600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Templates

Templates

- Templates are the C++ mechanism for doing *generic* programming
  - We can write code that works on many different data types, without manually overloading the code for each type we wish to support

- We've been using templates in the STL, the Standard Template Library
  - e.g., `vector` can hold `int`s, or `double`s, or `Course`s, or `Piece`s, etc.

- We can write our own template functions and template classes

Writing Template Functions

- Need to say that the function is templated, and what template argument(s) it takes:
  - template< typename T > or template < typename T , typename U >
  - Within the scope of the template statement, it's as if T had been typedef-ed to be the same as whatever the parameter was

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

template< typename T >
T add( T t1 , T t2 ){ return t1+t2; }

int main( void )
{
        cout << add( 1 , 2 ) << endl;
        cout << add( 2. , 3. ) << endl;
        cout << add( string( "dog" ) , string( "house" ) ) << endl;
        return 0;
}
```

```
>> ./a.out
3
5
doghouse
>>
```

Writing Template Functions

- Need to say that t
  argument(s) it tak
  - template< type
  - Within the scope of the template statement, it's as if T had been typedef-ed to be the same as whatever the parameter was

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function int main() :
main.cpp:11:23: error: no matching function for call to add(int, double)
   cout << add( 1 , 2. ) << endl;
                        ^
>>
```

main.cpp

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

template< typename T >
T add( T t1 , T t2 ){ return t1+t2; }

int main( void )
{
      cout << add( 1 , 2. ) << endl;
      return 0;
}
```

Writing Template Functions

- Templates don't *actually* generate code that runs on multiple types
  (The number of types would be limitless)

- Instead, each call with a different template parameter type causes the compiler to generate a new version on the fly
  - ⇒ Code is smaller even though the size of the compiled code is not
  - ⇒ The definition of the templated function has to be available at compile time
    - ⇒ Your `.hpp/.h` file needs to include the entire templated function/class
  - ⇒ The compiler may not catch bugs in template code if the code isn't called

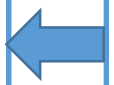
Writing Template Functions

main.cpp

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

int       add( int t1 , int t2 )           { return t1+t2; }
char      add( char t1 , char t2 )         { return t1+t2; }
double    add( double t1 , double t2 )     { return t1+t2; }
float     add( float t1 , float t2 )       { return t1+t2; }
string    add( string t1 , string t2 )     { return t1+t2; }

int main( void )
{
      cout << add( 1 , 2 ) << endl;
      cout << add( 'a' , 'b' ) << endl;
      cout << add( 2. , 3. ) << endl;
      cout << add( 4.f , 5.f ) << endl;
      cout << add( string( "dog" ) , string( "house" ) ) << endl;
      return 0;
}
```



main.cpp

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

template< typename T >
T add( T t1 , T t2 ){ return t1+t2; }

int main( void )
{
      cout << add( 1 , 2 ) << endl;
      cout << add( 'a' , 'b' ) << endl;
      cout << add( 2. , 3. ) << endl;
      cout << add( 4.f , 5.f ) << endl;
      cout << add( string( "dog" ) , string( "house" ) ) << endl;
      return 0;
}
```

Writing Template Functions

- The template parameter does not have to be used for the arguments
  - In this case the compiler can't know which version you want

  ⇒ You have to specify explicitly

```
>> ./a.out
94775
524272
524272
>>
```

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

template< typename T >
T average( const vector< int >& v )
{
	T sum = 0;
	for( size_t i=0 ; i<v.size() ; i++ ) sum += v[i];
	return sum/v.size();
}

int main( void )
{
	vector< int > v( 100000 );
	for( size_t i=0 ; i<v.size() ; i++ ) v[i] = rand();
	cout << average< int >( v ) << endl;
	cout << average< double >( v ) << endl;
	cout << average< long long >( v ) << endl;
	return 0;
}
```

Template Classes

- We used templated classes in the STL
- We can also write our own

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
	ListNode( T val , ListNode< T >* next );
	std::string toString( void ) const;
private:
	T _val;
	ListNode< T >* _next;
};
#include "templatedList.inl"
```

Template Classes

- We used templated classes in the STL
- We can also write our own

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
	ListNode( T val , ListNode< T >* next );
	std::string toString( void ) const;
private:
	T _val;
	ListNode< T >* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode< T >* next ) :
	_val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
	 std::stringstream ss;
	ss << _val;
	if( _next ) ss << " " << _next->toString();
	return ss.str();
};
```

Note:
This will work as long `operator << ( ostream& , T )` has been defined

Template Classes

main.cpp

```
#include "templatedList.h"
using namespace std;

int main( void )
{
	ListNode< int > l3( 3, nullptr );
	ListNode< int > l2( 2, &l3 );
	ListNode< int > l1( 1, &l2 );
	cout << l1.toString() << endl;

	ListNode< string > s3( "three" , nullptr );
	ListNode< string > s2( "two" , &s3 );
	ListNode< string > s1( "one" , &s2 );
	cout << s1.toString() << endl;

	return 0;
}
```

```
>> ./a.out
1 2 3
one two three
>>
```

