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:

```
#include <iostream>
#include <vector>
#include <list>

int main() {
std::vector<int> ivec(10)
std::vector<int> ivec0 = ivec; // ok
std::list<int> ilist0 = ivec; // ok?
                             // error: different containers
std::list<int> ilist(ivec.begin(), ivec.end()); ok?
                                  // ok: ctor exists
ilist.assign(ivec.begin(), ivec.end()); ok?
                                  // ok: member exists
}
```

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

Stack Revised: Member Template Functions

```
#ifndef DEFAULTARGUMENTSTACK_HPP
#define DEFAULTARGUMENTSTACK_HPP

#include <iostream>
#include <deque>

template <typename,typename> class Stack;

template <typename T, typename CONT>
std::ostream& operator<< (std::ostream &os, const Stack<T, CONT> &s);

template <typename T, typename CONT = std::deque<T>> class Stack {
public:
  friend std::ostream& operator<<<T>(std::ostream &, const Stack<T, CONT> &);
  void push(const T &);
  void pop();
  T& top();
  const T& top() const;
  bool empty() const;
private:
  CONT stack_;
};
```

Client Code

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

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

- What if two stacks have different element types?

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

Supporting Member Templates

Add to defaultArgumentsStack.hpp:

```
template <typename T, typename CONT = std::deque<T>> class Stack {
  // addition declarations:
  Stack() {} // must define default constructor

  // template function inside template class.
  template <typename T2, typename CONT2>
    Stack(const Stack<T2, CONT2>&);

  template <typename T2, typename CONT2>
    Stack(Stack<T2, CONT2>&&);

  template <typename Iter> Stack(Iter b, Iter e);

  template <typename T2, typename CONT2>
    Stack& operator=(const Stack<T2,CONT2> &);

  template <typename T2, typename CONT2>
    Stack& operator=(Stack<T2,CONT2> &&);

  template <typename Iter> void assign(Iter b, Iter e);
}
```

stack.h Expanded to Support Member Templates I

```
template <typename T, typename CONT>
template <typename T2, typename CONT2>
Stack<T,CONT>::Stack(const Stack<T2,CONT2> & s) {
  Stack<T2,CONT2> tmp(s);
  while (!tmp.empty()) {
    stack_.push_front(tmp.top());
    tmp.pop();
  }
}

template <typename T, typename CONT>
template <typename T2, typename CONT2>
Stack<T,CONT>::Stack(Stack<T2,CONT2> && s) {
  while (!s.empty()) {
    stack_.push_front(s.top());
    s.pop();
  }
}

template <typename T, typename CONT>
template <typename Iter>
Stack<T,CONT>::Stack(Iter b, Iter e) {
  for (; b != e; ++b)
    stack_.push_back(*b);
}
```

stack.h Expanded to Support Member Templates II

```
template <typename T, typename CONT>
template <typename T2, typename CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator=(const Stack<T2,CONT2> & s) {
  if ((void *)this == (void *)&s)
    return *this;

  Stack<T2,CONT2> tmp(s);
  stack_.clear();
  while (!tmp.empty()) {
    stack_.push_front(tmp.top());
    tmp.pop();
  }
  return *this;
}

template <typename T, typename CONT>
template <typename T2, typename CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator=(Stack<T2,CONT2> && s) {
  if ((void *)this == (void *)&s)
    return *this;
  stack_.clear();
  while (!s.empty()) {
    stack_.push_front(s.top());
    s.pop();
  }
  return *this;
}
```

stack.h Expanded to Support Member Templates III

```

template <typename T, typename CONT>
template <typename Iter>
void Stack<T,CONT>::assign(Iter b, Iter e) {
  stack_.clear();
  for (; b != e; ++b)
    stack_.push_back(*b);
}
```

- 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

```
#include "memberTemplatesStack.hpp"

int main() {
  float a[] = {1.1, 2.2, 3.3};
  Stack<float> fs(a, a+3);
  // instantiate Stack<float, deque<float>>(float*, float*)

  fs.assign(a, a+3);
  // Stack<float, deque<float>>::assign(float *, float*);
  std::cout << fs << std::endl;

  Stack<int> is = fs;
  // Stack<int, deque<int>>(const Stack<float, deque<float>> &)
  std::cout << is << fs << std::endl;

  is = fs;
  // Stack<int, deque<int>>::operator=(const Stack<float, deque<float>> &)
  std::cout << is << fs << std::endl;
}
```

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

```
#include <string>
#include "memberTemplatesStack.hpp"

int main() {
  float a[] = {1.1, 2.2, 3.3};
  Stack<float> fs(a, a+3);

  Stack<int> is = std::move(fs);
  std::cout << "is: " << is << "fs: " << fs << std::endl;

  fs = std::move(is);
  std::cout << "is: " << is << "fs: " << fs << std::endl;
}
```

OUTPUT:

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

Template Template Parameters (TTPs)

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

```
#include <iostream>
#include <vector>
#include "memberTemplatesStack.hpp"

int main(void) {
  Stack<int, std::vector<int>> s1;
  s1.push(1);
  s1.push(2);
  std::cout << "s1: " << s1 << std::endl;

  Stack<float, std::vector<int>> s2;
  s2.push(1.1);
  s2.push(2.2);
  std::cout << "s2: " << s2 << std::endl;

  Stack<int, std::vector<float>> s3;
  s3.push(1.1);
  s3.push(2.2);
  std::cout << "s3: " << s3 << std::endl;
}
```

