中山大學
SUN YAT-SEN UNIVERSITY

Lecture 11. C++ Templates

SMIE-121 Software Design II

zhangzizhen@gmail.com

School of Mobile Information Engineering, Sun Yat-sen University

Outline

Introduction

Function templates

Class templates

Templates with friends & static

Generic Programming

- In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.
- This approach, pioneered by ML in 1973, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication.

Example

对不同类型的数组，实现自加算法

1. 不使用函数模板

```
void selfAdd( int array[], int val, int size )  {
        for ( int i = 0; i < size; i++ )
                array[i] += val ;
}
void selfAdd( float array[], float val, int size )  {
        for ( int i = 0; i < size; i++ )
                array[i] += val ;
}
void selfAdd( double array[], double val, int size )  {
        for ( int i = 0; i < size; i++ )
                array[i] += val ;
}
```

实现代码相同，
支持数据类型不同

Example

对不同类型的数组，实现自加算法

2. 使用函数模板

```
Template < class T >
void selfAdd( T array[], T val, int size )  {
        for ( int i = 0; i < size; i++ )
                array[i] += val ;
 }
```

Templates

Easily create a large range of related functions or classes

将一段程序所处理的对象的数据类型参数化，以使得这段程序可以用于处理多种不同数据类型的对象。这避免了功能相同，数据类型不同的类出现, 实现代码复用。

将一个类所需要的数据类型参数化，使得该类成为能处理多种数据类型的通用类。在类的对象被创建时，通过指定参数所属的数据类型，来将通用类实例化。

这里的数据类型包括：

1. 数据成员的类型
2. 成员函数的参数的类型
3. 函数返回值的类型

Templates

Easily create a large range of related functions or classes

将一段程序所处理的对象的数据类型参数化，以使得这段程序可以用于处理多种不同数据类型的对象。这避免了功能相同，数据类型不同的类出现, 实现代码复用。

在template declarations（模板声明）中对模板类型参数定义时，"class" 和 "typename"是相同的。

```
template<class T> class Widget; // uses "class"
```

```
template<typename T> class Widget; // uses "typename"
```

Outline

Introduction

Function templates

Class templates

Templates with friends & static

Function Template vs. Template Function

- Function template - the blueprint of the related functions

  函数模板提供了一类函数的抽象，即代表一类函数。函数模板实例化后生成具体的模板函数。

- Template function - a specific function *made* from a function template

  模板函数是具体的函数。

  一般来说，模板函数在需要的时候才会生成。如

  ```
  template<class T>
  T add(T a, T b)
  ```

  在没有使用此函数模板的时候，并不会实例化任何函数模板，当调用add(1, 2)的时候，会生成模板函数int add(int a, int b)，当调用add(1.2, 1.4)的时候，才会生成模板函数double add(double a, doule b)

Instantiation

- 类和对象、函数模板和模板函数、类模板和模板类之间的关系。

类	实例化生成 →	对象
函数模板（一类函数）	实例化生成 →	模板函数（具体的函数）
类模板（类型参数化）	实例化生成 →	模板类（具体的类）

Function templates

- Function templates

  普通函数和**类的成员函数**可以声明为函数模板。

- Format:

```
template <class or typename T>
returnType functionName( parameterList ){
 //definition
 }
 Example:
```

```
Template < class T >
void selfAdd( T array[], T val, int size ) {
        for ( int i = 0; i < size; i++ )
                array[i] += val ;
}
```

Function templates

- 模板函数的参数分为：**函数实参**和**模板实参**。

```
Template < class T >
void selfAdd( T array[], T val, int size ) {…}
int main() {
    int a[10], val = 2 ;
    seftAdd( a, val, 10 ) ; //省略模板实参
    seftAdd<int>( a, val, 10 ) ;
    return 0 ;
}
```

函数实参

模板实参

Function templates

- 模板函数的模板实参可以省略，编译器将从函数实参的类型中推断。

- 下列情况，不能省略。
  - 1. 从函数实参获得的信息有矛盾
  - 2. 需要获得特定类型的返回值
  - 3. 虚拟类型参数没有出现在模板函数的形参表中
  - 4. 函数模板含有常规形参

Template parameters

1. 从函数实参获得的信息有矛盾：

```
template< typename T >
T add(T a,T b){return a+b;}
```

而调用语句为：

```
cout << add( 3.0, 5 ) << endl ; //error:歧义产生
cout << add< float >( 3.0, 5 ) << endl ; //OK!
```

Template parameters

2. 需要获得特定类型的返回值：

```
template< typename T >
T add(T a,T b){return a+b;}
```

需要add返回一个 int 型的值，
直接调用add< int >(a, b) ;

