Chapter 18

Templates

OBJECTIVES

- To use class templates to create a group of related types.
- To distinguish between class templates and class-template specializations.
- To overload function templates.

Topics

- **18.1 Introduction**
- 18.2 Function Templates
- 18.3 Overloading Function Templates
- 18.4 Class Templates
- 18.5 Nontype Parameters and Default Types for Class Templates

18.1 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

- 18.1 Introduction
- **18.2 Function Templates**
- 18.3 Overloading Function Templates
- 18.4 Class Templates
- 18.5 Nontype Parameters and Default Types for Class Templates

18.2 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

18.2 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

18.2 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

- 18.1 Introduction
- 18.2 Function Templates
- **18.3 Overloading Function Templates**
- 18.4 Class Templates
- 18.5 Nontype Parameters and Default Types for Class Templates

18.3 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

- 18.1 Introduction
- 18.2 Function Templates
- 18.3 Overloading Function Templates
- **18.4 Class Templates**
- 18.5 Nontype Parameters and Default Types for Class Templates

18.4 Class Templates

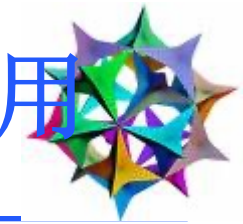
- vector< int > integers1(7);

  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

18.4 Class Te

```
class Stack
{
public:
    Stack(int =10);
    ~Stack();
    bool push(const int &);
    bool pop(int &);
    bool isEmpty()const;
    bool isFull()const;
private:
    int size;
    int top;
    int *stackPtr;
};
```



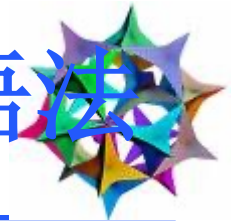
- 栈: FILO, 先进后出
- 数据成员
  - size: 栈的大小
  - top: 栈顶位置, 初始值为-1 (空栈)
  - stackPtr: 存储栈中元素的动态分配空间的数组
- 操作
  - push: 如果栈不满, 则向栈中增加一个元素, top增1
  - pop: 如果栈不空, 则从栈中弹出一个元素, top减1
  - isEmpty: 如果top为-1, 则栈为空
  - isFull: 如果top为size-1, 则栈满

18.4 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

18.4 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

Outline

```
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
```

```
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
```

18.4 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. }
```

- 非内联成员函数均作为类模板的函数模板来定义

```
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
```

```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 de

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

   cout << "Pushing elements onto doubleStack

   // 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 << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack; // default size 10
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   // push 10 integers onto intStack
   while ( intStack.push( intValue ) )
   {
      cout << intValue << ' ';
      intValue++;
   } // end while

   cout << "\nStack is full. Cannot push " << intValue
      << "\n\nPopping elements from intStack\n";

   // pop elements from intStack
   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
} // 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
```

18.4 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
- 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

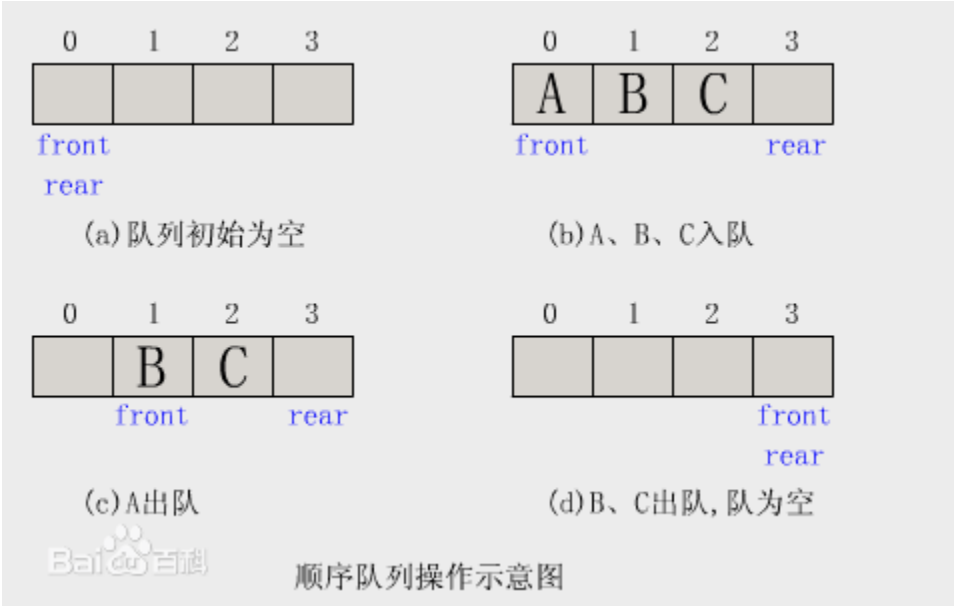
nplates

