第 7 章

深入理解类

本章将通过介绍如何使类对象更像 C++基本类型那样工作，来帮助读者更深入地理解类的知识。通过本章的学习，应该完成以下学习目标：

- ☑ 理解需要析构函数的原因并学会实现类的析构函数
- ☑ 理解何时必须编写类的复制构造函数并能够编写
- ☑ 理解运算符重载并能够实现类对象的+或*重载运算符
- ☑ 掌握类模板的定义和使用方法
- ☑ 学会在本地 C++程序中使用 string 类进行字符串操作

7.1 类的析构函数

析构函数用于销毁不再需要或不再有效的对象。当对象不再有效时，程序会自动调用类的析构函数。销毁对象的同时会释放该对象的数据成员(那些即使没有类对象存在时也将继续存在的静态成员除外)占用的内存。类的析构函数是与类同名的成员函数，只是类名前需要加个否定号(~)。析构函数不返回任何值，也没有定义的形参。例如对于 CBox 类来说，其析构函数的原型为：

```
~CBox();
```

注意：析构函数与动态内存分配有关。当我们在自由存储器中为类成员分配内存时，除必须利用构造函数外，还必须利用析构函数。另外，使用动态内存分配的类成员还要求我们自定义复制构造函数。

7.1.1 默认的析构函数

我们迄今为止所使用过的所有对象都是被类的默认析构函数自动销毁的，但默认的析构函数不能删除 new 运算符在自由存储器中分配的对象或对象成员。如果类成员占用的空间是在构造函数中动态分配的，那么我们必须自定义析构函数，然后显式使用 delete 运算符来释放构造函数中使用 new 运算符分配的内存，就像销毁普通变量一样。

为了帮助读者理解何时调用类的析构函数，在类 CBox 中添加析构函数：

```
#include<iostream>
using namespace std;
class CBox
{
```

```
  public:
    ~CBox()
    {
      cout<<"默认析构函数被调用。"<<endl;
    }
    CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                          m_Length(lv), m_Width(bv), m_Height(hv)
    {
      cout<<"构造函数被调用。"<<endl;
    }
    double Volume() const
   {
      return m_Length*m_Width*m_Height;
    }
    int Compare(CBox *pBox) const
    {
      return this->Volume()>pBox->Volume();
    }
  private:
    double m_Length;
    double m_Width;
    double m_Height;
};
int main()
{
  CBox boxes[5];
  CBox match(2.3,1.2,0.8);
  CBox cigar(2.0,2.1,4.8);
  CBox *pBox1=&cigar;
  CBox *pBox2=0;
  cout<<"cigar 的体积是："<<pBox1->Volume()<<endl;
  pBox2=boxes;
  boxes[2]=match;
  cout<<"boxes[2]的体积是："<<(pBox2+2)->Volume()<<endl;
  return 0;
}
```

程序的运行输出结果为：

```
构造函数被调用。
构造函数被调用。
构造函数被调用。
构造函数被调用。
构造函数被调用。
构造函数被调用。
构造函数被调用。
cigar 的体积是：20.16
boxes[2]的体积是：2.208
默认析构函数被调用。
默认析构函数被调用。
默认析构函数被调用。
默认析构函数被调用。
默认析构函数被调用。
```

```
默认析构函数被调用。
默认析构函数被调用。
```

示例说明：

CBox 类的析构函数仅仅显示一条被调用的信息。在程序结束时，每个对象都要调用一次析构函数。每出现一次构造函数的调用，就有一次匹配的析构函数调用。我们不需要在程序中显式调用析构函数，当某个对象无效时，编译器会自动调用该类的所有析构函数。

7.1.2 析构函数与动态内存分配

我们以后会发现，在 C++编程中经常需要为类的数据成员动态分配内存。我们可以在构造函数中使用 new 运算符来为对象成员分配空间。在这种情况下，我们必须提供适当的析构函数，在不需要该对象时释放其占用的内存空间。

假设我们希望定义一个类，其中每个对象都是描述性的信息。这个类应该尽可能高效地利用内存，因此不能将数据成员定义成足以容纳所需最大长度字符串的 char 数组。我们应该在创建对象时在自由存储器中为消息分配内存。类的定义如下所示：

```
class CMessage
{
  private:
    char* pmessage;
  public:
    void ShowIt() const
    {
      cout<<pmessage<<endl;
    }
    CMessage(const char* text = "Default message")
    {
      pmessage = new char[strlen(text) + 1];
      strcpy(pmessage,text);
    }
    ~CMessage();
};
```

上面的类定义了一个数据成员 pmessage，该成员是一个指向文本串的指针，是在类的 private 部分定义的，因此不能从类的外部访问它。在 public 部分，ShowIt()函数将 CMessage 对象输出到屏幕上。这里还定义了构造函数，并添加了析构函数的原型~CMessage()。

类的构造函数要求实参是字符串，如果不传递任何实参，它将使用为形参指定的默认字符串。在类的构造函数中，通过使用库函数 strlen()，获得字符串实参的长度。由于这样得出的长度不包括终止字符串，所以通过将返回值加 1，即可得出在自由存储器中存储该字符串所需的字节数。在使用 new 运算符获得供字符串使用的字符之后，我们使用库函数 strcpy()将为构造函数提供的字符串实参复制到为字符串分配的内存中。

接下来我们需要编写类的析构函数，以释放为消息分配的内存。如果不为该类提供析构函数，程序将无法释放为类对象分配的内存，而如果程序中创建了大量 CMessage 对象，那么将导致自由存储器被逐渐耗尽，直到程序失败。

注意：在一个被程序调用多次的函数中创建临时的 CMessage 对象时，您可以认为该对象在从函数返回时已被销毁，事实上并非如此，自由存储器中的内存并没有释放。因此，没调用一次该函数，就有更多的自由存储器内存被抛弃的 CMessage 对象占用。

由于是在类的外部定义析构函数，所以代码如下所示：

```
Cmessage::~Cmessage()
{
  cout <<"析构函数被调用。"<<endl;
  delete[] pmessage;
}
```

上面的析构函数的作用是显示一条消息，告诉我们析构函数被调用，然后使用 delete 运算符释放 pmessage 成员指向的内存。注意，delete 运算符后面的方括号是必需的，因为这里是在删除数组(char 数组)。

下面的程序演示了析构函数的用法：

```
#include <iostream>
#include <cstring>
using namespace std;
//类 Cmessage 的定义内容
//析构函数的定义内容
int main()
{
  Cmessage motto("差之毫厘，谬之千里。");
  Cmessage* pM = new Cmessage("失败乃成功之母。");
  motto.ShowIt();
  pM->ShowIt();
  //delete pM;
  return 0;
}
```

程序的运行输出结果为：

```
差之毫厘，谬之千里。
失败乃成功之母。
析构函数被调用。
```

示例说明：

在 main()函数的开始部分，我们声明并定义了一个已初始化的 CMessage 对象 motto。在第二条声明语句中，我们定义了一个指向 CMessage 对象的指针 pM，并使用 new 运算符为该指针指向的 CMessage 对象分配内存。对 new 运算符的调用将调用 CMessage 类的构造函数，结果是再次调用 new 运算符为数据成员 pmessage 指向的消息文本分配空间。

虽然我们创建了两个 CMessage 对象，但输出中只记录了一次析构函数调用。编译器为对象 motto 调用了析构函数，是因为虽然该对象的数据成员占用的内存是由构造函数在自由存储器中分配的，但它只是一个普通的自动对象。pM 指向的对象就不同了，我们在自由存储器中分配内存，而编译器不负责删除在自由存储器中创建的对象，所以必须使用 delete 将其删除。使 main()函数中 return 语句前的 delete 语句不再是注释形式，则可以得到

如下输出结果：

```
差之毫厘，谬之千里。
失败乃成功之母。
析构函数被调用。
析构函数被调用。
```

这样一来，析构函数被调用了两次。显然，delete 只处理函数中 new 运算符分配的内存，即只释放指针 pM 指向的内存。由于 pM 指向一个 CMessage 对象，所以 delete 还要调用析构函数来释放该对象的成员所占用的内存。因此，当我们使用 delete 删除 new 运算符动态创建的对象时，delete 将在释放该对象占用的内存之前，首先调用该对象的析构函数。

