HZ BOOKS

高等院校计算机教材系列

Windows 可视化程序设计

刘振安 主编

机械工业出版社
China Machine Press

为教师提供教学课件

高等院校计算机教材系列

Windows 可视化程序设计

刘振安 主编

机械工业出版社
China Machine Press

本书重点讲授Widows程序设计的基本结构和消息处理方法，目的是帮助读者尽快掌握可视化设计的核心技术。本书使用多文件编程、消息映射技术和模拟文档/视结构以解释Windows的消息处理思想及自动产生程序框架的可行性；接着引入MFC进行可视化程序设计，介绍基本的可视化程序的结构及其消息处理方法；然后通过设计多个实例，从不同则面讲解如何设计基于对话框、文档/视结构和多文档可视化程序；最后给出一个完整的课程设计实例，以便读者更全面地理解文档/视结构。

本书取材新颖、结构合理、概念清楚、实用性强，易于教学，适合作为高等院校的教材，也可以作为培训班教材、自学教材及工程技术人员的参考书。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

Windows可视化程序设计/刘振安主编. －北京：机械工业出版社，2007.1
（高等院校计算机教材系列）
ISBN 7-111-19715-1

Ⅰ. W…　Ⅱ. 刘…　Ⅲ. 窗口软件，Windows－程序设计－高等学校－教材　Ⅳ. TP316.7

中国版本图书馆CIP数据核字（2006）第089996号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：朱　劼
北京京北制版印刷厂印刷·新华书店北京发行所发行
2007年1月第1版第1次印刷
184mm×260mm　·　16.5印张
定价：26.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

前　言

人们通常把操作系统作为底层软件，应用软件作为高层软件。一般来说，层次愈高，软件技术含量愈低，但和用户的应用要求也愈密切。软件层次越高，其界面部分所占的比例也越高，就越适合采用所谓的“预制件”和可视化编程技术来提高开发效率。

目前，大多数介绍设计语言的教材都把重点放在基本词法、语法和简单的程序上，但读者学完之后，很难编出实用的程序。本书是作者在1994年以来开设的几门课程的基础上，通过把重点放在程序设计方法上，并进行合理组合与取舍而编写出来的，力求反映学科发展，展现它们的最新特征。本书使用多文件编程、消息映射技术和模拟文档/视结构以解释Windows的消息处理思想及自动产生程序框架的可行性；接着引入MFC进行可视化程序设计，介绍基本的可视化程序的结构及其消息处理方法；然后通过多个实例，从不同侧面讲解如何设计基于对话框、文档/视结构和多文档的可视化程序。本书最后给出一个完整的课程设计实例，通过这个课程设计，能使读者体会面向对象编程和可视化编程之间的关系。

为了方便学习，本书各章均配有相应的实验和习题。在写作时，作者力求达到取材新颖、结构合理、概念清楚、语言简洁、通俗易懂、实用性强的目标，以便用于教学。本书适合作为高等院校的教材，也可以作为培训班教材，并可作为自学者及工程技术人员的参考书。

本书共有10章。第1章是C++知识回顾，将简要回顾Windows可视化编程必须掌握的C++基础知识。第2章介绍Windows程序的基本风格，并手工编制一个Windows程序，说明编程的基本原理和消息处理机制。第3章介绍Windows程序的消息处理，通过引入多文件编程和消息映射机制，说明Windows程序结构。第4章介绍自动生成与消息处理，通过对给定程序框架进行改制并添加功能，进一步熟悉程序变换和消息处理方法，根据消息映射进行改制，以便为学习使用MFC和文档/视结构打下基础。第5章阐述使用MFC类库编程，通过模拟文档/视结构，引入使用向导和文档/视结构的概念，并介绍基于对话框的编程。第6章介绍MFC文档/视结构，主要说明应用程序数据的处理。第7章～第9章分别给出基于对话框的设计实例、单文档设计实例以及多文档设计实例，目的是通过多个设计实例，从不同角度说明Windows程序的结构，介绍数据和显示的基本组织方法，为进一步深造打下基础。第10章的课程设计以一个学生成绩管理为例，帮助读者练习如何实现运算符重载及文件存取功能，并实现可视化操作，从而进一步熟悉使用MFC文档/视结构实现程序设计的思想。

