;
    while ( ( pos = strSearchVal.find_first_of( letters,
pos )) != string::npos )
    {
        cout << "\n We found letter at index: " <<
pos
        << " element is: " << strSearchVal[pos]
<< endl;
        ++pos;
    }
    system("pause");
    return 0;
}
```

```

e:\hhw\hhwprogram\11.15_9.38_find_fist_of\o
Found number at index: 2 element is: 2
Found number at index: 4 element is: 4
Found number at index: 6 element is: 7
Found number at index: 8 element is: 4
Found number at index: 10 element is: 6
We found letter at index: 0 element is: a
We found letter at index: 1 element is: b
We found letter at index: 3 element is: c
We found letter at index: 5 element is: d
We found letter at index: 7 element is: R
We found letter at index: 9 element is: E
请按任意键继续. . .

```

第二个版本:

```

#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string strSearchVal( "ab2c4d7R4E6" );
    string numerics("0123456789");
    string
letters("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
NOPQRSTUVWXYZ");
    string::size_type pos = 0;
    // find num
    while ( ( pos =
strSearchVal.find_first_not_of( letters, pos )) !=
string::npos )
    {
        cout << "\n Found number at index: " <<
pos << " element is: " << strSearchVal[pos] << endl;
        ++pos;
    }

    // find letters

    pos = 0;

```

```

while ( ( pos =
strSearchVal.find_first_not_of( numerics, pos )) !=
string::npos )
{
    cout << "\n We found letter at index: " <<
pos
        << " element is: " << strSearchVal[pos]
<< endl;
    ++pos;
}
system("pause");
return 0;
}

```

```

e:\hhw\hhwprogram\11.15_9.38_find_first_r
Found number at index: 2 element is: 2
Found number at index: 4 element is: 4
Found number at index: 6 element is: 7
Found number at index: 8 element is: 4
Found number at index: 10 element is: 6
We found letter at index: 0 element is: a
We found letter at index: 1 element is: b
We found letter at index: 3 element is: c
We found letter at index: 5 element is: d
We found letter at index: 7 element is: R
We found letter at index: 9 element is: E
请按任意键继续. . .

```

39. 已知有如下 string 对象:

```

string line1 = " We were her pride of 10 she named
us: ";
string line2 = "Benjamin, Phoenix, the Prodigal";
string line3 = "and perspicacious pacific Suzanne";

```

string sentence = line1 + ' ' + line2 + ' ' + line3;  
编写程序计算 sentence 中有多少个单词, 并指出其中最长和最短单词。如果有多个最长或最短单词, 则将它们全部输出。

// 11.15\_9.39\_find\_howManyWords.cpp : 定义控制

台应用程序的入口点。

```
//

#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string separators(" :\\t\\v\\r\\n\\f");
    // find how many words
    string line1 = "We were her pride of 10 she
named us.";
    string line2 = "Benjamin, Phoenix, the Prodigal";
    string line3 = "and perspicacious pacific
Suzanne";
    string sentence = line1 + ' ' + line2 + ' ' + line3;
    string word;
    string::size_type maxLen, minLen, wordLen;
    vector<string> longestWords, shortestWords;

    cout << "\\n The sentence is :\\n" << sentence <<
endl;
    string::size_type startPos = 0, endPos = 0;
    size_t cnt = 0;
    while ( ( startPos =
sentence.find_first_not_of( separators, endPos )) !=
string::npos )
    {
        ++cnt;
        endPos = sentence.find_first_of( separators,
startPos );
        if ( endPos == string::npos )
            wordLen = sentence.size() - startPos;
        else
            wordLen = endPos - startPos;

        word.assign( sentence.begin() + startPos,
sentence.begin() + startPos + wordLen );
        startPos =
sentence.find_first_not_of( separators, endPos );
```

```
if ( cnt == 1 )
{
    maxLen = minLen = wordLen;
    longestWords.push_back( word );
    shortestWords.push_back( word );
}
else
{
    if ( wordLen > maxLen )
    {
        maxLen = wordLen;
        longestWords.clear();
        longestWords.push_back( word );
    }
    else if ( wordLen == maxLen )
    {
        longestWords.push_back( word );
    }

    if ( wordLen < minLen )
    {
        minLen = wordLen;
        shortestWords.clear();
        shortestWords.push_back( word );
    }
    else if ( wordLen == minLen )
    {
        shortestWords.push_back(word);
    }
}
}

// cout number of words
cout << "\\n\\t There are " << cnt << " words in the
sentence." << endl;
vector<string>::iterator iter;
// out the longest word
cout << "\\n\\t longest word : " << endl;
iter = longestWords.begin();
while ( iter != longestWords.end() )
{
    cout << *iter++ << endl;
}
```

```

// out the shortest word
cout << "\n\t shortest word :" << endl;
iter = shortestWords.begin();
while ( iter != shortestWords.end() )
{
    cout << *iter++ << endl;
}

system("pause");
return 0;
}

```

```

e:\hhw\hhwprogram\11.15_9.39_find_howmanywords\debug\11.15_9.39_find_...
The sentence is :
We were her pride of 10 she named us: Benjamin, Phoenix, the Prodigal and perspi
cacious pacific Suzanne

There are 17 words in the sentence.

longest word :
perspicacious

shortest word :
We
of
10
us
请按任意键继续. . .

```

40. 编写程序接受下列两个 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");
```

// 11.15\_9.40\_compare\_and\_append\_assign.cpp : 定义控制台应用程序的入口点。

//

```

#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string q1( "When lilacs last in the dooryard
bloom'd" );
    string q2( "The child is father of the man" );
    // compare q1 and q2

```

```

if ( q1.compare( q2 ) > 0 )
    cout << "\n\t==>> q1 > q2. " << endl;
else if ( q1.compare( q2 ) < 0 )
    cout << "\n\t==>> q1 < q2. " << endl;
else if ( q1.compare( q2 ) == 0 )
{
    cout << "\n\t==>> q1 == q2. " << endl;
}

// create sentence: string sentence("The child is
in the dooryard");
string sentence;
sentence.assign( q2.begin(), q2.begin() + 12 );
sentence.append( q1, 16, 16);
cout << "\n\tThen the sentence is :\n\t" <<
sentence << endl;

system("pause");
return 0;
}

```

```

e:\hhw\hhwprogram\11.15_9.40_compar
==>> q1 > q2.

Then the sentence is :
The child is in the dooryard
请按任意键继续. . .

```

41. 已知有如下 string 对象:

```
string generic1("Dear Ms Daisy:");
```

```
string generic2("MrsMsMissPeople");
```

编写程序实现下面函数:

```
string greet( string form, string lastname, string title,
string::size_type pos, int length );
```

该函数使用 replace 操作实现以下功能: 对于字符串 form, 将其中的 Daisy 替换为 lastname, 将其中的 Ms 替换为字符串 generic2 中从 pos 下标开始的 length 个字符。

例如, 下面的语句:

```
string lastName( " AnnaP" );
```

```
string salute = greet( generic1, lastName, generic2, 5,
4 );
```

将返回字符串:

```

Dear Miss AnnaP:
// 11.15_9.41_replace.cpp: 定义控制台应用程序的
入口点。
//

```

```

#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

```

```

string greet( string form, string lastname, string title,
string::size_type pos, int length )
{
    string::size_type pos_Daisy, pos_Ms;
    // replace "Dasiy" to lastname("AnnnaP")
    while ( ( pos_Daisy = form.find( "Daisy" ) ) !=
string::npos )
        form.replace( pos_Daisy, lastname.size(),
lastname );

    // replace "Ms" to "Miss"
    while ( ( pos_Ms = form.find( "Ms" ) ) !=
string::npos )
        form.replace( pos_Ms, 2, title, pos,
length );

    return form;
}

```

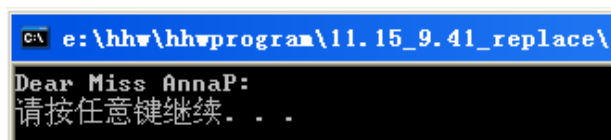
```

int _tmain(int argc, _TCHAR* argv[])
{
    string generic1("Dear Ms Daisy:");
    string generic2("MrsMsMissPeople");

    string lastName( "AnnaP");
    string salute = greet( generic1, lastName,
generic2, 5, 4 );

    cout << salute << endl;
    system("pause");
    return 0;
}

```



42.编写程序读入一系列单词，并将它们存储在 stack 对象中。

```

// 11.15_9.42_stack_deque.cpp: 定义控制台应用程序
的入口点。
//

```

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <deque>
#include <stack>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    stack<string> strStack;
    cout << "\tInput some words ( ctrl+z to
end):\n\t";
    string word;
    while ( cin >> word )
        strStack.push( word );

    // show out the elements of the stack
    cout << "\n\nThe elements of the stack are:" <<
endl;
    while ( ! strStack.empty() )
    {
        cout << strStack.top() << endl;
        strStack.pop();
    }

    system("pause");
    return 0;
}

```

```

C:\ e:\hhw\hhwprogram\11.15_9.42_stack_deque
Input some words ( ctrl+z to end ):
hello world davy hwang
^Z

The elements of the stack are:
hwang
davy
world
hello
请按任意键继续. . .

```

43.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。

// 11.15\_9.43\_stack\_application.cpp：定义控制台应用程序的入口点。

//

#include "stdafx.h"

// 11.15\_9.42\_stack\_deque.cpp：定义控制台应用程序的入口点。

//

#include "stdafx.h"

#include <iostream>

#include <string>

#include <stack>

#include <deque>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[])

{

stack<char> sExp;

string strExp;

cout << "Input a expression: ";

cin >> strExp;

// deal the sExp

string::iterator it = strExp.begin();

while ( it != strExp.end() )

```

{
    if ( *it != '(' )
        sExp.push( *it );
    else
    {
        while ( ( sExp.top() != '(' )
            & !sExp.empty() )
        {
            sExp.pop();
        }

        if ( sExp.empty() )
            cout << "It's not matched. " << endl;
        else
        {
            sExp.pop();
            sExp.push('@');
        }

        ++it;
    }

    // show out the elements of the stack
    cout << "\nThe elements of the stack are:" <<
endl;
    while ( ! sExp.empty() )
    {
        cout << sExp.top() << endl;
        sExp.pop();
    }

    system("pause");
    return 0;
}

```

## 第十章 关联容器

1.编写程序读入一些列 `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 `vector` 容器里。

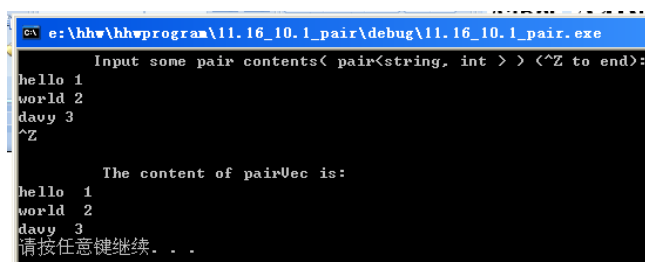
// 11.16\_10.1\_pair.cpp：定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <utility>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    string str;
    int iVal;

    cout << "\tInput some pair contents( pair<string,
int > ) (^Z to end):\n";
    vector< pair<string, int> > pairVec;
    while( cin >> str >> iVal )
        pairVec.push_back( pair< string, int > ( str,
iVal) );

    cout << "\n\t The content of pairVec is:\n" ;
    for ( vector< pair<string, int> >::iterator it =
pairVec.begin(); it != pairVec.end(); ++it )
    {
        cout << it->first << " " << it->second <<
endl;
    }

    system("pause");
    return 0;
}
```



2.在上一题中，至少可使用三种方法创建 pair 对象。编写三个版本的程序，分别采用不同的方法来创建 pair 对象。你认为哪一种方法更易于编写和理解，为什么？

```
string str;
int iVal;
```

```
vector< pair<string, int> > pairVec;
pair<string, int> newPair;
while( cin >> str >> iVal )
{
    // first method
    pairVec.push_back( pair< string, int > ( str,
iVal) );

    // the second method
    newPair = make_pair( str, iVal );
    pairVec.push_back( newPair );
}

// third method
while ( cin >> newPair.first >> newPair.second )
{
    pairVec.push_back( newPair );
}
```

第二种方法更好一些，更容易阅读和理解。因为它调用了 make\_pair 函数，可以明确地表明确实生成了 pair 对象这一行为。

3.描述关联容器和顺序容器的差别。

两者的本质差别在于：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。

4.举例说明 list、vector、deque、map 以及 set 类型分别使用的情况。

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，如以无序的方式读入一系列学生的数据；

vector 类型适用于需要随机访问元素的情况。如：在学号为 1...n 的学生中，访问第 x 学号的学生信息。

deque 类型适用于在容器的尾部或首部有插入和删除元素情况。如：对服务窗口先来先服务的情况。

map 适用于需要 key-value 对的集合的情况。如：字典电和话簿的建立和使用。

set 类型适用于使用键集合的情况。例如，黑名单的建立和使用。

5.定义一个 Map 对象，将单词与一个 list 对象关联



起来，该 list 对象存储对应的单词可能出现的行号。

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

6.可否定义一个 map 对象以 vector<int>::iterator 为键关联 int 型对象？如果以 list<int>::iterator 关联 int 型对象呢？或者，以 pair<int, string>关联 int？对于每种情况，如果允许，请解释其原因。

可以定义一个 map 对象以 vector<int>::iterator 和 pair< int, string > 为键关联 int 型对象。

不能定义第二种情况，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。

pair<int, string>关联 int 可以。

7.对于以 int 型对象为索引关联 vector<int>型对象的 map 容器，它的 mapped\_type、key\_type 和 value\_type 分别是什么？

分别是： vector<int> , int 和 pair< const int, vector<int> > 。

8.编写一个表达式，使用 map 的迭代器给其元素赋值。

```
map< string, int > m;
map< string, int >::iterator map_it = m.begin();
map_it->second = val; // 只能对 map 的值成员元素赋值。不能对键进行赋值。
```

9.编写程序统计并输出所读入的单词出现的次数。

// 11.16\_10.9\_wordcount.cpp：定义控制台应用程序的入口点。

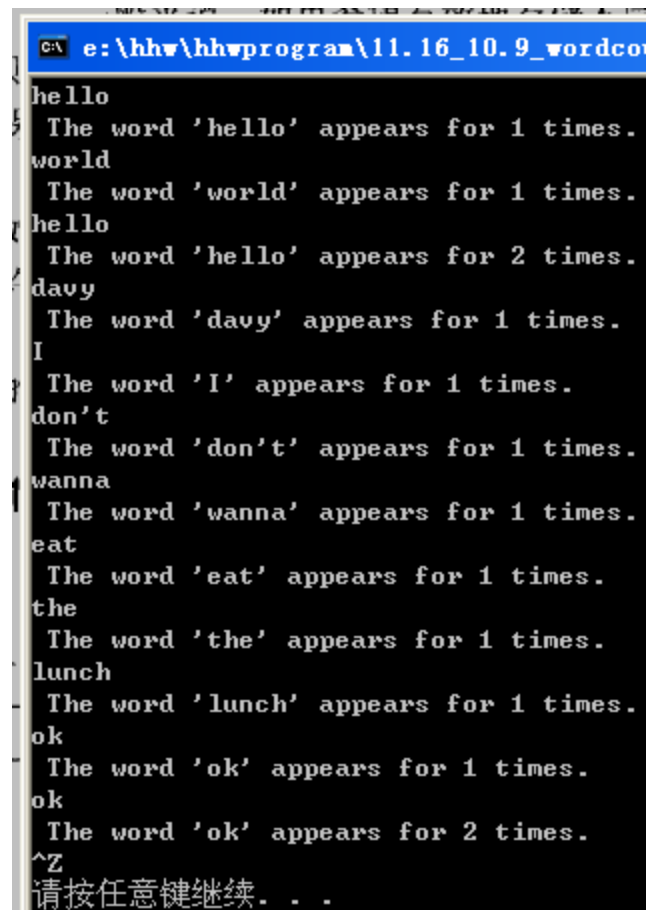
```
//
```

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <utility>
#include <map>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
```

```
    map<string, int > wordCount;
    string word;
    while ( cin >> word )
    {
        ++wordCount[ word ];
        cout << "The word '" << word << "' appears
for "
        << wordCount[ word ] << " times." <<
endl;
    }

    system("pause");
    return 0;
}
```



```
hello
The word 'hello' appears for 1 times.
world
The word 'world' appears for 1 times.
hello
The word 'hello' appears for 2 times.
davy
The word 'davy' appears for 1 times.
I
The word 'I' appears for 1 times.
don't
The word 'don't' appears for 1 times.
wanna
The word 'wanna' appears for 1 times.
eat
The word 'eat' appears for 1 times.
the
The word 'the' appears for 1 times.
lunch
The word 'lunch' appears for 1 times.
ok
The word 'ok' appears for 1 times.
ok
The word 'ok' appears for 2 times.
^Z
请按任意键继续...
```

10.解释下面程序的功能：

```
map<int, int> m;
m[0] = 1;
比较上一程序和下面程序的行为
vector<int> v;
v[0] = 1;
首先创建一个空的 map 容器 m，然后再 m 中增加一
```

个键为 0 的元素，并将其值赋值为 1，第二段程序将出现运行时错误，因为 v 为空的 vector 对象，其中下标为 0 的元素不存在。对于 vector 容器，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。

11. 哪些类型可用做 map 容器对象的下标？下标操作符返回的又是什么类型？给出一个具体例子说明，即定义一个 map 对象，指出哪些类型可用作其下标，以及下标操作符返回的类型。

可用作 map 容器的对象的下标必须是支持 < 操作符的类型。

下标操作符的返回类型为 map 容器中定义的 mapped\_type 类型。

如 map<string, int> wordCount;

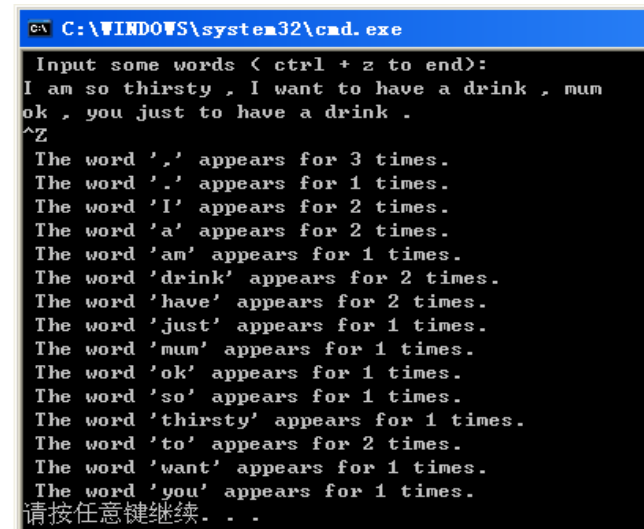
可用于其下标的类型为 string 类型以及 C 风格字符串类型，下标的操作符返回 int 类型。

12. 重写 10.3.4 节习题的单词统计程序，要求使用 insert 函数代替下标运算。你认为哪个程序更容易编写和阅读？请解释原因。

// 11.16\_10.12\_wordCount\_insert.cpp : 定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <utility>
#include <map>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Input some words ( ctrl + z to end): " << endl;
    map<string, int> wordCount;
    string word;
    while ( cin >> word )
    {
        pair< map< string, int>::iterator, bool> ret
=
wordCount.insert( make_pair( word, 1 ) );
        if ( !ret.second )
            ++ret.first->second;
```

```
    }
    for ( map< string, int>::iterator it =
wordCount.begin(); it != wordCount.end(); ++it )
    {
        cout << "The word '" << (*it).first << "'
appears for " << (*it).second << " times. " << endl;
    }
    system("pause");
    return 0;
}
```



13. 假设有 map<string, vector<int>> 类型，指出在该容器中插入一个元素的 insert 函数应具有的参数类型和返回值类型。

参数类型：pair< const string, vector<int>>

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

14. map 容器的 count 和 find 运算有何区别？

前者返回 map 容器中给定的键 K 的出现次数，且返回值只能是 0 或者 1，因为 map 只允许一个键对应一个实例。

后者返回给定键 K 元素索引时 指向元素的迭代器，若不存在此 K 值，则返回超出末端迭代器。

15. 你认为 count 适合用于解决哪一类问题？而 find 呢？

count 适合用于判断 map 容器中某键是否存在，find 适合用于在 map 容器中查找指定键对应元素。

16. 定义并初始化一个变量，原来存储调用键为 string，值为 vector<int>的 map 对象的 find 函数的返回结果。

```
map< string, vector<int> >::iterator it = xMap.find( K );
```

17. 上述转换程序使用了 find 函数来查找单词：

```
map<string, string>::const_iterator map_it =  
trans_map.find(word);
```

你认为这个程序为什么要使用 find 函数？如果使用下标操作符又会怎样？

使用 find 函数，是为了当文本中出现的单词 word 是要转换的单词时，获取该元素的迭代器，以便获取对应的转换后的单词。

如果使用下标，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不是预期效果

18. 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。为这个 map 容器输入至少六个条目。通过基于家族姓氏的查询检测你的程序，查询应输出该家族所有孩子的名字。

```
// 11..16_10.18_map_vector_children'sName.cpp :
```

定义控制台应用程序的入口点。

```
//
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

```
#include <string>
```

```
#include <vector>
```

```
#include <utility>
```

```
#include <map>
```

```
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
```

```
{
```

```
    map< string, vector<string> > children;
```

```
    string famName, childName;
```

```
    do
```

```
    {
```

```
        cout << " Input  families' name( ctrl + z to
```

```
end ): " << endl;
```

```
        cin >> famName;
```

```
        if ( !cin )
```

```
            break;
```

```
        vector<string> chd;
```

```
        pair< map< string, vector<string> >::iterator,
```

```
bool > ret =
```

```
children.insert( make_pair( famName, chd ) );
```

```
        if ( !ret.second )
```

```
        {
```

```
            cout << " Already exist the family name:
```

```
" << famName << endl;
```

```
            continue;
```

```
        }
```

```
        cout << "\n\tInput children's name ( ctrl + z  
to end ): " << endl;
```

```
        while ( cin >> childName )
```

```
        ret.first->second.push_back( childName );
```

```
        cin.clear();
```

```
    } while ( cin );
```

```
    cin.clear();
```

```
    cout << "\n\tInput a family name to search: " <<  
endl;
```

```
    cin >> famName;
```

```
    map< string, vector<string> >::iterator iter =  
children.find( famName );
```

```
    if ( iter == children.end() )
```

```
        cout << "\n\t Sorry, there is not this family  
name: " << famName << endl;
```

```
    else
```

```
    {
```

```
        cout << "\n\tchildren: " << endl;
```

```
        vector<string>::iterator it =
```

```
iter->second.begin();
```

```
        while ( it != iter->second.end() )
```

```
            cout << *it++ << endl;
```

```
    }
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

```

C:\WINDOWS\system32\cmd.exe
Input families' name( ctrl + z to end ):
wang
    Input children's name ( ctrl + z to end ):
qian hui
^Z
    Input families' name( ctrl + z to end ):
wang
    Already exist the family name: wang
    Input families' name( ctrl + z to end ):
hwang
    Input children's name ( ctrl + z to end ):
wei jiao kun
^Z
    Input families' name( ctrl + z to end ):
^Z
    Input a family name to search:
wang
    children:
qian
hui
请按任意键继续. . .