7.2 实现复制构造函数

当我们动态地为类成员分配空间时，会有些不合适的对象存在于自由存储器中。就 CMessage 类来说，默认的复制构造函数就不合适。分析如下两行代码：

```
CMessage motto1("镭辐射会导致基因突变。");
CMessage motto2(motto1);
```

上述语句中，默认复制构造函数的作用是将类对象 motto1 的指针成员存储的地址复制到 motto2 中。由于默认复制构造函数实现的复制过程只是将原来对象的数据成员中存储的数值复制到新对象中，因此，这两个对象将共享仅有的一个字符串，如图 7-1 所示。

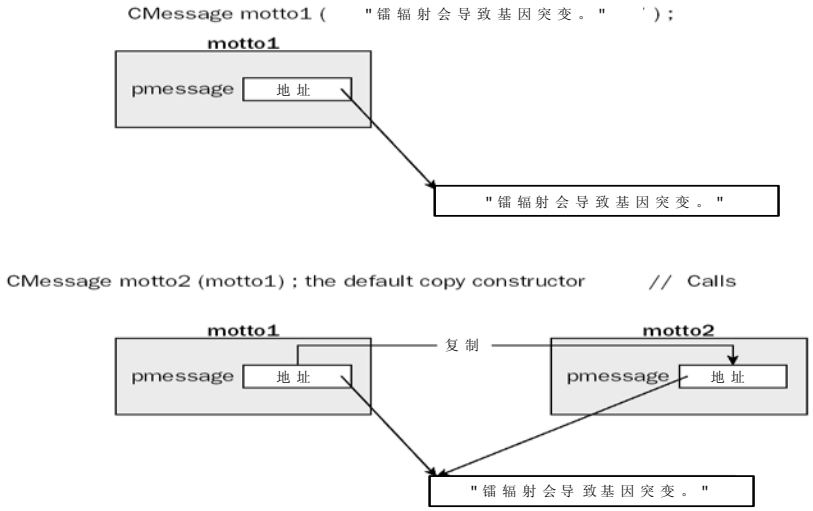

图 7-1 两个对象共享一个字符串

在任何一个对象中对字符串进行修改，都会在另一个对象中得到反映。例如，如果 motto1 被销毁，那么 motto2 中的指针将指向已经被释放、当前可能已用于其他对象的内存区域，导致出错。反之亦然。

解决方案是提供一个类复制构造函数来代替默认版本。如下所示，我们可以在类的

public 部分实现该函数：

```
CMessage(const CMessage& initM)
{
pmessage=new char[strlen(initM.pmessage)+1];
strcpy(pmessage,initM.pmessage);
}
```

第 6 章讲过，为了避免对复制构造函数的无穷调用，我们必须将形参指定为 const 引用。该复制构造函数首先分配足够的容纳 initM 对象中字符串的内存，将地址存入新对象的数据成员中，然后复制初始化对象中的文本串。这样一来，新对象与旧对象相同，但与旧对象完全无关。

请不要因为没有用另一个 CMessage 类对象初始化同类的对象就认为是安全的。考虑如下语句：

```
CMessage thought("Eye awl weighs yews my spell checker.");
DisplayMessage(thought);
```

其中 DisplayMessage()函数的定义如下所示：

```
void DisplayMessage(CMessage localMsg)
{
  cout << endl << "The message is: "<< localMsg.ShowIt();
  return;
}
```

代码看起来十分简单，但却存在着致命错误！问题就出在 DisplayMessage()函数的形参上。形参是一个 CMessage 对象，因此调用过程中实参是通过传值方式传递的。使用默认的复制构造函数，并顺序发生如下事件：

❶ 创建 thought 对象，在自由存储器中为消息 “Eye awl weighs yews my spell checker.” 分配空间。

❷ 调用 DisplayMessage()函数。由于实参是以按值方式传递的，所以使用默认的复制构造函数创建实参的副本 localMsg。现在，副本中的指针指向自由存储器中原来的对象所指向的字符串。

❸ DisplayMessage()函数结束时，局部对象 localMsg 不再有效。因此程序调用 CMessage 类的析构函数，通过释放 pmessage 指针所指向的内存，删除这个局部对象。

❹ 从 DisplayMessage()函数返回时，原来的对象 thought 所包含的指针仍然指向刚刚被释放的内存区域。当我们下次使用原来的对象时，程序就会发生异常。

如果某个类拥有动态定义的成员，我们又利用按值传递机制给函数传递该类的对象，那么对这个函数的任何调用都将出错。由此得出以下结论：

如果动态为本地 C++类的成员分配空间，则必须实现复制构造函数。

7.3 运算符重载

运算符重载是 C++中十分重要的功能，它使得我们能够使用像“+”、“－”、“*”这样的标准 C++运算符来处理复杂的自定义数据类型的对象。例如，我们可以重新定义运算符“>”，从而使该运算符用于前面定义的 CBox 类对象时，如果第一个实参的体积比第二个大，就返回值 true。

在 C++中，除了以下几个运算符外，任何其他运算符都是可以重载的：

- 作用域解析运算符——::
- 条件运算符——?:
- 直接成员访问运算符——•
- sizeof 运算符——sizeof
- 对指向类成员的指针解除引用的运算符——.*

很显然，应确保标准运算符的重载版本与原来的用途一致，或者至少在操作上相当直观。如果为某个类重载的“+”运算符却执行使类对象相乘的操作，这显然是不明智的。

7.3.1 实现重载的运算符

为了对某个类实现重载的运算符，我们必须编写特殊的函数。假设在类定义内重载>运算符的函数是 CBox 类的成员，则该函数的声明如下所示：

```
Class CBox
{
  public:
    bool operator>(CBox aBox) const;
}
```

这里的单词 operator 是个关键字，该关键字结合运算符符号或名称(上面是>)定义一个运算符函数。这里的函数名是 operator>()。关键字和运算符之间有无空格都行，但前提是没有歧义。歧义出现在运算符是名称而非符号的时候，比如 new 或 delete。因此，在编写这些运算符的重载函数时，必须在关键字 operator 和运算符名称间加个空格。注意上面代码中将函数 operator>()声明为 const，因为该函数不修改类 CBox 的数据成员。

在 operator>()运算符函数中，运算符的右操作数由函数形参定义，左操作数由 this 指针隐式定义。因此，对于语句

```
if(box1>box2)
    cout<<"box1 体积大于 box2!"<<endl;
```

来说，括号中的表达式将调用重载的运算符函数，它与下面的这个函数调用等价：

```
box1.operator>(box2);
```

表达式中的 CBox 对象与运算符函数形参之间的对应关系如图 7-2 所示。

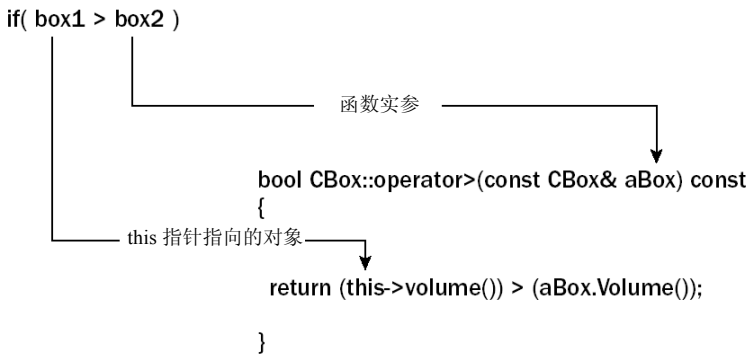

图 7-2　CBox 对象与运算符函数形参之间的对应关系

假设运算符函数 operator>()在类外部定义，代码如下：

```
bool CBox::operator>(const CBox& aBox) const
{
   return this->Volume() > aBox.Volume();
}
```

