Chapter 14

Templates

OBJECTIVES

- To use function templates to conveniently create a group of related (overloaded) functions.
- To distinguish between function templates and function-template specializations.
- To use class templates to create a group of related types.
- To distinguish between class templates and class -template specializations.
- To overload function templates.
- To understand the relationships among templates, friends, inheritance and static members.

Topics

- **14.1 Introduction**
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.1 Introduction

- 代码重用: 组合、继承、模板等
- 模板(template): 利用一种完全通用的方法来设计函数或类, 而不必预先指定将被使用的数据的类型— Generic Programming (泛型编程)
- 在C++语言中, 模板可分为类模板(class template)和函数模板(function template)
  - 模板(Template): 函数模板和类模板(参数化的类型)
  - 模板特化(Template Specialization): 模板函数和模板类

Topics

- 14.1 Introduction
- **14.2 Function Templates**
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.2 Function Templates

- 应用过程:
- 编程设计单个函数模板定义, 不指定若干参数的类型– 程序员
- 编译时, 由编译器根据调用时的实参确定这些数据的准确类型, 产生模板特化(目标代码), 即模板函数– 编译器
- 完成函数调用– 编译器

14.2 Function Templates

- GradeBook book[5];
  printArray(book, 5);

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

   cout << endl;
}
```

如何cout << GradeBook?

① 编译时 确认T为GradeBook

```
void printArray( const GradeBook *array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}
```

② 编译产生目标代码, 即针对GradeBook的printArray函数特化

14.2 Function Templates

- 注意:
- 模板仅仅是对函数或者类的说明, 本身无法编译和执行; 必须在编译时由编译器根据实际使用情况确认形式类型参数所对应的实际数据类型后, 才能编译成相应的目标代码—特化
- 如果模板被用户定义类型调用, 并且模板中对这些用户定义类型的对象使用了运算符(e.g., ==, +,<<) , 那么必须在该用户定义类型中overload这些运算符.

Topics

- 14.1 Introduction
- 14.2 Function Templates
- **14.3 Overloading Function Templates**
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.3 Overloading Function Templates

- 重载与函数模板密切相关
  - 函数模板特化之间
    1. printArray(int, ACOUNT);
    2. printArray(double, ACOUNT);
    3. printArray(char, ACOUNT);
  - 函数模板之间
    1. template< typename T >
       void printArray(const T *array, int count)
    2. template< typename T >
       void printArray(const T *array, int low, int high)
  - 函数模板与普通函数之间
    1. template< typename T >
       void printArray(const T *array, int count)
    2. void printArray(const int *array, int count)

```
template < typename T >
T max( T a, T b, T c )
{
    cout << "Template function called.\n";
    T m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    return m;
}
```

```
int max( int a, int b, int c )
{
    cout << "Ordinary function called.\n";
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;

    return m;
}
```

```
int main()
{
    cout << max(10, 20, 15) << endl;
    cout << max(10.0, 20.0, 15.0) << endl;
    return 0;
}
```

```
Ordinary function called.
20
Template function called.
20
```

- 如果仅有一个普通函数或者函数模板特化匹配函数调用，则使用该普通函数或者函数模板特化.
- 如果一个普通函数和一个函数模板特化均匹配函数调用，则使用普通函数.
- 否则，如果有多个匹配，则编译错误.

Topics

- 14.1 Introduction
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- **14.4 Class Templates**
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.4 Class Templates

- vector< int > integers1(7);
  类模板, 属于C++ STL

- Stack堆栈: 支持任何类型数据, 按照FILO原则进行入栈、出栈操作

- 类模板可称为参数化类型(parameterized types), 需要一个或多个类型参数来定制“通用类”以构成类模板特化.

14.4 Class Templates —应用

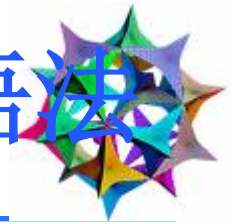
- 栈: FILO, 先进后出
- 数据成员
  - size: 栈的大小
  - top: 栈顶位置, 初始值为-1 (空栈)
  - stackPtr: 存储栈中元素的动态分配空间的数组
- 操作
  - push: 如果栈不满, 则向栈中增加一个元素, top增1
  - pop: 如果栈不空, 则从栈中弹出一个元素, top减1
  - isEmpty: 如果top为-1, 则栈为空
  - isFull: 如果top为size-1, 则栈满

14.4 Class Templates

- 栈: FILO, 先进后出

- 需求
- 建立栈的类模板, 可应用于不同类型的元素

14.4 Class Templates—语法



- (1) 类模板的定义