```

19.把上一题的 map 对象再扩展一下，使其 vector 对象存储 pair 类型的对象，记录每个孩子的名字和生日。相应地修改程序，测试修改后的测试程序以检查所编写的 map 是否正确。

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <map>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    map< string, vector< pair< string, string > > >
children;
    string famName, childName, childBday;
    do
    {
        cout << "\n Input  families' name( ctrl + z to
end ): " << endl;
        cin >> famName;
        if ( !cin )
            break;
        vector< pair< string, string > > chd;
        pair< map< string, vector< pair< string,
string > >::iterator, bool > ret =

```

```

children.insert( make_pair( famName, chd ) );
        if ( !ret.second )
        {
            cout << "\n Already exist the family
name: " << famName << endl;
            continue;
        }

        cout << "\tInput children's name and
birthday ( ctrl + z to end ): " << endl;
        while ( cin >> childName >> childBday )

            ret.first->second.push_back( make_pair( childNa
me, childBday ) );
        cin.clear();
    } while ( cin );

    cin.clear();
    cout << "\n\tInput a family name to search: " <<
endl;
    cin >> famName;

    map< string, vector< pair< string,
string > > >::iterator iter = children.find( famName );
    if ( iter == children.end() )
        cout << "\n\t Sorry, there is not this family
name: " << famName << endl;
    else
    {
        cout << "\n\tThe family's children and every
child's birthday is: " << endl;
        vector< pair< string, string > >::iterator it =
iter->second.begin();
        while ( it != iter->second.end() )
        {
            cout << (*it).first << " --- " <<
(*it).second << endl;
            it++;
        }
    }

    system("pause");
    return 0;

```

```
}
```

```

C:\WINDOWS\system32\cmd.exe

Input families' name( ctrl + z to end ):
wang
Input children's name and birthday ( ctrl + z to end ):
qian 2.23
hui 9.20
^Z

Input families' name( ctrl + z to end ):
hwang
Input children's name and birthday ( ctrl + z to end ):
wei 12.01
jiao 2.14
kun 10.08
^Z

Input families' name( ctrl + z to end ):
^Z

Input a family name to search:
wang

The family's children and every child's birthday is:
qian --- 2.23
hui --- 9.20
请按任意键继续. . .
```

20. 列出至少三种可以使用 `map` 类型的应用。为每种应用定义 `map` 对象，并指出如何插入和读取元素。

字典: `map< string, string > dictionary;`

电话簿: `map< string, string > telBook;`

超市物价表: `map< string, double > priceList;`

插入元素可以用: 下标操作符或 `insert` 函数; 用 `find` 函数读取元素。

21. 解释 `map` 和 `set` 容器的差别, 以及它们各自适用的情况。

差别: `map` 容器是 K-V 对的集合, `set` 容器是 键 的集合; `map` 类型适用于需要了解键与值的对应情况, 而 `set` 类型适用于只需判断某值是否存在的情况。

22. 解释 `set` 和 `list` 容器的差别, 以及它们各自适用的情况。

差别: `set` 容器中的元素不能修改, 而 `list` 容器中的元素可以修改;

`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。

23. 编写程序将排除的单词存储在 `vector` 对象中, 而不是存储在 `set` 对象中。请指出使用 `set` 的好处。

// 11.16\_10.23\_restricted\_wc.cpp: 定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <utility>
#include <vector>
#include <set>
#include <map>
using namespace std;

void restricted_wc( vector<string> strVec, map<string,
int> &wordCount )
{
    // create a set of excluded words
    set<string> excluded;
    for ( vector<string>::iterator it = strVec.begin();
it != strVec.end(); ++it )
    {
        excluded.insert( *it );
    }

    string word;
    cout << "Input some words to count the words in
wordCount(map) ( ctrl + z to end): "
        << endl;
    cin.clear();
    while ( cin >> word )
    {
        if ( !excluded.count( word ) )
            ++wordCount[ word ];
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Input some words to vector<string>
excluded ( ctrl + z to end): " << endl;
    vector<string> excludedVec;
    string excludedWord;
    while ( cin >> excludedWord )
        excludedVec.push_back( excludedWord );

    // use restricted_wc()
    map<string, int > wordCount;
```

```

restricted_wc( excludedVec, wordCount );

// show out the map:wordCount
cout << "\n\tShow out the map:wordCount: " <<
endl;

for ( map< string, int >::iterator it =
wordCount.begin(); it != wordCount.end(); ++it )
{
    cout << "The word " << (*it).first << "
appears for " << (*it).second << " times. " << endl;
}

system("pause");
return 0;
}

```

```

C:\e:\hhw\hhwprogram\11.16_10.23_restricted_wc\debug\11.16_10.23_restricted_wc>
Input some words to vector<string> excluded ( ctrl + z to end):
I am to the our my me so . . ?
^Z
Input some words to count the words in wordCount<map> ( ctrl + z to end):
hello friend .
today our school will have some guests to come .
^Z

Show out the map:wordCount:

' come '      appears 1 times.
' friend '    appears 1 times.
' guests '    appears 1 times.
' have '      appears 1 times.
' hello '     appears 1 times.
' school '    appears 1 times.
' some '      appears 1 times.
' today '     appears 1 times.
' will '      appears 1 times.

请按任意键继续. . .

```

24.编写程序通过删除单词尾部的's'生成单词的非负数版本。同时，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。例如，放在该排除集中的单词可能有 success 和 class。使用这个排除集编写程序，删除输入单词的复数后缀，而如果输入的是排除集中的单词，则保持该单词不变。

// 11.16\_10.25\_exclude\_s\_of\_words\_wc.cpp : 定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <set>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{

```

```

set<string> excluded;
// create set of excluded words
excluded.insert( "success" );
excluded.insert( "class" );

// get rid of 's'
string word;
cout << "\n Input some words : " << endl;
while ( cin >> word )
{
    if ( ! excluded.count( word ) )
        word.resize( word.size() - 1 );
    cout << "\n Not-plural version: " << word <<
endl
        << " \n Enter a word( ctrl+z to
end) " << endl;
}

system("pause");
return 0;
}

```

```

C:\e:\hhw\hhwprogram\11.16_10.25_exclude_s_of_words_wc>
Input some words :
takes

Not-plural version: take

Enter a word( ctrl+z to end)
success

Not-plural version: success

Enter a word( ctrl+z to end)
class

Not-plural version: class

Enter a word( ctrl+z to end)
trees

Not-plural version: tree

Enter a word( ctrl+z to end)
^Z
请按任意键继续. . .

```

25. 定义一个 `vector` 容器，存储你在未来六个月要阅读的书，再定义一个 `set`，用于记录你已经看过的书名。编写程序从 `vector` 中为你选择一本没有读过而现在要读的书。当它为你返回选中的书名后，应该将该书名放入记录已读书目的 `set` 中。如果实际上你把这本书放在一边没有看，则本程序应该支持从已读书目的 `set` 中删除该书的记录。在虚拟的六个月后，输出已读书目和还没有读的书。

// 11.16\_10.25\_book\_useVector\_set.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <cstdlib>
#include <ctime>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 1 create vector:bookVec
    vector<string> strVecBook;
    string bName;
    cout << " Input books' name you wanna read
( ctrl+z to end):\n";
    while ( cin >> bName )
        strVecBook.push_back( bName );

    size_t NumOfBooks = strVecBook.size(); //
record number of books

    // 2 create set:strSetReadedBook
    set<string> strSetReadedBook;
    string strChoice, bookName;
    bool _6MonthLater = false;
    // use system's time for seed of rand
    srand ( ( unsigned ) time(NULL) );
    while ( !_6MonthLater
&& !strVecBook.empty() )
    {
        cin.clear();
```

```
        cout << "\n==>>Do you wanna read a book?
( Yes / No ): ";
        cin >> strChoice;
        if ( strChoice[0] == 'Y' || strChoice[0] ==
'y' )
        {
            int i = rand() % strVecBook.size();
            bookName = strVecBook[i];
            cout << " ==>> We choose this book for
you: " << bookName << endl;

            strSetReadedBook.insert( bookName );
            strVecBook.erase( strVecBook.begin()
+ i );

            cout << "\tOne month later .... " <<
endl
                << " Did you read this book?
( Yes/ No ): ";
            //cin.clear();
            cin >> strChoice;
            if ( strChoice[0] == 'n' || strChoice[0]
== 'N' )
            {
                // delete the book from
strSetReadedBook and add to strVecBook

                strSetReadedBook.erase( bookName );

                strVecBook.push_back( bookName );
            }

            // 6 month later ?
            cout << " 6 month later ? ( Yes / No ): ";
            //cin.clear();
            cin >> strChoice;
            if ( strChoice[0] == 'Y' || strChoice[0] ==
'y' )
                _6MonthLater = true;
        }

        if ( _6MonthLater )
        {
            if ( !strSetReadedBook.empty() )
```



```

{
    cout << "\n ==>>>During the latest 6
months , you read: \n\t";
    for ( set<string>::iterator iter =
strSetReadedBook.begin(); iter !=
strSetReadedBook.end(); ++iter )
    {
        cout << *iter << " ";
    }
}
if ( !strVecBook.empty() )
{
    cout << "\n ==>>>The books you have
not read : \n\t";
    for ( vector<string>::iterator it =
strVecBook.begin(); it != strVecBook.end(); ++it )
    {
        cout << *it << " ";
    }
}
cout << endl;
}
if ( strSetReadedBook.size() == NumOfBooks )
    cout << "\t Good, you read all the
books. " << endl;

system("pause");
return 0;
}

```

```

e:\hhw\hhwprogram\11.16_10.25_book_usevector_set
Input books' name you wanna read < ctrl+z to end>:
b1 b2 b3 b4
^Z

==>>>Do you wanna read a book? < Yes / No >: yes
=>> We choose this book for you: b3
    One month later ...
Did you read this book? < Yes/ No >: yes
6 month later ? < Yes / No >: no

==>>>Do you wanna read a book? < Yes / No >: yes
=>> We choose this book for you: b4
    One month later ...
Did you read this book? < Yes/ No >: no
6 month later ? < Yes / No >: no

==>>>Do you wanna read a book? < Yes / No >: yes
=>> We choose this book for you: b2
    One month later ...
Did you read this book? < Yes/ No >: yes
6 month later ? < Yes / No >: yes

==>>>During the latest 6 months , you read:
    b2 b3
==>>>The books you have not read :
    b1 b4
请按任意键继续. . .

```

26.编写程序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,确保你的程序依然能正确执行。

// 11.16\_10.26\_multimap.cpp : 定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <map>
#include <utility>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    multimap< string, string > mmapAuthor;
    string author, book;
    cout << "\t==>> Input Author and his Book( ctrl + z
to end ):\n";
    while ( cin >> author >> book )
    {
        mmapAuthor.insert( make_pair( author,
book ) );
    }

    // search

```

```

C:\> cd /d h:\hhw\hhwprogram\11.16_10.26_multimap\debug\11.16_10.26
e:\hhw\hhwprogram\11.16_10.26_multimap\debug\11.16_10.26
=>> Input Author and his Book< ctrl + z to end >:
hhw C++_1ed
wwq C++_2ed
hhw C_L
wwq C_L_2ed
hhw C_sharp
^Z

=>> Input which author do you wanna search: hhw
^_^ We got the author: hhw
=>> his book: C++_1ed

^_^ We got the author: hhw
=>> his book: C_L

^_^ We got the author: hhw
=>> his book: C_sharp

We successfully to erase 3 books of the author.
请按任意键继续. . .

```

{

```

        cout << "\n\t ^_^ We got the author:  " <<
it->first << endl
        << "\t ==>> his book:\t" << it->second <<
endl;
        ++it;
    }

    // erase
    mmapAuthor.erase( pos.first, pos.second );
    if ( amount )
    {
        cout << "\n\tWe successfully to erase " <<
amount << " books of the author. " << endl;
    }
    cout << "\n\t==>> Now the content of multimap
is:" << endl;
    for ( author_it iter = mmapAuthor.begin(); iter !=
mmapAuthor.end(); ++iter )
    {
        cout << iter->first << " ---- " <<
iter->second << endl;
    }
    cout << endl;
    system("pause");
    return 0;
}

```

```

C:\e:\hhw\hhwprogram\11.16_10.27_equal_range\debug\11.16_10.27_equal_range>
=>> Input Author and his Book( ctrl + z to end ):
hhw c++
wwq C++Primer
hhw C_plus
wwq C_sharp
^Z

=>> Input which author do you wanna search: wwq

^_^ We got the author:  wwq
=>> his book:  C++Primer

^_^ We got the author:  wwq
=>> his book:  C_sharp

We successfully to erase 2 books of the author.

=>> Now the content of multimap is:
hhw ---- c++
hhw ---- C_plus
请按任意键继续. . .

```

28.沿用上题中的 multimap 容器,编写程序以下面的格式按姓名首字母的顺序输出作者名字:  
Author Names Bginning with 'A':  
Author, book, book, ...

```

...
Author Names Beginning with 'B':
...
// 11.16_10.28_multimap_Author_and_work.cpp : 定义控制台应用程序的入口点。
//
#include "stdafx.h"
#include <iostream>
#include <map>
#include <utility>
#include <string>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    multimap< string, string > mmapAuthor;
    string author, book;

    cin.clear();
    cout << "\t==>> Input Author and his Book( ctrl + z
to end ):\n";
    while ( cin >> author >> book )
    {
        mmapAuthor.insert( make_pair( author,
book ) );
    }

    typedef multimap<string, string>::iterator
author_it;
    author_it iter = mmapAuthor.begin();
    if ( iter == mmapAuthor.end() )
    {
        cout << "\n\t Empty multimap! " << endl;
        return 0;
    }
    string currAuthor, preAuthor;
    do
    {
        currAuthor = iter->first;
        if ( preAuthor.empty() || currAuthor[0] !=
preAuthor[0] )
        {
            cout << "Author Names Beginning with
' "

```

```

        << iter->first[0] << " " : " << endl;
    }
    cout << currAuthor;
    pair< author_it, author_it > pos =
mmapAuthor.equal_range( iter->first );
    while ( pos.first != pos.second )
    {
        cout << " " << pos.first->second;
        ++pos.first;
    }
    cout << endl;
    iter = pos.second;
    preAuthor = currAuthor;
} while ( iter != mmapAuthor.end() );

cout << endl;
system("pause");
return 0;
}

```

29. 解释本节最后一个程序的输出表达式使用的操作数 `pos.first->second` 的含义。

`pos` 存储的是一个 `pair` 对象，`pos.first` 是指 `pair` 的第一个迭代器，`pos.first->second` 是指那个迭代器所指的 `multimap` 对象的第二个元素，即作者的作品。

30. `TextQuery` 类的成员函数仅使用了前面介绍过的内容。先别查看后面章节，请自己编写这些成员函数。提示：唯一棘手的是 `run_query` 函数在行号集合 `set` 为空时应返回什么值？解决方法是构造并返回一个新的（临时）`set` 对象。

```

#include "stdafx.h"
#include "TextQuery.h"
#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
" );
}

void TextQuery::store_file( ifstream &is )
{
    string textline;
    while ( getline( is, textline ) )
    {
        lines_of_text.push_back( textline );
    }
}

void TextQuery::build_map()
{
    for ( line_no line_num = 0; line_num !=
lines_of_text.size(); ++ line_num )
    {
        istringstream
line( lines_of_text[ line_num ] );
        string word;
        while ( line >> word )
        {
            // get rid of punctuation
            word = cleanup_str( word );
            if ( word_map.count ( word ) == 0 )

                word_map[word].push_back( line_num );
            else
            {
                if ( line_num !=
word_map[word].back() )

                    word_map[word].push_back( line_num );
            }
        }
    }
}

```

```

vector< TextQuery::line_no >
TextQuery::run_query( const string& query_word )
const
{
    map<string, vector<line_no> >::const_iterator loc
= word_map.find( query_word );
    if ( loc == word_map.end() )
    {
        return vector<line_no>();
    }
    else
        return loc->second;
}

// get rid of punctuation
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;
}

```

31. 如果没有找到要查询的单词，main 函数输出什么？

strWord occurs 0 times.

32. 重新实现文本查询程序，使用 vector 容器代替 set 对象来存储行号。注意，由于行以升序出现，因此只有在当行号不是 vector 容器对象中的最后一个元素时，才能将新的行号添加到 vector 中。这两种实现方法的性能特点和设计特点分别是什么？你觉得哪一种解决方法更好？为什么？

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存在容器中，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，无需判断行号是否已存在，直接利用 insert 函

数插入元素即可，性能较高。

vector 容器的特点是适合用于需随机访问元素的情况。而此处对于存储的行号，不需进行随机访问，只需进行顺序访问，因此使用 set 容器操作简单，设计更为简洁，因此使用 set 容器好。

// 11.17\_10.32\_TextQuery.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include "TextQuery.h"
#include "functions.h"

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 )
{
    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;
    line_nums::const_iterator it = locs.begin();
    for ( ; it != locs.end(); ++it )
    {
        cout << "\t( line " << (*it) + 1 << " ) " <<
file.text_line( *it ) << endl;
    }
}

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    ifstream infile;
    if ( argc < 2 || !open_file( infile, argv[1] ) )
    {
        cerr << " No input file! " << endl;
        return EXIT_FAILURE;
    }
    TextQuery tq;
    tq.store_file( infile );
    while ( true )
    {

```

```

        cout << "Input word to look for, or q to quit:
";
        string s;
        cin >> s;
        string ret;
        for ( string::const_iterator it = s.begin(); it !=
s.end(); ++it )
        {
            ret += tolower( *it );
        }
        s = ret;

        if ( !cin || s == "q" || s == "Q" ) break;
        vector< TextQuery::line_no > locs =
tq.run_query( s );
        print_results( locs, s, tq );
    }

    system( "pause" );
    return 0;
}

```

```

#ifndef __TEXTQUERY_
#define __TEXTQUERY_

```

```

#include <map>
#include <utility>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <cstring>

```

```
using namespace std;
```

```

class TextQuery
{
public:
    typedef string::size_type str_size;
    typedef vector<string>::size_type line_no;

    // interface
    void read_file ( ifstream &is )
    {

```

```

        store_file( is );
        build_map();
    }

```

```

    vector<line_no> run_query ( const string& )
const;

```

```

    string text_line ( line_no ) const;
    void store_file( ifstream &);

```

```
private:
```

```

    void build_map();
    vector<string> lines_of_text;
    map< string, vector<line_no> > word_map;
    static string cleanup_str( const string& );

```

```
};
```

```
#endif
```

```

#include "stdafx.h"
#include "TextQuery.h"
#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
" );
}

```

```

void TextQuery::store_file( ifstream &is )
{

```

```

    string textline;
    while ( getline( is, textline ) )
    {
        lines_of_text.push_back( textline );
    }
}

```

```
void TextQuery::build_map()
```

```

{
    for ( line_no line_num = 0; line_num !=

```

```

lines_of_text.size(); ++ line_num )
{
    istringstream
line( lines_of_text[ line_num ] );
    string word;
    while ( line >> word )
    {
        // get rid of punctuation
        word = cleanup_str( word );
        if ( word_map.count ( word ) == 0 )

            word_map[word].push_back( line_num );
        else
        {
            if ( line_num !=
word_map[word].back() )

                word_map[word].push_back( line_num );
        }
    }
}

vector< TextQuery::line_no >
TextQuery::run_query( const string& query_word )
const
{
    map<string, vector<line_no> >::const_iterator loc
= word_map.find( query_word );
    if ( loc == word_map.end() )
    {
        return vector<line_no>();
    }
    else
        return loc->second;
}

// get rid of punctuation
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;
}

#include <fstream>
#include <string>
using namespace std;

string make_plural( size_t ctr, const string &word ,
const string &ending )
{
    return ( ctr == 1 ) ? word : word + ending;
}

ifstream& open_file( ifstream &in, const string &file )
{
    in.close();
    in.clear();
    in.open( file.c_str() );
    return in;
}

```

33.TextQuery::text\_line 函数为什么不检查它的参数是否为负数？

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 对象中出现的元素不可能为负数。

我的 C++Primer 作业，  
2010 年 11 月 6 日  
Davy

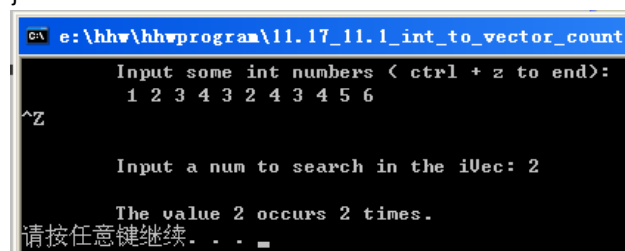
## 第十一章 泛型算法

1.algorithm 头文件定义了一个名为 count 的函数，其功能类似于 find。这个函数使用一对迭代器和一个值做参数，返回这个值出现的次数的统计结果。编写程序读取一系列 int 型数据，并将它们存储到



vector 对象中然后统计某个指定的值出现了多少次。  
// 11.17\_11.1\_int\_to\_vector\_count.cpp : 定义控制台应用程序的入口点。

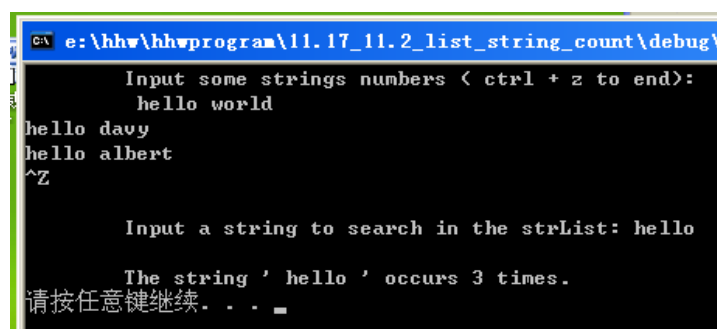
```
//  
  
#include "stdafx.h"  
#include <vector>  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int _tmain(int argc, _TCHAR* argv[])  
{  
    cout << "\tInput some int numbers ( ctrl + z to end):\n\t ";  
    vector<int> iVec;  
    int iVal;  
    while ( cin >> iVal )  
        iVec.push_back( iVal );  
  
    cout << "\n\tInput a num to search in the iVec: ";  
    cin.clear();  
    cin >> iVal;  
    int iCnt = 0;  
    if ( iCnt = count( iVec.begin(), iVec.end(), iVal ))  
    {  
        cout << "\n\tThe value " << iVal << " occurs "  
        << iCnt << " times." << endl;  
    }  
  
    system("pause");  
    return 0;  
}
```



2. 重复前面的程序，但是，将读入的值存储到一个 string 类型的 list 对象中。

// 11.17\_11.2\_list\_string\_count.cpp : 定义控制台应用程序的入口点。

```
//  
#include "stdafx.h"  
#include <string>  
#include <list>  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int _tmain(int argc, _TCHAR* argv[])  
{  
    cout << "\tInput some strings numbers ( ctrl + z to end):\n\t ";  
    list<string> strLst;  
    string str;  
    while ( cin >> str )  
        strLst.push_back( str );  
  
    cout << "\n\tInput a string to search in the strList: ";  
    cin.clear();  
    cin >> str;  
    size_t iCnt = 0;  
    if ( iCnt = count( strLst.begin(), strLst.end(), str))  
    {  
        cout << "\n\tThe string ' " << str << " ' occurs "  
        << iCnt << " times." << endl;  
    }  
    system("pause");  
    return 0;  
}
```



3. 用 accumulate 统计 vector<int> 容器对象中的元素之和。

// 11.19\_11.3\_accumulate\_vector\_int.cpp : 定义控制台应用程序的入口点。

//

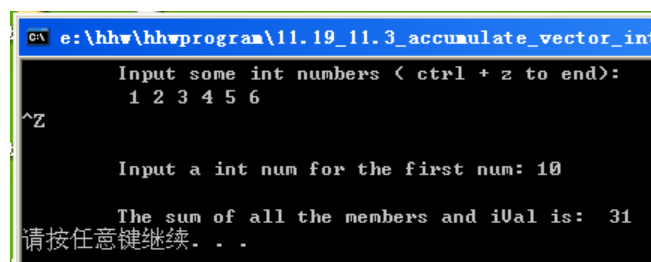
```
#include "stdafx.h"
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some int numbers ( ctrl + z to end):\n\t ";
    vector<int> iVec;
    int iVal;
    while ( cin >> iVal )
    {
        iVec.push_back( iVal );
    }

    cout << "\n\tInput a int num for the first num: ";
    cin.clear();
    cin >> iVal;

    if ( iVal= accumulate( iVec.begin(), iVec.end(),
iVal ) )
    {
        cout << "\n\tThe sum of all the members
and iVal is:  " << iVal << endl;
    }

    system("pause");
    return 0;
}
```



4. 假定 `v` 是 `vector<double>` 类型的对象，则调用 `accumulate( v.begin(), v.end(), 0 )` 是否有错？如果有，错在哪里？

没有错，`accumulate` 函数必须满足第三个实参的类型与容器内的意思匹配，或者可以转化为第三个实参的类型。本题中 `double` 可以转化为 `int` 类型，但是会有较大误差。