中国科学技术大学软件学院院长陈国良院士，原安徽大学副校长、计算机系程慧霞教授及南京大学计算机系陈本林教授在百忙之中审阅了本书的书稿并提出了许多宝贵意见。许多使用过本书初稿的兄弟院校的老师也提出了许多宝贵意见。本书在定稿之前，曾在多个软件工程硕士班及本科生班讲授，这些学生也提出了很好的建议并验证了书中的程序及实验，特此感谢。本书在写作中还参考了大量资料，有的收入参考文献之中，还有些没有收入其中，在此对这些作者表示感谢。

刘燕君、周军、潘剑锋、王文涛、孙忱等参加了本书的编写工作。由于时间仓促，本书的不妥之处在所难免，敬请同行及读者不吝赐教。

刘振安

于中国科学技术大学

目 录

此为试读，需要完整PDF请访问：www.ertongbook.com

第1章 C++知识回顾

本章将简要回顾进行Windows可视化编程必须掌握的C++基础知识，其中包括函数指针、重载、模板、虚函数及其多态性等内容，以便为下一步的学习打好基础。

另外，因为本章内容的回顾性质，所以不涉及知识点的顺序，而是使知识点相互交叉、有机融合。但在每节之后，尽可能联系它们在MFC中的应用实例，以方便以后的学习。

1.1 重载

重载分为函数重载和运算符重载两类，本节先回顾重载的相关知识。

1.1.1 函数重载和默认参数

C++允许为同一个函数定义多个版本，这称为函数重载。函数重载可以使一个函数名具有多种功能，即具有“多种形态”，所以这种性质称为多态性。本节只介绍实函数的多态性，虚函数的多态性将另外讲述。

【例1.1】函数重载产生多态性的例子。

```
#include <iostream>
using namespace std;
double max(double,double);          //2个实型参数的函数原型
int max(int,int);                   //2个整型参数的函数原型
char max(char,char);                //2个字符型参数的函数原型
int max(int,int,int);               //3个整型参数的函数原型
void main( ){
  cout<<max(2.5, 17.54)<<" "<<max(56,8)<<" "<<max('w','p')<<endl;
  cout<<"max(5,9,4)="<<max(5,9,4)<<" max(5,4,9)="<<max(5,4,9)<<endl;
}
double max(double m1, double m2)
{ return(m1>m2)?m1:m2;}
int max(int m1, int m2)
{return(m1>m2)?m1:m2;}
char max(char m1, char m2)
{return(m1>m2)?m1:m2;}
int max(int m1, int m2, int m3)
{  int t=max(m1,m2);
   return max(t,m3);
}
```

C++能够正确调用相应函数，程序输出结果如下:

```
17.54 56 w
max(5,9,4)=9 max(5,4,9)=9
```

从函数原型可见，这几个max函数的区别在于，一是参数类型不同，二是参数个数不同。

编译器在编译时，能根据源代码调用固定的函数标识符，然后由连接器接管这些标识符，并用物理地址代替它们，这就称为静态联编或先期联编。

同理，可以设计一个求整数之和的函数。不过，如果要求4个整数之和，使用函数重载则需要编写3个函数。这时可编写一个具有默认参数的函数。

【例1.2】编写一个具有默认参数的函数。

```
#include <iostream>
using namespace std;
int add(int m1=0, int m2=0, int m3=0, int m4=0)
{return m1+m2+m3+m4;}
void main()
{ cout<<add(1,3)<<","<<add(1,3,5)<<"," <<add(1,3,5,7)<<endl;}
```

程序输出结果为:

```
4,9,16
```

使用默认参数，就不能对少于参数个数的函数进行重载。例如，这里不能重载具有3个整型参数的add函数，只能对多于4个参数的add函数重载。因为编译器决定不了是使用3个参数的add函数，还是使用4个参数的add函数。另外，仅凭函数返回值不同也是无法区分重载函数的。当使用默认参数设计类的构造函数时，要特别注意。

一个类可以有多个构造函数，这也是典型的函数重载。可以使用域定义符“::”显式地指出调用的是基类的重载函数还是派生类的重载函数。

1.1.2 重载与名字支配规律的区别

如果基类和派生类的成员函数具有相同的参数表，则不属于函数重载。这时按名字支配规律调用，并且使用域定义符“::”防止二义性。下面是在基类和派生类中使用参数相同的同名函数的例子。

