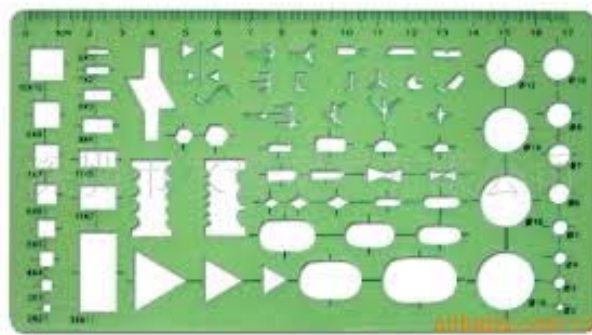


Template

Meng-Hsun Tsai
CSIE, NCKU



Introduction

- **Function templates** and **class templates** enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called **function-template specializations**—or an entire range of related classes—called **class-template specializations**.
- This technique is called **generic programming**.
- Note the distinction between **templates** and **template specializations**:
 - Function **templates** and class templates are like **stencils** out of which we trace shapes.
 - Function-**template specializations** and class-template specializations are like the **separate tracings** that all have the same shape, but could, for example, be drawn **in different colors**.

Introduction (cont.)

- Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, **templates are often defined in header files**, which are then `#include`d into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.

Function Templates

- Overloaded functions normally perform *similar or identical operations* on different types of data.
- If the operations are *identical* for each type, they can be expressed more compactly and conveniently *using function templates*.
- Initially, you write a *single* function-template definition.
- Based on the argument types provided explicitly or inferred from calls to this function, the *compiler generates separate source-code functions* (i.e., *function-template specializations*) to handle each function call appropriately.

Function Templates (cont.)

- All **function-template definitions** begin with keyword **template** followed by a list of **template parameters** to the function template enclosed in **angle brackets** (< and >); each template parameter that represents a type must be preceded by either of the interchangeable keywords **class** or **typename**, as in
 - **template<typename T>**
 - Or
 - **template<class ElementType>**
 - Or
 - **template<typename BorderType, typename FillType>**

Function Templates (cont.)

- The type template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
- Keywords **typename** and **class** used to specify function-template parameters actually mean “any fundamental type or user-defined type.”

Example: *printArray()*

```
1 #include <iostream>
2 using namespace std;
3 template <typename T>
4 void printArray(const T* const arr, int count)
5 {
6     for(int i=0;i<count;++i)
7         cout << arr[i] << " ";
8     cout << endl;
9 }
10 int main()
11 {
12     const int aCount=5;
13     const int bCount=7;
14     const int cCount=6;
15
16     int a[aCount] = {1,2,3,4,5};
17     double b[bCount] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
18     char c[cCount] = "HELLO";
19
```

```
20     cout << "Array a: ";
21     printArray(a, aCount);
22     cout << "Array b: ";
23     printArray(b, bCount);
24     cout << "Array c: ";
25     printArray(c, cCount);
26 }
```

Array a: 1 2 3 4 5

Array b: 1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array c: H E L L O

- T is referred to as a **type template parameter**, or **type parameter**.

Function Templates (cont.)

- When the compiler detects a `printArray` function invocation in the client program (e.g., lines 21, 23 and 25), the compiler uses its overload resolution capabilities to find a definition of function `printArray` that best matches the function call.
- The compiler compares the type of `printArray`'s first argument (`int *` at line 21) to the `printArray` function template's first parameter (`const T * const` at line 4) and deduces that replacing the type parameter `T` with `int` would make the argument consistent with the parameter.
- Then, the compiler substitutes `int` for `T` throughout the template definition and compiles a `printArray` specialization that can display an array of `int` values.

Function Templates (cont.)

- The **function-template specialization** for type `int` is
 - ```
void printArray(const int * const array, int count)
{
 for (int i = 0; i < count; i++)
 cout << array[i] << " ";
 cout << endl;
}
```
- As with function parameters, **the names of template parameters must be unique inside a template definition.**
- Template parameter names **need not be unique across different function templates.**

```
template < typename T >
int func1(T par) {...}
```

```
template < typename T >
double func2(T par) {...}
```

# Overloading Function Templates

- Function templates and overloading are intimately related.
- The function-template specializations generated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.
- A function template may be overloaded in several ways.
  - We can provide other function templates that specify the same function name but different function parameters.
  - We can provide nontemplate functions with the same function name but different function arguments.