5. 对于本节调用 `find_first_of` 的例程，如果不给 `it` 加 1，将会如何。

(1) 如果存在同时出现在两个 `list` 中的名字，则进入 `while` 循环，死循环；

(2) 不存在同时出现在两个 `list` 中的名字，循环不会被执行。

6. 使用 `fill_n` 编写程序，将一系列 `int` 型值设为 0。

// 11.18\_11.6\_fill\_n.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some int numbers ( ctrl + z to end):\n\t ";
    vector<int> iVec;
    int iVal;
    while ( cin >> iVal )
    {
        iVec.push_back( iVal );
    }

    cout << "\n\tThe content of ivec is :\n";
    for ( vector<int>::iterator it = iVec.begin(); it != iVec.end(); ++it )
        cout << *it << " ";
}
```

```

fill_n( iVec.begin(), iVec.size(), 0 );

cout << "\n\tAfter fill_n(), the content of ivec
is :\n";
for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
    cout << *it << " ";
cout << endl;

system("pause");
return 0;
}

```

7.判断下面的程序是否有错，如果有，请改正之：

```

(a) vector<int> vec; list<int> lst; int i;
while ( cin >> i )
    lst.push_back(i);
copy( lst.begin(), lst.end(), vec.begin() );
(b) vector<int> vec;
vec.reserve( 10 );
fill_n ( vec.begin(), 10, 0 );

```

(a) 有错，vec 是一个空容器，试图往一个空容器里复制数据，发生错误，应改为：

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

(b) 有错误，虽然为 vec 分配了内存，但是 vec 仍然是一个空容器，而在空 vec 上调用 fill\_n 会产生灾难，更正为：

```

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

```

8.前面说过，算法不改变它所操纵的容器的大小，为什么使用 back\_inserter 也不能突破这个限制？

在使用 back\_inserter 是，不是算法直接改变它所操作的容器的大小，而是算法操作迭代器 back\_inserter，迭代器的行为导致了容器的大小改变。

9.编写程序统计长度不小于 4 个单词，并输出输入序列中不重复的单词。在程序源文件上运行和测试你自己的程序。

// 11.18\_11.9\_wc\_GT4.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
using namespace std;

bool isShorter( const string &s1, const string &s2 )
{
    return s1.size() < s2.size();
}

bool GT4( const string &s )
{
    return s.size() >= 4;
}

string make_plural( size_t i, const string &s1, const
string &s2 )
{
    return ( i == 1 ) ? s1 : s1 + s2;
}

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some words ( ctrl + z to
end):\n\t ";
    vector<string> strVec;
    string strVal;
    while ( cin >> strVal )
        strVec.push_back( strVal );

    // sort
    sort ( strVec.begin(), strVec.end() );
    vector<string>::iterator end_unique = unique
( strVec.begin(), strVec.end() );
    strVec.erase( end_unique, strVec.end() );
}

```

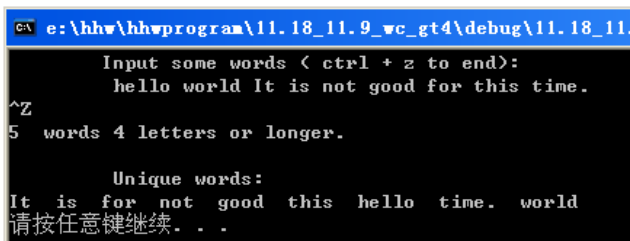
```

        stable_sort( strVec.begin(), strVec.end(),
isShorter );
        vector<string>::size_type wc = count_if
( strVec.begin(), strVec.end(), GT4 );
        cout << wc << " " << make_plural ( wc, "word",
"s" ) << " 4 letters or longer. " << endl;

        cout << "\n\t Unique words: " << endl;
        for ( vector<string>::iterator it = strVec.begin();
it != strVec.end(); ++it )
            cout << *it << " ";
        cout << endl;

        system("pause");
        return 0;
}

```



10. 标准库定义了一个 `find_if` 函数，与 `find` 一样，`find_if` 函数带有一对迭代器形参，指定其操作的范围。与 `count_if` 一样，该函数还带有第三个形参，表明用于检查范围内每个元素的谓词函数。`find_if` 返回一个迭代器，指向第一个使为此函数返回非零值的元素。如果这样的元素不存在，则返回第二个迭代器实参。使用 `find_if` 函数重写上述例题中统计长度大于 6 的单词个数的程序部分。

首先创建一个空的 `map` 容器 `m`，然后再 `m` 中增加一个键为 0 的元素，并将其值赋值为 1，第二段程序将出现运行时错误，因为 `v` 为空的 `vector` 对象，其中下标为 0 的元素不存在。对于 `vector` 容器，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。

```

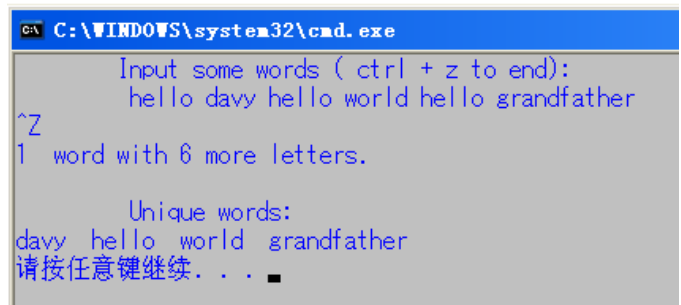
int iCnt = 0;
vector<string>::iterator wit = strVec.begin();
while ( ( wit = find_if( wit, strVec.end(), GT7 )) !=
strVec.end() )
{
    iCnt++;
    ++wit;
}

```

```

}
cout << iCnt << " " << make_plural ( iCnt,
"word", "s" ) << " with 6 more letters. " << endl;

```



11. 你认为为什么算法不改变容器的大小？  
为了使得算法能够独立于容器，从而普适性更好，真正成为“泛型”算法。

12. 为什么必须使用 `erase`，而不是定义一个泛型算法来删除容器中的元素。

泛型算法的原则就是不改变容器的大小。

13. 解释三种插入迭代器的区别。

区别在于插入的元素的位置不同：

`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` 实现插入，它还带有第二个实参：指向插入起始位置的迭代器。

14. 编程使用 `replace_copy` 将一个容器中的序列复制给另一个容器，并将前一个序列中给定的值替换为指定的新值。分别使用 `inserter`、`back_inserter` 和 `front_inserter` 实现这个程序。讨论在不同情况下输出序列如何变化。

// 11.18\_11.14\_replace\_copy.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    //vector< int > iVec;
    int ia[] = { 1, 2, 3, 4, 100, 5, 100 };
    vector< int > iVec( ia, ia+7 );
    // testing out iVec
    cout << "\n\tThe contents of iVec: ";
    for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
        cout << *it << " ";
    cout << endl;

    list<int> iLst;
    // copy iVec's member to iLst;
    cout << "\n Using inserter: " << endl;
    replace_copy( iVec.begin(), iVec.end(),
inserter( iLst, iLst.begin() ), 100, 0 );
    cout << "\tThe contents of iLst: ";
    for ( list<int>::iterator it = iLst.begin(); it !=
iLst.end(); ++it )
        cout << *it << " ";
    cout << endl;

    cout << "\n Using back_inserter: " << endl;
    iLst.clear();
    replace_copy( iVec.begin(), iVec.end(),
back_inserter( iLst ), 100, 0 );
    cout << "\tThe contents of iLst: ";
    for ( list<int>::iterator it = iLst.begin(); it !=
iLst.end(); ++it )
        cout << *it << " ";
    cout << endl;

    cout << "\n Using front_inserter: " << endl;
    iLst.clear();
    replace_copy( iVec.begin(), iVec.end(),
front_inserter( iLst ), 100, 0 );
    cout << "\tThe contents of iLst: ";
    for ( list<int>::iterator it = iLst.begin(); it !=
iLst.end(); ++it )
        cout << *it << " ";
    cout << endl;
    system("pause");
    return 0;
}

```

```

}

```

```

e:\hhw\hhwprogram\11.18_11.14_replace_copy\debug\11.18_11.14_replace_copy.exe

The contents of iVec: 1 2 3 4 100 5 100

Using inserter:
The contents of iLst: 1 2 3 4 0 5 0

Using back_inserter:
The contents of iLst: 1 2 3 4 0 5 0

Using front_inserter:
The contents of iLst: 0 5 0 4 3 2 1

请按任意键继续. . .

```

15. 算法标准库定义了一个名为 `unique_copy` 的函数，其操作与 `unique` 类似，唯一的区别在于：前者接受第三个迭代器实参，用于指定复制不重复元素的目标序列。编写程序使用 `unique_copy` 将一个 `list` 对象中不重复的元素复制到一个空的 `vector` 对象中。

// 11.18\_11.15\_unique\_copy.cpp : 定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    //vector< int > iVec;
    int ia[] = { 1, 2, 3, 4, 100, 100, 5 };
    list<int> iLst( ia, ia+7 );
    // testing out iLst
    cout << "\n\tThe contents of iLst:\n\t";
    for ( list<int>::iterator it = iLst.begin(); it !=
iLst.end(); ++it )
        cout << *it << " ";
    cout << endl;

    vector<int> iVec;
    // copy iLst's member to iVec;
    cout << "\n Using unique_copy, copy iLst's

```

```

member to iVec: " << endl;
    unique_copy( iLst.begin(), iLst.end(),
back_inserter( iVec ) );

    cout << "\n\tThe contents of iVec:\n\t";
    for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
        cout << *it << " ";
    cout << endl;
    system("pause");
    return 0;
}

```

16. 重写（11.3.2 节第 3 小节）的程序，使用 copy 算法将一个文件的内容写到标准输出中。

// 11.20\_11.16\_istream\_iterator.cpp：定义控制台应用程序的入口点。

// 使用copy算法将一个文件的内容写到标准输出中

```

#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <string>
#include <iterator>
#include <fstream>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    ostream_iterator<string> out_iter( cout, "\n" );
    ifstream inFile;
    inFile.open( "11.16.txt" );
    if ( !inFile )
    {
        cerr << " error file. " << endl;
        return -1;
    }
}

```

```

}
    istream_iterator<string> in_file_iter ( inFile ),
eof;
    copy( in_file_iter, eof, out_iter );

    inFile.close();
    cout << endl;
    system("pause");
    return 0;
}

```

17.使用一对 istream\_iterator 对象初始化一个 int 型的 vector 对象。

// 11.20\_11.17\_istream\_iterator\_vector.cpp：定义控制台应用程序的入口点。

//

```

#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    istream_iterator<int> cin_it( cin ), end;
    vector<int> iVec( cin_it, end );

    cout << "\n\tThe content of iVec:\n\t";
    for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
    {
        cout << *it << " ";
    }
}

```

```

}

cout << endl;
system("pause");
return 0;
}

```

18.编程使用 `istream_iterator` 对象从标准输入读入一些列整数。使用 `ostream_iterator` 对象将偶数写到第二个文件，每个写入的值都存放在单独的行中。

//  
11.20\_11.18\_istream\_iterator\_ostream\_iterator.cpp:  
定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <iterator>
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    ofstream outFile_even, outFile_odd;
    outFile_even.open( "outFile_even.txt" );
    outFile_odd.open( "outFile_odd.txt" );
    if ( !outFile_even || !outFile_odd )
    {
        cerr << " File can't be open. " << endl;
        return -1;
    }

    istream_iterator<int> in_iter( cin ), end;
    ostream_iterator<int> out_odd_iter( outFile_odd,
    " " );
    ostream_iterator<int>
    out_even_iter( outFile_even, "\n" );

```

```

while ( in_iter != end )
{
    if ( *in_iter % 2 == 0 )
        *out_even_iter++ = *in_iter++;
    else
        *out_odd_iter++ = *in_iter++;
}

```

```

outFile_odd.close();
outFile_even.close();
cout << endl;
system("pause");
return 0;
}

```

}

19.编写程序使用 `reverse_iterator` 对象以逆序输出 `vector` 容器对象的内容。

// 11.20\_11.19\_reverse\_iterator.cpp：定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

```



```
int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 0, 1, 2, 3, 4 };
    vector<int> iVec( ia, ia + 5 );
    for ( vector<int>::reverse_iterator rit =
iVec.rbegin(); rit != iVec.rend(); ++rit )
    {

        cout << *rit << " ";

    }
    cout << endl;
    system("pause");
    return 0;
}
```



20.现在，使用普通的迭代器逆序输出上题中的元素。

// 11.20\_11.20\_iterator\_reverse.cpp：定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 0, 1, 2, 3, 4 };
    vector<int> iVec( ia, ia + 5 );
    for ( vector<int>::iterator it = iVec.end() - 1; it >
iVec.begin(); --it )
    {
        cout << *it << " ";
    }
    cout << *iVec.begin();
    cout << endl;
    system("pause");
    return 0;
}
```



21.使用 find 在一个 int 型的 list 中需找值为 0 的最后一个元素。

// 11.20\_11.21\_list\_find.cpp：定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 1, 2, 3, 4, 0, 6 };
    list<int> iLst( ia, ia + 6 );

    for ( list<int>::reverse_iterator rit = iLst.rbegin();
rit != iLst.rend(); ++rit )
    {

        cout << *rit << " ";

    }
    cout << endl ;

    list<int>::reverse_iterator last_0_it =
find( iLst.rbegin(), iLst.rend(), 0 );
    if ( last_0_it != iLst.rend() )
        cout << "Get the last 0, it's value: " <<
*last_0_it << endl;
    else
        cout << "We can't find the last 0. " << endl;

    cout << endl;
    system("pause");
    return 0;
}
```

```

C:\ e:\hhw\hhwprogram\11.20_11.22_vector
6 0 4 3 2 1
Get the last 0, it's value: 0
请按任意键继续. . .

```

22. 假设有一个存储了 10 个元素的 vector 对象，将其中第 3~第 7 个位置上的元素以逆序复制给 list 对象。

// 11.20\_11.22\_vector\_reverseCopyToList.cpp : 定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <iterator>
#include <list>
#include <vector>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int ia[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    vector<int> iVec( ia, ia + 10 );
    cout << "\tThe original iVec's content is:\n";
    for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
        cout << *it << " ";

    list<int> iLst( iVec.rbegin() + 3, iVec.rbegin() + 8 );

    cout << endl << "\tAfter deal, list is:\n";
    for ( list<int>::iterator it = iLst.begin(); it !=
iLst.end(); ++it )
        cout << *it << " ";
    cout << endl;
    system("pause");
    return 0;
}

```

```

C:\ e:\hhw\hhwprogram\11.20_11.22_vector
The original iVec's content is:
1 2 3 4 5 6 7 8 9 10
After deal, list is:
7 6 5 4 3
请按任意键继续. . .

```

23. 列出五种迭代器类型及其各自支持的操作。

输入迭代器:	== != 自减, * ->
输出迭代器	++ *
前向迭代器	== != * ->
双向迭代器	== != ++ * -> --
随机访问迭代器	== != ++ * -> <
	<= > >= _ +=

24. List 容器拥有什么类型的迭代器？而 vector 呢？  
双向迭代器，vector 拥有随机访问迭代器。

25. 你认为 copy 算法需要使用哪种迭代器？而 reverse 和 unique 呢？

copy 至少需要输入和输出迭代器；  
reverse 算法至少需要双向迭代器；  
unique 算法至少需要前向迭代器。

26. 解释下列代码错误的原因，指出哪些错误可以在编译时捕获。

- (a) string sa[10];  
const vector<string> file\_names( sa, sa+6 );
- (b) const vector<int> ivec;  
fill ( ivec.begin(), ivec.end(), ival );
- (c) sort( ivec.begin(), ivec.rend() );  
sort( ivec1.begin(), ivec2.end() );
- (a) const 类型的 vector<string>对象，不可以写入，错误。  
(b) 错了，两个实参迭代器是 const 迭代器，不能用来修改容器中的元素。  
(c) 错了，用于算法的实参的两个迭代器必须是相同类型。  
(d) 错了，用于算法参数的迭代器必须属于同一个迭代器。  
前三个错误均可以在编译时捕获。(d) 不能在编译时捕获。

27.标准库定义了下面的算法:

```
replace( beg, end, old_val, new_val );
replace_if ( beg, end, pred, new_val );
replace_copy( beg, end, dest, old_val, new_val );
replace_copy_if ( beg, end, dest, pred, new_val );
```

只根据这些函数的名字和形参,描述这些算法的功能。

第一个:由 `beg` 和 `end` 指定的输入范围内值为 `old_val` 的元素用 `new_val` 替换。

第二个:由 `beg` 和 `end` 指定的输入范围内使得谓词 `pred` 为真的元素用 `new_val` 替换。

第三个:由 `beg` 和 `end` 指定的输入范围内的元素复制到 `dest`,并将值为 `old_val` 的元素用 `new_val` 替换。

第四个:由 `beg` 和 `end` 指定的输入范围内的元素复制到 `dest`,并将使得谓词为真的元素用 `new_val` 替换。

28.假设 `lst` 是存储了 100 个元素的容器,请解释下面的程序段,并修正你认为的错误。

```
vector<int> vec1;
reverse_copy ( lst.begin(), lst.end(), vec1.begin() );
```

这段程序试图将 `lst` 中的元素以逆序的方式复制到 `vec1` 容器中。

可能的错误为:

1. `vec1` 是一个空容器,尚未分配存储空间,可改为:  
`vector<int> vec1(100);`
2. 另外 `lst` 容器中存储的元素不一定为 `int` 型。应该保证两种容器内存储的值的类型相同或者可以进行转换。

29.用 `list` 容器取代 `vector` 重新实现 11.2.3 节编写的排除重复单词的程序。

// 11.20\_11.29\_list\_wordCount.cpp : 定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <iostream>
#include <list>
#include <string>
#include <algorithm>
#include <numeric>
```

```
using namespace std;
```

```
bool isShorter( const string &s1, const string &s2 )
{
    return s1.size() < s2.size();
}
```

```
bool GT4( const string &s )
{
    return s.size() >= 4;
}
```

```
string make_plural( size_t i, const string &s1, const
string &s2 )
{
    return ( i == 1 ) ? s1 : s1 + s2;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    cout << "\tInput some words to a list ( ctrl+ z to
end):\n\t ";
    list<string> strLst;
    string strVal;
    while ( cin >> strVal )
        strLst.push_back( strVal );

    // sort
    strLst.sort();
    strLst.unique();

    list<string>::size_type wc = count_if
( strLst.begin(), strLst.end(), GT4 );
    cout << "\t"<< wc << make_plural ( wc, " word",
"s" ) << " 4 letters or longer. " << endl;

    cout << "\n\t Unique words: " << endl;
    for ( list<string>::iterator it = strLst.begin(); it !=
strLst.end(); ++it )
        cout << *it << " ";
    cout << endl;

    system("pause");
    return 0;
}
```

```
}

CA e:\hhw\hhwprogram\11.20_11.29_list_wordcount\debug\11
Input some words to a list ( ctrl + z to end):
hello davy hello albert hello ZengKe
^Z

4 words 4 letters or longer.

Unique words:
ZengKe albert davy hello
请按任意键继续. . .
```

## 第十二章 类和数据抽象

12.1 编写一个名为 person 的类，表示人的名字和地址，使用 string 来保存每个元素。

答：

```
class person
{
public:
    person( string pName, string pAddress )
    {
        name = pName;
        address = pAddress;
    }
private:
    string name;
    string address;
};
```

12.2 为 person 提供一个接收两个 string 参数的构造函数。

见第一题。

12.3 提供返回名字和地址的操作。这些函数应为 const 吗？解释你的选择。

在 public 里添加成员函数：

```
string get_name() const
{
    return name;
}
string get_address() const
{
    return address;
}
```

这两个成员函数不应该修改其操作的对象的数据成员的值，应该声明为 const 类型。

12.4 指明 person 的哪个成员应声明为 public，哪个成员应声明为 private。解释。

数据成员 name 和 address 应为 private，保证只能被类的成员所用，外界不可访问。成员函数 get\_name() 和 get\_address() 应声明为 public，为外界提供接口访问类的数据成员。构造函数也应声明为 public，以便初始化类的对象。

12.5 C++ 类支持哪些访问标号？在每个访问标号之后应定义哪种成员？如果有的话，在类的定义中，一个访问标号可以出现在何处以及可出现多少次？约束条件是什么？

有 public, private, protect。public 后定义可被外界访问的接口，private 后定义只能被本类成员函数使用的成员；protect 后定义的成员称为受保护成员，只能由本类及本类的子类访问。

访问标号可以出现在任意成员定义之前且次数没有限制。

约束条件是：每个访问标号指定了随后的成员定义级别，这个级别持续有效，直到下一个访问标号出现，或者看到类定义体的右花括号为止。

12.6 用 class 关键字定义的类和用 struct 定义的类有什么不同。

默认访问标号不同，用 struct 关键字定义的，在第一个访问标号之前的成员是共有的，如果是用 class 关键字定义的，在第一个访问标号之前的成员是 private 成员。

12.7 什么事封装？为什么封装是有用的？

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。被封装的元素隐藏了它们的实现细节，可以调用一个函数但不能访问它所执行的语句，同样类也是一个封装的实体：它代表若干成员的聚集，大多数类类型隐藏了实现该类型的成员。

封装隐藏了内部元素的实现细节，提供了优点：避免类内部出现无意的可能破坏对象状态的用户级错误；在修改类的实现时不需要修改用户级代码，这些都很有用。

12.8 将 sales\_item::avg\_price 定义为内联函数。

```
inline double sales_item::avr_price() const
{
```

```

        if ( units_sole )
            return revenue/units_sold;
        else return 0;
    }

```

12.9 修改本节中给出的 screen 类，给出一个构造函数，根据屏幕的高度、宽度和内容的值来创建 screen。

```

class Screen
{
public:
    typedef std::string::size_type index;
    Screen( index hei, index wid , string content )
    {
        contents = content;
        height = hei;
        width = wid;
    }
private:
    std::string contents;
    index cursor;
    index height, width;
};

```

12.10 解释下述类中的每个成员：

```

class Record {
    typedef std::size_t size;
    Record(): byte_count(0) {}
    Record(size s): byte_count(s) {}
    Record(std::string s): name(s), byte_count(0)
}

    size byte_count;
    std::string name;

public:
    size    get_count()    const    {    return
byte_count; }
    std::string    get_name()    const    {    return
name; }
};

```

三个 Record() 函数是重载的三个构造函数，size byte\_count; std::string name; 这是两个 private 的数据成员，size get\_count() const ， std::string get\_name() const 这是两个 public 成员函数。

12.11 定义两个类 X 和 Y，X 中有一个指向 Y 的指针，Y 中有一个 X 类型的对象。

```

class Y;
class X
{
    Y *p;
};
class Y {
    X xobj;
};

```

12.12 解释类声明与类定义之间的差异。何时使用类声明？何时使用类定义？

类声明是不完全类型，只能以有限方式使用，不能定义该类型的对象，只能用于定义指向该类型的指针及引用，或者声明使用该类型作为形参类型或返回类型的函数。

类定义，一旦类被定义，我们就可以知道所有类的成员，以及存储该类的对象所需的存储空间。

在创建类的对象之前或者使用引用或指针访问类的成员之前必须定义类。

12.13 扩展 screen 类以及包含 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 );
```

// 12.13\_Screen.cpp：定义控制台应用程序的入口点。  
//

```

#include "stdafx.h"
#include <iostream>
#include <string>

```

```
using namespace std;
```

```

class Screen
{
public:
    typedef std::string::size_type index;
    Screen( index hei, index wid , const string
&content = "  ")
    {
        cursor = 0;