TL

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
	ListNode( T val , ListNode< T >* next );
	std::string toString( void ) const;
private:
	T _val;
	ListNode< T >* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode< T >* next ) :
	_val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
	 std::stringstream ss;
	ss << _val;
	if( _next ) ss << " " << _next->toString();
	return ss.str();
};
```

Template Classes

- We used templated classes in the STL
- We can also write our own

Note:

- Need a template statement before the definition of each method

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
	ListNode( T val , ListNode< T >* next );
	std::string toString( void ) const;
private:
	T _val;
	ListNode< T >* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode< T >* next ) :
	_val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
	 std::stringstream ss;
	ss << _val;
	if( _next ) ss << " " << _next->toString();
	return ss.str();
};
```

Template Classes

- We used templated classes in the STL
- We can also write our own

Note:

- Need a template statement before the definition of each method
- Need to specify the template param. when using the class name

  (Except constructors and destructors)

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
        ListNode( T val , ListNode< T >* next );
        std::string toString( void ) const;
private:
        T _val;
        ListNode< T >* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode< T >* next ) :
        _val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
         std::stringstream ss;
        ss << _val;
        if( _next ) ss << " " << _next->toString();
        return ss.str();
};
```

Template Classes

main.cpp

```
#include "templatedList.h"
using namespace std;

int main( void )
{
	ListNode< int > l3( 3, nullptr );
	ListNode< int > l2( 2, &l3 );
	ListNode< int > l1( 1, &l2 );
	cout << l1.toString() << endl;

	ListNode< string > s3( "three" , nullptr );
	ListNode< string > s2( "two" , &s3 );
	ListNode< string > s1( "one" , &s2 );
	cout << s1.toString() << endl;

	return 0;
}
```

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
	ListNode( T val , ListNode< T >* next );
	std::string toString( void ) const;
private:
	T _val;
	ListNode< T >* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode< T >* next ) :
	_val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
	 std::stringstream ss;
	ss << _val;
	if( _next ) ss << " " << _next->toString();
	return ss.str();
};
```

Template Classes

- We used templated classes in the STL
- We can also write our own

Note:

- Need a template statement before the definition of each method
- Need to specify the template param. when using the class name
- Within the class definition, don't need to specify the template parameter

templatedList.h

```
#include <iostream>
#include <string>
#include <sstream>

template< typename T >
class ListNode
{
public:
        ListNode( T val , ListNode* next );
        std::string toString( void ) const;
private:
        T _val;
        ListNode* _next;
```

templatedList.inl

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode* next ) :
        _val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
         std::stringstream ss;
        ss << _val;
        if( _next ) ss << " " << _next->toString();
        return ss.str();
};
```

Template Classes

- Compilation is on-demand

main.cpp

```
#include "templatedList.h"
using namespace std;

struct Foo{};

int main( void )
{

        ListNode< Foo > f2( Foo() , nullptr );
        ListNode< Foo > f1( Foo(), &f1 );
        cout << f1.size() << endl;

}
```

```
>> ./a.out
2
>>
```

templatedList.h

```
ream>
ng>

ename T >
e


( T val , ListNode* next );
ng toString( void ) const;
ze( void ) const;


* _next;


plateдList.inl"
```

templatedList.inl

```
#include <sstream>

template< typename T >
ListNode< T >::ListNode( T val , ListNode* next ) :
        _val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
         std::stringstream ss;
        ss << _val;
        if( _next ) ss << " " << _next->toString();
        return ss.str();
}

template< typename T >
size_t ListNode< T >::size( void ) const
{
        if( _next ) return 1 + _next->size();
        else return 1;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In instantiation of std::__cxx11::string ListNode<T>::toString() const [with T = Foo; std::__cxx11::string = std::__cxx11::basic_string<char>]:
main.cpp:10:23:   required from here
main.cpp:10:6: error: no match for operator<< (operand types are std::stringstream {aka std::__cxx11::basic_stringstream<char>} and const Foo)
   ss << _val;
   ~~~^~~~~~~
...
>>
```

main.cpp

```
#include "templatedList.h"
using namespace std;

struct Foo{};

int main( void )
{

      ListNode< Foo > f2( Foo() , nullptr );
      ListNode< Foo > f1( Foo(), &f1 );
      cout << f1.toString() << endl;

      return 0;
}
```

templatedList.h

```
ream>
ng>

ename T >
e


( T val , ListNode* next );
ng toString( void ) const;
ze( void ) const;


* _next;


plateList.inl"
```

```
template< typename T >
ListNode< T >::ListNode( T val , ListNode* next ) :
      _val( val ) , _next( next) { }

template< typename T >
std::string ListNode< T >::toString( void ) const
{
       std::stringstream ss;
      ss << _val;
      if( _next ) ss << " " << _next->toString();
      return ss.str();
}

template< typename T >
size_t ListNode< T >::size( void ) const
{
      if( _next ) return 1 + _next->size();
      else return 1;
}
```

Template Classes

- (Non-template) classes can have templated methods
  - The non-template methods should be defined in the `.cpp` file
  - The template methods should be part of the header

foo.h

```
#include <iostream>

class Foo
{
public:
	Foo( void );

	template< typename T > void bar( T t );
};
#include "foo.inl"
```

foo.inl

```
template< typename T >
void Foo::bar( T t ){ std::cout << t; }
```

foo.cpp

```
Foo::Foo( void ){ std::cout << "construcing foo" << std::endl; }
```

Piazza → Resources section → Resources tab → Exercise 13-1