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

```
#ifndef STACK_H  // The blue lines changed
#define STACK_H

#include<deque>
template <typename T, template <typename T, typename alloc = std::allocator<T>>
          class CONT> class Stack;

template <typename T,
template <typename T, typename alloc = std::allocator<T>> class CONT>
std::ostream& operator<<(std::ostream &, const Stack<T, CONT> &);

template <typename T,
template <typename T, typename alloc = std::allocator<T>> class CONT = std::deque>
class Stack {
public:   // interface same as before in Slide 3
  friend std::ostream& operator<<<T, CONT>(std::ostream &, const Stack<T, CONT> &);
  void push (const T &item);
  void pop ();
  T& top();
  const T& top() const;
  bool empty (void) const;
private:
  CONT<T> stack_; TTP: a Template Parameter that is a Template itself
};

```

Client Code

```
#include <vector>
#include "ttpStack.hpp"

int main(void) {
  Stack<int, std::vector > s1;

  s1.push(1);
  s1.push(2);

  std::cout << s1 << std::endl;

}
```

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

Partial Specialisation

- Provides a specialised version for pointer types:

```
template <typename T> class Stack<T*> {
public:
  void push(T*);
  void T* pop();
  T* top() const;
  bool empty() const;
private:
  std::vector<T*> stack_;
};
```

- 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:

```
template <> class Stack<EuclideanVector> {
public:
  void push(EuclideanVector);
  void EuclideanVector pop();
  EuclideanVector top() const;
  bool empty() const;
private:
  std::vector<EuclideanVector> stack_;
};
```

- 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:

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

- Header:

```
// stack.h
template <typename T> class Stack {
  ...
}
template <>
void Stack<const char*>::push(const char* const & );

// the definition of push in a separate cpp file
```

- Must also specialise pop, too
- Chapter 16

Updated Example from Thinking in C++: I

```
#ifndef SORTABLE_H
#define SORTABLE_H
#include <string>
#include <vector>

template<typename T>
class Sortable : public std::vector<T> {
public:  // extend std::vector to have a sort function.
  void sort();
};

template<typename T>
void Sortable<T>::sort() { // A simple sort
  for(std::size_t i = this->size(); i > 0; --i)
    for(std::size_t j = 1; j < i; ++j)
      if(this->at(j-1) > this->at(j)) {
        T t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
```

Updated Example from Thinking in C++: II

```

// Partial specialization for pointers:
template<typename T>
class Sortable<T*> : public std::vector<T*> {
public:
  void sort();
};
template<typename T>
void Sortable<T*>::sort() {
  for(std::size_t i = this->size(); i > 0; --i)
    for(std::size_t j = 1; j < i; ++j)
      if(*this->at(j-1) > *this->at(j)) {
        T* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}





```

Updated Example from Thinking in C++: III

```
// Full specialization for std::string
// Sorts by length rather than character by character
template<> inline void Sortable<std::string>::sort() {
  for(std::size_t i = this->size(); i > 0; --i)
    for(std::size_t j = 1; j < i; ++j)
      if(this->at(j-1).size() > this->at(j).size()) {
        std::string t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
#endif // SORTABLE_H ///:~
```

Updated Example from Thinking in C++: I

```
// Sortable-user.cpp

// Testing template specialization.
#include <iostream>
#include <array>
#include "Sortable.h"
#include "Urand.h"

int main() {
  std::array<std::string, 5> words = { "is", "running", "big", "dog", "a", };
  std::array<std::string, 3> words2 = { "short", "long", "longer", };
  Sortable<int> is;
  Urand<47> rnd;
  for(std::size_t i = 0; i < 15; ++i)
    is.push_back(rnd());
  std::cout << "Random numbers: ";
  for(std::size_t i = 0; i < is.size(); ++i)
    std::cout << is[i] << ' ';
  std::cout << std::endl;
  is.sort();
  std::cout << "Sorted numbers: ";
  for(std::size_t i = 0; i < is.size(); ++i)
    std::cout << is[i] << ' ';
  std::cout << std::endl;

  // Uses the template partial specialization:
  std::cout << "template partial specialization using pointers" << std::endl;
  Sortable<std::string*> ss;
  for(std::size_t i = 0; i < words2.size(); ++i)
```

Updated Example from Thinking in C++: II

```
    ss.push_back(new std::string(words2[i]));
  for(size_t i = 0; i < ss.size(); ++i)
    std::cout << *ss[i] << ' ';
  std::cout << std::endl;
  ss.sort();
  for(std::size_t i = 0; i < ss.size(); ++i) {
    std::cout << *ss[i] << ' ';
    delete ss[i];
  }
  std::cout << std::endl;

  // Uses the full std::string specialization:
  std::cout << "template partial specialization using std::string" << std::endl;
  Sortable<std::string> scp;
  for(std::size_t i = 0; i < words2.size(); ++i)
    scp.push_back(words2[i]);
  for(std::size_t i = 0; i < scp.size(); ++i)
    std::cout << scp[i] << ' ';
  std::cout << std::endl;
  scp.sort();
  for(std::size_t i = 0; i < scp.size(); ++i)
    std::cout << scp[i] << ' ';
  std::cout << std::endl;
} ///:~
```

Reading

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

Next Class: More Advanced Topics on Templates