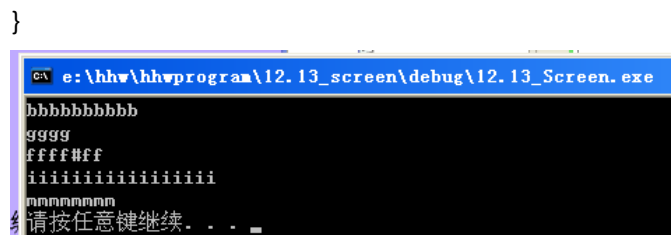
```

```

        contents = content;
        height = hei;
        width = wid;
    }
    // add 3 new members
    Screen & move(index r, index c)
    {
        index row = r * width;
        cursor = row + c;
        return *this;
    }
    Screen & set(char c)
    {
        contents[cursor] = c;
        return *this;
    }
    Screen & display( ostream &os )
    {
        do_display(os);
        return *this;
    }
    const Screen & display(ostream &os) const
    {
        do_display(os);
        return *this;
    }
private:
    std::string contents;
    index cursor;
    index height, width;
    void do_display(ostream & os) const
    {
        os << contents;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    Screen myscreen( 5, 5,
"bbbbbbbbbb\ngggg\nffffff\niiiiiiiiiiii\nmmmmmm\n");
    myscreen.move(4,0).set('#').display(cout);
    system("pause");
    return 0;
}

```



12.14 通过 this 指针引用成员虽然合法，但却是多余的。讨论显示地使用 this 指针访问成员的优缺点。

优点：当需要将一个对象作为整体引用而不是引用对象的一个成员时，使用 this，则该函数返回对调用该函数的对象的引用。

可以非常明确地指出访问的是调用该函数的对象的成员，且可以在成员函数中使用与数据成员同名的形参。

缺点：不必要使用，代码多余。

12.15 列出在类作用域中的程序文本部分。

类的定义体；在类外定义的成员函数：形参表、成员函数体，但不包含函数的返回类型。

12.16 如果定义 get\_cursor 时，将会发生什么？

```
Index Screen::get_cursor() const { return
cursor; }
```

编译错误，index 是在 Screen 类的作用域之外的，在类外没有定义 index 类型。

12.17 如果将 Screen 类中的类型别名放到类中的最后一行，将会发生什么？

发生编译错误，因为 Index 的使用出现在了其定义之前。

12.18 解释下述代码，指出每次使用 Type 或 initVal 时用到的是哪个名字定义。如果存在错误，说明如何改正。

```
Typedef string type;
Type initVal();
```

```
class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal(Type);
}
```

```

Type initVal();
private:
    int val;
};
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}

```

成员函数 `setVal` 的定义有错。进行必要的修改以便类 `Exercise` 使用全局的类型别名 `Type` 和全局函数 `initVal`。

答：在类的定义体内，两个成员函数的返回类型和 `setVal` 的形参使用的 `Type` 都是类内部定义的类型别名 `Type`；

在类外的成员函数 `setVal` 的定义体内，形参的类型 `Type` 是 `Exercise` 类内定义的类型别名 `Type`，返回类型的 `Type` 使用的是全局的类型别名 `Type`。

在 `setVal` 的定义体内使用的 `initVal` 函数，使用的是 `Exercise` 类的成员函数。

在 `setVal` 的函数定义有错：1. 缺少返回类型；2. 此函数的返回类型是 `string` 类型的，而类中声明的函数返回类型是 `double` 类型的，不合法的函数重载，可以改为：

```

Exercise::Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}

```

把 `setVal` 修改为如下，可保证类 `Exercise` 使用全局的类型别名 `Type` 和全局函数 `initVal`。

```

typedef string type;
Type initVal();
class Exercise {
public:
    // ...
Type setVal(Type);
private:
    Type val;
};
Type Exercise::setVal(Type parm)
{
    val = parm + initVal();
    return val;
}

```

12.1 9 提供一个或多个构造函数，允许该类的用户不指定数据成员的初始值或指定所有数据成员的初始值：

```

class NoName {
public:
    // constructor(s) go here ...
private:
    std::string *pstring;
    int ival;
    double dval;
};

```

解释如何确定需要多少个构造函数以及它们应该接受什么样的形参。

实参决定调用哪个构造函数，所以实参的种类数决定构造函数的数目，因此此题至少需要两个构造函数，一个是默认没有参数的构造函数，一个是带有三个形参的构造函数，类型要与三个数据成员的类型匹配。

构造函数：

```

class NoName {
public:
    // constructor(s) go here ...
    NoName() {}
    NoName( std::string *ps, int iv, double dv )
    {
        pstring = ps;
        ival = iv;
        dval = dv;
    }
private:
    std::string *pstring;
    int ival;
    double dval;
};

```

12.20 从下述抽象中选择一个（或一个自己定义的抽象），确定类中需要什么数据，并提供适当的构造函数集。解释你的决定：

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

(b) Date: 需要年: `int year`; 月: `int month`; 日: `int day`



构造两个构造函数，一个默认构造函数，用于创建空的 Date 对象，一个带有三个形参，用于创建指定日期的对象。

构造函数为：

```
class Date
{
public:
    Date() {}
    Date( int y, int m, int d )
    {
        year = y;    month = m;    day = d;
    }
private:
    int year, month, day;
};
```

12.21 使用构造函数初始化列表编写类的默认构造函数，该类包含如下成员：一个 const string，一个 int，一个 double\* 和一个 ifstream &。初始化 string 来保存类的名字。

```
class X
{
public:
    X( ): cStr(“ class name” ), iVal(0), pd(0) {}
private:
    const string cStr;
    int iVal;
    double *pd;
    ifstream & iFs;
};
```

12.22 下面的初始化式有错误。找出并改正错误。

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

struct X 的定义中，rem 先被定义，先被初始化，在构造函数初始化列表中的初始化顺序是先初始化 rem，而此时 rem 用还没有被初始化的 base 来初始化 rem，会产生 runtime error.

改正为：

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

12.23 假定有个命名为 NoDefault 的类，该类有一个接受一个 int 的构造函数，但没有默认构造函数。定义有一个 NoDefault 类型成员的类 C。为类 C 定义默认构造函数。

```
class c {
public:
    c() : i(0), Ndf(i) {}
private:
    int i;
    NoDefault Ndf;
};
```

12.24 上面的 Sales\_item 定义了两个构造函数，其中之一有一个默认实参对应其单个 string 形参，使用该 Sales\_item 版本，确定用哪个构造函数来初始化下述的每个变量，并列对象中数据成员的值。

Sales\_item first\_item(cin);

用的是第二个即：Sales\_item( std::istream &is ); 对象的数据成员的值：isbn 的值为 cin 输入的字符串。

```
int main() {
    Sales_item next;
    Sales_item last ( “9-999-99999-9”);
}
```

这两个用的是第一个即：有默认实参的构造函数，next 对象的数据成员的值：isbn 空串，units\_sold 值为 0，revenue 值为 0.0。

last 对象的数据成员的值：isbn 为“9-999-99999-9”，units\_sold 值为 0，revenue 值为 0.0。

12.25 逻辑上讲，我们可能希望将 cin 作为默认实参提供给接受一个 istream& 形参的构造函数。编写使用 cin 作为默认实参的构造函数声明。

Sales\_item( std::istream &is = std::cin );

12.26 接受一个 string 和接受一个 istream& 的构造函数都具有默认实参是合法的吗？如果不是，为什么？

么？

不合法，如果二者都具有默认实参，则造成了默认构造函数的重复定义，当定义一个 `Sales_item` 的对象而没有给出用于构造函数的实参时，将因为无法确定使用哪个构造函数而出错。

12.27 下面的陈述中哪个是不正确的（如果有的话）？为什么？

- (a) 类必须提供至少一个构造函数。
- (b) 默认构造函数的形参列表中没有形参。
- (c) 如果一个类没有有意义的默认值，则该类不应该提供默认构造函数。

(d) 如果一个类没有定义默认构造函数，则编译器会自动生成一个，同时将每个数据成员初始化为相关类型的默认值。

(a) 不提供构造函数时，由编译器合成一个默认构造函数。

(b) 错了，为所有形参提供了默认实参的构造函数也定义了默认构造函数，而这样的构造函数形参列表中有形参的。

(c) 不正确，如果一个类没有默认的构造函数，而只有自己写的构造函数，在编译器需要隐式使用默认构造函数时，这个类就不能使用，编译不通过，所以如果一个类定义了其他的构造函数，通常也应该提供一个默认的构造函数。

(d) 不正确，1，当一个类中自己定义了构造函数时，编译器就不会自动合成一个默认构造函数了，2，编译器不是将每个数据成员初始化为相关类型的默认值，而是使用与变量初始化相同的规则来初始化成员：类类型的成员执行各自的默认构造函数进行初始化，内置和复合类型的成员，只对定义在全局作用域中的对象才初始化。

12.28 解释一下接受一个 `string` 的 `Sales_item` 构造函数是否应该为 `explicit`。将构造函数设置为 `explicit` 的好处是什么？缺点是什么？

应该声明为 `explicit`，如果不声明为 `explicit`，编译器可以使用此构造函数进行隐式类型转换，即将一个 `string` 类型的对象转换为 `Sales_item` 对象，这种行为时容易发生语义错误的。

好处是：可以防止构造函数的隐式类型转换而

带来的错误，缺点是，当用户确实需要进行相应的类型转换时，不能依靠隐式类型转换，必须显示地创建临时对象。

12.29 解释在下面的定义中所发生的操作。

```
string null_isbn = "9-999-99999-9";  
Sales_item null1(null_isbn);  
Sales_item null("9-999-99999-9");
```

1，调用接收一个 C 风格字符串形参的 `string` 构造函数，创建一个临时的 `string` 对象，然后调用 `string` 类的复制构造函数，将 `null_isbn` 初始化为该该临时对象的副本。

2，使用 `string` 的对象 `null_isbn` 为实参，调用 `Sales_item` 类的构造函数创建 `Sales_item` 对象 `null1`。

3，使用接受一个 C 风格字符串形参的 `string` 类的构造函数，生成一个临时 `string` 对象，然后用这个临时对象作为实参，调用 `Sales_item` 类的构造函数来创建 `Sales_item` 类的对象 `null`。

12.30 编译如下代码：

```
f(const vector<int>&);  
int main() {  
    vector<int> v2;  
    f(v2); // should be ok  
    f(42); // should be an error  
    return 0;  
}
```

基于对 `f` 的第二个调用中出现的错误，我们可以对 `vector` 构造函数做出什么推断？如果该调用成功了，那么你能得出什么结论？

可以做出以下推断：`vector` 中没有定义接受一个 `int` 型参数的构造函数，或者即使定义了接受一个 `int` 型参数的构造函数，该函数也被声明为了 `explicit`。如果调用成功了，则说明 `vector` 中定义了可以接受一个 `int` 型参数的构造函数。

12.31 `pair` 的数据成员为 `Public`，然而下面这段代码却不能编译，为什么？

```
pair< int, int > p2 = (0, 42); // doesn't compile,  
why?
```

可能是因为 `pair` 类定义了构造函数，此时只能使用构造函数初始化成员，可改为：

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

### 12.32 什么是友元函数？什么是友元类？

被指定为某类的友元的函数称为该类的友元函数。  
被指定为某类的友元的类称为授予友元关系的那个类的友元类。

### 12.33 什么时候友元是有用的？讨论使用友元的优缺点。

在需要允许某些特定的非成员函数访问一个类的私有成员，而同时仍阻止一般的访问的情况下，友元机制是个有用的东西。

优点：可以灵活地实现需要访问若干类的私有或受保护成员才能完成的任务，便于与其他不支持类的语言进行混合编程；通过使用友元函数重载可以更自然地使用 C++ 语言的 I/O 流库。

缺点：一个类将对非公有成员的访问权授予其他的函数或类，会破坏该类的封装性，降低该类的可靠性和可维护性。

### 12.34 定义一个将两个 Sales\_item 对象相加的非成员函数。

首先将此函数定义：

```
Sales_item add2obj( 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;
}
```

然后再类 Sales\_item 中将此函数声明为友元：

```
friend Sales_item add2obj( const Sales_item&, const
sales_item&);
```

### 12.35 定义一个非成员函数，读取一个 istream 并将读入的内容存储到一个 Sales\_item 中。

```
stream& read( istream &is, Sales_item &obj )
```

```
{
    double price;
    is >> obj.isbn >> obj.units_sold >> price;
    if ( is )
        obj.revenue = obj.units_sold * price;
    return is;
}
```

然后将该函数指定为 Sales\_item 的友元。

### 12.36 什么是 static 类成员？static 成员的优点是什么？它们与普通成员有什么不同？

类成员声明前有关键字 static 的类成员，static 类成员不是任意对象的组成部分，但是由该类的全体对象所共享。

优点：1，static 成员的名字是在类的作用域中，因此可以避免与其他类的成员或全局对象名字冲突；2，可以实施封装，static 成员可以是私有成员，而全局对象不可以。3，通过阅读程序看出 static 成员是与特定的类关联的，这种可见性可清晰地显示程序员的意图。

不同点：普通成员是与对象相关联的，是某个对象的组成部分，而 static 成员与类相关联，由该类的全体对象所共享，不是任意对象的组成部分。

### 12.37 编写自己的 Account 类版本。

```
class Account
{
public:
    Account( string own, double am )
    {
        owner = own;
        amount = am;
    }

    void applyint()
    {
        amount += amount*interestRate;
    }

    static double rate()
    {
        return interestRate;
    }
}
```

```

        static void rate(double newRate)
        {
            interestRate = newRate;
        }
private:
    std::string owner;
    double amount;
    static double interestRate;
};
double Account::interestRate = 0.016;

```

12.38 定义一个命名为 Foo 的类，具有单个 int 型数据成员。为该类定义一个构造函数，接受一个 int 值并用该值初始化数据成员。为该类定义一个函数，返回其数据成员的值。

```

class Foo
{
private:
    int iVal;
public:
    Foo( int iV ): iVal( iV ) { }
    int get_val()
    {
        return iVal;
    }
};

```

12.39 给定上题中定义的 Foo 类定义另一个 Bar 类。Bar 类具有两个 static 数据成员：一个为 int 型，另一个为 Foo 类型。

```

class Bar
{
private:
    static int i;
    static Foo f;
};

```

12.40 使用上面两题中定义的类，给 Bar 类增加一对成员函数：第一个成员命名为 FooVal，返回 Bar 类的 Foo 类型 static 成员的值；第二个成员命名为 callsFooVal，保存 xval 被调用的次数。

```

class Bar

```

```

{
public:
    Foo FooVal()
    {
        callsFooVal++;
        return f;
    }
private:
    static int callsFooVal;
    static int i;
    static Foo f;
};

```

对 static 成员进行初始化：

```

int Bar::i = 42;
Foo Bar::f( 0 );
int Bar::callsFooVal = 0;

```

12.41 利用 12.6.1 节的习题中编写的类 Foo 和 Bar，初始化 Foo 的 static 成员。将 int 成员初始化为 20，并将 Foo 成员初始化为 0

```

int Bar::i = 20;
Foo Bar::f( 0 );

```

12.42 下面的 static 数据成员声明和定义哪些是错误的（如果有的话）？解释为什么。

```

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

```

```

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

```

因为非 const static 成员的初始化必须放在类定义体的外部。

下边对 static 成员 rate 和 vec 的定义也是错误的，因

为此处必须给出初始值。

更正为：

```
class Example {
public:
    static double rate;
    static const int vecSize = 20;
    static vector<double> vec;
};

// example C
#include "example.h"
double Example::rate= 6.5;
vector<double> Example::vec(vecSize);
```

## 第十三章 复制控制

### 1. 什么是复制构造函数？何时使用它？

只有单个形参，而且该形参是对本类类型对象的引用（常用 `const` 修饰），这样的构造函数叫复制构造函数。

copy constructor will be used under these situations:

根据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对象；

根据元素初始化式列表初始化数组元素。

### 2. 下面第二个初始化不能编译。可以从 `vector` 的定义得出什么推断？

```
vector<int> v1(42);
vector<int> v2 = 42;
```

若能编译成功说明，这是个复制初始化，创建 `v2` 时，首先调用接受一个 `int` 型形参的 `vector` 构造函数，创建一个临时 `vector` 对象，然后再调用复制构造函数用这个临时对象来初始化 `v2`。现在不能编译，说明 `vector` 没有定义复制构造函数。

### 3. 假定 `Point` 为类类型，该类类型有一个复制构造函数，指出下面程序段中每一个使用了复制构造函数的地方：

```
Point global;
Point foo_bar( Point arg ) // 调用此函数时，将实参
对象的副本传递给形参 Point 的对象 arg
{
    Point local = arg; // 调用复制构造函数，将局
部对象 local 初始化为形参 arg 的副本。
    Point *heap = new Point( global ); // 调用复
制构造函数用全局对象 global 来初始化
// Point 对象
*heap
    *heap = local;
    Point pa[4] = { local, *heap }; // 使用数组初
始化列表来初始化数组的每个元素。
    return *heap; // 从函数返回 Point 对象
*heap 的副本
}
```

### 4. 对于如下的类的简单定义，编写一个复制构造函数所有成员。复制 `pstring` 指向的对象而不是复制指针。

```
struct NoName {
    NoName(): pstring( new std::string ), i(0), d(0)
    {}
private:
    std::string *pstring;
    int i;
    double d;
};

NoName( const NoName& orig ) : pstring(new
string(*(orig.pstring))), i(orig.i), d(orig.d)
{
    // pstring = new string;
    // *pstring = *(orig.pstring);
}
```

### 5. 哪个类定义可能需要一个复制构造函数？

- (a) 包含四个 `float` 成员的 `Point3w` 类。
- (b) `Matrix` 类，其中，实际矩阵在构造函数中动态分配，在析构函数中删除。
- (c) `Payroll` 类，在这个类中为每个对象提供唯一 ID。
- (d) `Word` 类，包含一个 `string` 和一个以行列位置对为元素的 `vector`。



(b)需要, 涉及到指针及动态分配

(c)需要, 在根据已存在的 Payroll 对象创建其副本时, 需要提供唯一的 ID.

其他的均可以调用编译器的提供的复制构造函数, 或者调用类类型的 string 和 vector 的复制构造函数。

6.复制构造函数的形参并不限制为 const, 但必须是一个引用, 解释这个限制的基本原理, 例如, 解释为什么下面的定义不能工作,

```
Sales_item::Sales_item( const Sales_item rhs );
```

它不能工作的原因是: 当形参为非引用类型时, 将复制实参的值, 给这个 copy constructor,但是, 每当以传值方式传递参数时, 会导致调用复制构造函数, 因此, 如果要使用以传值方式传递参数的 copy constructor, 必须使用一个“不以传值方式传递参数”的 copy constructor, 否则就会导致 copy constructor 的无穷递归调用。这个“不以传值方式传递参数”的方法就是使用形参是一个引用的 copy constructor, 即以传地址的方式传递参数。

7.类何时需要定义复制操作符?

在需要定义复制构造函数时, 也需要定义赋值操作符, 即如果一个类 (1) 类中包含指针型数据成员, (2) 或者在进行赋值操作时需要做一些特定工作, 则该类需要定义赋值操作符。

8.对于习题 13.5 中列出的每个类型, 指出类是否需要赋值操作符。

(b)需要赋值操作符, 因为涉及指针和动态分配内存;  
(c) 需要, 在用已存在的 Payroll 对象给另一个 Payroll 对象赋值时, 需要提供唯一的 ID。

9.习题 13.4 中包括 NoName 类的简单定义, 确定这个类是否需要赋值操作符, 如果需要, 实现它。因为有指针类的数据成员, 所以需要赋值操作符, 实现为:

```
NoName& operator=( const NoName &Nn )
{
    i (Nn.i);
    d(Nn.d);
    pstring = new string;
    *pstring = *( Nn.pstring );
    return *this;
```

```

}
10.定义一个 Employee 类, 包含雇员名字和一个唯一的雇员标识, 为该类定义默认构造函数和参数为表示雇员名字的 string 构造函数。如果该类需要复制构造函数或赋值操作符, 实现这些函数。
```

```
class Employee
{
public:
    Employee(): ID(cnt) { cnt++; } // 默认构造函数
    Employee( const std::string &na ): name( na ), ID(cnt)
    { cnt++; } // 构造函数

    // 拷贝构造函数
    Employee( const Employee & rhs ) : name( rhs.name ), ID(cnt)
    { cnt++; }
    // 赋值操作符
    Employee & operator=( const Employee & rhs )
    {
        name = rhs.name;
        return *this;
    }
private:
    string name;
    int ID;
    static int cnt;
};
另外需要在类外对 static 成员进行初始化:
int Employee::cnt =1 ;
```

11.什么是析构函数? 合成析构函数有什么用? 什么时候会合成析构函数? 什么时候一个类必须定义自己的析构函数?

析构函数是一个成员函数, 它的名字与类的名字相同, 在名字前加一个代字符~, 没有返回值, 没有形参, 用于类的对象超出作用域时释放对象所获取的资源, 或删除指向动态分配对象的指针。

合成析构函数的作用: 1, 按对象创建时的逆序撤销每个非 static 成员, 2, 对于类类型的成员, 合成析构函数调用该成员的析构函数来撤销对象。

编译器总会为每个类合成一个析构函数。

当 1，需要释放指针成员的资源时，2，需要执行某些特定工作时，必须自己定义析构函数。

12. 确定在习题 13.4 中概略定义的 NoName 类是否需要析构函数，如果需要，实现它。

根据“三法则”需要显式定义析构函数，实现为：

NoName::~NoName

```
{
    delete pstring;
}
```

13. 理解复制控制成员和构造函数的一个良好方式是定义一个简单类，该类具有这些成员，每个成员打印自己的名字：

```
struct Exmp1 {
    Exmp1() { std::cout << "Exmp1()" <<
std::endl; }
    Exmp1( const Exmp1& )
    { std::cout << "Exmp1( const Exmp1& )" <<
std::endl; }
    // ...
};
```

编写一个像 Exmp1 这样的类，给出复制控制成员和其他构造函数。然后写一个程序，用不同方式使用 Exmp1 类型的对象：作为非引用和引用形参传递，动态分配，放在容器中，等等，研究何时执行哪个构造函数和复制控制成员，可以帮助你融会贯通第理解这些概念。

// 13.14\_CopyControlMember.cpp：定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <vector>
class Exmp
{
public:
    // constructor
    Exmp()
```

```
{
    std::cout << " Using Exmp(). " << std::endl;
}
//
// copy constructor
Exmp(const Exmp&)
{
    std::cout << " Using Exmp(const Exmp&)
_copy constructor." << std::endl;
}
// overload operator
Exmp& operator=( const Exmp&)
{
    std::cout << " Using Exmp&
operator=( const Exmp&) _overload operator." <<
std::endl;
    return *this;
}

// destructor
~Exmp()
{
    std::cout << " Using ~Exmp()." << std::endl;
}

};

void func1(Exmp obj)    // 形参为Exmp的对象
{
}
void func2(Exmp & obj) // 形参为Exmp对象的引用
{
}
Exmp func3()
{
    Exmp obj;
    return obj;        // 返回exmp
对象
}

int _tmain(int argc, _TCHAR* argv[])
{
    Exmp    exmp1;
    Exmp exmp2(exmp1);
```



```

    exmp2 = exmp2;
    func1( exmp1);

    func2( exmp1);

    exmp1 = func3();

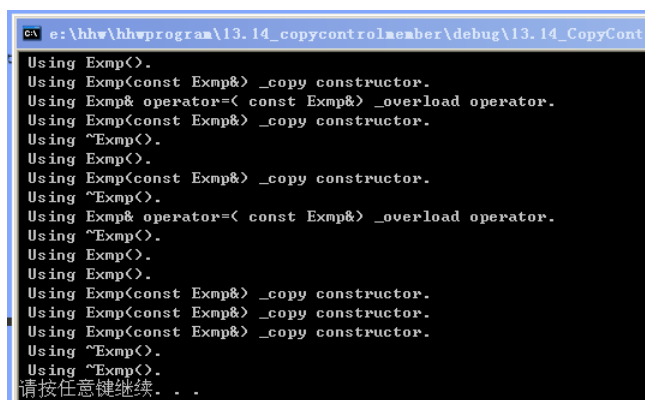
    Exmp *p = new Exmp;

    std::vector<Exmp> evec(3);

    delete p;

    system("pause");
    return 0;
}

```



```

e:\hhw\hhwprogram\13.14_copycontrolmember\debug\13.14_CopyCont
Using Exmp().
Using Exmp(const Exmp&)_copy constructor.
Using Exmp& operator=(const Exmp&)_overload operator.
Using Exmp(const Exmp&)_copy constructor.
Using ~Exmp().
Using Exmp().
Using Exmp(const Exmp&)_copy constructor.
Using ~Exmp().
Using Exmp& operator=(const Exmp&)_overload operator.
Using ~Exmp().
Using Exmp().
Using Exmp().
Using Exmp(const Exmp&)_copy constructor.
Using Exmp(const Exmp&)_copy constructor.
Using Exmp(const Exmp&)_copy constructor.
Using ~Exmp().
Using ~Exmp().
请按任意键继续...

