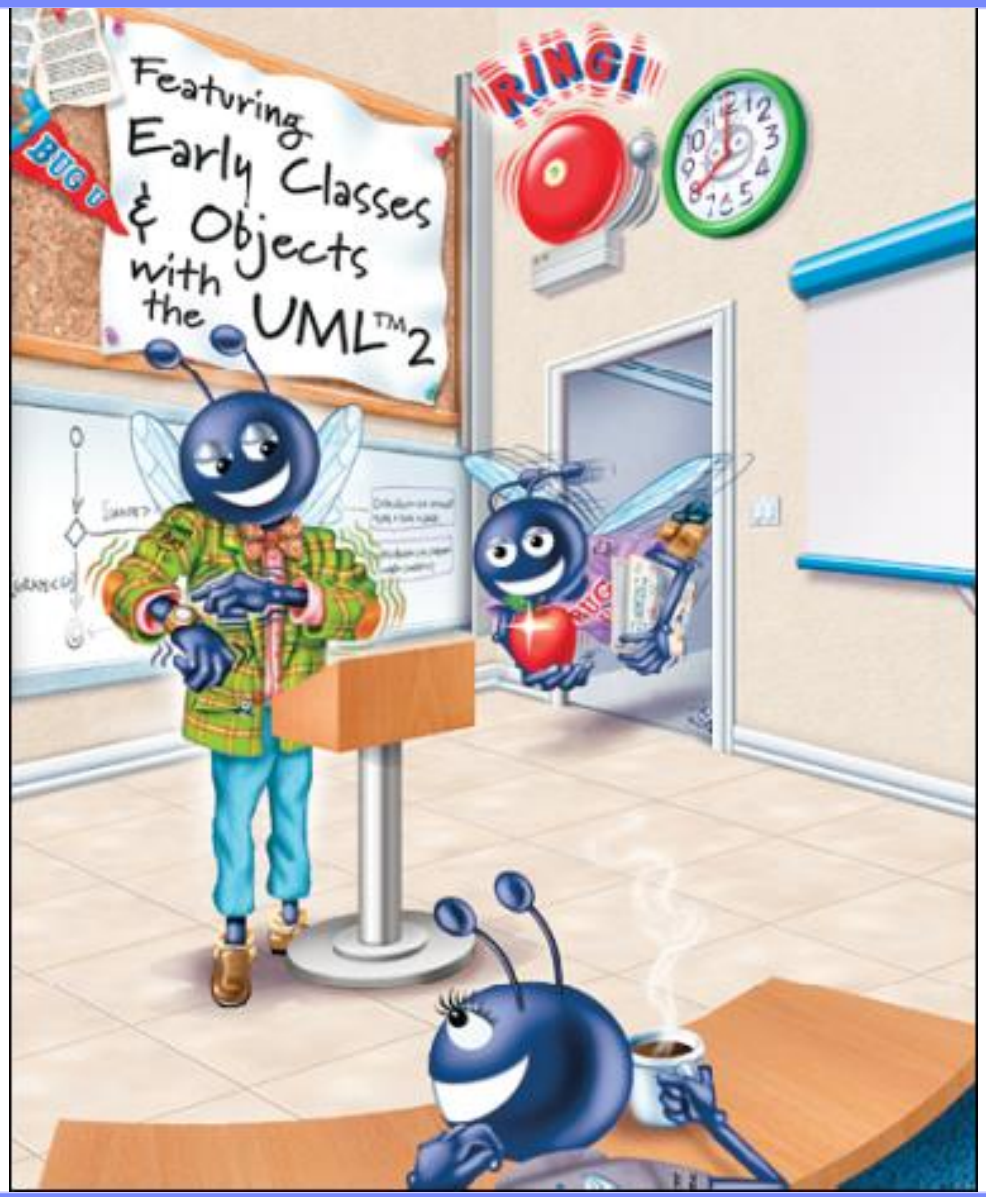


Lecture 11:
模板

第十四讲 模板

学习目标：

- 使用函数模板创建一组相关函数
- 使用类模板创建一组相关类型
- 模板、友元、继承和静态成员之间的关系

问题引入：参数多态性

设有复数类设计如下，该复数类的实部和虚部均为float类型：

```
class Complex1
{
private:
        float real;                         //实部
        float imag;                         //虚部
public:
        Complex1( float x = 0, float y = 0);    //构造函数
        ~Complex1();                            //析构方法
```

```
Complex1 Add(const Complex1 x) const;       //加
Complex1 Subtract(const Complex1 x) const;          //减
Complex1 Multiply(const Complex1 x) const;          //乘
Complex1 Divide(const Complex1 x) const;    //除
void show(void) const;                              //输出
    };
    (Complex1类成员函数的函数体定义省略)
```

