COMP6771
Advanced C++ Programming

Week 7
Part One: Member Templates and Specialisation

2016

www.cse.unsw.edu.au/~cs6771

Member Templates

Consider this STL code:

```
1  #include <iostream>
2  #include <vector>
3  #include <list>
4
5  int main() {
6  std::vector<int> ivec(10)
7  std::vector<int> ivec0 = ivec; // ok
8  std::list<int> ilist0 = ivec; // ok?
9                                // error: different containers
10 std::list<int> ilist(ivec.begin(), ivec.end()); ok?
11                                   // ok: ctor exists
12 ilist.assign(ivec.begin(), ivec.end()); ok?
13                                   // ok: member exists
14 }
```

`std::list<T>` is able to handle items of `std::vector<T>`

Stack Revised: Member Template Functions

```
1  #ifndef DEFAULTARGUMENTSTACK_HPP
2  #define DEFAULTARGUMENTSTACK_HPP
3
4  #include <iostream>
5  #include <deque>
6
7  template <typename,typename> class Stack;
8
9  template <typename T, typename CONT>
10 std::ostream& operator<< (std::ostream &os, const Stack<T, CONT> &s);
11
12 template <typename T, typename CONT = std::deque<T>> class Stack {
13 public:
14   friend std::ostream& operator<<<T>(std::ostream &, const Stack<T, CONT> &);
15   void push(const T &);
16   void pop();
17   T& top();
18   const T& top() const;
19   bool empty() const;
20 private:
21   CONT stack_;
22 };
```

Client Code

- The default ctor, copy-ctor and operator= are correct:

```
1 Stack<int> s1; // instantiate Stack<int>
2 s1.push(1);    // instantiate Stack<int>::push(const int&)
3 Stack<int> s2 = s1; // calls Stack<int>(const Stack<int>&)
4 s2 = s1; // calls Stack<int>::operator=(const Stack<int> &)
```

- What if two stacks have different element types?

```
1 Stack<int> is;
2 Stack<float> fs = is; // error: no such ctor
3 fs = is; // error: no such operator=
4 Stack<float> fs(is.begin(), is.end());
5           // error: ctor doesn't exist
6 fs.assign(is.begin(), is.end());
7           // error: assign doesn't exist
```

Supporting Member Templates

Add to defaultArgumentsStack.hpp:

```
1  template <typename T, typename CONT = std::deque<T>> class Stack {
2    // addition declarations:
3    Stack() {} // must define default constructor
4
5    // template function inside template class.
6    template <typename T2, typename CONT2>
7      Stack(const Stack<T2, CONT2>&);
8
9    template <typename T2, typename CONT2>
10     Stack(Stack<T2, CONT2>&&);
11
12   template <typename Iter> Stack(Iter b, Iter e);
13
14   template <typename T2, typename CONT2>
15     Stack& operator=(const Stack<T2,CONT2> &);
16
17   template <typename T2, typename CONT2>
18     Stack& operator=(Stack<T2,CONT2> &&);
19
20   template <typename Iter> void assign(Iter b, Iter e);
21 }
```

stack.h Expanded to Support Member Templates I

```
1  template <typename T, typename CONT>
2  template <typename T2, typename CONT2>
3  Stack<T,CONT>::Stack(const Stack<T2,CONT2> & s) {
4    Stack<T2,CONT2> tmp(s);
5    while (!tmp.empty()) {
6      stack_.push_front(tmp.top());
7      tmp.pop();
8    }
9  }
10
11 template <typename T, typename CONT>
12 template <typename T2, typename CONT2>
13 Stack<T,CONT>::Stack(Stack<T2,CONT2> && s) {
14   while (!s.empty()) {
15     stack_.push_front(s.top());
16     s.pop();
17   }
18 }
19
20 template <typename T, typename CONT>
21 template <typename Iter>
22 Stack<T,CONT>::Stack(Iter b, Iter e) {
23   for (; b != e; ++b)
24     stack_.push_back(*b);
25 }
26
27
28
29
```

stack.h Expanded to Support Member Templates II

```
30 template <typename T, typename CONT>
31 template <typename T2, typename CONT2>
32 Stack<T,CONT>&
33 Stack<T,CONT>::operator=(const Stack<T2,CONT2> & s) {
34   if ((void *)this == (void *)&s)
35     return *this;
36
37   Stack<T2,CONT2> tmp(s);
38   stack_.clear();
39   while (!tmp.empty()) {
40     stack_.push_front(tmp.top());
41     tmp.pop();
42   }
43   return *this;
44 }
45
46 template <typename T, typename CONT>
47 template <typename T2, typename CONT2>
48 Stack<T,CONT>&
49 Stack<T,CONT>::operator=(Stack<T2,CONT2> && s) {
50   if ((void *)this == (void *)&s)
51     return *this;
52   stack_.clear();
53   while (!s.empty()) {
54     stack_.push_front(s.top());
55     s.pop();
56   }
57   return *this;
58 }
```

stack.h Expanded to Support Member Templates III

```
59
60 template <typename T, typename CONT>
61 template <typename Iter>
62 void Stack<T,CONT>::assign(Iter b, Iter e) {
63   stack_.clear();
64   for (; b != e; ++b)
65     stack_.push_back(*b);
66 }
```

- Lazy Instantiation: Only member functions called are instantiated:
  - vector does not have push_front()
  - Can still use a vector as the internal container if you avoid assigning a stack with elements of a different type
- Member templates cannot be virtual

  Otherwise, the number of instantiations is not fixed. Cannot build vtable for the class unless the entire program has been compiled!