```

15. 下面的代码中发生了多少次析构函数的调用?

```

void fcn( const Sales_item *trans, Sales_item accm )
{
    Sales_item item1(*trans), item2(accm);
    if (!item1.same_isbn(item2)) return;
    if (item1.avg_price() <= 99) return;
    else if (item2.avg_price() <= 99) return;
    // ...
}

```

3 次, 分别用在当函数执行完毕后的撤销非 static 的

形参对象 accm 和 局部对象 item1,item2.

16. 编写本节中描述的 Message 类。

```

class Message
{
public:
    Message(const std::string &str = ""): contents (str)
    {}

    Message(const Message&);
    Message& operator=(const Message&);
    ~Message();

    void save(Folder &);
    void remove(Folder &);

    void addFldr(Folder*);
    void remFldr(Folder*);
private:
    std::string contents;
    std::set<Folder*> folders;

    void put_Msg_in_Folders(const
std::set<Folder*>&);
    void remove_Msg_from_Folders();
};

Message::Message(const Message &m) :
contents(m.contents), folders(m.folders)
{
    put_Msg_in_Folders(folders);
}

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);
    }
}

```

```

Message& Message::operator =(const Message &rhs )
{
    if (&rhs != this)
    {
        remove_Msg_from_Folders();
        contents = rhs.contents;
        folders = rhs.folders;
        put_Msg_in_Folders(rhs.folders);
    }
    return *this;
}

void Message::remove_Msg_from_Folders()
{
    for ( std::set<Folder*>::const_iterator beg =
folders.begin(); beg != folders.end(); ++beg )
    {
        (*beg)->remMsg(this);
    }
}

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

void Message::save(Folder &fldr)
{
    addFldr(&fldr);
    fldr.remMsg(this);
}

void Message::remove(Folder &fldr)
{
    remFldr(&fldr);
    fldr.remMsg(this);
}

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

void Message::remFldr(Folder* fldr)

```

```

{
    folders.erase(fldr);
}

17.为 message 类增加与 Folder 的 addMsg 和 remMsg
操作类似的函数。这些函数可以命名为 addFldr 和
remFldr,应接受一个指向 Folder 的指针并将该指针
插入到 folders。这些函数可为 private 的,因为它们
将仅在 Message 类的实现中使用。

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

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

```

18. 编写相应的 Folder 类。该类应保存一个 set<Message\*>, 包含指向 Message 的元素。

```

class Message;
class Folder
{
public:
    Folder() {}
    Folder( const Folder&);
    Folder& operator=(const Folder&);
    ~Folder();

    void save( Message &);
    void remove(Message&);

    void addMsg( Message &);
    void remMsg( Message &);

private:
    std::set<Message*> messages;
    void put_Fldr_In_Messages(const
std::set<Message*>&);
    void remove_Fldr_Messages();
};

Folder::Folder( const Folder

```

```

&f):messages(f.messages)
{
    put_Fldr_In_Messages(messages);
}
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);
}
Folder& Folder::operator =(const Folder &rhs)
{
    if (&rhs != this )
    {
        remove_Fldr_Messages();
        messages = rhs.messages;
        put_Fldr_In_Messages(rhs.messages);
    }
    return *this;
}

void Folder::remove_Fldr_Messages()
{
    for (std::set<Message*>::const_iterator beg =
messages.begin();
        beg != messages.end(); ++beg )
    {
        (*beg)->remFldr(this);
    }
}

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

void Folder::save(Message & msg)
{
    addMsg(&msg);
    msg.addFldr(this);
}
void Folder::remove(Message& msg)
{

```

```

remMsg(&msg);
msg.remFldr(this);
}

void Folder::addMsg(Message & msg)
{
    messages.insert(msg);
}
void Folder::remMsg(Message & msg)
{
    messages.erase(msg);
}

```

19.在 Message 类和 Folder 类中增加 save 和 remove 操作，这些操作应接受一个 Folder,并将该 Folder 加入到指向这个 Message 的 Folder 集中（或从其中删除该 Folder）。操作还必须更新 Folder 以反映它指向该 Message，这可以通过调用 addMsg 或 remMsg 完成。

```

void Message::save(Folder &fldr)
{
    addFldr(&fldr);
    fldr.remMsg(this);
}

void Message::remove(Folder &fldr)
{
    remFldr(&fldr);
    fldr.remMsg(this);
}

```

20.对于 HasPtr 类的原始版本(依赖于复制控制的默认定义)，描述下面代码中会发生什么：

```

int i =42;
HasPtr p1( &i, 42);
HasPtr p2 = p1;
cout << p2.get_ptr_val() << endl;  输出 42
p1.set_ptr_val(0);
cout << p2.get_ptr_val() << endl;  输出 0

```

21.如果给 HasPtr 类添加一个析构函数,用来删除指

针成员，会发生什么？

如果这样，则撤销一个 `HasPtr` 对象时会删除其指针成员所指向的对象，从而使得当一个 `HasPtr` 对象被撤销后，其他由该对象复制而创建的 `HasPtr` 对象中的指针成员也无法使用。

## 22. 什么是使用计数？

使用计数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，使用计数减少 1。赋值操作符和析构函数检查使用计数是否已减至 0，若是，则撤销对象。

## 23. 什么是智能指针？智能指针如何与实现普通指针行为的类相区别？

智能指针式一个行为类似指针但也提供其他功能的类。这个类与实现普通指针行为的类区别在于：智能指针通常接受指向动态分配对象的指针并负责删除该对象。用户分配对象，但由智能指针类删除它，因此智能指针类需要实现赋值控制成员来管理指向共享对象的指针。只有在撤销了指向共享对象的最后一个智能指针后，才能删除该共享对象。使用计数就是实现智能指针类最常用的一个方式。

## 24. 实现你自己的使用计数式 `HasPtr` 类的版本。

```
class U_Ptr
{
    friend class HasPtr;
    int *ip;
    size_t use;
    U_Ptr(int *p): ip(p), use(1) { }
    ~U_Ptr() { delete ip;
```

```
};
class HasPtr
{
public:
    HasPtr(int *p, int i): ptr(new U_Ptr(p), val(i)
    { }
    HasPtr(const HasPtr &rhs): ptr(rhs.ptr),
    val(rhs.val)
    {
        ++ptr->use;
    }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr()
    {
        if (--ptr->use == 0) delete ptr;
    }

    int *get_ptr() const { return ptr->ip; }
    int get_int() const { return val; }

    void set_ptr(int *p) { ptr->ip = p; }
    void set_int(int i) { val = i; }

    int get_ptr_val() const { return *ptr->ip; }
    void set_ptr_val(int i) { *ptr->ip = i; }
private:
    U_Ptr *ptr;
    int val;
};
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++rhs.ptr->use;
    if (--ptr->use == 0)
        delete ptr;
    ptr = rhs.ptr;
    val = rhs.val;
    return *this;
}
```

## 25. 什么是值型类？

是指具有值语义的类，其特征是：对该类对象进行赋值时，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。

26.实现自己的值型 HasPtr 类版本。

```
class HasPtr
{
public:
    HasPtr( const int &p, int i): ptr(new int(p)), val(i)
    { }

    HasPtr( const HasPtr &rhs ): ptr( new int
(*rhs.ptr)), val(rhs.val) { }

    HasPtr& operator=( const HasPtr&rhs)
    {
        *ptr = *rhs.ptr;
        val = rhs.val;
        return *this;
    }
    ~HasPtr() { delete ptr; }
    int *get_ptr() const { return ptr; }

    int get_ptr_val() const { return *ptr; }
    int get_int() const { return val; }

    void set_ptr( int *p ) { ptr = p; }
    void set_int( int i ) { val = i; }

    void set_ptr_val( int p ) const
    {
        *ptr = p;
    }
private:
    int *ptr;
    int val;
};
```

27.值型 HasPtr 类定义了所有复制控制成员。描述将会发生什么，如果该类：

(a) 定义了赋值构造函数和析构函数但没有定义赋值操作符。

(b) 定义了赋值构造函数和复制操作符但没有定义析构函数。

(c) 定义了析构函数但没有定义复制构造函数和赋值操作符。

(a)此种情况下，使用编译器合成的赋值操作符。因此若将一个 HasPtr 对象赋值给另一个 HasPtr 对象，则两个对象的 ptr 成员值相同，即二者指向同一基础 int 对象，当其中一个对象被撤销后，该基础 int 对象也被撤销，使得另一个对象中的 ptr 成员指向一个失效的基础 int 对象。而且当另一个对象被撤销时，会因为析构函数对同一指针进行重复删除而出现内存访问非法的错误。此外，被赋值的 HasPtr 对象的 ptr 成员原来所指向的基础 int 对象不能再被访问，但也没有被撤销。

(b) 此时使用编译器合成的析构函数，当一个对象被撤销后，其成员所指向的基础 int 对象不会被撤销，会一直存在，从而导致内存泄露。

(c) 此时，使用编译器合成的复制构造函数和赋值操作符。当将一个对象赋值给另一个 HasPtr 对象时或复制构造另一个 HasPtr 对象时，则两个 HasPtr 对象的 ptr 成员值相同，其他出现的问题与(a)的情况相同。

28.对于如下的类，实现默认构造函数和必要的复制控制成员。

```
(a) class TreeNode
{
public:
    // ...
private:
    std::string value;
    int count;
    TreeNode *left;
    TreeNode *right;
};
```

```
(a)
TreeNode::TreeNode() : count(0), left(0), right(0)
{ }
TreeNode::TreeNode( const TreeNode &rhs ) :
value( rhs.value)
{
    count = rhs.count;
```

```

    if ( rhs.left )
        left = new TreeNode( *rhs.left);
    else
        left = 0;
    if ( rhs.right )
        right = new TreeNode( *rhs.right);
    else
        right = 0;
}
TreeNode::~TreeNode()
{
    if (left)
        delete left;
    if ( right )
        delete right;
}

(b) class BinStrTree
{
public:
    // ...
private:
    TreeNode *root;
};

(b)
BinStrTree:: BinStrTree(): root(0) { }
BinStrTree:: BinStrTree( const BinStrTree & rhs )
{
    if ( rhs.root )
        root = new TreeNode(*rhs.root);
    else
        root = 0;
}

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

```

## 第十四章 重载操作符与转换

1. 在什么情况下重载操作符与内置操作符不同？在什么情况下重载操作符与内置操作符相同？

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，例如对 `&&` 和 `||` 的重载版本不再具有“短路求值”的特性，两个操作数都要进行求值，而且不规定操作数的求值顺序。

对于优先级和结合性及操作数的数目都不变。

2. 为 `Sales_item` 编写输入、输出。加以及复合赋值操作符的重载声明。

```

class Sales_item
{
    friend std::istream& operator >> ( std::istream&,
    Sales_item& );
    friend std::ostream& operator <<(std::ostream&,
    const Sales_item&);
public:
    Sales_item& operator += ( const Sales_item& );
};
Sales_item operator+ ( const Sales_item&, const
Sales_item& )

```

3. 解释如下程序，假定 `Sales_item` 构造函数的参数是一个 `string`，且不为 `explicit`。解释如果构造函数为 `explicit` 会怎样。

```
string null_book = "0-000-00000-0";
```

```
Sales_item item(cin);
```

```
item += null_book;
```

第一句：调用接受一个 C 风格的字符串形参的 `string` 构造函数，创建一个 `string` 临时对象，然后使用 `string` 复制构造函数用这个临时对象初始化 `string` 对象 `null_book`，

第二句：从标准输入设备读入数据，创建 `Sales_item` 对象 `item`。

第三句：首先调用接受一个 `string` 参数的 `Sales_item` 构造函数，创建一个临时对象，然后调用 `Sales_item` 的复合重载操作符 `+=`，将这个 `Sales_item` 临时对象加到 `item` 对象上，

如果构造函数为 `explicit`，则不能进行从 `string` 对象到 `Sales_item` 对象的隐式转换，第三句将不能被编译。

4. `string` 和 `vector` 类都定义了一个重载的 `==`，可用于比较这些类的对象，指出下面的表达式中应用了哪个 `==` 版本。

```
string s; vector<string> svec1, svec2;
"cobble" == "store" 应用了 C++语言内置版本的重载==
svec1[0] == svec2[0]; 应用了 string 版本的重载==
svec1 == svec2        应用了 vector 版本的重载==
```

5. 列出必须定义为类成员的操作符。

赋值 `=`，下标 `[]`，调用 `()`，成员访问箭头 `->`

6. 解释下面操作符是否应该为类成员，为什么？

(a) `+` (b) `+=` (c) `++` (d) `->` (e) `<<` (f) `&&` (g) `==` (h) `()`

`+`, `<<`, `==`, `&&` 通常定义为非成员；`->`和`()`必须定义为成员，否则会出现编译错误；`+=` 和 `++` 会改变对象的状态，通常会定义为类成员。

7. 为下面的 `CheckoutRecord` 类定义一个输出操作符：

```
class CheckoutRecord
{
public:
    // ..
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;
    pair<string, string> borrower;
    vector<pair<string, string>* > wait_list;
};

ostream&
operator<< (ostream& out, const CheckoutRecord&
s)
{
    out << s.book_id << "\t" << s.title<< "\t" <<
s.date_borrowed<< "\t" << s.date_due
```

```
    << "\t";
    out<< " borrower:" <<s.borrower.first << ", "<<
s.borrower.second << endl;
    out << " wait_list: " << endl;
    for (vector<
pair<string, string>* >::const_iterator it =
s.wait_list.begin();
        it != s.wait_list.end(); ++it )
    {
        out << "\t" << (*it)->first << ", " <<
(*it)->second << endl;
    }
    return out;
}
```

8. 在 12.4 节的习题中，你编写了下面某个类的框架：(b) `Date`

为所选择的类编写输出操作符。

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

class Date
{
public:
    Date() {}
    Date( int y, int m, int d )
    {
        year = y;    month = m;    day = d;
    }
    friend ostream& operator<< ( ostream&, const
Date& );
private:
    int year, month, day;
};

ostream&
operator<<( ostream& out, const Date& d )
{
    out << " year: " << d.year << "\t"
    << " month: " << d.month << "\t"
    << " day: " << d.day << endl;
}

int main()
{
    Date dt(1988, 12, 01);
```

```

    cout << dt << endl;
    system("pause");
    return 0;
}

```

9. 给定下述输入, 描述 Sales\_item 输入操作符的行为。

- (a) 0-201-99999-9 10 24.95  
 (b) 10 24.95 0-201-99999-9

(a) 将 0-201-99999-9 读入赋给对象的 isbn 成员, 将 10 给 units\_sold 成员, revenue 成员被设置为 249.5  
 (b) 首先将形参对象的 isbn 成员设置为 10, 然后因为输入的数据不符合要求, 导致输入失败, 从而执行 else 语句, 将 Sales\_item 对象复位为空对象, 此时 isbn 为空 string, units\_sold 和 revenue 都为 0

10. 下述 Sales\_item 输入操作符有什么错误?

```

istream& operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}

```

如果将习题 14.9 中的数据作为输入, 将会发生什么?

上述的输入操作符中 缺少了对错误输入情况的判断与处理, 会导致错误的情况发生。

(a) 的输入没有问题, 但是(b)的输入, 将形参 Sales\_item 对象的 ISBN 成员值为 10, units\_sold 和 revenue 成员保持原值不变。

11. 为 14.2.1 节习题中定义的 CheckoutRecord 类定义一个输入操作符, 确保该类操作符处理输入错误。

```

class CheckoutRecord
{
public:
    // ...

```

```

friend istream& operator>>(istream&,
CheckoutRecord& ); // 声明为类的友元
// ...
};
istream& operator>>(istream& in, CheckoutRecord&
c)
{
    cout << "Input bookid(double) and
title(string) :\n";
    in >> c.book_id >> c.title;

    // Input Data data_borrowed and data_due
    cout << "Input data_borrowed (3 ints: year,
month , day) :\n";
    in >> c.date_borrowed; //.year >>
c.date_borrowed.month >> c.date_borrowed.day;
    // Input Data data_due and data_due
    cout << "Input data_due (3 ints: year,
month , day) :\n";
    in >> c.date_due; //.year >>
c.date_due.month >> c.date_due.day;

    // Input the pair<string,string> borrower
    cout << "Input the pair<string,string>
borrower (string) :\n";
    in >> c.borrower.first >> c.borrower.second;
    if ( !in )
    {
        c = CheckoutRecord();
        return in;
    }

    // Input wait_list
    cout << "Input the wait_list (string) :\n";
    c.wait_list.clear();
    while ( in )
    {
        pair<string, string> *ppr = new
pair<string, string>;
        in >> ppr->first >> ppr->second;
        if ( !in )
        {
            delete ppr;
            return in;

```



```

    }
    c.wait_list.push_back( ppr );
}

return in;
}

```

输入错误的情况:

```

C:\WINDOWS\system32\cmd.exe
Input bookid(double) and title(string) :
davy 01
Input data_borrowed (3 ints: year, month , day) :
Input data_due (3 ints: year, month , day) :
Input the pair<string,string> borrower (string) :
bookid:-9.25596e+061 title:
date_borrowed:-858993460/-858993460/-858993460
date_due:-858993460/-858993460/-858993460
borrower:.
wait_list:
请按任意键继续. . .

```

输入正确的情况:

```

C:\e:\hhw\hwp\program\14.7_ostream_operator_overload\debug\14.7_ostream.o...
Input bookid(double) and title(string) :
1 C++Primer
Input data_borrowed (3 ints: year, month , day) :
1800 1 1
Input data_due (3 ints: year, month , day) :
1799 12 31
Input the pair<string,string> borrower (string, string) :
huanghuawei davy
Input the wait_list (string, string) < Ctrl+z to end >:
Lidia Lord
^Z
After assign object1 to object2 :
bookid: 1 title: C++Primer
date_borrowed: 1800/1/1
date_due: 1799/12/31
borrower: huanghuawei,davy
wait_list:
Lidia, Lord
请按任意键继续. . .

```

12. 编写 Sales\_item 操作符, 用+进行实际加法, 而 +=调用+。与本节中操作符的实现方法相比较, 讨论这个方法的缺点。

```

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

```

将下面定义的非成员+=操作符声明为类 Sales\_item 的友元:

```

Sales_item operator+=( Sales_item& lhs, const

```

```

Sales_item& rhs )

```

```

{
    lhs = lhs + rhs;
    return lhs;
}

```

这个方法缺点: 在+=操作中需要创建和撤销一个临时 Sales\_item 对象, 来保存+操作的结果, 没有本节中的方法简单有效。

13. 如果有, 你认为 Sales\_item 还应该有哪些其他的算术操作符? 定义你认为的该类应包含的那些。还应有-操作符, 定义为类的非成员, 相应地还应该有 -= 复合操作符并定义为类的成员。

```

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;
}

```

14. 定义一个赋值操作符, 将 isbn 赋值给 Sales\_item 对象。

```

Sales_item& Sales_item::operator=( const string & s )
{
    sbn = s;
    return *this;
}

```

10. 为 14.2.1 节习题中介绍的 CheckoutRecord 类定义赋值操作符。

主函数中定义了两个 CheckoutRecord 类的对象, 调用了CheckoutRecord 类的 =赋值操作符, 效果如下截图:

```

int _tmain(int argc, _TCHAR* argv[])

```

```

{
    CheckoutRecord c1;
    cin >> c1;
    CheckoutRecord c2;
    c2 = c1;    // 调用了类的赋值操作符=
    std::cout << c2 << std::endl; // 输出对象c2的内容

    system("pause");
    return 0;
}

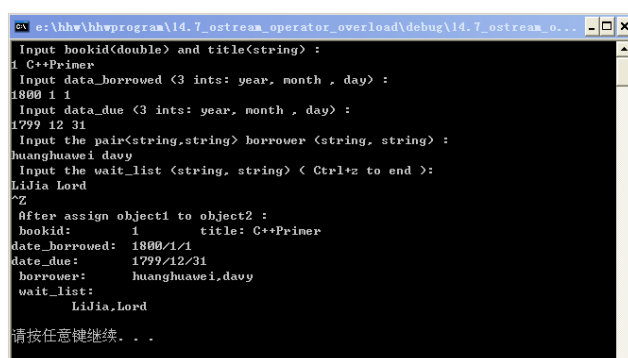
// CheckoutRecord 类中赋值操作符定义为:
// 重载操作符=
CheckoutRecord& CheckoutRecord::operator=(const
CheckoutRecord& cr)
{
    book_id = cr.book_id;
    title = cr.title;
    date_borrowed = cr.date_borrowed; // 前提:
    必须在Date类里也重载操作符=
    date_due = cr.date_due;           // as
    before
    // 对pair进行赋值操作
    borrower.first = cr.borrower.first;
    borrower.second = cr.borrower.second;

    // 对vector进行赋值操作
    wait_list.clear(); // 首先清空
    for (vector<
pair<string,string>*>::const_iterator it =
cr.wait_list.begin();

                                it != cr.wait_list.end(); ++ it )
    {
        pair<string, string> *ppr = new pair<string,
string>;
        ppr->first = (*it)->first;
        ppr->second = (*it)->second;
        wait_list.push_back( ppr );
    }

    return *this;
}

```



```

e:\hhw\hhwprogram\14.7 ostream_operator_overload\debug\14.7 ostream_o...
Input bookid(double) and title(string) :
1 C++Primer
Input date_borrowed <3 ints: year, month, day> :
1800 1 1
Input date_due <3 ints: year, month, day> :
1799 12 31
Input the pair<string,string> borrower <string, string> :
huanghuawei davy
Input the wait_list <string, string> < Ctrl+z to end> :
LiJia Lord
~Z
After assign object1 to object2 :
bookid: 1 title: C++Primer
date_borrowed: 1800/1/1
date_due: 1799/12/31
borrower: huanghuawei,davy
wait_list:
LiJia, Lord
请按任意键继续. . .

```

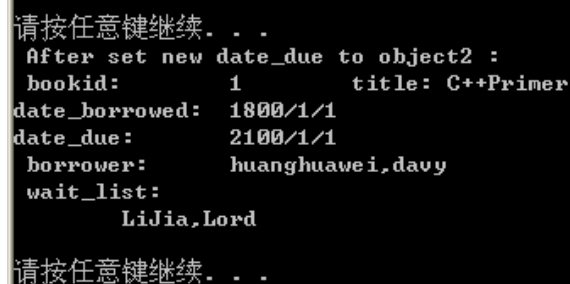
16。CheckoutRecord 类还应该定义其他赋值操作符吗？如果是，解释哪些类型应该用作操作数并解释为什么。为这些类型实现赋值操作符。

从应用角度考虑，可能会修改预约时间 date\_due，可通过设置新的赋值操作符来实现；或者是往 wait\_list 里添加排队读者，也可以通过设置新的赋值操作符来实现。

```

// set new date_due
CheckoutRecord&
CheckoutRecord::operator=(const& new_due)
{
    date_due = new_due;
    return *this;
}

```



```

请按任意键继续. . .
After set new date_due to object2 :
bookid: 1 title: C++Primer
date_borrowed: 1800/1/1
date_due: 2100/1/1
borrower: huanghuawei,davy
wait_list:
LiJia, Lord
请按任意键继续. . .

```

```

// add new readers who wait for some books
CheckoutRecord& CheckoutRecord::operator=(const
std::pair<string,string>& new_waiter)
{
    pair<string,string> *ppr = new pair<string,
string>;
    *ppr = new_waiter;
    wait_list.push_back( ppr );
    return *this;
}

```