该函数使用引用形参，以避免被调用时不必要的复制开销。由于该函数不需要修改调用它的对象，而且要比较的是 CBox 类的 const 对象，所以将其声明为 const。return 表达式使用成员函数 Volume()计算 this 指向的 CBox 对象的体积，然后使用基本运算符>将结果与对象 aBox 的体积进行比较，比较结果将自动转换为该运算符函数的返回类型 bool。

使用上面实现的 operator>()运算符函数，我们仍然有很多事情不能做。例如，CBox 对象可能涉及诸如“aBox>20”的表达式。为了使 operator>()函数支持这种比较 CBox 对象与数值的表达式，需要修改函数的定义。首先，将类定义内的成员函数声明为：

```
bool operator>(const double& value) const;
```

>运算符的右操作数对应于这里的函数形参，作为左操作数的 CBox 对象是由隐式指针 this 传递的。该重载运算符的实现同样很容易：

```
bool CBox::operator>(const double& value) const
{
   return this->Volume() > value;
}
```

那么，对于诸如“20.0>aBox”的情况呢？也许有人认为，通过实现接受 double 类型右实参的 operator<()运算符函数，然后相应得改写这样的表达式，就可以完成该功能。但实际上，实现<运算符也是比较 CBox 对象所必需的。在实现对某种对象类型的支持时，我们不应该人为限制在表达式中使用这种对象的方式。对象的使用应尽可能地自然。

成员运算符函数总是以指针 this 作为左边的实参。由于这里左边的实参是 double 类型，所以我们不能以成员函数的形式实现该运算符。那么就只剩下普通函数或友元函数这两种选择。我们不需要访问 CBox 类的 private 成员，所以该函数不必是友元函数。因此，我们可以将左操作数属于 double 类型的重载>运算符实现为普通函数。由于该函数不是成员函

数，其原型应放在类定义的外部：

```
bool operator>(const double& value，const CBox& aBox);
```

该函数的具体实现如下所示：

```
bool operator>(const double& value，const CBox& aBox)
{
   return value>aBox.Volume();
}
```

下面的程序展示了>运算符重载函数的工作过程：

```
#include<iostream>
using namespace std;
class CBox
{
  public:
    CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                          m_Length(lv), m_Width(bv), m_Height(hv)
     {
       cout<<"构造函数被调用。"<<endl;
     }
    double Volume() const
    {
       return m_Length*m_Width*m_Height;
    }
    bool operator>(const CBox& aBox) const
     {
       return this->Volume() > aBox.Volume();
     }
    bool operator>(const double& value) const
     {
       return this->Volume() > value;
     }
   ~CBox()
    {
        cout<<"析构函数被调用。"<<endl;
    }
  private:
     double m_Length;
     double m_Width;
     double m_Height;
};
 bool operator>(const double& value,const CBox& aBox)
 {
    return value>aBox.Volume();
 }
int main()
{
   CBox smallBox(4.0,2.0,1.0);
   CBox mediumBox(10.0,4.0,2.0);
   if(mediumBox>smallBox)
```

```
        cout<<"mediumBox 的体积大于 smallBox。"<<endl;
    if(mediumBox>50.0)
        cout<<"mediumBox 的体积大于 50.0。"<<endl;
    else
           cout<<"mediumBox 的体积不超过 50.0。"<<endl;
    if(10.0>smallBox)
        cout<<"smallBox 的体积小于 10.0。"<<endl;
    else
        cout<<"smallBox 的体积不小于 10.0。"<<endl;
    return 0;
}
```

程序的运行输出结果如下所示：

```
构造函数被调用。
构造函数被调用。
mediumBox 的体积大于 smallBox。
mediumBox 的体积大于 50.0。
smallBox 的体积小于 10.0。
析构函数被调用。
析构函数被调用。
```

7.3.2　重载赋值运算符

编译器默认会为类提供重载的赋值运算符函数，但默认版本仅仅提供逐个成员的复制过程，与默认复制构造函数的功能类似。但是，请不要混淆默认复制构造函数与默认赋值运算符。默认复制构造函数是通过声明以现有同类对象进行初始化的类对象，或者通过以传值方式给函数传递对象而被调用的；而默认赋值运算符是在赋值语句的左边和右边是同类对象时被调用的。

通常情况下，使用类的默认赋值运算符没有任何问题。但对于那些动态为成员分配空间的类而言，我们就需要仔细考虑这些类的需求了。以前面讨论复制构造函数时使用的 CMessage 类为例，该类有个成员 pmessage，它是指向字符串的指针。现在考虑默认赋值运算符可能产生的结果：

```
motto2=motto1;//motto1 和 motto2 均为 CMessage 对象
```

为该类使用默认赋值运算符的后果基本上与使用默认复制构造函数相同，因为两个对象都有一个指向相同对象的指针，任何一个对象的字符串发生变动，受影响的都将是两个对象。另外还有一个问题，就是当该类的实例之一被销毁时，其析构函数将释放该字符串所占用的内存，因此，另一个对象包含的指针就可能指向已被其他对象占用的内存。我们可以使用自己的赋值运算符函数来解决上述问题，要做的事情就是将文本复制到目标对象所拥有的内存区域。该函数被定义在类定义的外部：

```
CMessage& operator=(const CMessage& aMess)
{
delete[] pmessage;
pmessage = new char[strlen(aMess.pmessage)+1];
strcpy(this->pmessage, aMess.pmessage);
```

```
return *this;
}
```

这里的赋值看起来比较简单。我们从赋值运算符函数中返回的是引用。从表面上看，返回引用意味着不需要返回任何东西，但我们需要进一步考虑该运算符的使用方式。我们有时会在表达式的右边使用赋值操作的结果，例如：

```
motto1=motto2=motto3;
```

由于赋值运算符具有右结合性，即首先执行将 motto3 赋给 motto2 的操作，所以该语句可以被翻译成：

```
motto1=(motto2.operator=(motto3));
```

语句最终变成：

```
motto1.operator=(motto2.operator=(motto3));
```

要使上述语句工作，我们当然必须有返回对象。括号内对 operator=()函数的调用必须返回一个对象，以作为另一个 operator=()函数调用的实参。对于本例，返回类型为 CMessage 或 CMessage&都可以。

考虑如下语句：

```
(motto1=motto2)=motto3;
```

这是完全合法的代码，括弧旨在确保首先执行最左边的赋值。该语句可以被翻译成：

```
(motto1.operator=(motto2))=motto3;
```

当我们将剩下的赋值操作表示成显式的重载函数调用时，该语句最终变成：

```
(motto1.operator=(motto2)).operator=motto3;
```

现在的情况是，从函数 operator=()返回的对象被用来调用 operator=()函数。如果返回类型仅仅是 CMessage，则该语句是不合法的，因为实际返回的是原始对象的临时副本，编译器不允许使用临时对象调用成员函数。确保上述语句能够正常编译和工作的唯一方法是返回可以作为左值的引用。因此，如果希望实现使用赋值运算符处理类对象的灵活性，则唯一可能的返回类型是 CMessage&。

operator=()运算符函数仍然存在缺点。考虑如下语句：

```
motto1=motto1;
```

很显然，我们不会这么做。但可能会使用如下语句：

```
motto1=*pMess;
```

如果指针 pMess 指向 motto1，那么这条语句实质上就是上面的那条赋值语句。这种情

况下，目前的赋值运算符函数将释放供 motto1 使用的内存，然后基于已经被删除的字符串的长度另外分配一些内存，并试图复制当时可能已经被破坏的旧内存。通过在函数的开头检查左右操作数是否相同，我们可以修正上述问题：

```
CMessage& operator=(const CMessage& aMess)
{
   if(this==&aMess)
      return *this;
    ……
```