Client Code

```
1  #include "memberTemplatesStack.hpp"
2
3  int main() {
4    float a[] = {1.1, 2.2, 3.3};
5    Stack<float> fs(a, a+3);
6    // instantiate Stack<float, deque<float>>(float*, float*)
7
8    fs.assign(a, a+3);
9    // Stack<float, deque<float>>::assign(float *, float*);
10   std::cout << fs << std::endl;
11
12   Stack<int> is = fs;
13   // Stack<int, deque<int>>(const Stack<float, deque<float>> &)
14   std::cout << is << fs << std::endl;
15
16   is = fs;
17   // Stack<int, deque<int>>::operator=(const Stack<float, deque<float>> &)
18   std::cout << is << fs << std::endl;
19 }
```

OUTPUT:

```
1.1 2.2 3.3
1 2 3 1.1 2.2 3.3
1 2 3 1.1 2.2 3.3
```

Client Code (with Move Semantics)

moveSemanticsStack-user.cpp

```
1  #include <string>
2  #include "memberTemplatesStack.hpp"
3
4  int main() {
5    float a[] = {1.1, 2.2, 3.3};
6    Stack<float> fs(a, a+3);
7
8    Stack<int> is = std::move(fs);
9    std::cout << "is: " << is << "fs: " << fs << std::endl;
10
11   fs = std::move(is);
12   std::cout << "is: " << is << "fs: " << fs << std::endl;
13 }
```

OUTPUT:

```
is: 1 2 3 fs:
is: fs: 1 2 3
```

Template Template Parameters (TTPs)

`template <typename T, typename CONT>> class Stack`

```
1  #include <iostream>
2  #include <vector>
3  #include "memberTemplatesStack.hpp"
4
5  int main(void) {
6    Stack<int, std::vector<int>> s1;
7    s1.push(1);
8    s1.push(2);
9    std::cout << "s1: " << s1 << std::endl;
10
11   Stack<float, std::vector<int>> s2;
12   s2.push(1.1);
13   s2.push(2.2);
14   std::cout << "s2: " << s2 << std::endl;
15
16   Stack<int, std::vector<float>> s3;
17   s3.push(1.1);
18   s3.push(2.2);
19   std::cout << "s3: " << s3 << std::endl;
20 }
```

Output:

```
s1: 1 2
s2: 1 2
s3: 1 2
```

- Prefer to write
  `Stack<int, vector>`
  rather than
  `Stack<int, vector<int>>`
- T and the element type in CONT may not be the same

Problem: CONT is a type! – so both T and the element type in CONT are not related - but they should be!

Stack in Slide 3 Changed to Use a TTP

```
1  #ifndef STACK_H  // The blue lines changed
2  #define STACK_H
3
4  #include<deque>
5  template <typename T, template <typename T, typename alloc = std::allocator<T>>
6            class CONT> class Stack;
7
8  template <typename T,
9  template <typename T, typename alloc = std::allocator<T>> class CONT>
10 std::ostream& operator<<(std::ostream &, const Stack<T, CONT> &);
11
12 template <typename T,
13 template <typename T, typename alloc = std::allocator<T>> class CONT = std::deque>
14 class Stack {
15 public:   // interface same as before in Slide 3
16   friend std::ostream& operator<<<T, CONT>(std::ostream &, const Stack<T, CONT> &);
17   void push (const T &item);
18   void pop ();
19   T& top();
20   const T& top() const;
21   bool empty (void) const;
22 private:
23   CONT<T> stack_; TTP: a Template Parameter that is a Template itself
24 };
25
```

Client Code

```
1  #include <vector>
2  #include "ttpStack.hpp"
3
4  int main(void) {
5    Stack<int, std::vector > s1;
6
7    s1.push(1);
8    s1.push(2);
9
10   std::cout << s1 << std::endl;
11
12 }
```

// cannot write stack<int, vector<float>> any more!

