COMP6771
Advanced C++ Programming

Week 6
Part Three: Class Templates

2016

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

Why Class Templates?

- In C++, class names cannot be overloaded. Thus:

```
class IntStack {
public:
  void push(int&);
  void pop();
  int& top();
  const int& top() const;
  ...
private:
  vector<int> stack_;
};
```

```
class DoubleStack {
public:
  void push(double&);
  void pop();
  double& top();
  const double& top() const;
  ...
private:
  vector<double> stack_;
};
```

- Two drawbacks:
  - The lexical complexity: IntStack, DoubleStack, ...
  - Administrative nightmare

What Are Class Templates? – stack.h

- The interface Stack.h:

```
#ifndef STACK_H
#define STACK_H
#include <iostream>
#include <vector>
template <typename> class Stack;
template <typename T> std::ostream& operator<< (std::ostream &os, const Stack<T> &s);
template <typename T> class Stack {
public:
friend std::ostream& operator<<<T>(std::ostream &, const Stack<T> &);
  void push(const T &);
  void pop();
  T& top();
  const T& top() const;
  bool empty() const;
private:
  std::vector<T> stack_;
};
#endif
```

- A prescription for compiler to generate class types

The Stack Implementation: Stack.h

```
template <typename T>
void Stack<T>::push(const T &item) {
  stack_.push_back(item);
}

template <typename T>
void Stack<T>::pop() {
  stack_.pop_back();
}

template <typename T>
T& Stack<T>::top() {
  return stack_.back();
}

template <typename T>
const T& Stack<T>::top() const {
  return stack_.back();
}

template <typename T>
bool Stack<T>::empty() const {
  return stack_.empty();
}
```

The Stack implementation: Stack.h

```
template <typename T>
std::ostream& operator<<(std::ostream &os, const Stack<T> &s) {
  for (unsigned int i = 0; i < s.stack_.size(); i++) {
    os << s.stack_[i] << " ";
  }
  return os;
}
```

NB

A friend to a class is part of the class's interface!

Client Code: stack-user.cpp

```
#include <iostream>

#include "stack.hpp"

int main() {
  Stack<int> s1; // int: template argument
  s1.push(1);
  s1.push(2);
  Stack<int> s2 = s1;
  std::cout << s1 << s2 << std::endl;
  s1.pop();
  s1.push(3);
  std::cout << s1 << s2 << std::endl;
}
```

- Use the compiler-generated default ctor
  `stack<int> s; // create an empty stack`
- Use the compiler-generated copy-control members:
  OK: because C++ STL containers use value semantics
- Use the compiler-generated dtor (which does nothing)
  OK: vector's dtor will take care of stack_'s deallocation

The Default C'tor and Big Five

```
template <typename T>
Stack<T>::Stack() { }

template <typename T>
Stack<T>::Stack(const Stack<T> &s) : stack_{s.stack_} { }

template <typename T>
Stack<T>::Stack(Stack<T> &&s) : stack_(std::move(s.stack_)); { }

template <typename T>
Stack<T>& Stack<T>::operator=(const Stack<T> &s) {
  stack_ = s.stack_;
}

template <typename T>
Stack<T>& Stack<T>::operator=(Stack<T> &&s) {
  stack_ = std::move(s.stack_);
}

template <typename T>
Stack<T>::~Stack() { }
```

Inclusion Compilation Model

```
// stack.h:
template <typename T> class Stack { ... };
```

```
// user-file1.h:
#include "stack.h"
Stack<int> s1;
Stack<double> s2;
```

```
// user-file2.h:
#include "stack.h"
Stack<int> s3;
Stack<char> s4;
```

- Program must behave as if only one `Stack<int>` were instantiated even when it is instantiated multiple times
- Drawbacks:
    - Implementation details available in Stack.h
    - Compiling the same template many times can be slow

Class Template Instantiation

```
Stack<int> s1; // (1) stack<int> instantiated

Stack<char> *ps;
ps->push('a'); // (2) stack<char> instantiated
```

- Instantiation: the generation of a class from a template
- On-demand instantiation: a template is instantiated when an object is defined with a type that is a class template
- (1) and (2): point of instantiation
- Lazy instantiation: only member functions used are instantiated (e.g, empty() not instantiated if never called)

Name Resolution

- Same two steps as for function templates
- The member function push modified to:

```
template <typename T>
void Stack<T>::push(const T &item) {
    std::cout << "value pushed: "; (1)
    std::cout << item (2)
    stack_.push_back(item);
}
```

- Type-independent names are resolved when the template is defined, e.g., operator<< for strings at (1)
- Type-dependent names are resolved when the template is instantiated, e.g., operator<< for item at (2)

Name Resolution

- SmallInt

```
class SmallInt {
  friend ostream&
  operator<<(ostream &os, const SmallInt &s);
public:
  SmallInt(int v) : value_(v) { }
private:
  int value_;
};
```

<< for 2 found

- Client code:

```
#include "stack.h"
#include "SmallInt.h"
Stack<SmallInt> s; // point of instantiation
std::cout << s;
```

No Class Template Argument Deduction

Stack s; // which instantiation?