下面的程序演示了重载赋值运算符的使用过程：

```
#include <iostream>
#include <cstring>
using namespace std;
class CMessage
{
  private:
    char* pmessage;
  public:
    void ShowIt() const
    {
      cout<<pmessage<<endl;
    }
    void Reset()
    {
      char* temp=pmessage;
      while(*temp)
        *(temp++)='$';
    }
    CMessage& operator=(const CMessage& aMess)
    {
      if(this == &aMess)
        return *this;
      delete[] pmessage;
      pmessage=new char[strlen(aMess.pmessage)+1];
      strcpy(this->pmessage, aMess.pmessage);
      return *this;
    }
    CMessage(const char* text = "Default message")
    {
      pmessage = new char[ strlen(text) +1 ];
      strcpy(pmessage, text);
    }
    ~CMessage()
    {
      cout<<"析构函数被调用!"<<endl;
      delete[] pmessage;
    }
};
int main()
{
```

```
    CMessage motto1("活到老，学到老");
    CMessage motto2;
    cout<<endl<<"motto2 包含--";
    motto2.ShowIt();
    motto2 = motto1;
    cout<<endl<<"motto2 包含--";
    motto2.ShowIt();
    motto1.Reset();
    cout<<endl<< "motto1 现在包含--";
    motto1.ShowIt();
    cout<<endl<< "motto2 现在包含--";
    motto2.ShowIt();
    return 0;
}
```

程序的运行输出结果如下所示：

```
motto2 包含--Default message
motto2 包含--活到老，学到老
motto1 现在包含--$$$$$$$$$$$$
motto2 现在包含--活到老，学到老
析构函数被调用!
析构函数被调用!
```

示例说明：

从该程序的输出可以看出，motto1 和 motto2 的消息之间没有任何联系，除非我们显式地使它们相等。Reset()函数的作用是将消息重置为美元符号字符串。

通过本节的学习，我们可以得到另一条重要结论：

如果需要给类的数据成员动态分配空间，则必须实现赋值运算符。

7.3.3 重载加法运算符

以两个 CBox 对象相加为例，我们希望相加的结果是两个 CBox 对象的体积的和。那么，CBox 类的加法运算符的相加过程可以用图 7-3 来表示。下面的程序演示了其具体工作过程。

```
#include <iostream>
using namespace std;
class CBox
{
  public:
    CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0): m_Height(hv)
    {
      m_Length = lv > wv? lv: wv;
```

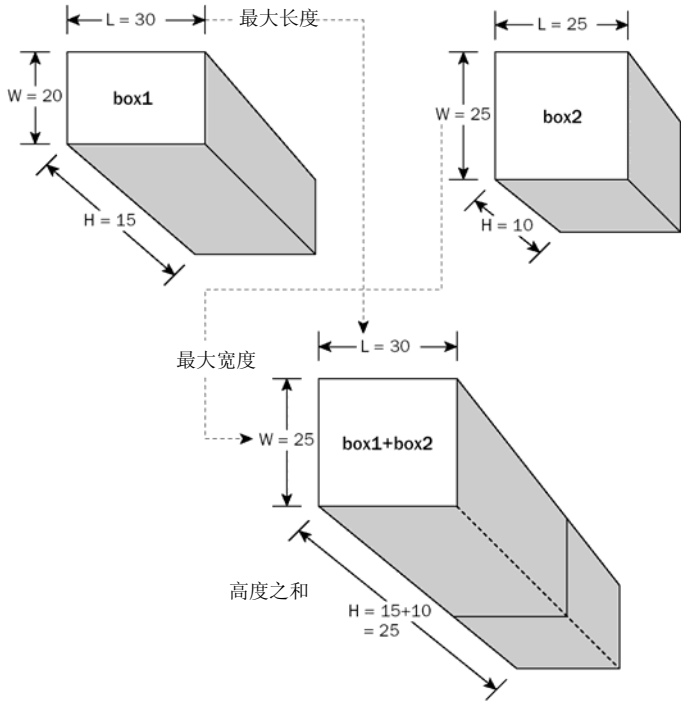

图 7-3　CBox 类的加法运算符的相加过程

```
      m_Width = wv < lv? wv: lv;
    }
    double Volume() const
    {
      return m_Length*m_Width*m_Height;
    }
    CBox operator+(const CBox& aBox) const
    {
      return CBox(m_Length > aBox.m_Length? m_Length:aBox.m_Length,
                  m_Width > aBox.m_Width? m_Width:aBox.m_Width,
                  m_Height + aBox.m_Height);
    }
    void ShowBox() const
    {
      cout<<m_Length<<" "<<m_Width<<" "<<m_Height<<endl;
    }
  private:
    double m_Length;
    double m_Width;
    double m_Height;
};
int main()
{
  CBox smallBox(4.0, 2.0, 1.0);
  CBox mediumBox(10.0, 4.0, 2.0);
  CBox aBox;
  CBox bBox;
  aBox = smallBox + mediumBox;
```

```
    cout << "aBox 的尺寸是：";
    aBox.ShowBox();
    bBox = aBox + smallBox + mediumBox;
    cout << "bBox 的尺寸是：";
    bBox.ShowBox();
    return 0;
}
```

程序的运行输出结果如下所示：

```
aBox 的尺寸是：10  4   3
bBox 的尺寸是：10  4   6
```

示例说明：

CBox 类定义中删除了析构函数(因为它对 CBox 类而言不是必需的)，构造函数被修改成可以保证 m_Length 成员不小于 m_Width 的成员，另外还添加了一个输出 CBox 对象尺寸的 ShowBox()函数。使用该函数，我们可以验证重载的加法操作是否像预期的那样工作。

7.3.4 重载递增和递减运算符

在本地 C++中，对应于递增、递减运算符的前缀和后缀形式的重载运算符是不同的。例如，下面是在名为 Width 的类中定义这些运算符的方法：

```
class Width
{
  private:
     double wid;
  public:
     Width& operator++();
     const Width operator++(int);
     Width& operator--();
     const Width operator--(int);
}
```

Width 类只需要存储 double 类型的数值，该类十分简单，我们这里旨在说明如何重载递增和递减运算符。

区分重载运算符前缀和后缀形式的首要方法是利用形参列表。前缀形式没有形参，后缀形式有一个 int 类型的形参(该形参只是为了将其同前缀形式区分开，别无它用)。

前缀形式的递增和递减运算符在操作数的值被用于表达式之前将其递增或递减，因此在当前对象被递增或递减之后，我们只需返回该对象的引用即可。例如：

```
Width& Width::operator++()
{
    ++(this->len);
    return *this;
}
```

而在后缀形式中，操作数是在其当前值被表达式使用之后递增或递减的。要实现这一点，需要在递增或递减当前对象之前创建当前对象的副本，并在修改过当前对象之后返回新创建的副本对象。例如：

```
const Width& Width::operator++(int)
{
    Width width=*this;//复制当前对象
    ++*this;//递增当前对象
    return width;//返回原来的对象，因为当前值在表达式使用之后才递增
}
```

我们这里将返回值声明为 const，是为了防止类似 data++++这样的表达式被误编译。

7.4　类　模　板

类模板不是类，而只是编译器用来生成类代码的一种“配方”，具有类似函数模板的机制。和函数模板一样，类模板也是通过模板中尖括号内的形参类型来确定希望生成的类，如图 7-4 所示。