【例1.3】演示在基类和派生类中使用参数相同的同名函数的例子。

```
#include <iostream>
using namespace std;
class Point
{
  private:
    int x,y;
  public:
    Point(int a, int b){x=a; y=b;}
    void Show(){cout<<"x="<<x<<",y="<<y<<endl;};                    //基类的Show()函数
};
class Rectangle : public Point
{
  private:
    int H, W;
  public:
    Rectangle(int, int, int, int);                                  //构造函数原型
    void Show()
```

```
    {
        cout<<"H="<<H<<",W="<<W<<endl;
    }
    void Display()
    {
        Point::Show();                                    //使用基类的Show()函数
        Rectangle::Show();                                //使用派生类的Show()函数
    }

};
Rectangle::Rectangle(int a, int b, int h, int w):Point(a,b)   //定义构造函数
{H=h; W=w;}
void main()
{
  Point a(3,4);
  Rectangle r1(3,4,5,6);
  a.Show();              //基类对象调用基类Show()函数
  r1.Show();             //派生类对象调用派生类Show()函数
  r1.Point::Show();      //派生类对象调用基类Show()函数
  r1.Display();
}
```

程序输出如下:

```
x=3,y=4
H=5,W=6
x=3,y=4
x=3,y=4
H=5,W=6
```

派生类的Display( )函数使用域定义符“::”指明调用的是基类还是派生类的Show()函数。其实，调用派生类本身的Show()函数不需要使用“Rectangle::”来限定，它根据名字支配规律即可正确地调用自己的Show()函数，这里是有意使用显示方式“Rectangle”，以便帮助读者进一步理解域定义符“::”的作用。

如果不需要单独显示派生类的H和W的数值，可将void()函数直接定义为如下形式:

```
void Show()
{
    Point::Show();                                  //显示x和y的数值
    cout<<"H="<<H<<",W="<<W<<endl;
}
```

如果同名函数的参数类型不同或者参数个数不同，则属于重载。这时，也可以使用域定义符显式地指明被调用的函数。

1.1.3 运算符重载

因为任何运算都是通过函数来实现的，所以运算符重载其实就是函数重载，要重载某个运算符，只要重载相应的函数就可以了。在重载运算符时，需要使用新的关键字“operator”，它经常和C++的一个运算符连用，构成一个运算符函数名，例如“operator +”。通过这种构成

方法，用户就可以像重载普通函数那样重载运算符函数operator+( )。由于C++已经为各种基本数据类型定义了该运算符函数，所以只需要为自己定义的类型重载operator + ( )就可以了。

一般来说，为用户定义的类型重载运算符，都要求能够访问这个类型的私有成员。完成这项工作只有两条路可走：要么将它重载为这个类型的成员函数，要么将它重载为这个类型的友元。为区别这两种情况，将作为类的成员函数的运算符称为类运算符，而将作为类的友元的运算符称为友元运算符。

C++的大部分运算符都可以重载，不能重载的只有“.”、“::”、“*”和“?:”。前3个运算符因为在C++中有特定的含义，所以不能重载，以免引起不必要的麻烦；“?:”运算符则不值得重载。另外，“sizeof”和“#”不是运算符，因而不能重载，而= 、()、 [] 、->这4个运算符只能用类运算符来重载。

【例1.4】使用友元函数重载运算符“<<”和“>>”。

```
#include <iostream.h>
class test{
  private:
          int i;
          float f;
          char ch;
   public:
          test(int a=0, float b=0, char c='\0'){i=a; f=b; ch=c;}
          friend ostream &operator << (ostream & , test);
          friend istream &operator >> (istream & , test &);
};
ostream &operator << (ostream & stream, test obj)
{
   stream<<obj.i<<",";
   stream<<obj.f<<",";
   stream<<obj.ch<<endl;
   return stream;
}
istream &operator >> (istream & t_stream, test&obj)
{
   t_stream>>obj.i;
   t_stream>>obj.f;
   t_stream>>obj.ch;
   return t_stream;
}
void main()
{
   test A(45,8.5,'W');
   operator <<(cout,A);
   test B,C;
   cout<<"Input as i f ch:";
   operator >>(cin,B);
   operator >>(cin,C);
   operator << (cout,B);
```