```
1.  template< typename T >
2.  class Stack
3.  {
4.      // class definition body
5.  };
```

- 其中T 可以用于成员函数(形参、局部变量和返回值)和数据成员

14.4 Class Templates —语法

- (2) 类模板的实现– 成员函数定义

```
1.  template< typename T >
2.  bool Stack< T >::push( const T &pushValue )
3.  {
4.     if ( !isFull() )
5.     {
6.        stackPtr[ ++top ] = pushValue; // place item on Stack
7.        return true; // push successful
8.     }
9.
10.    return false; // push unsuccessful
11. }
```

程序解读（P451）

- 非内联成员函数均作为类模板的函数模板来定义

14.4 Class Templates —语法

- (3) 类模板的使用(实例化1)

  Stack< double > doubleStack ;

- 和普通类实例化的差别在于类名需指定类型参数, 编译器依靠该信息对类模板进行特化
- 该程序为何仅两个文件?
  - 当客户程序使用模板时, 大部分编译器要求模板的完整定义均被包含入客户程序的源代码中, 因此:
  - (1) 类模板在头文件中定义
  - (2) 类模板的成员函数也在该头文件中定义

14.4 Class Templates

- (3) 类模板的使用(实例化2)

  Stack< Date > doubleStack ;

- 以对象作为栈元素时的两点限制:
- Fig 14.2.44: new动态分配数组内存– 缺省构造函数
- Fig 14.2.56/70: 对象赋值– 赋值符号重载

```
11      Stack< double > doubleStack( 5 ); // size
12      double doubleValue = 1.1;
14      cout << "Pushing elements onto doubleS
16      // push 5 doubles onto doubleStack
17      while ( doubleStack.push( doubleValue )
18      {
19         cout << doubleValue << ' ';
20         doubleValue += 1.1;
21      } // end while
23      cout << "\nStack is full. Cannot push " <<
24            << "\n\nPopping elements from dou
26      // pop elements from doubleStack
27      while ( doubleStack.pop( doubleValue ) )
28         cout << doubleValue << ' ';
30      cout << "\nStack is empty. Cannot pop\n
```

```
32      Stack< int > intStack; // default size 10
33      int intValue = 1;
34      cout << "\nPushing elements onto intSta
36      // push 10 integers onto intStack
37      while ( intStack.push( intValue ) )
38      {
39         cout << intValue << ' ';
40         intValue++;
41      } // end while
43      cout << "\nStack is full. Cannot push " <<
44            << "\n\nPopping elements from ints
46      // pop elements from intStack
47      while ( intStack.pop( intValue ) )
48         cout << intValue << ' ';
50      cout << "\nStack is empty. Cannot pop" <<
```

- 类型不一样 → 模板的形式类型参数
- 数值不一样 → 函数参数

```
template< typename T >
void testStack( T value,  T increment,
                const string stackName,
                int stackSize = 10 )
{
   Stack<T> theStack(stackSize);
   cout << "\nPushing elements onto "
        << stackName << '\n';

   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from "
        << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
}
```