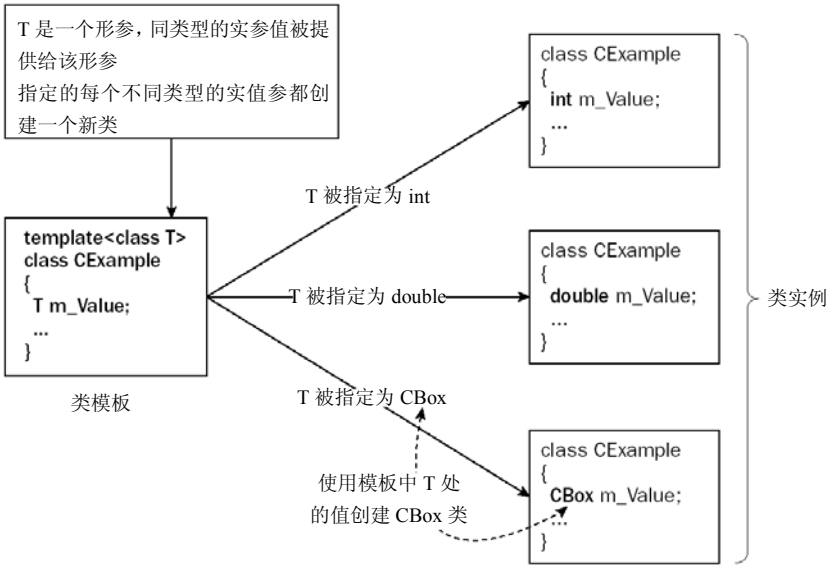

图 7-4　类模板的工作机制

以这种方式生成的类被称作类模板的实例，根据模板创建类的过程被称为实例化模板。当我们以特定的类型实例化模板类的某个对象时，编译器将生成适当的类定义，因此，一个类模板可以衍生出任意数量、各不相同的类。

7.4.1　定义类模板

下面我们通过一个简单的示例来说明如何定义和使用类模板。假设我们希望定义几个可以存储大量数据样本的类，每个类都应当提供一个求出所存储样本最大值的 Max()函数。我们可以定义一个类模板，用以生成可以存储任何类型样本的 CSample 类。

```
template <class T> class CSample
{
public:
    //接收多个样本的构造函数
```

```
    CSample(const T values[],int count)
    {
        m_Free = count < 100 ? count:100;  //是否溢出数组长度
        for(int i = 0 ; i < m_Free ; i++)
            m_Values[i] = values[i];    //存储样本数据
    }
    //接收单个样本的构造函数
    CSample(const T& value)
    {
        m_Values[0] = value;
        m_Free = 1;
    }
    //默认的构造函数
    CSample()
    {
        m_Free= 0;//没有存储样本数据，所以数组第一个元素为空
    }
    //添加样本值
    bool Add(const T& value)
    {
        bool OK = m_Free < 100;
       if(OK)
            m_Values[m_Free++] = value;
        return OK;
    }
    //获取样本最大值
    T Max() const
    {
        T theMax = m_Free ? m_Values[0] : 0;
        for(int i = 1;i < m_Free; i++)
            if(m_Values[i] > theMax)
                theMax = m_Values[i];
        return theMax;
    }
private:
    T m_Values[100];//用于存储样本数据的数组
    int m_Free;//数组索引
}
```

为了指出正在定义的是模板而非简单的类，我们在关键字 class 和类名 CSample 之前，插入了关键字 template 和尖括号内的类型形参 T。T 是类型变量，它将在我们声明类对象时被具体类型代替(可以是基本类型或类，但必须在类模板的上下文中有意义)。一般而言，如果需要，我们可以在类模板中指定多个形参。

虽然理论上我们可以创建或处理任何数据类型的 CSample 类的对象，但这并不意味着所创建的对象能够编译，且像我们预期的那样工作。实际情况取决于类模板定义所让做的处理，通常一个模板仅仅适用于特定的类型范围。例如，Max()函数隐含地认为>运算符可以用于被处理的任何类型，否则程序将不能编译。毫无疑问，通常我们定义的模板只是为了处理某些类型，但我们却无法限制施加到模板上的类型。

我们也可以将类模板的成员函数定义在模板定义的外部，但首先应将函数声明放在类

模板定义的内部：

```
template <class T> class CSample
{
    ……
    T Max() const;
    ……
}
```

接下来，我们需要为该成员函数的定义创建单独的函数模板，创建时必须使用模板类的名称加上尖括号内的形参，以标识函数模板所属的类模板：

```
template <class T> T CSample<T>::Max() const
{
    T theMax = m_Values[0];
    for(int i = 1;i < m_Free; i++)
        if(m_Values[i] > theMax)
            theMax = m_Values[i];
    return theMax;
}
```

因为该函数模板是形参为 T 的类模板的成员，所以这里的函数模板定义应该有与类模板定义相同的形参。这里只有一个形参 T，但通常可能有多个。如果类模板有两个或更多形参，则每个定义成员函数的模板也应该有同样的形参。作用域解析运算符之前只能使用附带形参名称 T 的类模板名，这是必需的——形参对于识别出根据该模板生成的函数属于哪个类非常重要。类模板的类型是 CSample<T>，其中 T 是我们创建类模板实例时指定的类型。指定的类型将被插入类模板中，从而生成类定义。还将被插入函数模板中，从而生成类中 Max()函数的定义。每个根据类模板生成的类都需要有自己的 Max()函数定义。

在类模板外部定义构造函数和析构函数的方法与上面类似，例如，可以将接收样本数组的那个构造函数在类模板外部定义成：

```
template <class T> T CSample<T>::CSample(T values[],int count)
    {
        m_Free = count < 100 ? count:100;
        for(int i = 0 ; i < m_Free ; i++)
            m_Values[i] = values[i];
    }
```

7.4.2　根据类模板创建对象

当我们使用函数模板定义的函数时，编译器能够根据使用的实参类型生成函数。函数模板的类型形参是通过使用特定的函数隐式确定的。类模板则有所不同，为了以类模板为基础创建对象，我们必须在声明中指定类名后面的类型形参。例如，为了声明一个 CSample<>对象来处理 double 类型的样本，需要将声明写成如下形式：

```
CSample<double> myData(10.0);
```

该语句定义了一个 CSample<double>类型的对象，它可以存储 double 类型的样本。该对象使用值为 10.0 的一个样本创建的。下面的程序演示了类模板的用法：

```
#include<iostream>
using namespace std;
class CBox
{
  public:
     CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                          m_Length(lv), m_Width(bv), m_Height(hv){    }
     double Volume() const
     {
        return m_Length*m_Width*m_Height;
     }
     int CBox::operator>(const CBox& aBox) const
     {
        return this->Volume() > aBox.Volume();
     }
     int operator>(const double& value) const
     {
       return Volume() > value;
     }
  private:
    double m_Length;
    double m_Width;
    double m_Height;
};
template <class T> class CSample
……   //前面类模板的定义放在此处。
int main()
{
  CBox boxes[] = {
                 CBox(8.0, 5.0, 2.0),
                 CBox(5.0, 4.0, 6.0),
                 CBox(4.0, 3.0, 3.0)
                };
  CSample<CBox> myBoxes(boxes, sizeof boxes / sizeof CBox);
  CBox maxBox = myBoxes.Max();
  cout << "这个最大的箱子的体积是："
       << maxBox.Volume();
  return 0;
}
```

程序的运行输出结果是：

```
这个最大的箱子的体积是：120
```

示例说明：

CBox 类的定义中重载了>运算符，这是因为类模板中 Max()函数的表达式中用到了 CBox 对象的比较。在 main()函数中，我们创建了一个包含 3 个 CBox 对象的数组，然后使用该数组初始化一个可以存储 CBox 对象的 CSample 对象。CSample 对象的声明基本上与普通对象的声明相同，只是在模板名称后面增加了以尖括号包围的类型形参。

注意，当我们创建类模板的实例时，不能理解成用于创建函数成员的那些函数模板的实例也被创建。编译器只能创建程序中实际调用的那些成员函数的模板实例。事实上，我

们的函数模板甚至可以包含编码错误，而且只要不调用该模板生成的成员函数，编译器就不会报错。大家可以试着在 Add()函数中加入一些错误，可以发现该程序仍然可以编译和运行。

7.4.3　使用多个形参的类模板

要在类模板中使用多个类型形参，只需对前面看到的使用单个形参的示例做些简单的扩展即可。例如，可以定义一个使用两个类型形参的类模板：

```
template <class T1,class T2> class CSample
{
   ……//类模板定义的其他部分
  private:
     T1 m_Value1;
     T2 m_Value2;
}
```