此为试读，需要完整PDF请访问：www.ertongbook.com

```
    operator << (cout,C);
}
```

运行示例如下:

```
45,8.5,W
Input as i f ch:5 5.8 A 2 3.4 a    //假设输入两组
5,5.8,A
2,3.4,a
```

将主函数写成上面的函数调用形式主要是为了演示运算符的重载。一般在使用时，则直接使用运算符。下面是常见的使用方式:

```
void main()
{
   test A(45,8.5,'W');
   cout<<A;
   test B,C;
   cout<<"Input as i f ch:";
   cin>>B>>C;
   cout<<B<<C;
}
```

显然，运算符“<<”的重载函数有两个参数，第1个是ostream 类的一个引用，第2个是自定义类型的一个对象。这个运算符的重载方式是友元重载。另外，该函数的返回类型是一个ostream 类型的引用，在函数中实际返回的是该函数的第1个参数，这样做可以使“<<”能够连续使用。例如，对于语句

```
cout << a << b;  //a,b均为自定义类型的对象
```

的处理如下: 首先系统把 cout << a 作为

```
operator << ( cout,a);
```

来处理，返回cout，紧接着又把刚返回的cout连同后面的“<< b”一起作为

```
operator << (cout,b);
```

处理，再返回cout，从而实现了运算符“<<”的连续使用。

【例1.5】使用类运算符重载“++”运算符。

```
#include <iostream>
using namespace std;
class number {
          int num;
   public:
          number( int i ) {  num=i; }
          int operator ++ ( );                    // 前缀: ++n
          int operator ++ ( int );                // 后缀: n++
          void print( ) { cout << "num="<<num << endl; }
};
int number :: operator ++ ( )
{
```

```
        num ++;
        return num;
    }
    int number :: operator ++ ( int )                    //不用给出形参名
    {
        int i=num;
        num ++;
        return i;
    }
    void main( )
    {
       number n(10);
       int i = ++n;                                      // i=11, n=11
       cout <<"i="<<i<<endl;                             // 输出i=11
       n.print();                                        // 输出n=11
       i=n++;                                            // i=11, n=12
       cout <<"i="<< i << endl;                          // 输出i=11
       n.print( );                                       // 输出n=12
    }
```

同理，如果主函数的第2条和第5条语句使用函数调用方式，则分别为:

```
   int i=n.operator ++ ( );
   i=n.operator ++(0);
```

由此可见，只要定义正确，就不必再使用函数调用方式，直接使用运算符即可。

1.1.4 友元运算符、类运算符及其参数

如果运算符所需的操作数，尤其是第一个操作数希望进行隐式类型转换，则该运算符应该通过友元来重载。另一方面，如果一个运算符的操作需要修改类对象的状态，则应当使用类运算符，这样更符合数据封装的要求。

【例1.6】使用对象作为友元函数参数来定义运算符“+”的例子。

```
#include <iostream.h>
class complex {
      private:
          double real, imag;
      public:
          complex(double r=0, double i=0) { real=r; imag=i;}
          friend complex operator + (complex, complex);
          void show(){cout<<real<<"+"<<imag<<"i";}
};
complex operator + (complex a,complex b)
{
          double r = a.real + b.real;
          double i = a.imag + b.imag;
          return complex(r,i);
}
void main()
```

```
{
    complex x(5,3), y ;
    y =x+7;                          //语句2
    y =7+y;                          //语句3
    y.show();
}
```

程序运行正常，因为语句2和语句3可以分别解释为:

```
y =operator +(x,7);
y =operator +(7,y);
```

而“7”均可通过构造函数转换成complex类型的对象，使其参数匹配，从而保证正常工作。如果使用类运算符，假设为如下形式:

```
complex  operator + (complex a)
{
   double r = a.real + real;
   double i = a.imag + imag;
   return complex(r,i);
}
```

因为“y =7+y;”等价为“y=7.operator+(y);”，所以系统无法解释这个式子的含义。由此可见，如果对象作为重载运算符函数的参数，则可以使用构造函数将常量转换成该类型的对象。如果使用引用作为参数，则这些常量不能作为对象名使用，否则编译系统就要报错。如果将上面友元和类运算符均使用引用作为参数，则“y =x+7;”和“y =7+y;”都不能通过编译。在使用它们时，必须分清场合及其使用方法。