Template parameters

3. 虚拟类型参数没有出现在模板函数的形参表中（多义性）。如下图所示，为避免T2的数据类型未知，必须指定T2的数据类型。。

```
template<typename T1,typename T2,typename T3>
T2 add(T1 a,T3 b){return a+b;}
void main() {
      cout<<showpoint;
      cout<<add<double,int>(3,5L)<<endl;
```

```
程序运行结果为:
8
8.00000
```

```
}
```

Template parameters

- 当模板定义含有常规形参时，如果此常规形参并未同时出现在函数模板的函数形参表中，则在调用时，无法通过函数实参，初始化此参数，故而必须显式的给出对应的模板实参。

```
template < class T, int rows>
sum(T data[],T &result)
{result=0;
 for(int i=0;i<rows;i++)
      result+=data[i];}
int main()
{int d[3]={1,2,3};int r;
 sum< int,3>(d,r); //此处必须显式给出对应于常规参数的模板实参
```

Overloading A Function Template

- Which version to use?

Sequence of matching:

(1)The common function with matching parameter list (no type conversion);

(2)The matching function template (no type conversion);

(3)The common function with matching parameter list after implicit type conversion;

(4) Otherwise, compiling error.

(a)	template <class TYPE> TYPE max(TYPE x, TYPE y);
(b)	template <class TYPE> TYPE max(TYPE x, TYPE y, TYPE z);
(c)	template <class TYPE> TYPE max(TYPE x[], int n);
(d)	double max(int x, double y);

Example:

(1) max(1, 1.2);

(2) max(2, 3);

(3) max(3, 4, 5);

Example:

(4) max(array1, 5);

(5) max(2.1, 4.5)

(6) max('B', 9)

Outline

Introduction

Function templates

Class templates

Templates with friends & static

Class templates

- Class templates
  - Allow type-specific versions of generic classes
- Format:

```
template <class T>
class ClassName{
  T var;
  // other definitions ...
  } ;
```

  - Need not use "T", any identifier will work
  - To create an object of the class, type

```
ClassName < type > myObject;
```

Example: `Stack< double > doubleStack;`

Class templates

- Function Template in class
  - Defined normally, but preceded by `template< class T>`
    - Generic data in class listed as type `T`
  - Binary scope resolution operator used

    `MyClass< T >::MyClass(int size)`

  - Function template in class definition:

```
//Constructor definition - creates an array of type T
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

Template parameters

- 模板形参的名字不能在模板内部重用，也就是说一个名字在一个模板中只能使用一次:

  template<class u, class u> //error

- 类模板的声明和构造子定义中模板形参的名字可以不同：

  声明：template<class T>

  class A{...}

  构造子：template<class U>

  A<U>::A(){...}

```
// Fig. 22.1: tstack1.h
// Class template Stack
#ifndef TSTACK1_H
#define TSTACK1_H

template< class T >
class Stack {
public:
   Stack( int = 10 );    // default constructor (stack size 10)
    ~Stack() { delete [] stackPtr; } // destructor
    bool push( const T& ); // push an element onto the stack
    bool pop( T& );         // pop an element off the stack
 private:
    int size;              // # of elements in the stack
    int top;               // location of the top element
    T *stackPtr;           // pointer to the stack

    bool isEmpty() const { return top == -1; }      // utility
   bool isFull() const { return top == size - 1; } // functions
 }; // end class template Stack

```

```
// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;               // Stack is initially empty
   stackPtr = new T[ size ]; // allocate space for elements
} // end Stack constructor

// Push an element onto the stack
// return 1 if successful, 0 otherwise
template< class T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue; // place item in Stack
      return true;  // push successful
   } // end if
   return false;     // push unsuccessful
} // end function template push

```

```
// Pop an element off the stack
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
   } // end if
   return false;     // pop unsuccessful
} // end function template pop

#endif
```


tstack1.h (Part 3 of 3)

```
// Fig. 22.1: fig22_01.cpp
// Test driver for Stack template
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"
```

fig22_01.cpp (Part 1 of 3)

```cpp
int main()
{
   Stack< double > doubleStack( 5 );
   double f = 1.1;
   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( f ) ) { // success true returned
      cout << f << ' ';
      f += 1.1;
   } // end while

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

   while ( doubleStack.pop( f ) )  // success true returned
      cout << f << ' ';

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

   Stack< int > intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( i ) ) { // success true returned
      cout << i << ' ';
      ++i;
   } // end while
```

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

   while ( intStack.pop( i ) )  // success true returned
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop\n";
    return 0;
} // end function main
```

fig22_01.cpp (Part 3 of 3)

```
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
```

```cpp
// Fig. 22.2: fig22_02.cpp
// Test driver for Stack template.
// Function main uses a function template to manipulate
// objects of type Stack< T >.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"

// Function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,    // reference to the Stack< T >
   T value,                 // initial value to be pushed
   T increment,             // increment for subsequent values
   const char *stackName )  // name of the Stack < T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

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

```

Outline

fig22_02.cpp (Part 1 of 2)