```
11      Stack< double > doubleStack( 5 ); // size
12      double doubleValue = 1.1;
14      cout << "Pushing elements onto doubleS
16      // push 5 doubles onto doubleStack
17      while ( doubleStack.push( doubleValue )
18      {
19         cout << doubleValue << ' ';
20         doubleValue += 1.1;
21      } // end while
23      cout << "\nStack is full. Cannot push " <<
24           << "\n\nPopping elements from dou
26      // pop elements from doubleStack
27      while ( doubleStack.pop( doubleValue ) )
28         cout << doubleValue << ' ';
30      cout << "\nStack is empty. Cannot pop\n
```

```
32      Stack< int > intStack; // default size 10
33      int intValue = 1;
34      cout << "\nPushing elements onto intSta
36      // push 10 integers onto intStack
37      while ( intStack.push( intValue ) )
38      {
39         cout << intValue << ' ';
```

```
int main()
{
   testStack( 1.1, 1.1, "doubleStack", 5 );
   testStack( 1, 1, "intStack" );
   return 0;
}
```

- 类型不一样 → 模板的形式类型参数
- 数值不一样 → 函数参数

```
template< typename T >
void testStack( T value,  T increment,
                const string stackName,
                int stackSize = 10 )
{
   Stack<T> theStack(stackSize);
   cout << "\nPushing elements onto "
        << stackName << '\n';

   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from "
        << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
```

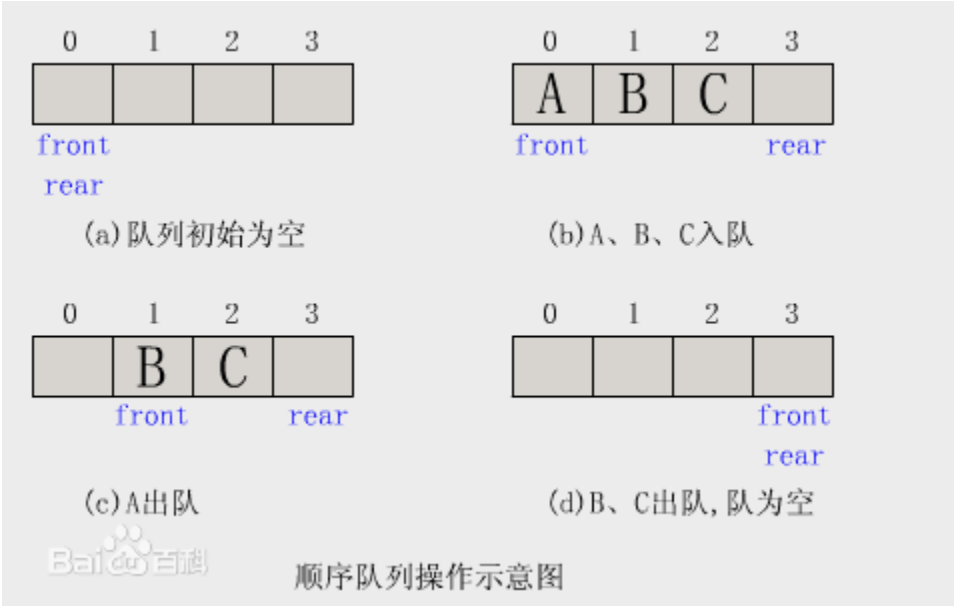
nplates

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class Stack{
5.  public:
6.      template <typename T>
7.      void test(T);
8.  };
9.  template <typename T>
10. void Stack::test( T num )
11. {
12.     cout << num << endl;
13. }
14. int main()
15. {
16.     A a;
17.     a.test( "hello" );
18.     a.test( 10 );
19.     return 0;
20. }
```