```

请按任意键继续. . .
After add new waiter to object2 :
bookid:      1          title: C++Primer
date_borrowed: 1800/1/1
date_due:     2100/1/1
borrower:     huanghuawei,davy
wait_list:
    LiJia,Lord
    hhw1,hhw2
请按任意键继续. . .

```

17. 14.2.1 节习题中定义了一个 CheckoutRecord 类, 为该类定义一个下标操作符, 从等待列表中返回一个名字。

下标重载:

```

pair<string, string>& CheckoutRecord::operator[ ]
( const vector< pair<string, string>* >::size_type
index )
{
    return *wait_list.at( index );
}
// 下标操作符重载
const pair<string, string>&
CheckoutRecord::operator[ ]
( const vector< pair<string, string>* >::size_type
index ) const
{
    return *wait_list.at( index );
}

```

18. 讨论用下标操作符实现这个操作的优缺点。

优点: 使用简单。

缺点: 操作的语义不够清楚, 因为 CheckoutRecord 不是一个通常意义上的容器, 而且等待者也不是 CheckoutRecord 容器中的一个元素, 不易使人弄明白怎样用。

19. 提出另一种方法定义这个操作。

可以将这个操作定义成普通的函数, pair<string, string>& get\_a\_waiter( const size\_t index ) 和 const pair<string, string>& get\_a\_waiter ( const size\_t index ) const.

20. 在 ScreenPtr 类的概略定义中, 声明但没有定义赋值操作符, 请实现 ScreenPtr 赋值操作符。

ScreenPtr& operator=( const ScreenPtr& sp )

```

{
    ++sp.ptr->use;
    if ( --ptr->use == 0 )
        delete ptr;
    ptr = sp.ptr;
    return *this;
}

```

21. 定义一个类, 该类保存一个指向 ScreenPtr 的指针。为该类定义一个重载的箭头操作符。

```

class NoName
{
public:
    NoName( *p ): ps( new ScreenPtr(p) ) { }
    ScreenPtr operator->()
    {
        return *ps;
    }
    const ScreenPtr operator->() const
    {
        return *ps;
    }

    ~NoName()
    {
        delete ps;
    }

private:
    ScreenPtr *ps;
};

```

22. 智能指针可能应该定义相等操作符和不等操作符, 以便测试两个指针是否相等或不等。将这些操作加入到 ScreenPtr 类。

```

class ScreenPtr
{
public:
    // ...
    friend inline bool operator==( const ScreenPtr
&, const ScreenPtr &);
    friend inline bool operator!=( const ScreenPtr &,
const ScreenPtr &);
private:

```

```

        ScrPtr *ptr;
};
// operator ==
inline bool operator==( const ScreenPtr &p1, const
ScreenPtr &p2 )
{
    return p1.ptr == p2.ptr;
}

// operator !=
inline bool operator!=( const ScreenPtr &p1, const
ScreenPtr &p2 )
{
    return !(p1.ptr == p2.ptr);
}

```

23. CheckedPtr 类表示指向数组的指针。为该类重载下标操作符和解引用操作符。使操作符确保 CheckedPtr 有效：它应该不可能对超出数组末端的元素进行解引用或索引。

// 下标操作符重载

```

int& CheckedPtr::operator[]( const size_t index )
{
    if ( beg + index >= end )
        throw out_of_range( " invalid index" );
    return *( beg + index );
}

```

```

const int & CheckedPtr::operator[] ( const size_t
index ) const

```

```

{
    if ( beg + index >= end )
        throw out_of_range( " invalid index" );
    return *( beg + index );
}

```

// 解引用操作符重载

```

int CheckedPtr::operator*()
{
    if ( curr == end )
        throw out_of_range( " invalid current
pointer" );
    return *curr;
}
const int& CheckedPtr::operator*() const

```

```

{
    if ( curr == end )
        throw out_of_range( " invalid current
pointer" );
    return *curr;
}

```

24. 习题 14.23 中定义了解引用操作符或下标操作符，是否也应该检查对数组起点之前的元素进行的解引用或索引？解释你的答案。

对于下标操作符，应该进行检查，因为当用户给出的下标索引值小于 0 时，编译器不会出现编译错误，而会出现运行时错误。应修改为：

// 下标操作符重载

```

int& CheckedPtr::operator[]( const size_t index )
{
    if ( beg + index >= end || beg + index < beg )
        throw out_of_range( " invalid index" );
    return *( beg + index );
}

```

```

const int & CheckedPtr::operator[] ( const size_t
index ) const

```

```

{
    if ( beg + index >= end || beg + index < beg )
        throw out_of_range( " invalid index" );
    return *( beg + index );
}

```

而对于解引用操作符，返回 curr 所指向的数组元素，在创建对象时已经将 curr 初始化为指向数组的第一个元素，只有当执行—操作时才会对 curr 进行减的操作，而—操作符已经对 curr 的值与数组起点进行了检查，所以不用再在这里检查。

25. 为了表现得像数组指针，CheckedPtr 类应实现相等和关系操作符，以便确定两个 CheckedPtr 对象是否相等，或者一个小于另一个，诸如此类。为 CheckedPtr 类增加这些操作。

应将这些操作符声明为 CheckedPtr 的友元。

// 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.beg == rhs.beg && lhs.end == rhs.end
        && lhs.curr > rhs.curr;
}
bool operator<=( const CheckedPtr & lhs, const
CheckedPtr& rhs )
{
    return !( lhs.curr > rhs.curr );
}
bool operator >=( const CheckedPtr & lhs, const
CheckedPtr& rhs )
{
    return !( lhs.curr < rhs.curr );
}

```

26. 为 CheckedPtr 类定义加法或减法，以便这些操作符实现指针运算。

将这两个操作符声明为类的友元。

```

CheckedPtr operator+( const CheckedPtr& lhs, const
size_t n )
{
    CheckedPtr temp( lhs );
    temp.curr += n;
    if ( temp.curr > temp.end )
        throw out_of_range(“ Too large n, over.”);
    return temp;
}

```

```

}
CheckedPtr operator-( const CheckedPtr& lhs, const
size_t n )
{
    CheckedPtr temp( lhs );
    temp.curr -= n;
    if ( temp.curr < temp.beg )
        throw out_of_range(“ Too large n, over.”);
    return temp;
}

```

用到了复制构造函数，定义为：

```

CheckedPtr::CheckedPtr( const CheckedPtr& cp ):
beg( cp.beg ), end( cp.end ), curr( cp.curr )
{ }

```

27. 讨论允许将空数组实参传给 CheckedPtr 构造函数的优缺点。

优点：构造函数的定义简单。缺点：导致构造的对象没有指向有效的数组，从而失去使用价值。应该在构造函数里控制参数确保 beg<end.

28. 没有定义自增和自减操作符的 const 版本，为什么？

const 所修饰的 operator，不可以改变对象，而自增和自减肯定改变了对象，与 const 意义相反，所以不定义 const 版本。

29. 我们也没有实现箭头操作符，为什么？

因为此类所指向的是 int 型数组，int 型为内置类型，而箭头操作符必须返回一个类类型的指针，所以不实现。

30. 定义一个 CheckedPtr 版本，保存 Screen 数组。为该实现重载的自增、自减。解引用和箭头等操作符。

```

class CheckedPtr
{
public:
    CheckedPtr( Screen *b, Screen *e ): beg( b ),
end( e ), curr( b ) { }

    // 自增、自减
    CheckedPtr& operator++();
    CheckedPtr& operator--();
}

```

```

    CheckPtr& operator++( int );
    CheckPtr& operator--( int );

    // 箭头操作符
    Screen* operator->();
    const Screen* operator->() const;

    // 解引用操作
    Screen& operator*();
    const Screen& operator*();
private:
    Screen *beg;
    Screen *end;
    Screen *curr;
};

CheckPtr& CheckPtr::operator++()
{
    if ( curr == end )
        throw out_of_range( "increment past the
end of CheckedPtr");
    ++curr;
    return *this;
}
CheckPtr& CheckPtr::operator--()
{
    if ( curr == begin )
        throw out_of_range( "decrement past the
beginning of CheckedPtr");
    --curr;
    return *this;
}
// 后缀式
CheckPtr& CheckPtr::operator++( int )
{
    CheckedPtr temp( *this );
    ++*this;
    return temp;
}
CheckPtr& CheckPtr::operator--( int )
{
    CheckedPtr temp( *this );
    --*this;
    return temp;
}

```

```

}
// 箭头操作符
Screen* CheckedPtr::operator->()
{
    return curr;
}
const Screen* CheckedPtr::operator->() const
{
    return curr;
}

Screen& CheckedPtr::operator*()
{
    if ( curr == end )
        throw out_of_range( "invalid current
pointer.");
    return *curr;
}
const Screen& CheckedPtr::operator*() const
{
    if ( curr == end )
        throw out_of_range( "invalid current
pointer.");
    return *curr;
}

```

31。定义一个函数对象执行“如果……则……否则”操作；该函数对象应接受三个形参；它应该测试第一个形参；如果测试成功，则返回第二个形参；否则，返回第三个形参。

```

class NoName
{
public:
    NoName() { }
    NoName( int i1, int i2, int i3 ): iVal1( i1 ),
iVal2( i2 ), iVal3( i3 ) { }
    int operator()( int i1, int i2, int i3 )
    {
        return i1? i2: i3;
    }
private:
    int iVal1;
    int iVal2;
    int iVal3;
}

```

```
};
```

32. 一个重载的函数调用操作符可以接受多少个操作数？

0 个或多个。

33. 使用标准库算法和 GT\_cls 类，编写一个程序查找序列中第一个比指定值大的元素。

// 14.33\_Gt\_cls\_and\_algorithm.cpp：定义控制台应用程序的入口点。

```
//
```

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
```

```
class GT_cls
{
public:
    GT_cls( const string  gW = " " ) : givenWord( gW )
    { }
```

```
    bool operator()( const string &s )
    {
        return ( s > givenWord );
    }
```

```
private:
```

```
    std::string givenWord;
```

```
};
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "Input some words( ctrl + z to end ):" << std::endl;
    vector<string> text;
    string word;
    while ( std::cin >> word )
    {
        text.push_back( word );
    } // end of input the text
```

```
// input the given word
```

```
std::cin.clear();
std::cout << "Then , input one given word:\t";
string gWord;
std::cin >> gWord;
```

```
// deal with text , to realize find the first word which is bigger than the given word
```

```
vector<string>::iterator it = find_if( text.begin(), text.end(), GT_cls( gWord ) );
```

```
if ( it != text.end() )
```

```
    std::cout << "\n Then the first word in the text you input," << std::endl
```

```
        << " \twhich is bigger
```

```
than the given word is:" << std::endl
```

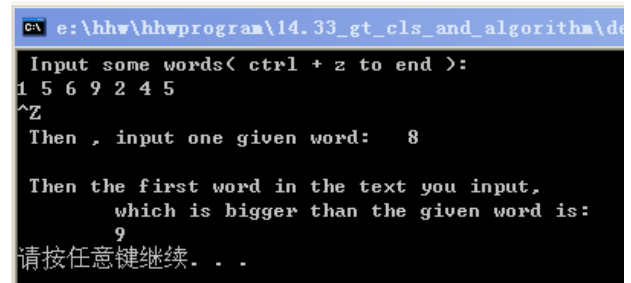
```
        << "\t" << *it <<
```

```
std::endl;
```

```
system("pause");
```

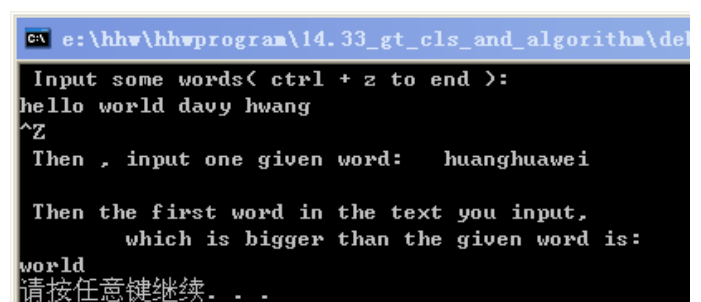
```
return 0;
```

```
}
```



```
C:\e:\hhw\hhwprogram\14.33_gt_cls_and_algorithm\de
Input some words( ctrl + z to end ):
1 5 6 9 2 4 5
^Z
Then , input one given word: 8

Then the first word in the text you input,
which is bigger than the given word is:
9
请按任意键继续. . .
```



```
C:\e:\hhw\hhwprogram\14.33_gt_cls_and_algorithm\de
Input some words( ctrl + z to end ):
hello world davy hwang
^Z
Then , input one given word:  huanghuawei

Then the first word in the text you input,
which is bigger than the given word is:
world
请按任意键继续. . .
```

34. 编写类似于 GT\_cls 的类，但测试两个值是否相等。使用该对象和标准库算法编写程序，替换序列中给定值的所有实例。

// 14.34\_GT\_cls.cpp：定义控制台应用程序的入口点。

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class GT_cls
{
public:
    GT_cls( int iVal = 0 ) : val( iVal ) {    }

    bool operator()( const int & iv )
    {
        return ( iv == val );
    }

private:
    int val;
};

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "Input some nums( ctrl + z to end ):"
<< std::endl;
    vector<int> iVec;
    int iVal;
    while ( std::cin >> iVal )
    {
        iVec.push_back( iVal );
    } // end of input the iVec

    cin.clear();

    cout << "Input a num which will be replaced :\t";
    int rp;
    cin >> rp;

    cout << "Input a given num to insert:\t";
    int givenNum;
    cin >> givenNum;

    // replace

```

```

        replace_if( iVec.begin(), iVec.end(), GT_cls( rp ),
givenNum );

        cout << " Now , the new vector<int> iVec, is :
\n\t";
        for ( vector<int>::iterator it = iVec.begin(); it !=
iVec.end(); ++it )
        {
            cout << *it << " ";
        }

        system("pause");
        return 0;
    }
}

```

35。编写类似于 GT\_cls 的类，但测试给定 string 对象的长度是否与其边界相匹配。报告输入中有多少单词的长度在 1 到 10 之间。

// 14.35\_BT\_cls.cpp：定义控制台应用程序的入口点。  
//

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class BT_cls
{
public:
    BT_cls( size_t len1 = 0, size_t len2 = 0 )
    {
        if ( len1 < len2 )
        {
            minlen = len1;
            maxlen = len2;
        }
        else

```



```

        {
            minlen = len2;
            maxlen = len1;
        }
    }

    bool operator() ( const string &s )
    {
        return ( s.size() >= minlen && s.size() <=
maxlen );
    }

private:
    std::string::size_type minlen, maxlen;
};

bool isShorter( const string &s1, const string &s2 )
{
    return s1.size() < s2.size();
}

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "Input some words( ctrl + z to
end ):" << std::endl;
    vector<string> text;
    string word;
    while ( std::cin >> word )
    {
        text.push_back( word );
    } // end of input the text

    // deal with text
    sort( text.begin(), text.end() );
    text.erase( unique( text.begin(), text.end() ),
text.end() );
    stable_sort( text.begin(), text.end(), isShorter );

    // to count how many words' length are between
1-10
    vector<string>::size_type cnt =
count_if( text.begin(), text.end(), BT_cls( 1, 10 ) );
    std::cout << " There are " << cnt << " words'
length is between 1-10." << std::endl;
}

```

```

        system("pause");
        return 0;
    }
}

```

```

e:\hhw\hhwprogram\14.35_bt_cls\debug\14.35_BT_cls.exe
Input some words( ctrl + z to end ):
hello everybody
I am dauyHuang
and my name is huanghuawei
That is my real name
^Z
There are 11 words' length is between 1-10.
请按任意键继续. . .

```

36。修改前一段程序以报告在 1 到 9 之间以及 10 以上的单词的数目。

// 14.36\_BT\_cls\_Words.cpp : 定义控制台应用程序的入口点。

```

//
#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class BT_cls
{
public:
    BT_cls( size_t len1 = 0, size_t len2 = 0 ) :
minlen( len1 ), maxlen( len2 ) { }

    bool operator() ( const string &s )
    {
        return ( s.size() >= minlen && s.size() <=
maxlen );
    }

private:
    std::string::size_type minlen, maxlen;
};

int _tmain(int argc, _TCHAR* argv[])
{

```

```

std::cout << "Input some words( ctrl + z to
end );" << std::endl;
vector<string> text;
string word;
while ( std::cin >> word )
{
    text.push_back( word );
} // end of input the text

std::cin.clear();

// deal with text , to count how many words'
length are between 1-9
size_t cnt = count_if( text.begin(), text.end(),
BT_cls( 1, 9 ) );
std::cout << "\nThere are "<< cnt << " words'
length are 1-9." << std::endl;
cout << " And there are "<< text.size() - cnt << "
words' length is 10 or more." << endl;

system("pause");
return 0;
}

```

37. 使用标准库函数对象和函数适配器，定义一个对象用于：

- (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
- ```

(a) find_if( vec.begin(), vec.end(),
bind2nd( greater<int>(), 1024 ) );
(b) find_if ( svec.begin(), svec.end(),
bind2nd( not_equal_to<string>(), "pooh" );
(c) transform( ivec.begin(), ivec.end(), ivec.begin(),
bind2nd( multiplies<int>(), 2 ) );

```

38. 最后一个 count\_if 调用中，用 not1 将 bind2nd(less\_equal<int>(), 10) 的结果求反。为什么使用 not1 而不用 not2?

not1 用于将一元函数对象的真值求反，not2 用于将二元函数对象的真值求反，函数适配器 bind2nd 将 less\_equal<int>对象的右操作数绑定到 10，从而 less\_equal 对象由二元函数对象转换为一元函数对象，所以用 not1 求反。

39. 使用标准库函数对象代替 GT\_cls 来查找指定长度的单词。

输入一个指定长度：

```

vector<string> text;
// ... // input words into text
string::size_type len;
cin >> len;
string *w;
for ( vector<string>::iterator it = text.begin(); it !=
text.end(); ++it )
{
    w = find_if ( it, text.end(),
bind2nd( equal_to<string::size_type>(), len ) );
}
cout << " The words whose length is "<< len << "
was found., It is: "
<< *w << endl;

```

40. 编写可将 Sales\_item 对象转换为 string 类型和 double 类型的操作符。你认为这些操作符应返回什么值？你认为定义这些操作符是个好办法吗？解释你的结论。

```

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

```

不是好办法，因为不必在需要 string 类型和 double

类型对象的地方使用 `Sales_item` 对象，这种做法没有太大用处。

41. 解释这两个转换操作符之间的不同：

```
class Integral
{
public:
    operator const int();
    operator int() const;
};
```

不同：前者将对象转换为 `const int` 值，后者将对象转换为 `int` 值，前者太严格，限制使用在可以使用 `const int` 值的地方，只保留后者，将变得更为通用。

42. 为 14.2.1 节习题中的 `CheckoutRecord` 类定义到 `bool` 的转换操作符。

```
CheckoutRecord::operator bool() const
{
    return wait_list.empty();
}
```

43. 解释 `bool` 转换操作符做了什么。这是这个 `CheckoutRecord` 类型转换唯一可能的含义吗？解释你是否认为这个转换是一种转换操作的良好使用。将 `wait_list` 为空的 `CheckoutRecord` 对象转换为 `bool` 类型的 `true`，将不为空的对象转换为 `false`。不是唯一的含义，还可以将 `bool` 转换符定义为当前日期是否已经超过 `date_due`，即判断某本借出的书是否已经过期。

上述定义可用于判断是否有读者在等待借阅，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着。因此这个转换是一种转换操作的良好使用。

44. 为下述每个初始化列出可能的类类型转换序列。每个初始化的结果是什么？

```
class LongDouble
{
public:
    operator double();
    operator float();
};
```

`LongDouble ldObj;`

(a) `int ex1 = ldObj;` (b) `float ex2 = ldObj;`

(a) 既可以先使用从 `Longdouble` 到 `double` 的转换操作，再是偶那个从 `double` 到 `int` 的标准转换，也可以使用从 `LongDouble` 到 `float` 的转换操作，再使用从 `float` 到 `int` 的标准转换，二者没有优劣之分，具有二义性。

(b) 使用从 `LongDouble` 到 `float` 的转换操作，将 `ldObj` 对象转换为 `float` 值用于初始化 `ex2`

45. 哪个 `calc()` 函数是如下函数调用的最佳可行函数？列出调用每个函数所需的转换序列，并解释为什么所选定的是最佳可行函数。

```
class LongDouble
{
public:
    LongDouble( double );
    // ...
};
void calc( int );
void calc( LongDouble );
double dval;
calc( dval ); // which function?
```

最佳可行函数是 `void calc(int)`，调用此函数的转换为：将实参 `double` 类型转换为 `int` 类型的，为标准转换；调用 `void calc( LongDouble)` 函数时，将实参从 `double` 转换为 `LongDouble` 类型，为类类型转换，因为标准转换优于类类型转换，所以第一个函数为最佳可行函数。

46. 对于 `main` 中的加操作，哪个 `operator` 是最佳可行函数？列出候选函数。可行函数以及对每个可行函数中实参的类型转换。

```
class Complex
{
public:
    Complex( double );
    // ...
};
class LongDouble
{
    friend LongDouble operator+( LongDouble,
int );
public:
    LongDouble( int );
```

```

operator double();
LongDouble operator+( const Complex & );
// ...
};
LongDouble operator+( const LongDouble &,
double);
LongDouble ld( 16.08 );
double res = ld + 15.05; // which operator+?

```

(a) 内置类型的+操作符

将 ld 从 LongDouble 转换为 double 类型。

(b) 友元 friend LongDouble operator+( LongDouble, int );

将 15.05 从 double 类型转换为 int 类型（使用标准转换）。

(c) LongDouble operator+( const Complex & );

将 15.05 转换为 Complex 型，使用 Complex 类的构造函数。

(d) 全局函数 LongDouble operator+( const LongDouble &, double);

无需进行类型转换，因此虽然这 4 个都为候选函数，和可行函数，但是这个是最佳可行函数。

## 第十五章 面向对象编程

1. 什么是虚成员？

在类中被声明为 virtual 的成员，基类希望这种成员在派生类中重定义。除了构造函数外，任意非 static 成员都可以为虚成员。

2. 给出 protected 访问标号的定义。它与 private 有何不同？

protected 为受保护的访问标号，protected 成员可以被该类的成员、友元和派生类成员（非友元）访问，而不可以被该类型的普通用户访问。

而 private 成员，只能被基类的成员和友元访问，派生类不能访问。

3. 定义自己的 Item\_base 类版本。

```
class Item_base
```

```

{
public:
    Item_base( const string &book = "", double
sales_price = 0.0 ) :
        isbn( book ), price( sales_price )    {}
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n )const
    {
        return price * n;
    }
    virtual ~Item_base() {}

private:
    string isbn;
protected:
    double price;
};

```

4. 图书馆可以借阅不同种类的资料——书、CD、DVD 等等。不同种类的借阅资料有不同的登记、检查和过期规则。下面的类定义了这个应用程序可以使用的基类。指出在所有借阅资料中，哪些函数可能定义为虚函数，如果有，哪些函数可能是公共的。（注：假定 LibMember 是表示图书馆读者的类，Date 是表示特定年份的日历日期的类。）

```

class Library
{
public:
    bool check_out( const LibMember& );
    bool check_in( const LibMember& );
    bool is_late( const Date& today );
    double apply_fine();
    ostream& print( ostream& = count );
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};

```

因为有不同的登记、检查、和过期规则，所以  
bool check\_out( const LibMember& );

```

bool check_in( const LibMember& );
bool is_late( const Date& today );
double apply_fine();
ostream& print( ostream& = count );

```

这几个函数应该被定义为虚函数，print 函数可能用于打印不同的项目的内同，也定义为虚函数。

其他几个函数，因为操作都有公共性质，所以应该是公共的。

5. 如果有，下面的声明中哪些是错误的？

```

class Base { ... };
(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : Private Base { ... };
(d) class Derived : public Base;
(e) class Derived : inherits Base { ... };

```

(a) 错误，不能用类本身作为类的基类，(d) 声明类时，不可以包含派生列表。

(e) 派生不能用 inherits

6. 编写自己的 Bulk\_item 类版本。

```

class Bulk_item : public item_base
{
public:
    double net_price( size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }

private:
    size_t min_qty;
    double discount;
};

```

7. 可以定义一个类型实现有限折扣策略。这个类可以给低于某个上限的购书量一个折扣，如果购买的数量超过该上限，则超出部分的书应按正常价格购买。定义一个类实现这种策略。