Conversions for Member Functions

```
Stack<int> s; // instantiates Stack<int>
short s = 10;
int i = 10;
s.push(s); // instantiates Stack<int>::push(const int&)
s.push(i); // uses Stack<int>::push(const int&)
```

- Unlike function templates, normal conversions are allowed on arguments to function parameters that were defined using the template parameters of the class template
- However, member templates are still function templates

Default Arguments

- Can supply default arguments for template types
- For instance, if we wanted the user to be able to change the underlying container (defaultArgumentStack.hpp):

```
#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_;
};
```

Default Arguments

- Client code:

```
#include<vector>

#include "defaultArgumentStack.hpp"

int main() {
  Stack<int> s1;
  Stack<int, std::deque<int>> s2 = s1;
  Stack<int, std::vector<int>> s3;

  s1.push(1);
  s1.push(2);
  s2 = s1;
//  s3 = s1; // OK? - one is a deque and the other is a vector
  std::cout << s1 << s2 << std::endl;
  s1.pop();
  s1.push(3);
  std::cout << s1 << s2 << std::endl;
}
```

- Will look at template template parameters in the next lecture

Static Members

- Each instantiation has its own set of static members
- Add a static data member to count the no. of stacks created:

```
// staticStack.hpp:
static int numStacks_; // declaration inside class declaration

template <typename T, typename CONT>
int Stack<T,CONT>::numStacks_ = 0; // definition

template <typename T, typename CONT> inline
Stack<T, CONT>::Stack() { numStacks_++; }

template <typename T, typename CONT> inline
Stack<T, CONT>::~Stack() {
  if (numStacks_)
    std::cout << typeid(this).name() << ": " << numStacks_
<< std::endl;
  numStacks_ = 0;
}
```

Static Members

- Client code:

```
#include <vector>

#include "staticStack.hpp"

int main() {
  Stack<float> fs;
  Stack<int> is1, is2, is3;
  Stack<int, std::vector<int>> is4;
}
```

- At the closing }, the destructors are called:

```
P5StackIiSt6vectorIiSaIiEEE: 1
P5StackIiSt5dequeIiSaIiEEE: 3
P5StackIfSt5dequeIfSaIfEEE: 1
```

Friend Declarations in Class Templates

Three kinds of friend declarations:

- Ordinary friend functions/classes
- Friend template instantiations
- Friend templates

Ordinary Friends

```
class Foo {
  void bar();
};

template <typename T> class Stack {
  friend class foobar;
  friend void foo();
  friend void Foo::bar();
}
```

Friend Templates

- Each Manager instantiation has all instantiations of a friend
- The mapping is one-to-many (unbounded)

```
template <typename T>
class Manager {
  template <typename U> friend class Task;
  template <typename U>
     friend class Schedule<U>::dispatch(Task<U>*);
  ...
};
```

- No need to give the corresponding template declarations since the friend declarations are treated as template declarations
- The scope of a friend declaration is exposed to the scope surrounding the class (Chapter 16)

Friend Template Instantiations

- Each Stack instantiation has one unique instantiation of each friend
- The mapping is one-to-one (bounded)

```
template <typename T> class foobar { ... }
template <typename T> void foo(Stack<T> &);
template <typename T> class Foo { void bar(); }

template <typename T> class Stack {
  friend class foobar<T>;
  friend void foo<T>(Stack<T>);
  friend void Foo<T>::bar();
}
```

- operator<< in Stack is another example

The typename Keyword for nested types

```
template <typename T> class X {
  typename T::id *i; // error without typename
public:
  void f() { i.g(); }
};
class Y {
public:
  class id {
    ...
  };
};
int main() {
  X<Y> xy;
}
```

- Use typename for a nested type
- By default, T::id is assumed to be a non-type

Summary

- Templates are used extensively in C++ STL
- They are relatively straightforward to understand
- But the syntax can be overwhelming initially
- The error messages can be overwhelming
- C++ programmers need to be proficient in templates

Reading

- Chapter 16, C++ Primer:
- Chapter 5, Bruce Eckel, Thinking in C++, Vol 2
- C++ FAQ Lite: templates: http://yosefk.com/c++fqa/templates.html

Next Lecture: Templates