如果在实际的软件设计中，要求的Complex类的实部和虚部均为double类型时，我们就要重新设计Complex类如下：

```
class Complex2
{
private:
        double real;                                        //实部
        double imag;                                        //虚部
public:
        Complex2( double x = 0, double y = 0);              //构造函数
        ~Complex2();                                        //析构方法
        Complex2 Add(const Complex2 x) const;               //加
        Complex2 Subtract(const Complex2 x) const;          //减
        Complex2 Multiply(const Complex2 x) const;          //乘
        Complex2 Divide(const Complex2 x) const;            //除
        void show(void) const;                              //输出
};
        (Complex2类成员函数的函数体定义省略)
```

重复劳动，容易出错，很大的维护和调试工作量，增加可执行文件的大小！

对于上述问题，通常有两种解决方法：

1. 把类中的数据类型定义为一个抽象的、需根据具体问题要求确定的数据类型。在外部程序具体定义该类的对象前，首先定义该数据类型为某个具体存在的数据类型。
2. 把类中的数据类型定义为一个参数。在外部程序具体定义该类的对象时，同时用实际需要的数据类型指定该参数。

方法2是一种比方法1更为有效的实现参数多态性的方法。

本章讨论的模板就是C++语言用方法2实现的参数多态性。

C++的解决之道：模板

模板是一种基于类型参数生成类或函数的机制。函数模板和类模板可以使程序员只需要制定一个单独的代码段，就可表示一整套称为函数模板特化的相关（重载）函数或者是表示一整套称为类模板特化的相关的类。这种技术称为泛型程序设计(Generic programming)。本质是适应面更宽。

1 Introduction

- 函数模板和类模板
  - 使程序员可以声明一组相关函数和相关类
  - 泛型编程（Generic programming）

2 Function Templates

- 函数模板定义
  - 模板头
    - ✓关键字 template
    - ✓模板参数列表
      - 尖括号内（< >）
      - 每个模板参数前面加关键字 class 或 typename
      - 用来声明函数模板参数类型，局部变量和返回值类型

2 Function Templates

- 函数模板定义
  - 模板头
    - ✓例如：
      - `template< typename T >`
      - `template< class ElementType >`
      - `template< typename BorderType, typename Filltype >`

```cpp
// Fig. 14.1: fig14_01.cpp
// Using template functions.
#include <iostream>
using std::cout;
using std::endl;

// function template printArray definition
template< typename T >
void printArray( const T *array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function template printArray

int main()
{
   const int ACOUNT = 5; // size of array a
   const int BCOUNT = 7; // size of array b
   const int CCOUNT = 6; // size of array c

   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // 6th position for null

   cout << "Array a contains:" << endl;
```

Type template parameter **T** specified in template header

- 该函数模板把惟一的形式参数T(T一般作为类型参数)声明为函数printArray打印的数组类型。
- 当编译器检测到程序源代码中调用函数printArray时，用printArray的第一个参数的类型替换掉整个模板定义中的T，并建立用来打印指定类型数组的一个完整的特化函数，然后再编译这个新建的函数。

```
28
29    // call integer function-template specialization
30    printArray( a, ACOUNT );
31
32    cout << "Array b contains:" << endl;
33
34    // call double function-template specialization
35    printArray( b, BCOUNT );
36
37    cout << "Array c contains:" << endl;
38
39    // call character function-template specialization
40    printArray( c, CCOUNT );
41    return 0;
42 } // end main
```

Creates a function-template specialization of **printArray** where **int** replaces **T**

Creates a function-template specialization of **printArray** where **double** replaces **T**

Creates a function-template specialization of **printArray** where **char** replaces **T**