MFC中大量使用运算符重载，其中包括类型转换运算符(该运算符没有返回值)。常见的是CPoint类和CString类。CdumpContext和CArchive类中定义了大量“>>”和“<<”运算符的重载版本。

1.2 模板

我们知道，模板有函数模板和类模板两种类型，下面我们回顾一下这两种模板的相关知识。

1.2.1 函数模板及其显式调用规则

在程序设计时并不给出相应数据的实际类型，而是在编译时，才由编译器利用实际的类型给予实例化，使数据类型满足需要。由此可见，可使编译器成为一种在函数模板引导下，制作符合要求的代码的程序设计辅助工具。

函数模板声明的一般方法如下:

```
template <函数模板参数>
返回类型 函数名{//函数体};
```

从声明中可以看出，模板以template关键字和一个形参表开头。在尖括号里只需要声明一个类型参数的标识符，例如定义一个求最大值函数:

```
template <class T>
T max(T m1, T m2)
{ return(m1>m2)?m1:m2;}
```

class意为“用户定义的或固有的类型”。字母T标识这个模板有一个参数类型。当在程序中使用max(2,5)时，编译器能推断出这个T为int，并使用如下版本产生具体的模板函数:

```
int max(int m1, int m2){ return(m1>m2)?m1:m2;}
```

而对于max(2.5,8.8)则使用如下版本:

```
double max(double m1, double m2){ return(m1>m2)?m1:m2;}
```

由此可见，在调用函数模板时，函数参数的类型决定到底使用模板的哪个版本。也就是说，模板的参数是由函数推断出来的，这种使用方法称为默认方式。一般形式为:

函数模板名(参数列表)

也可以使用max<int>(2,5)明确指出参数类型为int，这称为显式参数比较准则。一般形式为:

函数模板名<模板参数>(参数列表)

每次调用都显式地给出比较准则比较麻烦。显式规则一般在特殊场合使用，通常使用默认方式即可。对于一个默认调用，从函数的参数推断出模板参数的能力是最关键的。编译器能够从一个调用推断出类型参数和非类型参数，从而省去显式调用的麻烦。条件是由这个调用的函数参数表能够惟一地标识出模板参数的一个集合。

另外，C++还专门定义一个仅仅用在模板中的关键字typename，它的用途之一是代替template参数列表中的关键字class。

1.2.2 模板函数专门化和模板重载

1. 模板函数专门化

虽然按照默认约定，定义一个模板，用户可以使用能想到的任何模板参数(或者模板参数组合)，但用些用户却宁肯选择另外的实现方法。例如定义的模板函数max:

```
template <typename T>              //声明模板
T max(T m1, T m2)                  //求两个数的最大值
{ return(m1>m2)?m1:m2;}
```

它虽然可以处理字符串，但用户希望换一种处理方法。用户的方案是：如果模板参数不是指针，就使用这个模板；若果是指针，就使用如下的处理方法:

```
char *max(char *a, char *b){     return (strcmp(a,b)>=0?a:b);}
```

由于普通函数优先于模板函数，因此在执行如下语句

```
cout<<max("ABC","ABD")<<","<<max('W','T')<<" ";
```

时，第一个字符串参数是调用普通函数，第二个单字符参数则调用模板函数。不过，为了形成完整的模板系，便于管理，并保证在无调用时不会生成任何无用代码，希望仍使用模板形式。这可以通过提供不同的定义方式来处理，并由编译器基于在使用处提供的模板参数，在

这些定义中做出选择。对一个模板的这些可以互相替换的定义称为用户定义的专门化，或简称为用户专门化。

前缀“template <>”说明这是一个专门化，在描述时不用模板参数。可以写成:

```
template <>char *max<char*>(char *a, char *b){return (strcmp(a,b)>=0?a:b);}
```

函数名之后的<char*>说明这个专门化应该在模板参数是char*的情况下使用。由于模板参数可以从函数的实际参数列表中推断，所以不需要显式地描述它，即上述定义可以简化为:

```
template <>char *max<>(char *a, char *b){return (strcmp(a,b)>=0?a:b);}
```

它给出了template <>前缀，第二个<>也属多余，可以简单地写成如下形式:

```
template <>char *max(char *a, char *b){return (strcmp(a,b)>=0?a:b);}
```