# Error: Ambiguous Definitions

- A **compilation error** occurs if **no matching function** definition can be found for a particular function call **or** if there are **multiple matches** that the compiler considers ambiguous.



```
1 #include <iostream>
2 using namespace std;
3 template <typename T>
4 void fun(T t) {cout << t;}
5 template <typename U>
6 void fun(U u) {cout << u;}
7 int main()
8 {
9 int x;
10 fun(x);
11 return 0;
12 }
```

```
ambiguous.cpp:6: error: redefinition of
'template<class U> void fun(U)'
ambiguous.cpp:4: error: 'template<class T> void
fun(T)' previously declared here
```

# Example: Overloading vs. Templates

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

template <typename T>
int func(T t) { cout << t << ": template function" << endl; }

int func(int Y)
{ cout << Y << ": function overloading with int" << endl; }

int func(double Z)
{ cout << Z << ": function overloading with double" << endl; }

int main()
{
 int x = 5;
 func(x);
 return 0;
}
```

**Output:**  
5: function overloading with int

# Class Templates

- It's possible to understand the **concept of a “stack”** (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) **independent of the type of the items** being placed in the stack.
- **We need the means for describing the notion of a stack generically** and instantiating classes that are type-specific versions of this generic stack class.
- **C++ provides this capability through class templates.**
- Class templates are called **parameterized types**, because they require one or more type parameters to **specify how to customize a “generic class” template to form a class-template specialization.**

# Stack.h

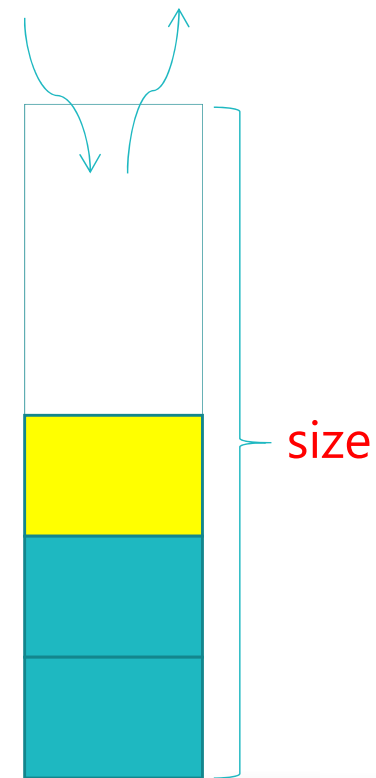
```
1 #ifndef STACK_H
2 #define STACK_H
3 template <typename T>
4 class Stack {
5 public:
6 Stack(int=10);
7 ~Stack()
8 {
9 delete [] stackPtr;
10 }
11 bool push(const T&);
12 bool pop(T&);
13 bool isEmpty() const
14 {
15 return top == -1;
16 }
17 bool isFull() const
18 {
19 return top == size -1;
20 }
```

```
21 private:
22 int size;
23 int top;
24 T* stackPtr;
25 };
26 template <typename T>
27 Stack<T>::Stack(int s): size(s>0?s:10),
28 top(-1),stackPtr(new T[size]) { }
29 template <typename T>
30 bool Stack<T>::push(const T& val)
31 {
32 if(!isFull())
33 {
34 stackPtr[++top] = val;
35 return true;
36 }
37 return false;
38 }
```

push() pop()

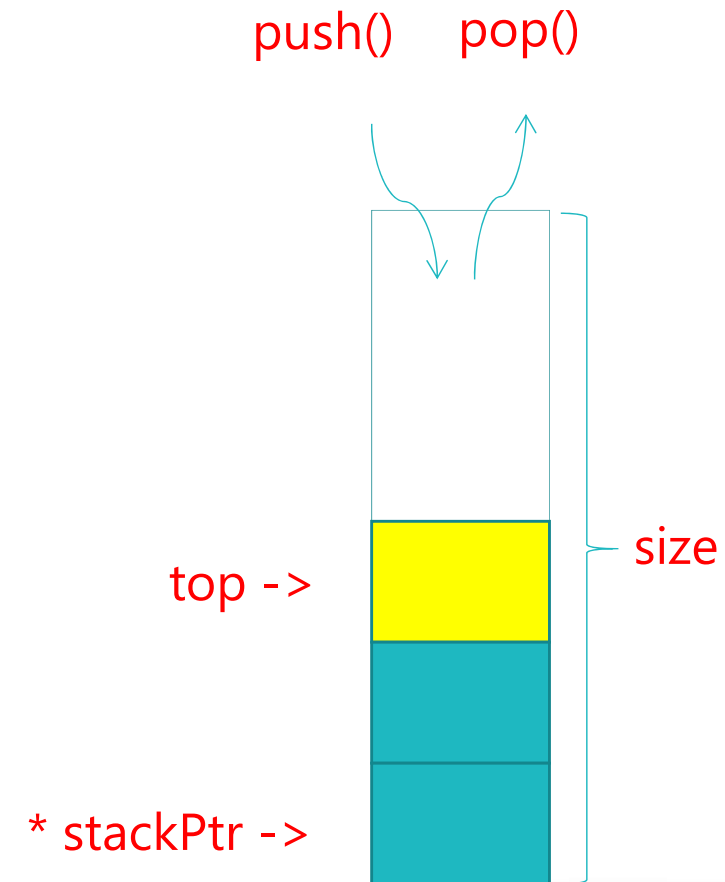
top ->

\* stackPtr ->



## Stack.h (cont.)