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

程序首先实例化int数组a、double数组b和char数组c，长度分别为5、7、6。

3 Overloading Function Templates

- 从函数模板初始化的函数模板特化都具有相同的名字，所以编译器会用重载的方式来调用适当的函数。
- 函数模板本身可以用多种方式重载。
  - 提供其他函数模板，有相同的函数名字和不同的函数参数
  - 用其他非模板函数(同名而参数不同)重载
- 编译器通过匹配过程确定调用哪个函数，如果找不到匹配或产生多个匹配，就全产生编译错误。
  - 首先找出所有和调用函数的函数名相匹配的函数模板
  - 然后根据调用函数的参数类型对函数模板进行特化
  - 接着找出所有与调用函数里的普通函数相匹配的函数
    - ✓ 若某一普通函数或者函数模板特化是与调用函数最匹配的，那么该普通函数或者函数模板特化将被调用
    - ✓ 若某一普通函数和易函数模板特化跟调用函数的匹配程度相同，那么这个普通函数被调用
    - ✓ 若多个函数和调用函数相匹配，报错。

4 Class Templates

- 类模板（或参数化类型）
  - 类模板定义前需要有模板头

    如：`template< typename T >`
  - 类型参数 T 可以在成员函数和数据成员中作为数据类型使用
  - 额外的类型参数用逗号分隔

    如：`template< typename T1, typename T2 >`

类模板格式

```
template <类型参数表>
class 类模板名
{
    <类成员的声明>
};
```

函数模板格式

```
template <类型参数表>
void 函数名
{
    <函数实现>
};
```

类模板中的成员函数的定义

- 可以放在类模板的定义体中（此时与类中的成员函数的定义方法一致）
- 也可以放在类模板的外部定义成员函数，此时成员函数的定义格式如下：

```
template <类型形式参数表>
返回值类型 类模板名<类型名表> ::<函数名>(<参数表>)
{
    <函数体>
}
```

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T& ); // push an element onto the Stack
19    bool pop( T& ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
```

Create class template **Stack** with type parameter **T**

Member functions that use type parameter **T** in specifying function parameters

```
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
```

Data member **stackPtr** is a pointer to a **T**

在类模板定义之外的成员函数定义都要以下面的形式开头： **template<class T>**

Member-function template definitions that appear outside the class-template definition begin with the template header

泛型类名是Stack<T>

```
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif
```

4 类模板的使用

- 类模板定义的只是对类的描述，它本身还不是一个实实在在的类。
- 使用前首先要定义类模板的对象(即实例)，才能使用。
- 需要用下列格式的语句：

类模板名 <实际的类型> 对象名[(实际参数表)];

注：该处对象名实际是指一个真正的类的名字。

在创建对象时指定数据类型,即可更改该类操作中的数据类型

```cpp
// Fig. 14.3: fig14_03.cpp
// Stack class template test program.
#include <iostream>
using std::cout;
using std::endl;

#include "Stack.h" // Stack class template definition

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleStack\n";

   // push 5 doubles onto doubleStack
   while ( doubleStack.push( doubleValue ) )
   {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";

   // pop elements from doubleStack
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';
```

用double取代类型参数T，生成模板类特化Stack< double >,即实例化成一个真正的类，然后再实例化为对象

```
29
30    cout << "\nStack is empty. Cannot pop\n";
31
32    Stack< int > intStack; // default size 10
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    // push 10 integers onto intStack
37    while ( intStack.push( intValue ) )
38    {
39       cout << intValue << ' ';
40       intValue++;
41    } // end while
42
43    cout << "\nStack is full. Cannot push " << intValue
44       << "\n\nPopping elements from intStack\n";
45
46    // pop elements from intStack
47    while ( intStack.pop( intValue ) )
48       cout << intValue << ' ';
49
50    cout << "\nStack is empty. Cannot pop" << endl;
51    return 0;
52 } // end main
```

Create class-template specialization **`Stack< int >`** where type **`int`** is associated with type parameter **`T`**，然后实例化成对象

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

```
1  // Fig. 14.4: fig14_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack<T>. 创建函数模板测试类模板
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;
10
11 #include "Stack.h" // Stack class template definition
12
13 // function template to manipulate Stack< T >
14 template< typename T >
15 void testStack(
16    Stack< T > &theStack, // reference to Stack< T >
17    T value, // initial value to push
18    T increment, // increment for subsequent values
19    const string stackName ) // name of the Stack< T > object
20 {
21    cout << "\nPushing elements onto " << stackName << '\n';
22
23    // push element onto Stack
24    while ( theStack.push( value ) )
25    {
26       cout << value << ' ';
27       value += increment;
28    } // end while
```

Use a 函数模板 to process **Stack** class-template specializations