Partial Specialisation

- Provides a specialised version for pointer types:

```
1 template <typename T> class Stack<T*> {
2 public:
3   void push(T*);
4   void T* pop();
5   T* top() const;
6   bool empty() const;
7 private:
8   std::vector<T*> stack_;
9 };
```

- The specialised implementation will be used:

  `Stack<int*> s;`

- May have different members but doing so is bad usually

Specialisation

- Provides a specialised version for EuclideanVector:

```
1 template <> class Stack<EuclideanVector> {
2 public:
3   void push(EuclideanVector);
4   void EuclideanVector pop();
5   EuclideanVector top() const;
6   bool empty() const;
7 private:
8   std::vector<EuclideanVector> stack_;
9 };
```

- The specialised implementation will be used:

  `Stack<EuclideanVector> s;`

- May have different members but doing so is bad usually

Specialising Members but Not the Class

- Specialise push to copy the char array (rather than the pointer), for example:

```
1 template<>
2 void Stack<const char*>::push(const char* const & s) {
3   char* item = new char[strlen(s)+1];
4   strncpy(item, s, strlen(s)+1);
5   stack_.push_back(item);
6 }
```

- Header:

```
1 // stack.h
2 template <typename T> class Stack {
3   ...
4 }
5 template <>
6 void Stack<const char*>::push(const char* const & );
7
8 // the definition of push in a separate cpp file
```

- Must also specialise pop, too
- Chapter 16

Updated Example from Thinking in C++: I

```
1  #ifndef SORTABLE_H
2  #define SORTABLE_H
3  #include <string>
4  #include <vector>
5
6  template<typename T>
7  class Sortable : public std::vector<T> {
8  public:  // extend std::vector to have a sort function.
9    void sort();
10 };
11
12 template<typename T>
13 void Sortable<T>::sort() { // A simple sort
14   for(std::size_t i = this->size(); i > 0; --i)
15     for(std::size_t j = 1; j < i; ++j)
16       if(this->at(j-1) > this->at(j)) {
17         T t = this->at(j-1);
18         this->at(j-1) = this->at(j);
19         this->at(j) = t;
20       }
21 }
```

Updated Example from Thinking in C++: II

```
22
23 // Partial specialization for pointers:
24 template<typename T>
25 class Sortable<T*> : public std::vector<T*> {
26 public:
27   void sort();
28 };
29 template<typename T>
30 void Sortable<T*>::sort() {
31   for(std::size_t i = this->size(); i > 0; --i)
32     for(std::size_t j = 1; j < i; ++j)
33       if(*this->at(j-1) > *this->at(j)) {
34         T* t = this->at(j-1);
35         this->at(j-1) = this->at(j);
36         this->at(j) = t;
37       }
38 }
39
40
41
42
43
```

Updated Example from Thinking in C++: III

```
44 // Full specialization for std::string
45 // Sorts by length rather than character by character
46 template<> inline void Sortable<std::string>::sort() {
47   for(std::size_t i = this->size(); i > 0; --i)
48     for(std::size_t j = 1; j < i; ++j)
49       if(this->at(j-1).size() > this->at(j).size()) {
50         std::string t = this->at(j-1);
51         this->at(j-1) = this->at(j);
52         this->at(j) = t;
53       }
54 }
55 #endif // SORTABLE_H ///:~
```

Updated Example from Thinking in C++: I

```
1  // Sortable-user.cpp
2
3  // Testing template specialization.
4  #include <iostream>
5  #include <array>
6  #include "Sortable.h"
7  #include "Urand.h"
8
9  int main() {
10   std::array<std::string, 5> words = { "is", "running", "big", "dog", "a", };
11   std::array<std::string, 3> words2 = { "short", "long", "longer", };
12   Sortable<int> is;
13   Urand<47> rnd;
14   for(std::size_t i = 0; i < 15; ++i)
15     is.push_back(rnd());
16   std::cout << "Random numbers: ";
17   for(std::size_t i = 0; i < is.size(); ++i)
18     std::cout << is[i] << ' ';
19   std::cout << std::endl;
20   is.sort();
21   std::cout << "Sorted numbers: ";
22   for(std::size_t i = 0; i < is.size(); ++i)
23     std::cout << is[i] << ' ';
24   std::cout << std::endl;
25
26   // Uses the template partial specialization:
27   std::cout << "template partial specialization using pointers" << std::endl;
28   Sortable<std::string*> ss;
29   for(std::size_t i = 0; i < words2.size(); ++i)
```

Updated Example from Thinking in C++: II

```
30      ss.push_back(new std::string(words2[i]));
31    for(size_t i = 0; i < ss.size(); ++i)
32      std::cout << *ss[i] << ' ';
33    std::cout << std::endl;
34    ss.sort();
35    for(std::size_t i = 0; i < ss.size(); ++i) {
36      std::cout << *ss[i] << ' ';
37      delete ss[i];
38    }
39    std::cout << std::endl;
40
41    // Uses the full std::string specialization:
42    std::cout << "template partial specialization using std::string" << std::endl;
43    Sortable<std::string> scp;
44    for(std::size_t i = 0; i < words2.size(); ++i)
45      scp.push_back(words2[i]);
46    for(std::size_t i = 0; i < scp.size(); ++i)
47      std::cout << scp[i] << ' ';
48    std::cout << std::endl;
49    scp.sort();
50    for(std::size_t i = 0; i < scp.size(); ++i)
51      std::cout << scp[i] << ' ';
52    std::cout << std::endl;
53  } ///:~
```

Reading

- Chapter 5, Thinking in C++ (Eckel)
- Chapter 15, C++ Templates (Vandevoorde and Josuttis)

Next Class: More Advanced Topics on Templates