类模板中的形参不仅仅限于数据类型，我们还可以在类定义中使用一些需要以常量或常量表达式进行替换的形参。例如，在前面的类模板 CSample 中，我们将数组 m_Values 定义成包含 100 个元素。我们还可以让该模板的用户在实例化对象时选择数组的大小，方法是将类模板定义成如下形式：

```
template <class T,int Size> class CSample
{
private:
    T m_Values[Size];//用于存储样本数据的数组
    int m_Free;//数组索引
public:
    CSample(const T values[],int count)
    {
        m_Free = count < Size ? count:Size;  //是否溢出数组长度
        for(int i = 0 ; i < m_Free ; i++)
            m_Values[i] = values[i];    //存储样本数据
    }
    CSample(const T& value)
    {
        m_Values[0] = value;
        m_Free = 1;
    }
    CSample()
    {
        m_Free= 0;//没有存储样本数据，所以数组第一个元素为空
    }
    bool Add(const T& value)
    {
        bool OK = m_Free < Size;
        if(OK)
            m_Values[m_Free++] = value;
        return OK;
    }
    T Max() const
```

```
    {
        T theMax = m_Free ? m_Values[0] : 0;
        for(int i = 1;i < m_Free; i++)
            if(m_Values[i] > theMax)
                theMax = m_Values[i];
        return theMax;
    }
}
```

创建对象时给 Size 提供的数值将代替整个模板定义中使用该形参的所有实例。例如，我们可以采用如下形式声明前面那个示例的 CSample 对象：

```
CSample<CBox,3> myBoxes(boxes, sizeof boxes / sizeof CBox);
```

使 Size 成为模板形参的结果是，那些存储相同类型的对象但 Size 形参值不同的模板实例完全是不同的类，而且不可能被混淆，例如 CSample<CBox,3>类型的对象和 CSample<CBox,4>类型的对象。

当实例化模板时，我们必须小心处理那些包含比较运算符的表达式。语句

```
CSample<aType,x>y?10:20>MyType();
```

将不能被正确编译，因为表达式中 y 前面的>被解释成右尖括号。应该写成

```
CSample<aType,(x>y?10:20)> MyType();
```

以确保第二个模板实参的表达式不会与尖括号混淆。

7.5 字符串进阶

前面提到过，<string>标准头文件中定义了表示字符串的 string 类和 wstring 类。这两个类在<cstring>头文件中被定义为类模板 basic_string<T>的实例：string 类被定义为 basic_string<char>，wstring 类被定义为 basic_string<wchar_t>。因此 string 类表示 char 类型的字符串，wstring 类表示 wchar_t 类型的字符串。这些字符串类型比以空字符结尾的字符串容易使用得多，它们还配备了一整套有用的函数。

提示：由于 string 和 wstring 都是同一类模板的实例，所以它们提供的功能相同。

7.5.1 创建字符串对象

字符串的创建十分简单，也有多种方式。例如，可以用如下形式创建并初始化一个字符串对象：

```
string sentence = "this is a sentence."
```

由于用来初始化 sentence 的字符串字面值的末尾没有空字符，因此 sentence 的字符串长度是字符串中的字符个数。我们可以在任何时候通过调用字符串对象的 length()函数来查

看字符串对象所封装的字符串的长度。例如：

```
sentence.length();
```

可以采用如下方式将字符串读到字符串对象中：

```
cin>>sentence;
```

然而，以这种方式读取字符时会忽略开头的空格，直到发现非空格的字符为止。另外，当我们在一个或多个非空字符后面输入一个空格后，这种方式将结束读取。而实际上，我们常常希望将文本读入一个包括空格的字符串对象中。此时，可以使用在<string>头文件中定义的 getline()函数模板。例如：

```
getline(cin,sentence,'*');
```

getline()函数模板专门用来将数据从流中读入字符串或 wstring 对象中。它有 3 个参数，第 1 个是输入流，这里是 cin；第 2 个是接收输入的对象，这里是 sentence；第 3 个终止读取的字符，这里是*。

当然，也可以用函数表示法来初始化 string 对象：

```
string sentence("this is a sentence.");
```

如果在创建 string 对象时没有指定初始字符串字面值，那么该对象会包含一个空字符串，调用该字符串的 length()函数将返回结果 0。

另一种方式是通过重复一个字符指定的次数来初始化字符串对象，例如：

```
string bees(7, 'b');//字符串对象是"bbbbbbb"。
```

最后一种方式是用另一个字符串对象的全部或一部分来初始化字符串对象，例如：

```
string letters(bees);
```

这里用 bees 中包含的字符串来初始化 letters 对象。

为了选择一个字符串对象的一部分作为初始化器，我们可以调用带 3 个实参的字符串构造函数：第 1 个实参是作为初始化字符串来源的字符串对象；第 2 个实参是要选择的第 1 个字符的索引位置；第 3 个实参是要选择的字符的个数。例如：

```
string sentence = "this is a sentence."
string part(sentence,5,14);//第 1 个字符的索引位置是 0，结果为"is a sentence."
```

我们还可以创建 string 对象的数组，并用常规方法来初始化它们。例如：

```
string animals = {"dog","cat","horse","lion","donkey"};
```

7.5.2　连接字符串

字符串最常见的运算可能就是连接两个字符串以形成一个新的字符串了。string 类实现了 operator+()，因此我们可以使用+运算符来直接连接两个字符串，但这也意味着+运算

符的操作数之一必须是 string 对象，我们不能用它来连接两个字符串字面值。例如：

```
string sentence1 = "this is a sentence."
string sentence2 = "this is another sentence."
string combined;           //创建一个空的字符串。
sentence1 = sentence1+"\n";  //连接的字符串将新起一行。
combined = sentence1+sentence2;
cout<<combined<<endl;
```

最后一行的 cout 语句的输出结果会是：

```
this is a sentence.
this is another sentence.
```

我们也可以用+运算符将一个字符连接到一个字符串对象上，因此上面代码中用于转行的语句可以写成：

```
sentence1 = sentence1+'\n';
```

或者

```
sentence1 += '\n';
```

但两者有一个显著区别：+运算符会创建一个包含合并后的字符串的新字符串对象；而+=运算符将作为右操作数的字符串或字符附加到作为左操作数的 string 对象后，因此直接修改 string 对象，而不创建新的对象。

7.5.3 访问与修改字符串

可以通过使用下标运算符[]来访问 string 对象中的任何字符从而读取或重写它，例如：

```
string sentence = "hello kitty!";
for(size_t i=0 ; i<sentence.length(); i++)
if{sentence[i] == ' '}
   sentence[i] = '*';
```

这段代码会依次检查 sentence 字符串中的每个字符，看看它是否为空格，如果是，则用*替换该空格。

用 at()成员也可以得到相同的结果：

```
string sentence = "hello kitty!";
for(size_t i=0 ; i<sentence.length(); i++)
if{sentence.at(i) == ' '}
   sentence.at(i) = '*';
```

两者的区别在于：利用下标值比使用 at()函数快，不过使用下标值的缺点是没有检查索引的有效性，如果索引超出了范围，那么使用下标运算符的结果将不确定；而 at()函数虽然会慢一些，但它会检查索引的有效性，如果索引无效，该函数会抛出一个 out_of_range 的异常。因此，当索引值有可能超出范围时，可以使用 at()函数，将其放在 try 代码块中，并恰当地抛出异常；当我们能确信不会出现索引超出范围的情况时，就使用下标值。

我们可以使用 substr()函数提取现有 string 对象的一部分作为一个新的 string 对象，例如：

```
string sentence = "hello kitty!";
string substring = sentence.substr(6,6);
```

substr()函数的第一个实参是要提取的子串的第一个字符，第二个实参是子串中的字符个数。

通过使用字符串对象的 append()函数，我们可以在字符串末尾添加一个或多个字符、一个字符串字面值或一个 string 对象。例如：

```
string phrase("fall into the pit");
string word("a gain in your wit");
phrase.append(1,',');
phrase.append(word);
phrase.append(2,'!');
```