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

   while ( theStack.pop( value ) )  // success true returned
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";
} // end function template testStack

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
} // end function 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
```


Program Output

Template parameters

- Can use non-type parameters in class templates
  - 可以是常整数（包括枚举）、指向外部链接对象的指针，而浮点数，指向内部链接对象的指针则不行。
  - must be constant at compile time

Example:

```
Template < class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

- Defines object of type `Stack< double, 100>`

Template parameters

Non-type parameters are resolved at compile time, not runtime

Example:

This may appear in the class definition:

```
Template < class T, int elements >
```

```
T stackHolder[ elements ]; //array to hold stack
```

The array is created at compile time, rather than dynamic allocation at execution time

Template parameters

- Class templates can have default arguments for type or value parameters.
- template <class T = long> class A;
- Function templates CANNOT have default arguments

Class template specialization

- Classes can be overridden
  - For template class `Array`, define a class named `Array<myCreatedType>`
  - This new class overrides the class template for `myCreatedType`
  - The template remains for unoverriden types

Class templates specialization

- 应用场景：即想使用模板，同时又需要对一个特殊类型做不同的实现。

```
// class template:
template <class T>
class specTemplate  {
   T m_var;
public:
   specTemplate (T inData)
   { m_var = inData; }
   T increase () { return ++m_var; }
};
```

```
// class template specialization:
template <>
class specTemplate <char>  {
   char m_var;
public:
   specTemplate (char arg) { m_var = arg; }
   char upperCase ()   {
      if ((m_var >= ’ a’ ) && (m_var
<= ’ z’ ))
         m_var += ’ A’ -’ a’ ;
      return m_var;    }  } ;
```

Template and inheritance

- A non-template class can be derived from a template class（普通类继承模板类）
- A template class can be derived from a non-template class（模板类继承了普通类（非常常见））
- A class template can be derived from a class template（类模板继承类模板）
- A template class can be derived from a class template（模板类继承类模板，即继承模板参数给出的基类）

Template and inheritance

1. 普通类继承模板类

```
template<class T>
class TBase{
      T data;
……
};
class Derived:public TBase<int>{
……
};
```

Templates and inheritance

2. 模板类继承了普通类（非常常见）

```
class Base{
……
};
template<class T>
class TDerived:public Base{
T data;
……
};
```

Templates and inheritance

3. 模板类继承模板类

```
template<class T>
class TBase{
T data1;
……
};
template<class T1,class T2>
class TDerived:public TBase<T1>{
T2 data2;
……
};
```

```
template<typename T>
```

```
void main()
{
    Derived<BaseA> x;// BaseA作为基类
    Derived<BaseB> y;// BaseB作为基类
    Derived<BaseC<int> > z(3); // BaseC<int>作为基类
}
```

```
};
```

```
程序运行结果为:
BaseA founed
Derived founed
BaseB founed
Derived founed
BaseC founed 3
Derived founed
```

Outline

Introduction

Function templates

Class templates

Templates with friends & static

Templates and friends

- Friendships allowed between a class template and
  - **Global function**
  - **Member function of another class**
  - **Entire class**

- `friend` functions, Inside definition of class template **`X`**:
  - `friend void f1();`
    - `f1()` a `friend` of all template classes
  - `friend void f2( X< T > & );`
    - `f2( X< int > & )` is a `friend` of `X< int >` only. The same applies for `float`, `double`, etc.
  - `friend void A::f3();`
    - Member function `f3` of class `A` is a `friend` of all template classes
  - `friend void C< T >::f4( X< T > & );`
    - `C<float>::f4( X< float> & )` is a `friend` of `class X<float>` only

Templates and friends

- `friend` classes, Inside definition of class template **`X`**:
  - `friend class Y;`
    - Every member function of `Y` a friend with every template class made from `X`
  - `friend class Z<T>;`
    - Class `Z<float>` a `friend` of class `X<float>`, etc.

Templates and static Members

- Non-template class
  - `static` data members shared between all objects
- Template classes
  - Each class (`int`, `float`, etc.) has its own copy of `static` data members
  - `static` variables initialized at file scope
  - Each template class gets its own copy of `static` member functions

Template is NOT OO

- Bjarne Stroustrup has described the C++ programming language that he created as

- "*a general-purpose programming language that supports procedural programming, data abstraction, object-oriented programming, and generic programming.*"

- Strictly speaking, Template is a technique of Generic Programming, NOT of Object-Oriented Programming

- The C++ class templates (with inheritance rules) is the application of GP on OO

Template vs. other dynamic techniques

- C++ Template vs. Polymorphism
  - Compile time vs. Runtime
  - template is also called a kind of compile time polymorphism

- C++ Template vs. Marco
  - Generic programming vs. Metaprogramming
  - Template is not the only way of Generic programing
    - e.g. Generic in Java

Thank you!