```
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

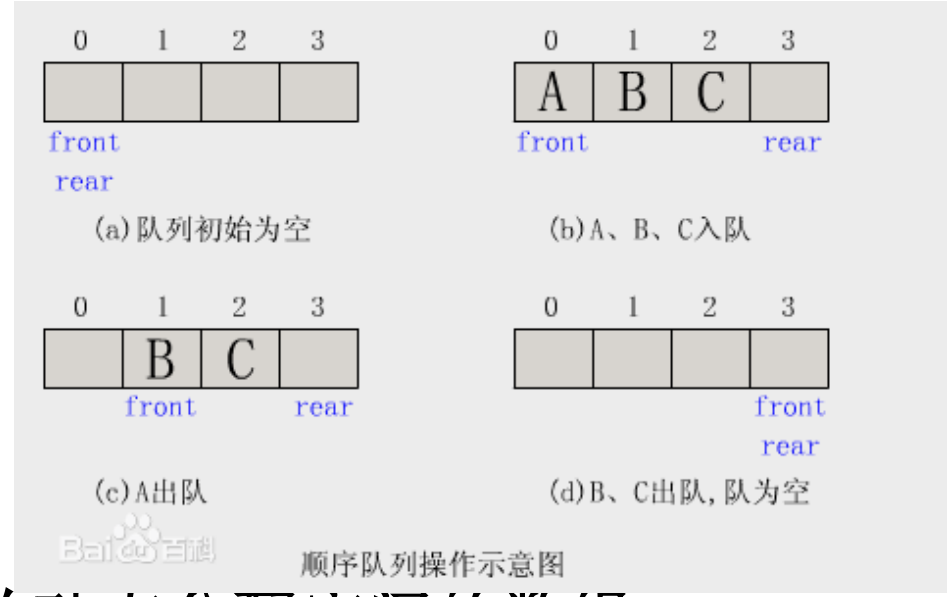
- 队列: FIFO, 先进先出
  - 顺序队列
  - 循环队列



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

```
template <typename T>
class Queue
{
private:
    T *ptr;
    int size, head, tail;
public:
    Queue(int s)
    : head(0)
    , tail(0)
    , size(s+1)
    { ptr = new T[size]; }

    ~Queue()
    { delete[] ptr; }

    bool isEmpty() const
    { return head == tail; }

    bool isFull() const
    { return (tail + 1) % size == head; }

    bool enqueue(const T &val);

    bool dequeue(T &val);
};
```

```
template <typename T>
bool Queue<T>::enqueue(const T &val)
{
    if (!isFull()) {
        ptr[tail]=val;
        tail=(tail+1)%size;
        return true;
    }
    return false;
}

template <typename T>
bool Queue<T>::dequeue(T &val)
{
    if (!isEmpty()) {
        val = ptr[head];
        head = (head + 1) % size;
        return true;
    }
    return false;
}
```

队列

```
int main()
{
    Queue<int> myQueue(5);
    int val=1;
    while(myQueue.enqueue(val))
    {
        cout<<val<<" ";
        val+=1;
    }
    cout<<"队列已满! \n";
    while(myQueue.dequeue(val))
    {
        cout<<val<<" ";
    }
    cout<<"队列已空! \n";
    for(int i=0;i<3;i++)
        myQueue.enqueue(i);
    myQueue.dequeue(val);
    myQueue.dequeue(val);
    for(int i=0;i<3;i++)
        myQueue.enqueue(i);
    while(myQueue.dequeue(val))
    {
        cout<<val<<" ";
    }
}
```

```
1 2 3 4 5 队列已满!
1 2 3 4 5 队列已空!
2 0 1 2
```

Topics

- 18.1 Introduction
- 18.2 Function Templates
- 18.3 Overloading Function Templates
- 18.4 Class Templates
- **18.5 Nontype Parameters and Default Types for Class Templates**

18.5 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;
```

18.5 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

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

Homework

- 实验必选题目：

  18.4