当执行完这段代码后，phrase 会被改成“fall into the pit, a gain in your wit!!”。当使用带两个实参的 append()版本时，第一个实参指明将被附加到第二个实参指定的字符的次数。当我们调用 append()函数时，函数返回对调用该函数的对象的引用，因此我们可以在一条语句中写出以上 3 个 append()调用：

```
phrase.append(1,',').append(word).append(2,'!');
```

我们可以用 append()将字符串字面值的一部分或 string 对象的一部分附加到一个现有字符串后面，例如：

```
string phrase("the more the merrier.");
string query("any");
query.append(phrase,3,5).append(1,'?');
```

上面这段代码的执行结果是 query 会包含字符串“any more？”。在最后一行语句中，对 append()函数的调用有 3 个实参：第 1 个实参 phrase 是从中提取字符并附加到 query 后面的 string 对象；第 2 个实参 3 是要提取的第 1 个字符的索引位置；第 3 个实参 5 是要附加的字符总数。

有时，仅仅向字符串末尾添加字符还不够，可能还需要在字符串中间的某个位置插入一个或多个字符，这可以通过 insert()函数来实现。例如：

```
string saying("A horse");
string word("blind");
string sentence("He is as good as gold.");
string phrase("a wink too far");
saying.insert(1, " ");
saying.insert(2, word);
saying.insert(2, "nodding", 3);
saying.insert(5, sentence, 2, 15);
saying.insert(20, phrase, 0, 9);
saying.insert(29, " ").insert(30, "a poor do", 0, 2);
```

上面这段代码的执行结果是 saying 将包含字符串“A nod is as good as a wink to a blind

horse”。在 insert()的各个版本中，返回对调用该函数的 string 对象的一个引用。这样，就可以像前面代码片段中的最后一个语句那样将所有调用链接在一起。

insert()的各版本的形参如表 7-1 所示。

表 7-1　insert()的各版本的形参

函 数 原 型	说　　明
string& insert(size_t index, const char* pstring)	在 index 位置插入以空字符结尾的字符串 pstring
string& insert(size_t index, const string astring)	在 index 位置插入 string 对象 astring
string& insert(size_t index, const char* pstring,size_t count)	在 index 位置插入以空字符结尾的字符串 pstring 中的前 count 个字符
string& insert(size_t index, const string astring,size_t count)	在 index 位置插入 string 对象 astring 中的前 count 个字符
string& insert(size_t index, const string astring,size_t start, size_t count)	插入 string 对象 astring 中从 start 位置的字符开始的 count 个字符；子串插在 index 位置
string& insert(size_t index, const char* pstring,size_t start, size_t count)	插入以空字符结尾的字符串 pstring 中从 start 位置的字符开始的 count 字符；子串插在 index 位置

我们可以通过调用 swap()成员函数来交换封装在两个 string 对象之间的字符串，例如：

```
string phrase("The more the merrier.");
string query("Any");
query.swap(phrase);
```

query 中的结果包含字符串“The more the merrier.”phrase 包含字符串“Any”。当然，执行 phrase.swap(query)也能得到同样的效果。

如果我们需要将一个字符串对象转换为以空字符结尾的字符串，可以用 c_str()函数来完成。例如：

```
string phrase("The higher the fewer");
const char *pstring = phrase.c_str();
```

c_str()函数返回一个指向以空字符结尾的字符串的指针，该字符串的内容与 string 对象相同。

我们也可以通过调用 data()成员函数来获得一个字符串对象(作为 char 类型的元素的数组)的内容。注意，该数组仅包含字符串对象中的字符，不包括结尾的空字符。

我们可以通过调用字符串对象的 replace()成员函数来替换字符串对象的一部分。它也有几个版本，如表 7-2 所示。

表 7-2　replace()的各版本的形参

函 数 原 型	说　　明
string& replace(size_t index, size_t count,const char* pstring)	用 pstring 中的前 count 个字符替换从 index 位置开始的 count 个字符

(续表)

函 数 原 型	说　　明
string& replace(size_t index, size_t count,const string astring)	用 astring 中的前 count 个字符替换从 index 位置开始的 count 个字符
string& replace(size_t index, size_t count1,const char* pstring, size_t count2)	用 pstring 中第一个到第 count2 个字符替换从 index 位置开始的 count1 个字符。这个版本可以替换比所替换的子串长或短的子串
string& replace(size_t index1, size_t count1,const string astring, size_t index2,size_t count2)	用 astring 中从 index2 位置开始的 count2 个字符替换从 index1 位置开始的 count1 个字符
string& replace(size_t index, size_t count1,size_t count2, char ch)	用 count2 个字符 ch 替换从 index 位置开始的 count1 个字符

表 7-2 的各个版本都会返回对调用该函数的 string 对象的引用，例如：

```
string proverb("A nod is as good as a wink to a blind horse");
string sentence("It's bath time!");
proverb.replace(38, 5, sentence, 5, 3);
```

这段代码采用表 7-2 中 replace()函数的第 4 个版本，用“bat”替换字符串 proverb 中的“horse”。

7.5.4　比较字符串

我们有一整套比较运算符，用来比较两个字符串对象，或者比较一个字符串对象与一个字符串字面值。string 类中对下列运算符实现了运算符重载：

```
== != < <= > >=
例如：
string dog1("St Bernard");
string dog2("Tibetan Mastiff");
if(dog1 < dog2)
cout << "dog2 comes first!" << endl;
else if(dog1 > dog2)
cout << "dog1 comes first!" << endl;
```

当我们比较两个字符串时，实际上是比较对应的字符，直到发现一对不同的字符，或者到达一个或两个字符串的末尾。当发现两个对应字符不相同时，字符代码的值决定比较结果。如果没有发现不同的字符对，那么字符较少的字符串小于另一个字符串。如果两个字符串包含相同的字符个数，而且对应的字符也相同，则这两个字符串相等。

7.5.5　搜索字符串

可以使用 find()函数来搜索给定的字符串或子串，find()函数有 4 个版本，如表 7-3 所示。

表 7-3　find()的各个使用版本

函　　数	说　　明
size_t find(char ch, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始的字符 ch。可以省略第二个实参，在这种情况下默认值为 0
size_t find(char char* pstr, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始的以空字符结尾的字符串 pstr。可以省略第二个实参，在这种情况下默认值为 0
size_t find(char char* pstr, size_t offset,size_t count)	在 string 对象中搜索从 offset 索引位置开始的以空字符结尾的字符串 pstr 的前 count 个字符
size_t find(const string str, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始的字符串对象 str。可以省略第二个实参，在这种情况下默认值为 0

find()函数的各个版本都返回发现字符或子串的第一个字符的索引位置。如果没有找到要找的条目，该函数会返回值 string::npos。后一个值是在 string 类中定义的一个常量，表示 string 对象中的一个非法位置，它通常用来标识搜索失败。

下面的代码片段演示了 find()函数的部分用法：

```
string phrase("so near and yet so far");
string str("so near");
cout << phrase.find(str) << endl; //输出 0
cout << phrase.find("so far") << endl; //输出 16
cout << phrase.find("So near") << endl;//输出 string::npos = 4294967295
```

string::nops 的值可能根据不同的 C++编译器实现而不同，因此，为了测试它，我们应总是使用 string::npos，而不是使用显式的值。

下面是反复扫描同一个字符串以搜索出现的特定子串的又一示例：

```
string str( "Smith, where Jones had had \"had had\", \"had had\" had."
            " \"Had had\" had had the examiners' approval.");
string substr("had");
cout << "被搜索的字符串是:" << str << endl;
size_t offset = 0;
size_t count = 0;
size_t increment = substr.length();
while(true)
{
  offset = str.find(substr, offset);
  if(offset == string::npos)
  break;
  offset += increment;
  ++count;
}
cout << " 字符串 " << substr << " 被找到的次数是：" << count << endl;
```