模板函数专门化的具体使用方法见例1.7。

2. 模板重载

C++模板的机制也是重载。模板提供的语法与多态性相似，当提供细节时，模板就可以生成模板函数。因为选择调用哪一个函数是在编译时实现的，所以是静态联编。下面通过重载进一步扩大已定义模板max的适用范围。

【例1.7】专门化和重载的例子。

```
#include <iostream>
using namespace std;
template <typename T>                    //声明模板
T max(T m1, T m2)                        //求两个数的最大值
{ return(m1>m2)?m1:m2;}
template <typename T>                    //声明函数模板时，需要重写template
T max(T a, T b, T c)                     //重载
{ return max(max(a,b),c);}
template <class T>                       //声明函数模板时，需要重写template
T max(T a[], int n)                      //重载，求数组中的最大值
{
   T maxnum=a[0];
   for(int i=0; i<n;i++)
          if (maxnum<a[i]) maxnum=a[i];
   return maxnum;
}
template <>                              //专门化
char *max(char *a, char *b)              //使用指针
{return (strcmp(a,b)>=0?a:b);}
int  max(int m1, double m2)              //普通函数
{ int m3=(int)m2; return(m1>m3)?m1:m3;}
void main( )
{
   cout<<max("ABC","ABD")<<" ";          //1
   cout<<max("ABC","ABD","ABE")<<" ";    //2
   cout<<max('W','T','K')<<" ";          //3
   cout<<max(2.0,5.,8.9)<<" ";           //4
   cout<<max(2,6.7)<<" ";                //5
```

```
    double d[]={8.2,2.2,3.2,5.2,7.2,-9.2,15.6,4.5,1.1,2.5};    //定义实数数组d
    int a[]={-5,-4,-3,-2,-1,-11,-9,-8,-7,-6};                   //定义整数数组a
    char c[]="acdbfgweab";                                      //定义字符串数组c
    cout<<"intMax="<<max(a,10)<<" doubleMax="<<max(d,10)
        <<" charMax="<<max(c,10)<<endl;
}
```

程序输出结果为:

```
ABD ABE W 8.9 6 intMax=-1 doubleMax=15.6 charMax=w
```

注意执行语句2和3的区别。它们执行重载的过程一样，但在重载函数调用时，前者使用专门化(指针参数)模板，后者使用定义(选择单字符参数)模板。语句5不调用模板，而是使用普通的函数。

1.2.3 类模板

如果将类看做包含某些数据类型的框架，那么支持该数据类型的不同操作可理解为：将数据类型从类中分离出来，允许单个类处理通用的数据类型T。其实，这种类型并不是类，而仅仅是类的描述，常称之为类模板。在编译时，由编译器将类模板与某种特定数据类型联系起来，从而产生一个真实的类。由此可见，利用类模板进行程序设计，就如烹调食物一样，只要购买了原料，就可以做出不同口味的菜肴。

类模板声明的一般方法如下:

```
template <类模板参数>class 类名{//类体};
```

类模板也称为参数化类型。初始化类模板时，给它传递具体的数据类型，就产生了模板类。使用模板类时，编译器自动产生处理具体数据类型的所有成员（数据成员和成员函数）。

只要赋给模板一种实际的数据类型，就会产生新的类，而且以特定类型替代模板参数。定义对象的一般格式如下:

```
类名<模板实例化参数类型>对象名(构造函数实参列表);
类名<模板实例化参数类型>对象名;    //默认或者无参数构造函数
```

“模板实例化参数类型”包括数据类型和值，编译器不能从构造函数列表推断出模板实例化参数类型，所以必须显式地给出它们的参数类型。

在类体之外定义成员函数时，必须用template重写模板函数声明。一般格式如下:

```
template<模板参数>
返回类型 类名<模板类型参数>::成员函数名(函数参数列表){//函数体}
```

“模板类型参数”是指template的“< >”内使用class(或typename)声明的类型参数，也就是使用< T_1, T_2, … , T_n>的类型参数列表。构造函数和析构函数没有返回类型。

【例1.8】演示对4个数字求和的类模板程序。

```
#include <iostream>
using namespace std;
template <class T, int size=4>                    //可以传递程序中的整数参数值
class Sum
{
```

此为试读，需要完整PDF请访问：www.ertongbook.com