```
38 template <typename T>
39 bool Stack<T>::pop(T& val)
40 {
41 if(!isEmpty())
42 {
43 val = stackPtr[top--];
44 return true;
45 }
46 return false;
47 }
48 #endif
```



## use\_stack.cpp

```
1 #include <iostream>
2 #include "Stack.h"
3 using namespace std;
4 int main()
5 {
6 Stack <double> dStk(5);
7 double dVal = 1.1;
8 cout << "push elements: ";
9 while(dStk.push(dVal))
10 {
11 cout << dVal << ' ';
12 dVal += 1.1;
13 }
14 cout << "\nStack is full, cannot push " << dVal;
15 cout << "\npop elements: ";
16 while(dStk.pop(dVal))
17 cout << dVal << ' ';
18 cout << "\nStack is empty\n";

19 Stack<int> iStk;
20 int iVal = 1;
21 cout << "push elements: ";
22 while(iStk.push(iVal))
23 cout << iVal++ << ' ';
24 cout << "\nStack is full, cannot push " << iVal;
25 cout << "\npop elements: ";
26 while(iStk.pop(iVal))
27 cout << iVal << ' ';
28 cout << "\nStack is empty\n";
29 }
```

push elements: 1.1 2.2 3.3 4.4 5.5  
Stack is full, cannot push 6.6  
pop elements: 5.5 4.4 3.3 2.2 1.1  
Stack is empty  
push elements: 1 2 3 4 5 6 7 8 9 10  
Stack is full, cannot push 11  
pop elements: 10 9 8 7 6 5 4 3 2 1  
Stack is empty



# Class Templates (cont.)

- The `Stack` class-template definition looks like a conventional class definition, except that it's preceded by the header (line 3)

- `template< typename T >`

to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created.

- The type of element to be stored on this `Stack` is mentioned generically as `T` throughout the `Stack` class header and member-function definitions.
- Due to the way this class template is designed, there are two constraints for non-fundamental data types used with this `Stack`
  - they must have a default constructor
  - their assignment operators must properly copy objects into the `Stack`

# Class Templates (cont.)

- The member-function definitions of a class template are function templates.
- The member-function definitions that appear outside the class template definition each begin with the header
  - `template< typename T >`
- Thus, each definition resembles a conventional function definition, except that the Stack element type always is listed generically as type parameter T.
- The binary scope resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.

# Class Templates (cont.)

- The program begins by instantiating object dStk of size 5.
- This object is declared to be of class `Stack< double >` (pronounced “Stack of double”).
- The compiler associates type double with type parameter T in the class template to produce the source code for a Stack class of type double.
- Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

# Class Templates (cont.)

- Notice that the code in function `main` is almost identical for both the `dStk` manipulations in lines 6–18 and the `iStk` manipulations in lines 19–28.
- This presents another opportunity to use a function template.
- In the next example, we define function template `testStack` (lines 5–19) to push a series of values onto a `Stack< T >` and pop the values off a `Stack< T >`.
- Function template `testStack` uses template parameter `T` (specified at line 5) to represent the data type stored in the `Stack< T >`.

## test\_stack.cpp

```
1 #include <iostream>
2 #include <string>
3 #include "Stack.h"
4 using namespace std;
5 template <typename T>
6 void testStack(Stack<T> &theStack,
7 T value, T increment, const string stackName)
8 {
9 cout << "push elements from " << stackName << ":\n";
10 while(theStack.push(value))
11 {
12 cout << value << ' ';
13 value += increment;
14 }
15 cout << "\nStack is full. Cannot push "
16 << value << "\npop elements from "
17 << stackName << ":\n";
18 while(theStack.pop(value))
19 {
20 cout << value << ' ';
21 }
22 cout << "\nStack is empty.\n";
23 }
```

```
21 int main()
22 {
23 Stack<double> dStk(5);
24 Stack<int> iStk;
25 testStack(dStk, 1.1, 1.1, "dStk");
26 testStack(iStk, 1, 1, "iStk");
27 return 0;
28 }
```

push elements from dStk:  
1.1 2.2 3.3 4.4 5.5  
Stack is full. Cannot push 6.6  
pop elements from dStk:  
5.5 4.4 3.3 2.2 1.1  
Stack is empty.  
push elements from iStk:  
1 2 3 4 5 6 7 8 9 10  
Stack is full. Cannot push 11  
pop elements from iStk:  
10 9 8 7 6 5 4 3 2 1  
Stack is empty.

21