```
29
30    cout << "\nStack is full. Cannot push " << value
31       << "\n\nPopping elements from " << stackName << '\n';
32
33    // pop elements from Stack
34    while ( theStack.pop( value ) )
35       cout << value << ' ';
36
37    cout << "\nStack is empty. Cannot pop" << endl;
38 } // end function template testStack
39
40 int main()
41 {
42    Stack< double > doubleStack( 5 ); // size 5
43    Stack< int > intStack; // default size 10
44
45    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
46    testStack( intStack, 1, 1, "intStack" );
47
48    return 0;
49 } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

[*] EXAMPLE10_02A.h | EXAMPLE10_2.CPP | EXAMPLE10_02B.h

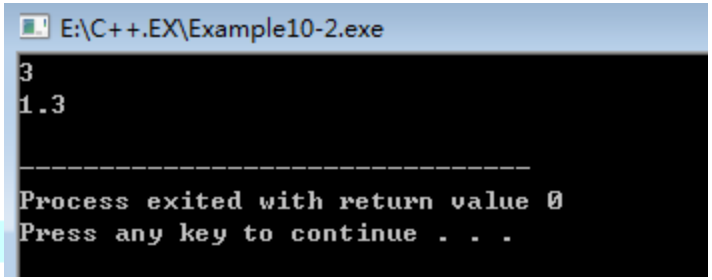
```
1   //EXAMPLE10_02A.H
2   #ifndef EXAMPLE10_02A_H
3   #define EXAMPLE10_02A_H
4   template <class T>
5   class Max      //声明类模板Max
6   {
7        private:
8           T item1, item2, item3;      //类型为T,T在该类的对象生成时具体化
9
10       public:
11
12          Max(T thefirst,T thesecond, T thethird);
13
14          T GetMaxItem();    //求得3个元素中的最大值并按类型T返回
15
16          void SetItem(T thefirst,T thesecond, T thethird); //设置类中的3个元素的值
17
18  };
19  #endif
20
```

[*] EXAMPLE10_02A.h | EXAMPLE10_2.CPP | EXAMPLE10_02B.h

```
1  //类模板的实现
2  #ifndef EXAMPLE10_02B_H
3  #define EXAMPLE10_02B_H
4
5  #include "EXAMPLE10_02A.h"
6
7  template <class T>
8  Max<T>::Max(T thefirst, T thesecond, T thethird):
9          item1(thefirst),item2(thesecond),item3(thethird)
10 {
11
12 };
13
14 template <class T>
15 void Max<T>::SetItem(T thefirst, T thesecond, T thethird)
16 {
17       item1=thefirst;
18       item2=thesecond;
19       item3=thethird;
20 };
21
22 template <class T>
23 T Max<T>::GetMaxItem()
24 {
25       T maxitem;
26       maxitem=item1>item2 ? item1 : item2;
27       maxitem=maxitem>item3 ? maxitem : item3;
28       return maxitem;
29 };
30 #endif
31
```

[*] EXAMPLE10_02A.h | EXAMPLE10_02B.h | EXAMPLE10_2.CPP

```
1  //EXAMPLE10_2.CPP
2  //主程序
3  #include <iostream>
4  #include "EXAMPLE10_02A.H"
5  #include "EXAMPLE10_02B.H"
6  using namespace std;
7
8  int main()
9  {
10       Max<int> nmyMax(1,2,3);
11       cout<<nmyMax.GetMaxItem()<<endl;
12
13       Max<double> dblmyMax(1.2,1.3,-1.4);
14       cout<<dblmyMax.GetMaxItem()<<endl;
15
16
17       return 0;
18 }
19
```



5 类模板的非类型参数和默认类型参数

- 非类型模板参数
  - 有默认的参数
  - 作为常量处理
  - Example
    - ✓ 模板头部：template< typename T, int elements >
      声明: Stack< double, 100 > salesFigures;

- 类型参数可以指定默认类型
  - Example
    - ✓ 模板头部: template< typename T = string >
      声明: Stack<> jobDescriptions;

- 显式的类模板特化
  - 如果一个特定的用户定义类型不能使用
  - Example：为 Employee 对象创建显式的Stack模板特化，构建Stack< Employee > 类
    - ✓
```
template<>
class Stack< Employee >
{
    ...
};
```
  - 完全替换了 Stack类模板
    - ✓ 没有使用原来类模板的任何内容甚至可以包含不同的成员