```

class Ltd_item : public item_base
{
public:
    Ltd_item( const string& book = "", double
sales_price, size_t qty = 0, double disc_rate = 0 ) :
        item_base( book, sales_price),
max_qty( qty ), discount( disc_rate ) { }
    double 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;
    double discount;
};

```

8. 对于下面的类，解释每个函数：

```

struct base
{
    string name() { return basename; } // 返回
私有成员 basename
    virtual void print( ostream &os ) { os <<
basename; } // 打印私有成员 basename
private:
    string basename;
};

struct derived
{
    void print() { print( ostream &os ); os <<
“ “ << mem; } // 首先调用 base 类的 print 函数，
然后打印一个空格和私有成员 mem
private:
    int mem;
};

```

如果该代码有问题，如何修正？

问题：没有声明类 derived 是从 base 派生过来的，改正为如下：

```

struct base
{
    base( string szNm ): basename( szNm ) { }
    string name() { return basename; } // 返回
私有成员 basename
    virtual void print( ostream &os ) { os <<
basename; } // 打印私有成员 basename
private:
    string basename;
};

struct derived : public base
{
    derived( string szName, int iVal ): base( szName ),
mem( iVal ) ( )
    void print() { print( base :: ostream &os ); os <<
“ “ << mem; }
private:
    int mem;
};

```

9. 给定上题中的类和如下对象，确定在运行时调用哪个函数：

```

base bobj; base *bp1 = &base; base &br1 = bobj;
derived dobj; base *bp2 = &dobj; base &br2 = dobj;

```

- (a) bobj.print();
- (b) dobj.print();
- (c) bp1->name();
- (d) pb2->name();
- (b) br1.print();
- (f) br2.print();

- (a) bobj.print(); 用的是基类的 print 函数
- (b) dobj.print(); 用的是派生类的 print 函数
- (c) bp1->name(); 用的是基类的 name 函数
- (d) pb2->name(); 用的是基类的 name 函数
- (b) br1.print(); 用的是基类的 print 函数
- (f) br2.print(); 用的是派生类的 print 函数

10. 在 15.2.1 节的习题中编写一个表示图书馆借阅政策的基类。假定图书馆提供下列种类的借阅资料，每一种有自己的检查和登记政策。将这些项目组成

一个继承层次：

```

book          atdio book      record
children's puppet  sega video game  video
cdrom book      Nintendo video game
rental book
sony playstation video game

```

类 book、record、children's puppet、video 继承自 Library;

类 audio book ,cdrom book, rental book 继承自类 book;

类 sega video game, Nintendo video game, sony playstation video game 继承自类 video.

11. 在下列包含一族类型的一般抽象中选择一种(或者自己选择一个)，将这些类型组织成一个继承层次。

- (a) 图像文件格式 ( 如 gif, tiff, jpeg, bmp )
- (b) 几何图元 (如矩形，圆，球形，锥形)
- (c) C++语言的类型 (如类，函数，成员函数)

对(b)中的几何图元组织成一个继承层次，

基类 Figure,

矩形 Rectangle, 圆 cicle, 球形 sphere, 锥形 Cone 继承自 Figure 类。

12. 对上题中选择的类，标出可能的虚函数以及 public 和 protected 成员。

虚函数，比如计算图形面积的函数 virtual double area(); 计算体积的函数 virtual double cubage(); 求周长的函数 virtual double perimeter();

Figuer 类的 public 成员可能有两个图元的尺寸：xSize, ySize, 其他类的 protected 成员可能有 cone 类和球形的 zSize 即 Z 轴尺寸，还有

13. 对于下面的类，列出 C1 中的成员函数访问 ConcreteBase 的 static 成员的所有方式，列出 C2 类型的对象访问这些成员的所有方式。

```

struct ConcreteBase
{
    static std::size_t object_count();
protected:
    static std::size_t obj_count;
};

```

```
struct C1 : public ConcreteBase { //... }
struct C2 : public ConcreteBase { // ... }
```

C1 中的成员函数访问基类的 static 成员可以用

- (1) ConcreteBase::成员名
- (2) C1::成员名
- (3) 通过 C1 类对象或对象的引用, 使用 ( . ) 操作符访问
- (4) 通过 C1 类对象的指针, 使用箭头 (->) 操作符访问
- (5) 直接使用成员名。

C2 类型的对象访问时, 只可以访问基类的成员函数, 假如 C2 对象为 obj\_c2, 可用

- (1) obj\_c2.object\_count()
- (2) ConcreteBase::object\_count()
- (3) C2::object\_count()

14. 重新定义 Bulk\_item 和 item\_base 类, 使每个类只需定义一个构造函数。

```
class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0 ) :
        isbn( book ), price( sales_price )    {}
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n ) const
    {
        return price * n;
    }
    virtual ~Item_base() {}

private:
    string isbn;
protected:
    double price;
```

```
};

class Bulk_item : public Item_base
{
public:
    Bulk_item ( const string &book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc = 0.0 ):
        Item_base ( book, sales_price ),
        min_qty( qty ), discount ( disc ) {}
    double net_price( size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }

private:
    size_t min_qty;
    double discount;
};
```

15. 对于 15.2.5 节第一个习题中描述的图书馆类层次, 识别基类和派生类构造函数。

基类为 IS\_A 结构中的上边的类为下边的类的基类, 派生类构造函数应在初始化列表中包含其直接基类以初始化继承成员和派生类自己的成员。

16. 对于下面的基类定义:

```
struct Base {
    Base(int val): id(val) { }
protected:
    int id;
};

解释为什么下述每个构造函数是非法的。
(a) struct C1 : public Base {
    C1(int val): id(val) { }
};
(b) struct C2 : public C1 {
    C2(int val): Base(val), C1(val){ }
```



```

};
(c) struct C3 : public C1 {
    C3(int val): Base(val) { }
};
(d) struct C4 : public Base {
    C4(int val) : Base(id + val) { }
};
(e) struct C5 : public Base {
    C5() { }
};

```

- (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变量 id
- (e) 缺少了初始化列表,基类没有默认构造函数,其派生类必须用初始化列表对基类的构造函数传递实参。

17. 说明在什么情况下类应该具有虚析构函数。  
作为基类使用的类应该具有虚析构函数,以保证在删除指向动态分配对象的基类指针时,根据指针实际指向的对象所属的类型运行适当的析构函数。

18. 虚析构函数必须执行什么操作?  
虚析构函数可以为空,即不执行任何操作,而当类中有指针类成员时,则需要自己定义虚析构函数,以对指针成员进行适当的清除。

19. 如果这个类定义有错,可能是什么错?

```

class AbstractObject {
public:
    virtual void doit();
    // other members not including any of the copy-control functions
};

```

可能是因为该类可能作为基类使用,所以这时必须定义虚析构函数。而这个类没有定义虚析构函数。

20. 回忆在 13.3 节习题中编写的类,该类的复制控制成员打印一条消息,为 Item\_base 和 Bulk\_item 类的构造函数增加打印语句,定义复制控制成员,使

之完成与合成版本相同的工作外,还打印一条消息,应用使用了 Item\_base 类型的那些对象和函数编写一些程序,在每种情况下,预测将会创建和撤销什么对象,并将你的预测与程序所产生的结果进行比较。继续实验,直至你能够正确地预测对于给定的代码片段,会执行哪些复制控制成员。

```

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

class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0) :
        isbn( book ), price( sales_price )
    {
        cout << "Using Item_base's constructor."
<< endl;
    }
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n ) const
    {
        return price * n;
    }
    virtual ~Item_base()
    {
        cout << "Using Item_base's destructor."
<< endl;
        system("pause");
    }

    friend ostream &operator << ( ostream &,
Item_base &);

private:
    string isbn;
protected:
    double price;

```

```

};

ostream&
operator << ( ostream & os, Item_base & ib)
{
    os << "\tUsing operator << ( ostream & ,
Item_base & );" << endl
        << "\tVisit Item_base 's  book():\t" <<
ib.isbn << endl
        << "\tVisit Item_base 's  net_price():"
        <<  "\t3  《" << ib.book() << "》 , the price
is:\t" << ib.net_price( 3  ) << endl;
    return os;
}

class Bulk_item : public Item_base
{
public:
    Bulk_item ( const string &book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc = 0.0):
        Item_base ( book, sales_price ), min_qty( qty ),
discount ( disc )
    {
        cout << "Using Bulk_item's constructor." <<
endl;
    }
    double net_price( size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }

    ~Bulk_item()
    {
        cout << "Using Bulk_item's destructor." <<
endl;
        system("pause");
    }

private:
    size_t min_qty;

```

```

double discount;
};

ostream&
operator << ( ostream & os, Bulk_item & bi )
{
    os << "\tUsing operator << ( ostream & ,
Bulk_item & )" << endl
        << "\tVisit Bulk_item's  book():\t" <<
bi .book() << endl
        << "\tVisit Bulk_item's  net_price():"
        <<  "\t5  《" << bi.book() << "》 , the price
is:\t" << bi .net_price( 5 ) << endl;
    return os;
}

int _tmain(int argc, _TCHAR* argv[])
{
    Item_base base( "C++Primer", 42.00 );
    Bulk_item bulk( "How to program", 50.32, 3, 0.2);
    cout << base << endl;
    cout << bulk << endl;
    system("pause");
    return 0;
}

```

21。重新定义 Item\_base 层次以包含 Disc\_item 类。

```

class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0 ) :
        isbn( book ), price( sales_price )    {}
    string book( ) const
    {

```

```

        return isbn;
    }

    virtual double net_price( size_t n )const
    {
        return price * n;
    }
    virtual ~Item_base() {}

private:
    string isbn;
protected:
    double price;
};

class Disc_item: public Item_base
{
public:
    Disc_item ( const string& book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc = 0.0 ) :
        Item_base ( book, sales_price ), quantity
( qty ), dicount ( disc ) {}
    double net_price( size_t cnt )
    {
        if ( cnt >= quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;

    }
    std::pair<size_t, double > discount_policy() const
    { return std::make_pair( quantity, discount ); }

protected:
    size_t quantity;
    double discount;
};

```

22. 重新定义 Bulk\_base 和习题 15.2.3 节中实现的那个表示有限折扣策略的类，继承含 Disc\_item 类。

```

class Disc_item: public Item_base
{
public:

```

```

    Disc_item ( const string& book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc = 0.0 ) :
        Item_base ( book, sales_price ), quantity
( qty ), discount ( disc ) {}
    double net_price( size_t cnt )
    {
        if ( cnt >= quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;

    }
    std::pair<size_t, double > discount_policy() const
    { return std::make_pair( quantity, discount ); }

protected:
    size_t quantity;
    double discount;
};

class Bulk_item : public Disc_item
{
public:
    Bulk_item ( const string &book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc_rate =
0.0 ):
        Disc_item ( book, sales_price, qty, disc_rate )
    {}
    double net_price( size_t cnt ) const
    {
        if ( cnt < quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price - quantity * discount
* price;
    }
};

```

23. 对于下面的基类和派生类定义:

```
struct Base {
    foo(int);
protected:
    int bar;
    double foo_bar;
};

struct Derived : public Base {
    foo(string);
    bool bar(Base *pb);
    void foobar();
protected:
    string bar;
};
```

找出下述每个例子中的错误并说明怎样改正:

```
(a) Derived d; d.foo(1024);
(b) void Derived::foobar() { bar = 1024; }
(c) bool Derived::bar(Base *pb)
    { return foo_bar == pb->foo_bar; }
```

(a) 调用的是 Derived 类中的 foo(string) 函数, 所以参数为 1024 就错了, 应改为使用 string 类型的实参。

(b) 函数体内, bar 是 string 类型的, 用 int 类型的值 1024 对 bar 进行赋值错误。应赋以 string 类型的对象给 bar。

(c) 通过指向 Base 类对象的指针访问其 protected 的成员 foo\_bar 错误, 应该将 pb 定义为指向子类 Derived 类的对象的指针。

24. 对于如下代码:

```
Bulk_item bulk;
Item_base item(bulk);
Item_base *p = &bulk;
为什么表达式
p->net_price(10);
调用 net_price 的 Bulk_item 实例, 而表达式
```

```
item.net_price(10);
```

调用 Item\_base 实例?

因为 net\_price 是基类中的虚函数, 当用指向基类对象的指针调用时, 将动态绑定虚函数, 即调用该指针实际指向的对象所属类型中定义的函数。p 是指向 Base\_item 类对象的指针, 但实际上它指向的是 Bulk\_item 类对象 bulk, 所以 p->net\_price 调用的是 Bulk\_item 实例。

而通过基类的对象直接调用虚函数时, 总是调用的是该对象所属的函数, 所以 item 是基类的对象, 调用的是 Item\_base 的实例。

25. 假定 Derived 继承 Base, 并且 Base 将下面的函数定义为虚函数, 假定 Derived 打算定义自己的这个虚函数的版本, 确定在 Derived 中哪个声明是错误的, 并指出为什么错。

```
(a) Base* Base::copy( Base* );
    Base* Derived::copy( Derived* );
(b) Base* Base::copy( Base* );
    Derived* Derived::copy( Base* );
(c) ostream& Base::print( int, ostream& = cout );
    ostream& Derived::print( int, ostream& );
(d) void Base::eval() const;
    void Derived::eval();
```

(a) 错了, 因为 Derived 中声明的 copy 是一个非虚函数, 而不是对 Base 中的虚函数 copy 的重载, 因为派生类的重定义的虚函数必须与基类中的虚函数具有相同原型。而且此时 Derived 中定义的 copy 函数还屏蔽了基类 Base 的 copy 函数。

26. 使你的 Disc\_item 类版本成为抽象类。

```
double net_price( size_t ) const = 0;
```

27. 试着定义 Disc\_item 类型的一个对象, 看看会从编译器得到什么错误。

```
h:\program\sales_item_base\sales_item_base\sales_item_base.cpp(14) : error C2259: "Disc_item": 不能实例化抽象类
由于下列成员:
    "double Disc_item::net_price(size_t) const": 是抽象的
e:\hbw\hbwprogram\sales_item_base\sales_item_base\disc_item.h(13) : 参见 "Disc_item::net_price" 的声明
E保存在 "file:///e:/hbw/HBWProgram/Sales item Item base/Sales item Item base/Debug/BuildLog.htm"
Item_base - 1 个错误, 0 个警告
= 生成: 0 已成功, 1 已失败, 0 最新, 0 已跳过 =====
```

会如上提示, Disc\_item 是抽象的, 不能实例化抽象类。

28. 定义一个 vector 来保存 Item\_base 类型的对象,

并将一些 Bulk\_item 类型对象赋值到 vector 中，遍历 vector 并计算容器中元素的 net\_price。

```
// Item_base.h
#include <iostream>
#include <ostream>
#include <string>
using namespace std;

class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0) :
        isbn( book ), price( sales_price )
    { }
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n ) const
    {
        return price * n;
    }
    virtual ~Item_base()
    { }

    friend ostream &operator << ( ostream &,
Item_base &);

private:
    string isbn;
protected:
    double price;
};

ostream&
operator << ( ostream & os, Item_base & ib)
{
    os << "\tUsing operator << ( ostream &,
Item_base & );" << endl
        << "\tVisit Item_base's  book():\t" <<
ib.isbn << endl
        << "\tVisit Item_base's  net_price():"
```

```
<< " 3  《" << ib.book() << "》 , the price
is:\t" << ib.net_price( 3 ) << endl;
    return os;
}

class Bulk_item : public Item_base
{
public:
    Bulk_item ( const string &book = "", double
sales_price = 0.0,
        size_t qty = 0, double disc = 0.0):
        Item_base ( book, sales_price ), min_qty( qty ),
discount ( disc )
    { }
    double net_price( size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }

    ~Bulk_item()
    { }
private:
    size_t min_qty;
    double discount;
};

ostream&
operator << ( ostream & os, Bulk_item & bi )
{
    os << "\tUsing operator << ( ostream &,
Bulk_item & )" << endl
        << "\tVisit Bulk_item's  book():\t" <<
bi .book() << endl
        << "\tVisit Bulk_item's  net_price():"
        << " 5  《" << bi.book() << "》 , the price
is:\t" << bi .net_price( 5 ) << endl;
    return os;
}

// main
// Sales_item_Item_base.cpp : 定义控制台应用程序
的入口点。
```

```
//
#include "stdafx.h"
#include "Disc_item.h"
#include <vector>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    Item_base base0( "C++Primer_0", 50.00 );
    Bulk_item bulk0( "How to program_0", 50.0, 3,
0.2);

    cout << base0 << endl;

    cout << bulk0 << endl;

    vector<Item_base> basket;
    // create some more objects
    Item_base base1( "C++Primer_1", 50.00);

    Bulk_item bulk1( "How to program_1", 50, 3,
0.25);
    basket.push_back(base0);
    basket.push_back(base1);
    basket.push_back(bulk0);
    basket.push_back(bulk1);

    double SumPrice = 0.0;
    for ( vector<Item_base>::iterator it =
basket.begin(); it != basket.end(); ++it )
    {
        SumPrice += (*it).net_price(5);
    }
    cout << "\n\tThe sum price of the basket is :\t"
<< SumPrice << endl;
    system("pause");
    return 0;
}
```

29. 重复以上程序，但这次存储 Item\_base 类型的对

象的指针。比较结果总和。

把 main 函数改为如下：

// Sales\_item\_Item\_base.cpp : 定义控制台应用程序的入口点。

```
//

#include "stdafx.h"
#include "Disc_item.h"
#include <vector>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    Item_base base0( "C++Primer_0", 50.00 );
    Bulk_item bulk0( "How to program_0", 50.0, 3,
0.2);

    vector<Item_base *> basket;
    // create some more objects
    Item_base base1( "C++Primer_1", 50.00);

    Bulk_item bulk1( "How to program_1", 50, 3,
0.25);

    Item_base *p1 = &base0;
    Item_base *p2 = &base1;
    Bulk_item *p3 = &bulk0;
    Bulk_item *p4 = &bulk1;

    cout << *p1 << endl;
    cout << *p2 << endl;
    cout << *p3 << endl;
    cout << *p4 << endl;

    basket.push_back(p1);
    basket.push_back(p2);
    basket.push_back(p3);
    basket.push_back(p4);

    cout << "\n\tEvery book's price is 50$, when you
put 4*5 books in the basket," << endl;
    double SumPrice = 0.0;
    for ( vector<Item_base*>::iterator it =
basket.begin(); it != basket.end(); ++it )
```

```

{
    SumPrice += (*it)->net_price(5);    //
    net_price 's reference is the num of the book you
    bought for this time
}
cout << "\n\tthe sum price of the basket is :\t" <<
SumPrice << endl;
system("pause");
return 0;
}

```

30. 解释上两题程序所产生总和的差异。如果没有差异，解释为什么没有。

有差异，第 28 题通过 `Item_base` 类型的对象调用虚函数 `net_price`，不实现动态绑定，调用的是 `Item_base` 类中定义的版本，第 29 题通过 `Item_base` 类型的指针调用虚函数 `bet_price`，实现动态绑定，后两个指针实际指向 `Bulk_item` 对象，所以结果会产生差异。

31. 为 15.2.3 节的习题中实现的有限折扣类定义和实现 `clone` 操作。

```

// 基类
class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0) :
        isbn( book ), price( sales_price )    {}
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n )const

```

```

{
    return price * n;
}
virtual ~Item_base() {}
// Add clone function at base class
virtual Item_base* clone() const
{
    return new Item_base( *this );
}

```

private:

string isbn;

protected:

double price;

};

// 子类

class Ltd\_item : public item\_base

{

public:

Ltd\_item( const string& book = "", double
sales\_price, size\_t qty = 0, double disc\_rate = 0 ) :

item\_base( book, sales\_price),

max\_qty( qty ), discount( disc\_rate ) { }

double net\_price( size\_t cnt ) const

{

if ( cnt <= max\_qty )

return cnt \* ( 1 - discount ) \* price;

else

return cnt \* price - max\_qty \*

discount \* price;

}

// overload clone function

Ltd\_item \* clone() const

{

return new Ltd\_item( \*this );

}

private:

size\_t max\_qty;

double discount;

};

32. 实际上，程序不太可能在第一次运行或第一次用真实数据运行时就能正确运行。在类的设计中包



括调试策略经常是有用的。为 Item\_base 类层次实现一个 debug 虚函数，显示各个类的数据成员。

// 15\_32\_Item\_base\_Debug\_virtualFcn.cpp : 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <ostream>
#include <string>
using namespace std;
```

```
class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0 ) :
        isbn( book ), price( sales_price )
    { }
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n ) const
    {
        return price * n;
    }
    virtual ~Item_base()
    { }

    friend ostream &operator << ( ostream &,
Item_base &);

protected:
    string isbn;
protected:
    double price;

public:
    virtual void debug() const
    {
        cout << "Debug: Item_base's members are:"
```

```
<< endl
        << "\tThe isbn is : \t" << isbn << endl
        << "\tThe price is : \t" << price << endl;
    }
};

ostream &operator << ( ostream & out , Item_base &
ib )
{
    out << "\tItem_base's members are:" << endl
        << "\tThe isbn is : \t" << ib.isbn << endl
        << "\tThe price is : \t" << ib.price << endl;
    return out;
}

class Bulk_item : public Item_base
{
public:
    Bulk_item ( const string &book = "", double
sales_price = 0.0,
        size_t qty = 0, double disc = 0.0):
        Item_base ( book, sales_price ), min_qty( qty ),
discount ( disc )
    { }
    double net_price( size_t cnt ) const
    {
        if ( cnt > min_qty )
            return cnt * ( 1- discount ) * price;
        else
            return cnt * price;
    }
    ~Bulk_item() { }

public:
    virtual void debug( ostream& os = cout ) const
    {
        os << "\nDebug: Bulk_item's members
are(the 4 members):" << endl;
        Item_base::debug();
        os << "\tThe min_qty is : \t" << min_qty <<
endl
        << "\tThe discount is : \t" << discount <<
endl;
    }
private:
```

```

size_t min_qty;
double discount;
};

int _tmain(int argc, _TCHAR* argv[])
{
    Item_base ib("0-001-100", 100.0);
    Bulk_item bi("0-001-101", 90.0, 10, 0.4);
    Item_base *pIb = &ib;
    Bulk_item *pBi = &bi;
    cout << *pIb << endl;
    cout << *pBi << endl;
    pIb->debug();

    pBi->debug();

    system("pause");
    return 0;
}

```

33. 对于 Item\_base 层次的包括 Disc\_item 抽象基类的版本, 指出 Disc\_item 类是否应实现 clone 函数, 为什么?