- test为普通类Stack的函数模板
- 加<T>的目的是区分类模板的函数模板和普通类的函数模板

队列

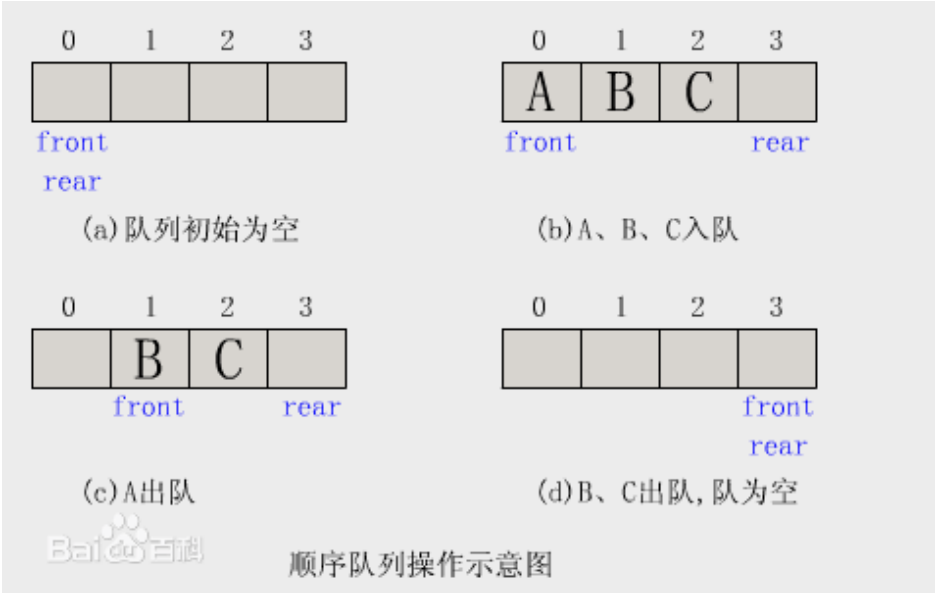
- 队列: **FIFO**, 先进先出
  - 顺序队列
  - 循环队列



顺序队列操作示意图

队列

- 队列: FIFO, 先进先出
  - 循环队列
- 数据成员
  - **size**: 队列的大小
  - **head**: 对头位置.
  - **tail**: 对尾位置
  - **ptr**: 存储队列中元素的动态分配空间的数组
- 操作
  - **enqueue**: 入队
  - **dequeue**: 出队
  - **isEmpty**: 判断队列是否为空
  - **isFull**:判断队列是否为满



顺序队列操作示意图

Topics

- 14.1 Introduction
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- **14.5 Nontype Parameters and Default Types for Class Templates**
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.5 Nontype Parameters and Default Types for Class Templates

- **Nontype template parameters(非类型模板参数)**
  - 模板声明

```
template< typename T, int elements >
class Stack{
    .........
};
```

  - 模板使用

```
Stack< double, 100 > myStack;
```

- **意义: 类模板在编译时进行特化, 因此elements在编译时即可确定值, 在模板类中可以当常量使用.**

```
template <class T, int size>
class Stack
{
   T buffer [size];
   int top;
public:
   Stack() { top = -1; }
   bool push( const T &x);
   bool pop( T &x );

   bool isEmpty() const
   {
      return top == -1;
   }

   bool isFull() const
   {
      return top == size - 1;
   }
};
```

```
template <class T, int size>
bool Stack <T, size>::push( const T &x )
{
  if ( !isFull() )
  {
    buffer[ ++top ] = x;
    return true;
  }
  return false;
}
template <class T, int size>
bool Stack <T, size>::pop( T &x )
{
  if ( !isEmpty() )
  {
    x = buffer[ top-- ];
    return true;
  }
  return false;
}
```

```
Stack <int,100> st1;
Stack <double, 200> st2;
```

14.5 Nontype Parameters and Default Types for Class Templates

- **Default Types** for Class Templates
  (形式类型参数的缺省值)
- • 模板声明

  template < typename T = string >
- • 模板使用

  Stack<> stringStack;

- • 要求: 只能是尾部的若干类型参数带缺省值

14.5 Nontype Parameters and Default Types for Class Templates

- **Explicit specializations** (显式特化)