6 Notes on Templates and Inheritance

- 模板和继承
  - 类模板可以从类模板特化派生
  - 类模板可以从非类模板类派生
  - 类模板特化可以继承自类模板特化
  - 非模板类可以继承自类模板特化

7 Notes on Templates and Friends

- 模板和友元
  - 假定类模板 X 具有类型参数 T：

    template< typename T > class X

    - ✓模板类中的友元函数可以是每个模板特化类的友元函数

      friend void f1();

      f1 是 X< double >, X< string >, 等的友元

7 Notes on Templates and Friends

- 使用类模板，可以声明各种各样的友元关系。友元可以在类模板与全局函数间、另一个类(可能是类模板特化)的成员函数、甚至整个类中(可能是类模板特化)建立。
  - template< typename T > class X
    - ✓ 所有从类X的类模板实例化得到的类模板特化的友元函数f1
      - ◈ friend void f1();

        f1 就是 X< double >, X< string >, 等的友元
    - ✓ 仅仅包含相同类型参数的类模板特化的友元函数f2，
      - ◈ friend void f2(X< T > &);

        f2(X< float > &)只是类模板特化 X< float >的友元而不是 X< string >的友元

- ✓ 将其他类的成员函数声明为是类模板产生的所有类模板特化的友元。只要用类名和二元作用域运算符指定其它类的成员函数名
  - friend void A::f3();
    - A类的成员函数f3称为之前声明的、类模板实例化产生的友元，例如X< double >, X< string >等
- ✓ 另一个类的成员函数只能是包含相同类型参数的类模板特化的友元。
  - friend void C< T >::f4(X< T > &);
    - C< float >::f4(X< float > &) 只能成员类模板特化X< float > but not a的友元，而不是 X< string >的友元

- ✓ 将整个这个类的成员函数设定为类模板的友元。

  friend class Y;

  Y类的每个成员函数都成为每个类模板X产生的类模板特化X< double >, X< string >等的友元

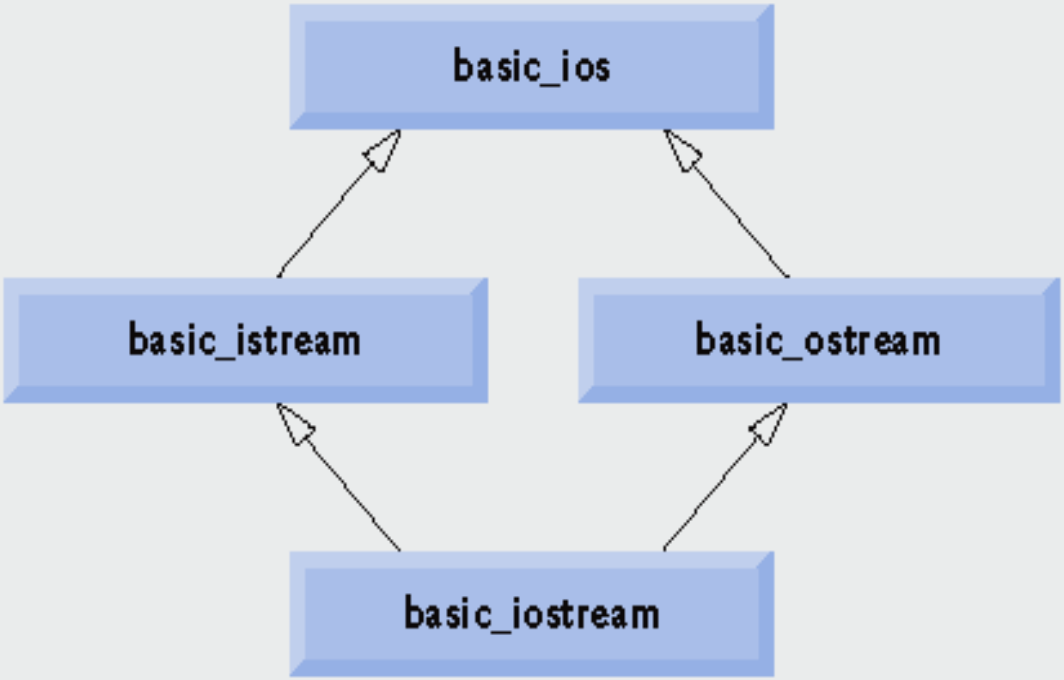
- ✓ 使一个类模板特化的所有成员函数都成为另一个包含相同类型参数的类模板特化的友元