别实现了, 因为 Disc\_item 抽象基类不能创建对象。

34. 修改调试函数以允许用户打开或关闭调试。用两种方式实现控制:

- (a) 通过定义 debug 函数的形参。
- (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。

基类中的 debug 函数修改为:

```

virtual void debug( bool bDisplay ) const
{
    if ( bDisplay )
    {
        cout << "Debug: Item_base's members are:" << endl
                << "\tThe isbn is :\t" << isbn
                << endl
                << "\tThe price is :\t" << price
                << endl;
    }
}

```

子类中的 debug 函数修改为:

```

virtual void debug( bool bD ) const
{
    if ( bD )
    {
        cout << "\nDebug: Bulk_item's members are(the 4 members):" << endl;
        Item_base::debug(bD);
        cout << "\tThe min_qty is :\t" << min_qty << endl
                << "\tThe discount is :\t" << discount << endl;
    }
}

```

主函数中增加一段代码:

```

bool bDisp = false;
cout << "Do you wanna display the Debug info? ( 1/0 ) :\n";
while ( cin >> bDisp )
{
    pIb->debug(bDisp);
    pBi->debug(bDisp);
    if ( !bDisp )
    {

```

```
cout << "\n\t====>> Won't display the
debug information." << endl;
```

```
    }
}
```

```
e:\hhw\hhwprogram\15_32_item_base_debug_virtualfcn
Item_base's members are:
The isbn is : 0-001-100
The price is : 100

Item_base's members are:
The isbn is : 0-001-101
The price is : 90

Do you wanna display the Debug info? < 1/0 > :
0

====>> Won't display the debug information.

1
Debug: Item_base's members are:
The isbn is : 0-001-100
The price is : 100

Debug: Bulk_item's members are(the 4 members):
Debug: Item_base's members are:
The isbn is : 0-001-101
The price is : 90
The min_qty is : 10
The discount is : 0.4

^Z
请按任意键继续...
```

35. 编写自己的 compare 函数和 Basket 类的版本并使用它们管理销售。

```
#ifndef ITEM_H
#define ITEM_H

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

class Item_base
{
public:
    Item_base( const string &book = "", double
sales_price = 0.0) :
        isbn( book ), price( sales_price )
    {
    }
    string book( ) const
    {
        return isbn;
    }

    virtual double net_price( size_t n ) const
```

```
{
    return price * n;
}
virtual ~Item_base()
{
}

virtual Item_base* clone() const
{
    return new Item_base( *this );
}

private:
    string isbn;
protected:
    double price;
};

class Disc_item: public Item_base
{
public:
    Disc_item ( const string& book = "", double
sales_price = 0.0,
                size_t qty = 0, double disc = 0.0) :
        Item_base ( book, sales_price ), quantity ( qty ),
        discount ( disc )
    {
    }

    virtual double net_price( size_t ) const = 0;

    std::pair<size_t, double > discount_policy() const
    { return std::make_pair( quantity, discount ); }

    ~Disc_item()
    {
    }

protected:
    size_t quantity;
    double discount;
};

class Bulk_item : public Disc_item
{
public:
```

```

        Bulk_item ( const string &book = "", double
sales_price = 0.0,
        size_t qty = 0, double disc = 0.0):
        Disc_item( book, sales_price, qty, disc ) {
double net_price( size_t cnt ) const
{
    if ( cnt > quantity )
        return cnt * ( 1- discount ) * price;
    else
        return cnt * price;
}

virtual Bulk_item* clone() const
{
    return new Bulk_item( *this );
}
};

class Ltd_item : public Disc_item
{
public:
    Ltd_item( const string& book = "", double
sales_price = 0.0, size_t qty = 0, double disc_rate =
0.0 ) :
        Disc_item( book, sales_price, qty, disc_rate )
{ }

    double net_price( size_t cnt ) const
    {
        if ( cnt <= quantity )
            return cnt * ( 1- discount ) * price;
        else
            return cnt* price - quantity *
discount * price;
    }

    virtual Ltd_item* clone() const
    {
        return new Ltd_item( *this );
    }
};

#endif
#endif SALES_ITEM_H

```

```

#define SALES_ITEM_H
#include "Item.h"

class Sales_item
{
public:
    Sales_item() : p(0), use( new std::size_t (1) ) {}
    Sales_item( const Item_base& item ):
p( item.clone() ), use( new std::size_t (1) ) {}
    Sales_item ( const Sales_item &i):p( i.p ),
use( i.use ) { ++*use; }
    ~Sales_item() { decr_use(); }

    Sales_item& operator=(const Sales_item& rhs)
    {
        ++*rhs.use;
        decr_use();
        p = rhs.p;
        use = rhs.use;
        return *this;
    }

    const Item_base *operator-> () const
    {
        if(p)
            return p;
        else
            throw std::logic_error( "unbound
Sales_item" );
    }

    const Item_base &operator *() const
    {
        if ( p )
            return *p;
        else
            throw std::logic_error( "unbound
Sales_item" );
    }

private:
    Item_base *p;
    std::size_t *use;

    void decr_use()

```

```

    {
        if ( --*use == 0 )
        {
            delete p;
            delete use;
        }
    }
};

#endif
#ifndef __BASKET_H_
#define __BASKET_H_

#include "Sales_item.h"
#include <set>

inline bool compare( const Sales_item &lhs, const
Sales_item &rhs )
{
    return lhs->book() < rhs->book();
}

class Basket
{
    typedef bool (*Comp) (const Sales_item&, const
Sales_item&);
public:
    typedef std::multiset<Sales_item, Comp>
set_type;

    typedef set_type::size_type size_type;
    typedef set_type::const_iterator const_iter;

    Basket(): items( compare ) { }

    void add_item( const Sales_item &item )
    {
        items.insert( item );
    }

    size_type size( const Sales_item &i ) const
    {
        return items.count(i);
    }
};

```

```

    }

    double total() const;
private:
    std::multiset<Sales_item, Comp> items;
};

double Basket::total() const
{
    double sum = 0.0;
    for ( const_iter iter = items.begin(); iter !=
items.end(); iter = items.upper_bound(*iter))
    {
        sum +=
(*iter)->net_price( items.count( *iter ));
    }
    return sum;
}

#endif

// 15.35_Basket(multiset).cpp : 定义控制台应用程序
的入口点。
//

#include "stdafx.h"
#include <iostream>
#include "Basket.h"
#include "Sales_item.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    Basket basket;
    Sales_item item1 ( Bulk_item( "0-0001-0001-1",
99, 20, 0.5 ));
    Sales_item item2 ( Bulk_item( "0-0001-0001-2",
50 ));
    Sales_item item3 ( Bulk_item( "0-0001-0001-3",
59, 200, 0.3 ));
}

```

```
Sales_item item4 ( Bulk_item( "0-0001-0001-1",
99, 20, 0.2 ));
```

```
basket.add_item(item1);
basket.add_item(item2);
basket.add_item(item3);
basket.add_item(item4);
```

```
cout << basket.total() << endl;
```

```
system("pause");
return 0;
```

```
}
```



36. Basket::const\_iter 的基础类型是什么？  
即 mutiset 容器的元素类型 Sales\_item.

37. 为什么在 Basket 的 private 部分定义 Comp 类型别名？  
因为该别名仅在类 Basket 里使用。

38. 为什么在 Basket 中定义两个 private 部分？

这是为了在类里面逻辑地区分开不同模块所侧重的不同点：显示地区分开类型别名与数据成员。

39. 给定 s1、s2、s3 和 s4 均为 string 对象，确定下述 Query 类的使用创建什么对象：

- (a) Query(s1) | Query(s2) & - Query( s3);
- (b) Query(s1) | (Query(s2) & - Query(s3));
- (c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));

(a) 共创建 12 个对象：6 个 Query\_base 对象以及其相关联的句柄。6 个 Query\_base 对象分别是 3 个 WordQuery 对象，一个 NotQuery 对象，一个 AndQuery 对象，一个 OrQuery 对象。

(b) 与(a)同。

(c) 共创建 14 个对象：7 个 Query\_base 对象及其相关联的句柄。7 个 Query\_base 对象分别是 4 个 WordQuery 对象，2 个 AndQuery 对象，1 个 OrQuery 对象。

40. 对图 15-4 中建立的表达式：

```
Query q = Query("fiery") & Query( " bird " ) |
Query( " wind " );
```

(a) 列出处理这个表达式所执行的构造函数。

(b) 列出执行 cout << q 所调用的 display 函数和重载的 << 操作符。

(c) 列出计算 q.eval 时所调用的 eval 函数。

- (a) Query( const std::string& );  
WordQuery( std::string& );  
AndQuery( Query, Query );  
BinaryQuery( Query, Query, std::string );  
Query\_base();  
OrQuery(Query,Query);  
Query( Query\_base\* );

(b) Query 类的 operator<<

Query 类的 display, BinaryQuery 类的 display, WordQuery 类的 display

(c) Query 类的 eval, OrQuery 类、AndQuery 类、WordQuery 类的 eval.

41. 实现 Query 类和 Query\_base 类，并为第 10 章的 TextQuery 类增加需要的 size 操作。通过计算和打印如图 15-4 所示的查询，测试你的应用程序。  
由于程序牵涉到第 10 章的过多内容，等以后再完善这个程序吧。

42. 设计并实现下述增强中的一个：

- (a) 引入基于同一句子而不是同一行计算单词的支持。
- (b) 引入历史系统，用户可以用编号查阅前面的查询，并可以在其中增加内容或与其他查询组合。
- (c) 除了显示匹配数目和所有匹配行之外，允许用户对中间查询计算和最终查询指出要显示的行的范围。

(a) 设计: 改为支持基于同一句子而不是同一行计算单词, 只需要将文本按句子而不是按行存储到 `vector` 中。将 `TextQuery` 类的成员函数 `store_file` 修改为对句子的相应处理。

```
void TextQuery::store_file( ifstream &is )
{
    char ws[] = { '\t', '\r', '\v', '\f', '\n' };
    char eos[] = { '?', ',', '!' };
    set<char> whitespace( ws, ws + 5 );
    set<char> endOfSentence( eos, eos + 3 );
    string sentence;
    char ch;
    while ( is.get(ch) )
    {
        if ( !whiteSpace.count(ch) )
            sentence += ch;
        if ( endOfSentence.count(ch) )
        {
            lines_of_text.push_back( sentence );
            sentence = "";
        }
    }
}
```

Davy\_H

2010-11-09

## 第十六章 部分选做习题

16.1 编写一个模板返回形参的绝对值, 至少用三种不同类型的值调用模板。注意: 在 16.3 节讨论编译器怎样处理模板实例化之前, 你应该将每个模板定义和该模板的所有使用放在同一文件中。

// 16.1\_template.cpp: 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
template<typename T> inline T abs( T tVal)
```

```
{
    return tVal > 0 ? tVal : -1*tVal;
}

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "\n\tabs( false ) is:\t" << abs(false)
    << std::endl
    << "\n\tabs( -3 ) is:\t\t" <<
    abs( -3 ) << std::endl
    << "\n\tabs( -8.6 ) is:\t\t" <<
    abs( -8.6 ) << std::endl;

    system("pause");
    return 0;
}
```

16.2 编写一个函数模板, 接受一个 `ostream` 引用和一个值, 将该值写入流。用至少四种不同类型调用函数。通过写至 `cout`、写至文件和写至 `stringstream` 来测试你的程序。

// 16.2\_template.cpp: 定义控制台应用程序的入口点。

//

```
#include "stdafx.h"
#include <iostream>
#include <ostream>
#include <fstream>
#include <sstream>
#include <cassert>
using namespace std;

template<typename T> inline
void testOstream( ostream& os, T tVal )
{
    os << tVal;
```



```

}

int _tmain(int argc, _TCHAR* argv[])
{
    // cout
    cout << "Test of testOstream( cout, 6 ):\t";
    testOstream( cout, 6 );
    cout << "\n Test of testOstream( cout, true ):\t";
    testOstream( cout, true );
    cout << "\n Test of testOstream( cout, 8.6 ):\t";
    testOstream( cout, 8.6 );
    cout << "\n Test of testOstream( cout,
\"Good!\"):\t";
    testOstream( cout, "Good!" );
    cout << endl;

    // ofstream
    ofstream outfile;
    outfile.open("testOf_Ofstream.txt");
    assert( outfile );
    if ( !outfile )
    {
        return NULL;
    }
    cout << "\n\tTest of ofstream, Please check the
file just create.\n";
    testOstream(outfile, "\n Test of
testOstream( outfile, 6 ):\t");
    testOstream( outfile, 6 );
    testOstream(outfile, "\n Test of
testOstream( outfile, true ):\t");
    testOstream( outfile, true );
    testOstream(outfile, "\n Test of
testOstream( outfile, 8.6 ):\t");
    testOstream( outfile, 8.6 );
    testOstream(outfile, "\n Test of
testOstream( outfile, \"Good!\"):\t");
    testOstream( outfile, "Good!" );
    outfile.close();
    cout << endl;

    // stringstream
    stringstream oss;
    cout << "\n Test of testOstream( oss, 6 )/( oss,

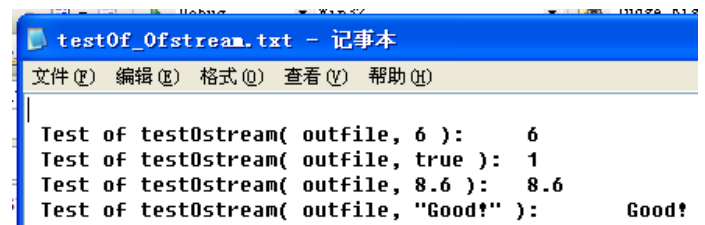
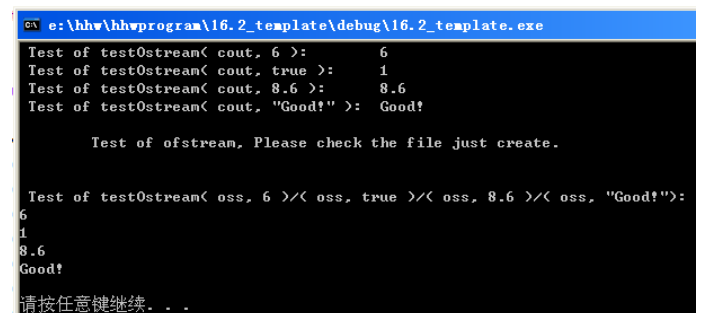
```

```

true )/( oss, 8.6 )/( oss, \"Good!\"): " << endl;
    testOstream( oss, 6 );
    testOstream( oss, "\n" );
    testOstream( oss, true );
    testOstream( oss, "\n" );
    testOstream( oss, 8.6 );
    testOstream( oss, "\n" );
    testOstream( oss, "Good!");
    testOstream( oss, "\n" );
    cout << oss.str() << endl;

    system("pause");
    return 0;
}

```

16.3 当调用两个 string 对象的 compare 时，传递用字符串字面值初始化的两个 string 对象。如果编写以下代码会发生什么？

```
compare ( "hi", "world" );
```

将会出编译错误，根据第一个实参"hi" 可将模板形参 T 推断为 char[3]，而第二个实参"world"可将模板形参 T 推断为 char[6],T 被推断为两个不同的类型，编译器无法实例化。

16.5 定义一个函数模板，返回两个值中较大的一个。

```
template < typename T > T Max( T &val1, T &val2 )
{
    return val1 > val2 ? val1 : val2;
}
```

16.6 （标记但暂时不做，待 stl 学习过后做）

16.8 如果有，解释下面哪些声明是错误的并说明为什么。

```
(a) template <class Type> Type bar ( Type, Type );
    template <class Type> Type bar ( Type, Type );
(b) template <class T1, class T2 > void bar ( T1,
T2);
    template <class C1, typename C2 > void bar ( C1,
C2);
```

都正确。

16.9 （标记但暂时不做，待 stl 学习过后做）

16.11 何时必须使用 typename?

如果要在函数模板内部使用在类中定义的类型成员，必须在该成员名前加上关键字 `typename`，告诉编译器将该成员当做类型。

16.12 编写一个函数模板，接受表示未知类型迭代器的一对值，找出在序列中出现得最频繁的值。

```
template <typename T> T::value_type MostFreq( T
first, T last )
{
    // 计算需分配内存的大小
    std::size_t amount = 0;
    T start = first;
    while ( start != last )
    {
        amount++;
        start++;
    }

    // 定义类型别名
    typedef std::vector<typename T::value_type>
VecType;
```

// 创建vector对象，用于保存输入序列的副本

```
VecType vec(amount);
VecType::iterator newFirst = vec.begin();
VecType::iterator newLast = vec.end();
```

// 将输入序列复制到vector对象

```
std::uninitialized_copy( first, last, newFirst );
```

std::sort ( newFirst, newLast ); // 对副本序列  
排序，使得相同的值出现在相邻位置

std::size\_t maxOccu = 0, occu = 0; // 出现最频繁  
的次数，当前值的出现次数

VecType::iterator preIter = newFirst; //指向当前  
值的前一个值

VecType::iterator maxOccuElement = newFirst; //  
指向当前出现最频繁的值

```
while( newFirst != newLast )
{
    if ( *newFirst != *preIter ) // 当前值与前  
一值不同
    {
        if ( occu > maxOccu ) // 当前  
值的出现次数为目前最大次数
        {
            maxOccu = occu; //  
修改最大次数
            maxOccuElement = preIter; //  
修改指向当前出现最频繁的值的迭代器
        }
        occu = 0;
    }
    ++occu;
    preIter = newFirst;
    ++newFirst;
}

// 最后一个值的出现次数与目前的最大次数  
进行比较
if ( occu > maxOccu )
{
    maxOccu = occu;
    maxOccuElement = preIter;
}
```

```

    return *maxOccuElement;
}

```

16.17 在 3.3.2 节的“关键概念”中，我们注意到，C++程序员习惯于使用 != 而不用 <，解释这一习惯的基本原理。

标准库中的类及泛型算法大多定义为模板类及模板函数，它们的实例化版本是否合法取决于用作模板实参的类型是否支持模板所要求的操作。对于用作模板实参的类型，支持相等操作符 == 和不等操作符 != 的可能性要比支持关系操作符 < 的可能性更大，因此使用 != 而不用 <。

16.21 指出对模板实参推断中涉及的函数实参允许的类型转换。

**const 转换：**接受 const 引用或 const 指针的函数可以分别用非 const 对象的引用或指针来调用，无需产生新的实例化。如果函数接受非引用类型，形参类型和实参都忽略 const，即无论传递 const 或非 const 对象给接受非引用类型的函数，都使用相同的实例化。

**数组或函数到指针的转换：**如果模板形参不是引用类型，则对数组或函数类型的实参应用常规指针转换。数组实参将当做指针其第一个元素的指针，函数实参将当做指向函数类型的指针。

16.22 对于下面的模板：

```

template < class Type>
Type calc ( const Type* array, int size);
template < class Type >
Type fcn( Type p1, Type p2)

```

下面这些调用有错吗？如果有，哪些是错误的？为什么？

```

double dobj; float fobj; char cobj;
int tai[5] = { 511, 16, 8, 63, 34 };

```

- (a) calc ( cobj, 'c' );
- (b) calc( dobj, fobj);
- (c) fcn( ai, cobj);

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

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

(c) 两个实参的类型不一样，不能使用函数模板 fcn 进行实例化。

16.26 对于下面的 sum 模板定义：

```

template <class T1, class T2, class T3 > T1 sum( T2, T3 );

```

解释下面的每个调用，是否有错？如果有，指出哪些是错误的，对每个错误，解释错在哪里。

```

double dobj1, dobj2; float fobj1, fobj2; char cobj1, cobj2;

```

- (a) sum ( dobj1, dobj2 );
- (b) sum< double, double, double > ( fobj2, fobj2);
- (c) sum<int> ( cobj1, cobj2 );
- (d) sum < double, , double > (fobj2, dobj2);

(a)错误，没有显示指定 T1 实参类型。

(b)正确，产生 double sum ( double, double) 的实例，并将两个函数实参由 float 转换为 double 类型来调用该实例。

(c)正确，产生 int sum( char, char )的实例。

(d)错误，只有最右边形参的显示模板实参可以省略，不能用空格代替被省略的显示模板实参。

16.28 如果所用的编译器支持分别编译模型，将类模板的成员函数和 static 数据成员的定义放在哪里？为什么？

如果编译器支持分别编译模型，则类模板的内联成员函数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被编译器所见，而非内联函数一般并不希望被用户看见，所以将其放在实现文件中。

16.37 下面哪些模板实例化是有效的？解释为什么实例化无效？

```

template <class T, int size> class Array { };
template< int hi, int wid > class Screen { };

```

(a) const int hi = 40, wi = 80; Screen< hi, wi + 32> sObj;

(b) `const int arr_size = 1024; Array< string, arr_size > a1;`

(c) `unsigned int asize = 255; Array<int, asize > a2;`

(e) `const double db = 3.1415; Array< double, db > a3;`

(a) 和 (b) 有效。

(c) 无效, 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` 做模板实参。

(d) 无效, 因为 `db` 是 `double` 型常量, 与模板中的 `int` 型模板实参不符。

16.38 编写 `Screen` 类模板, 使用非类型形参定义 `Screen` 的高度和宽度。

```
template< int hi, int wid >
class Screen
{
public:
    typedef std::string::size_type index;
    Screen() : contents( hi * wid, '#' ), cursor(0),
height( hi ), width( wid ) { }
    Screen ( const std::string & cont );

    char get() const
    {
        return contents[ cursor ];
    }
    char get( index ht, index wd ) const;

    index get_cursor() const { return cursor; }
    Screen& move( index r, index c );
    Screen& set( char );
    Screen& display( ostream &os );
    const Screen& display( ostream &os ) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};

template<int hi, int wid> Screen < hi, wid>::
Screen< hi, wid> ( const std::string& cont ) :
    cursor ( 0 ), height( hi ), width( wid )
{
    contents.assign( hi*wid, ' ' );
```

```
if ( cont.size() != 0 )
    contents.replace (0, cont.size(), cont);
}
// ...
```

16.53 编写一个程序调用上题中定义的 `count` 函数, 首先传给该函数一个 `double` 类型 `vector`, 然后传递一个 `int` 型 `vector`, 最后传递一个 `char` 型 `vector`。

// 16.53\_template\_count\_vector.cpp: 定义控制台应用程序的入口点。

```
//
#include "stdafx.h"
#include <vector>
#include <iostream>
using namespace std;

template< typename T1, typename T2 > int
count( vector<T1> & vec, T2 tVal )
{
    int iCnt = 0;
    for ( vector<T1>::iterator it = vec.begin(); it !=
vec.end(); ++it )
    {
        if ( (*it) == tVal )
        {
            iCnt++;
        }
    }
    return iCnt;
}

int _tmain(int argc, _TCHAR* argv[])
{
    vector<double> dVec(10, 8.6);
    vector<int> iVec(9, 6);
    vector<char> cVec(8, 'd');

    cout << "\n\tThe '8.6' in the vector<double>
dVec(10, 8.6), its time is:\t" << count( dVec, 8.6 ) << endl;

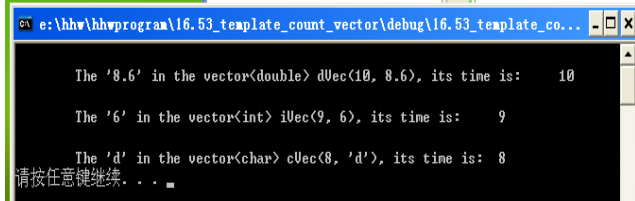
    cout << "\n\tThe '6' in the vector<int> iVec(9, 6),
its time is:\t" << count( iVec, 6 ) << endl;

    cout << "\n\tThe 'd' in the vector<char> cVec(8,
'd'), its time is:\t" << count( cVec, 'd' ) << endl;
```

```

system("pause");
return 0;
}

```



16.60 讨论这两个设计的优缺点：为 `const char*` 定义该类的特化版本和只特化 `push` 和 `pop` 函数。具体而言，比较 `front` 的行为的异同以及用户代码中的错误破坏 `Queue` 元素的可能性。

在为 `const char*` 定义该类的特化版本时，优点是：`front` 函数返回 `string` 对象，该 `string` 对象是 `Queue` 中 `QueueItem` 节点中 `item` 部分所包含的 `string` 对象的副本，用户代码对该副本对象的任意使用都不会破坏 `Queue` 元素；缺点是：定义一个特化类需要修改全部的成员。

只特化 `push` 和 `pop` 函数时，优点是：`front` 函数返回类型为 `const char*` 的指针，该指针是 `Queue` 中 `QueueItem` 节点中 `item` 部分所包含的指针的副本。两个指针指向同一块内存，用户代码对该指针的使用有可能破坏 `Queue` 元素；优点是只需要修改 `push` 和 `pop` 函数为特化版本即可。

16.61 实现 `compare` 函数的三个版本。在每个函数中包含一个输出语句，指出正在调用哪个函数。使用这些函数检查对其余问题的回答。

```

template<typename T> int compare ( const T& v1,
const T& v2 )
{
    cout << "Using compare 2 objects: int compare
( const T& v1, const T& v2 )" << endl;
    if ( v1 < v2 ) return -1;
    if ( v2 < v1 ) return 1;
    return 0;
}

```

```

template < class U, class V > int compare ( U v1, U v2,
V beg )
{
    cout << " Using compare elements int 2

```

```

sequences" << endl;
return 0;
}

```

```

int compare( const char* p1, const char* p2 )
{
    cout << " Using ordinary function to handle
C-style character strings" << endl;
    return strcmp( p1, p2 );
}

```

## 第十七章 用于大型程序的工具

## 第十八章 特殊工具与技术