  当某数据类型不能使用通用的类模板时, 可以定制处理(即: 重定义该类型的类模板)

```
template<>
class Stack< Employee >
{
    ......
};
```

- 可以用完全不同的代码替换原来的类模板

Topics

- 14.1 Introduction
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- **14.6 Notes on Templates and Inheritance**
- 14.7 Notes on Templates and Friends
- 14.8 Notes on Templates and static Members

14.6 Notes on Templates and Inheritance

- Templates and Inheritance
- • 类模板可以继承类模板特化
- • 类模板可以继承非模板类(即普通的类)
- • 类模板特化可以继承类模板特化
- • 非模板类可以继承类模板特化

Topics

- 14.1 Introduction
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- **14.7 Notes on Templates and Friends**
- 14.8 Notes on Templates and static Members

14.7 Notes on Templates and Friends

template< typename T > class X

- (1) 声明函数为类模板X的所有特化的友元:

  1对N

  friend void f1();

  - f1 is a friend of X< double >, X< string >, etc.

```
template< typename T >
class X{
    friend void f1( );
public:
    ......
};
```

14.7 Notes on Templates and Friends

template< typename T > class X

- (2) 声明函数为类模板X的相同类型参数的单个特化的友元: 1对1

  friend void f2(X<T> &);

  - f2(X< float > &) is a friend of X< float >, but not a friend of X< string >

```
template< typename T >
class X{
    friend void f2( X<T> & );
};
```

14.7 Notes on Templates and Friends

- (3) 声明类A的成员函数为类模板X的所有特化的友元: 1对N

```
friend void A::f3();
```

  - f3 of class A is a friend of X< double >, X< string >, etc.

- (4) 声明类模板C的成员函数为类模板X相同类型参数的单个特化的友元: 1对1

```
friend void C< T >::f4( X< T > & );
```

  - C< float >::f4(X< float > &) is a friend of X< float >, but not a friend of X< string >

14.7 Notes on Templates and Friends

- (5) 声明类Y为类模板X的所有特化的友元: 1对N

```
friend class Y;
```

  - Every member function of class Y is a friend of X< double >, X< string >, etc.

- (6) 声明类模板Z为类模板的相同类型参数的单个特化的友元: 1对1

```
friend class Z< T >;
```

  - Class-template specialization Z< float > is a friend of X< float >, Z< string > is a friend of X< string >, etc.

Topics

- 14.1 Introduction
- 14.2 Function Templates
- 14.3 Overloading Function Templates
- 14.4 Class Templates
- 14.5 Nontype Parameters and Default Types for Class Templates
- 14.6 Notes on Templates and Inheritance
- 14.7 Notes on Templates and Friends
- **14.8 Notes on Templates and static Members**

14.8 Notes on Templates and static Members

- 类模板的每个特化都有自己的static数据成员，static数据成员不在类模板的所有特化之间共享
- 该特化的所有对象共享这个static数据成员
- static数据成员必须在类定义外部定义和初始化

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

template <typename T>
class A
{
public:
   static int x;
};

template <typename T>
int A<T>::x = 0;
```

```
int main()
{
   A<int>::x = 10;
   A<double>::x = 20;

   A<int> a1, a2; // a1和a2共享一个x
   cout << a1.x << " ";
   a1.x = 15;
   cout << a2.x << endl;
   A<double> b1, b2; // b1和b2共享一个x
   cout << b1.x << " " << b2.x << endl;
   return 0;
}
```

```
10 15
20 20
```

Summary

- 函数模板和函数模板重载
- 类模板
  - 定义
  - 类型参数和非类型参数
  - 实例创建
- 模板和继承
- 模板和友元
- 类模板中的静态成员

Homework

- 实验必选题目：

  14.3 14.7

- 实验任选题目：

- 作业题目(Homework)：