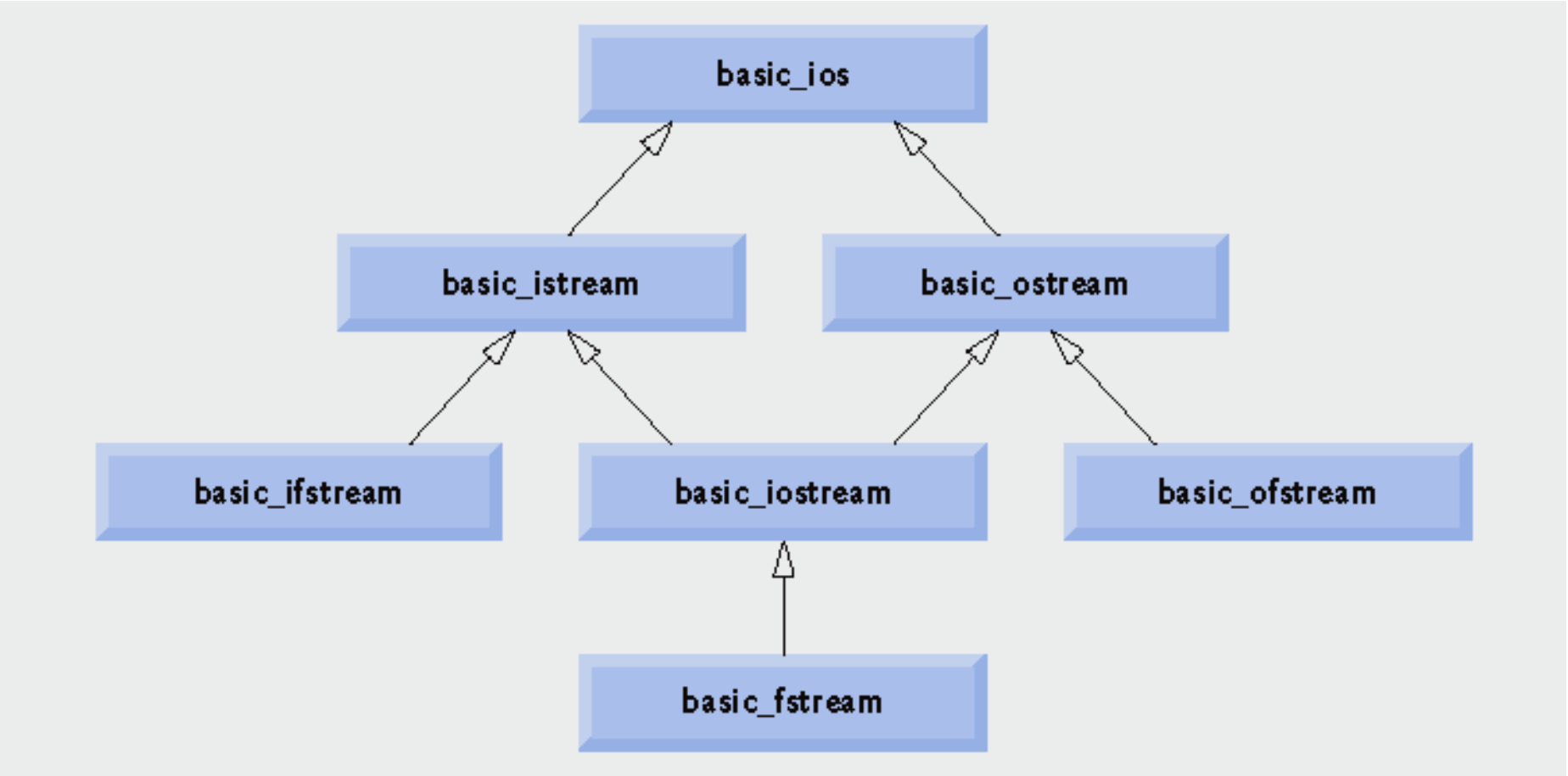
  friend class Z< T >;