在这段代码中，我们搜索 str，查看其中出现“had”的次数。此搜索在 while 循环中完成，其中 offset 记录发现的位置，该位置也用作搜索的起始位置。该搜索从索引位置 0(字符串的开头)开始，每次发现子串时，就将下一次搜索的新起始位置设置为发现的位置加上子串的长度，以确保绕过发现的子串。每次发现子串时，计数就递增。如果 find()返回 stri-

ng::npos，就表示没有发现子串，搜索结束。执行该代码片段会产生如下输出：

```
被搜索的字符串是:

Smith, where Jones had had "had had", "had had" had.
 "Had had" had had the examiners' approval

字符串"had"被找到的次数是：10
```

find_first_of()和 find_last_of()成员函数在 string 对象中搜索出现的给定集合中的任何字符。例如，我们可能在字符串中搜索空格或标点符号(它们可以用来将一个字符串分解为单个单词)。这两个函数都有几个版本，如表 7-4 所示。

表 7-4　find_first_of()和 find_last_of()的各个使用版本

函　　数	说　　明
find_first_of(char ch, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始出现的字符 ch，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_first_of(char* pstr, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始第一次出现的以空字符结尾的字符串 pstring 中的任何字符，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_first_of(char* pstr, size_t offset, size_t count)	在 string 对象中搜索从 offset 索引位置开始第一次出现的以空字符结尾的字符串 pstring 中的前 count 个字符中的任何字符，并返回发现字符的索引位置值，类型为 size_t
find_first_of(string str, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始第一次出现的字符串 str 中的任何字符，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_last_of(char ch, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的字符 ch，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 nops
find_last_of(char* pstr, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的以空字符结尾的字符串 pstr 中的任何字符，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 nops
find_last_of(char* pstr, size_t offset, size_t count)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的以空字符结尾的字符串 pstr 中的前 count 个字符，并返回发现该字符的索引位置值，类型为 size_t
find_last_of(string str, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的字符串 str 中的任何字符，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 npos

对于 find_first_of()和 find_last_of()函数的所有版本，如果没有发现匹配的字符，都会返回 string::npos。

使用与上一个代码片段中相同的字符串，可以看到 find_last_of()函数对字符串“had”所执行的搜索。

```
size_t count = 0;
size_t offset = string::npos;
while(true)
{
offset = str.find_last_of(substr, offset);
if(offset == string::npos)
break;
--offset;
++count;
}
cout << endl << " Characters from the string \"" << substr
<< "\" were found " << count << " times in the string above."
<< endl;
```

这次我们从字符串末尾索引位置 string::npos 开始向后搜索，因为这是默认开始位置。该代码片段的输出为：

```
The string to be searched is:
Smith, where Jones had had "had had", "had had" had. "Had had" had had
the examiners' approval.
Characters from the string "had" were found 38 times in the string above.
```

结果不出意料。记住，我们在搜索字符串 str 中出现的“had”中的任何字符。“Had”和“had”单词中有 32 个，其他单词中有 6 个。因为我们在沿着字符串向后搜索，因此当我们发现一个字符时，递减循环内的 offset。

最后一组搜索函数是 find_first_not_of()和 find_last_not_of()函数的各个版本，如表 7-5 所示。

表 7-5　find_first_not_of()和 find_last_not_of()的各个使用版本

函　　数	说　　明
find_first_not_of(char ch, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始出现的不是字符 ch 的字符，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_first_not_of(char* pstr, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始第一次出现的不在以空字符结尾的字符串 pstring 中的字符，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_first_not_of(char* pstr, size_t offset, size_t count)	在 string 对象中搜索从 offset 索引位置开始第一次出现的不在以空字符结尾的字符串 pstring 中的前 count 个字符中的任何字符，并返回发现字符的索引位置值，类型为 size_t
find_first_not_of(string str, size_t offset=0)	在 string 对象中搜索从 offset 索引位置开始第一次出现的不在字符串 str 中的任何字符，并返回发现字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为 0
find_last_not_of(char ch, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的不是字符 ch 的字符，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 nops

(续表)

函　　数	说　　明
find_last_not_of(char* pstr, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的不在以空字符结尾的字符串 pstr 中的任何字符，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 nops
find_last_not_of(char* pstr, size_t offset, size_t count)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的不在以空字符结尾的字符串 pstr 中的前 count 个字符中的字符。该函数返回发现该字符的索引位置值，类型为 size_t
find_last_not_of(string str, size_t offset=npos)	在 string 对象中向后搜索从 offset 索引位置开始最后一次出现的不在字符串 str 中的任何字符，并返回发现该字符的索引位置值，类型为 size_t。如果省略第二个实参，offset 的默认值就为字符串的末尾字符 nops

对于前面的这些搜索函数，如果没有搜索到匹配的字符，将返回 string::npos。这些函数有很多用法，通常用来在字符串中查找可能由各种字符隔开的令牌(token)。例如，用空格和标点符号隔开的单词组成的文本，因此，我们还可以用这些函数在文本块中查找单词。

本 章 小 结

本章介绍了如何使类对象更像 C++基本类型那样工作，来帮助读者更深入地理解类的知识。下一章向读者介绍面向对象编程的核心主题——类继承。本章的要点如下：

- 对象是构造函数创建的，构造函数的首要任务是给类对象的数据成员(字段)赋值。
- 对象是由析构函数销毁的。为了销毁那些包含动态分配成员的对象，本地 C++类中必须定义析构函数，因为默认的析构函数不能完成这项任务。
- 如果没有为本地 C++类定义复制构造函数，则编译器将自动提供一个。默认复制构造函数不能正确处理包含在自由存储器上分配的数据成员的类对象。
- 当我们在本地 C++类中自定义复制构造函数时，必须使用引用形参。
- 决不能在数值类中定义复制构造函数，数值类对象的副本总是通过复制字段创建的。
- 编译器没有为引用类提供默认的复制构造函数，不过需要时我们可以自行定义。
- 如果我们没有为本地 C++类定义赋值运算符，则编译器将提供默认的版本。与复制构造函数一样，默认的赋值运算符不能正确处理包含在自由存储器上分配的数据成员的类对象。
- 对于那些包含 new 运算符分配的成员的本地 C++类来说，我们必须提供析构函数、复制构造函数和赋值运算符。
- 决不能在数值类中定义赋值运算符，数值类对象的赋值总是通过复制字段完成的。
- 编译器没有为引用类提供默认的赋值运算符，不过需要时我们可以自行定义。
- 为了提供类对象所特有的动作，大多数基本运算符都可以被重载。我们只应该以符合基本运算符正常解释的方式，实现自定义类的运算符函数。

- 类模板用来创建结构相同的类，但支持不同的数据类型。
- 我们可以定义拥有多个形参的类模板，形参甚至可以是常量值而非类型。
- 本地 C++的标准库中的 string 类提供了一种强有力的处理程序中的字符串的方式。

习　　题

填空题

1. 我们可以在构造函数中使用 new 运算符来为对象成员分配空间。在这种情况下，我们必须提供适当的_____，在不需要该对象时释放其占用的内存空间。
2. 如果动态为本地 C++类的成员分配空间，则必须实现_____。
3. 为了提供类对象所特有的动作，大多数基本运算符都可以被_____。我们只应该以符合基本运算符正常解释的方式，实现自定义类的运算符函数。
4. 根据模板创建类的过程被称为_____。
5. 我们可以使用+运算符来直接连接两个字符串，但操作数之一必须是_____对象。
6. 可以使用_____或_____来访问 string 对象的任何字符。

简答题

7. 析构函数有什么作用，在什么情况下我们必须自定义类的析构函数？
8. 在什么情况下，我们必须实现类的复制构造函数？
9. 为何要进行运算符的重载，有何原则？
10. 使用下标值和 at()函数访问 string 对象的字符时，两者有何不同？

上机操作题

11. 在本地 C++中实现一个简单的字符串类，包含两个私有数据成员：一个 char*字符串和一个整型长度。提供接收 const char*类型实参的构造函数，并实现复制构造函数、赋值运算符和析构函数。验证这个类能够正常工作。
12. 上题中的字符串类还需要添加哪些构造函数？列出清单并编写它们的代码。
13. 为前面的字符串类重载+和+=运算符，以支持字符串的连接。