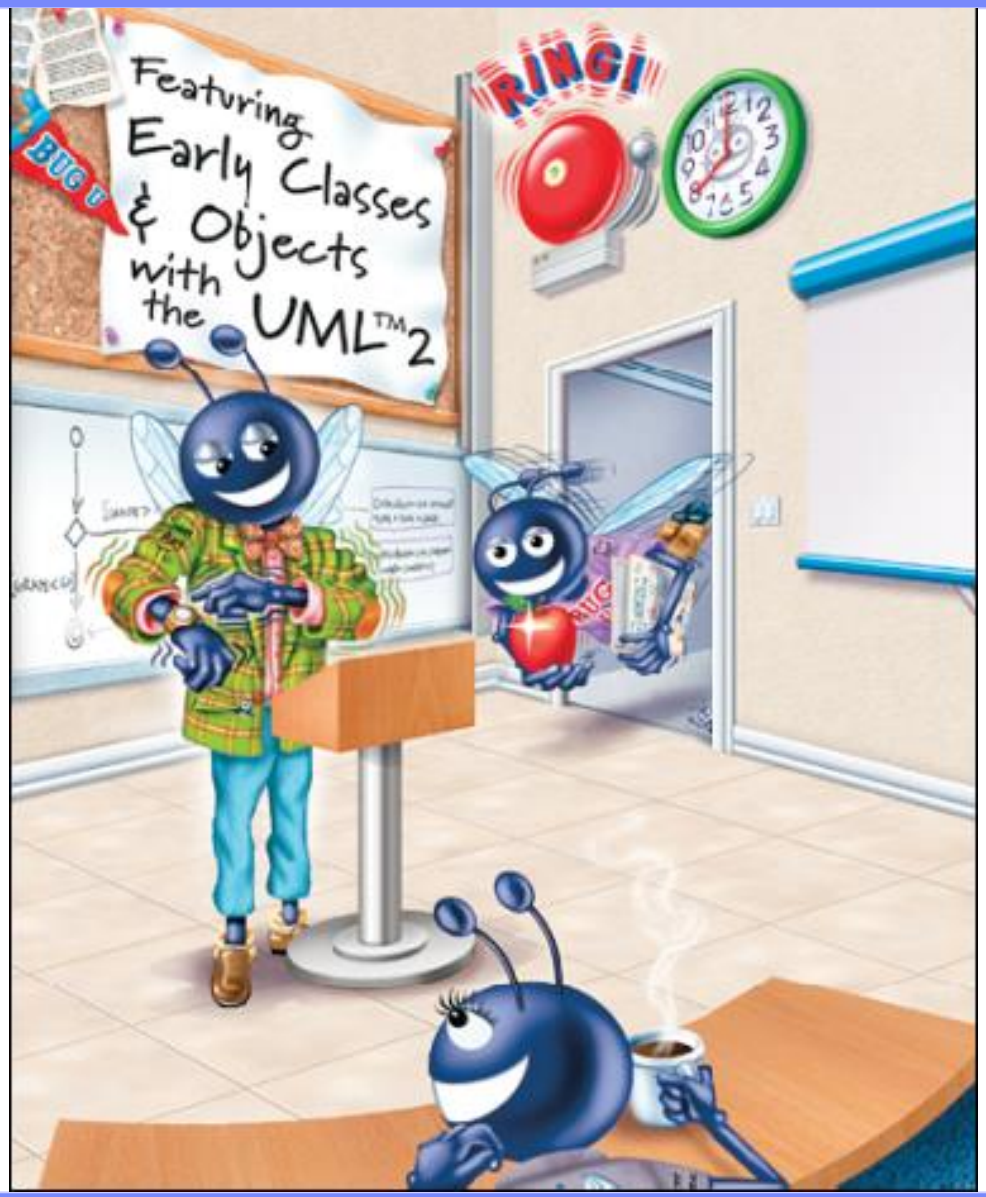
  类模板特化 Z< float >的所有成员函数都成为类模板特化 X< float >的友元，类模板特化Z< string >的所有成员函数是 类模板特化X< string >的友元

7 Notes on Templates and static Members

- 类模板中的静态成员
  - 每个模板特化类具有每个静态数据成员的拷贝
    - ✓ 所有模板特化类的对象共享一个静态数据成员
    - ✓ 静态数据成员必须定义和初始化
  - 每个模板特化类具有静态成员函数的拷贝


basic_ios
basic_istream
basic_ostream
basic_iostream


basic_ios
basic_istream
basic_ostream
basic_ifstream
basic_iostream
basic_ofstream
basic_fstream



Thank you!