Intermediate Programming

Day 31

Outline

- Exercise 30
- Template functions
- Template classes
- What goes where?
- Review questions

Exercise 30

Implement the **<<** operator for the **int_set** class.

int_set.cpp

```
...
std::ostream &operator << ( std::ostream &os , const int_set &s )
{
	os << "{";
	for( const int_node *node=s.head ; node ; node=node->get_next() )
	{
		os << node->get_data();
		if( node->get_next() ) os << ", ";
	}
	os << "}";
	return os;
}
```

Exercise 30

Implement the **+=** operator for the **int_set** class.

int_set.cpp

```
...
int_set &int_set::operator += ( int new_value )
 {
     add( new_value);
     return *this;    //for consistency - assignment ops return the value assigned
}
```

Exercise 30

Implement the copy constructor for the **int_set** class.

int_set.cpp

```
...
int_set::int_set( const int_set &other )
{
	head = nullptr;
	for( const int_node *node= other.head ; node ; node=node->get_next() )
		add( node->get_data() );
}
```

Exercise 30

Implement the assignment operator for the **int_set** class.

int_set.cpp

```
...
int_set::int_set( const int_set &other )
{
	head = nullptr;
	for( const int_node *node= other.head ; node ; node=node->get_next() )
		add( node->get_data() );
}
int_set &int_set::operator = ( const int_set &other )
{
	clear();
	for( const int_node *node=other.head ; node ; node=node->get_next() )
		add( node->get_data() );
	return *this;
}
```

Exercise 30

Implement the assignment operator for the **int_set** class.

int_set.cpp

```
...
int_set::int_set( const int_set &orig ) : head( nullptr )
{
	copy( orig );
}
int_set &int_set::operator = ( const int_set &other )
{
	clear() ; copy( other ) ; return *this;
}
void int_set::copy( const int_set &other )
{
	for( const int_node *node=orig.head ; node ; node=node->get_next() )
		add( node->get_data() );
}
```

Exercise 30

Implement the destructor for the **int_set** class.

int_set.cpp

```
...
int_set::~int_set( void )
{
	clear();
}
```

Outline

- Exercise 30
- Template functions
- Template classes
- What goes where?
- Review questions

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 `Complex`s, etc.
- We can write our own template functions and template classes

Template Functions

- Use **template< typename T , … >** to indicate that the function is templated, and what template argument(s) it takes.

main.cpp

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

template< typename T >
T add( T t1 , T t2 )
{
	T t = t1+t2;
	return t;
}
int main( void )
{
	cout << add< double >( 5. , add< double >( 2. , 3. ) ) << endl;
	cout << add< string >( string( "a" ) , string( "b" ) ) << endl;
	return 0;
}
```

Template Functions

- Use **template< typename T , … >** to indicate that the function is templated, and what template argument(s) it takes.
- In the scope of the template statement, it's as if T had been **typedef**-ed to be the same as whatever the parameter was

main

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

template< typename T >
T add( T t1 , T t2 )
{
	T t = t1+t2;
	return t;
}
int main( void )
{
	cout << add< double >( 5. , add< double >( 2. , 3. ) ) << endl;
	cout << add< string >( string( "a" ) , string( "b" ) ) << endl;
	return 0;
}
```

```
>> ./a.out
10
ab
>>
```

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 **instantiate** a new version on the fly

  ⇒ Source 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 `.h` file needs to include the definition of the templated function/class

  ⇒ The compiler may not catch bugs in template code if the code isn't called

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< int >( 1 , 2 ) << endl;
	cout << add< char >( 'a' , 'b' ) << endl;
	cout << add< double >( 2. , 3. ) << endl;
	cout << add< float >( 4.f , 5.f ) << endl;
	cout << add< string >( string( "a" ) , string( "b" ) ) << 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< int >( 1 , 2 ) << endl;
	cout << add< char >( 'a' , 'b' ) << endl;
	cout << add< double >( 2. , 3. ) << endl;
	cout << add< float >( 4.f , 5.f ) << endl;
	cout << add< string >( string( "a" ) , string( "b" ) ) << endl;
	return 0;
}
```

Template Functions

- The compiler may be able to determine which instance of the function you are using based on the argument.

main.cpp

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

template< typename T >
T add( T t1 , T t2 )
{
	T t = t1+t2;
	return t;
}
int main( void )
{
	cout << add( 2. , 3. ) << endl;
	cout << add( string( "a" ) , string( "b" ) ) << endl;
	return 0;
}
```

Template Functions

- The compiler may be able to determine which instance of the function you are using based on the argument.
- If it cannot (e.g. the function does not take that type of argument) you need to specify the template type explicitly.

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

```
#include <iostrea
#include <vector>
using namespace s

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 Functions

- The compiler may be able to determine which instance of the function you are using based on the argument.
- If it cannot (e.g. the function does not take that type of argument) you need to specify the template type explicitly.

main.cpp

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

template< typename T >
T add( T t1 , T t2 )
{
	T t = t1+t2;
	return t;
}
int main( void )
{
	cout << add( 2 , 3. ) << endl;
	return 0;
}
```

Note:
If you do not explicitly specify the template parameters the arguments must unambiguously define the parameter type.

```
>> g++ main.cpp ...
main.cpp: In function 'int main()':
main.cpp:12:22: error: no matching function for call to 'add(int, double)'
   12 |  cout << add( 2 , 3. ) << endl;
      |                      ^
main.cpp:5:25: note: candidate: 'template<class T> T add(T, T)'
    5 | template< typename T >T add( T t1 , T t2 )
      |                         ^~~
main.cpp:5:25: note:   template argument deduction/substitution failed:
main.cpp:12:22: note:   deduced conflicting types for parameter 'T' ('int' and 'double')
   12 |  cout << add( 2 , 3. ) << endl;
      |                      ^
>>
```

take that type of argument) you need to specify the template type explicitly.

```
    return t;
}
int main( void )
{
    cout << add( 2 , 3. ) << endl;
    return 0;
}
```

Note:
If you do not explicitly specify the template parameters the arguments must unambiguously define the parameter type.

Outline

- Exercise 30
- Template functions
- Template classes
- What goes where?
- Review questions

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;
	size_t size( void ) const;
private:
	ListNode< T > *_next;
	T _val;
};
#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;
	size_t size( void ) const;
private:
	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

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;
        size_t size( void ) const;
private:
        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:
toString will work as long operator << (ostream& , T) is 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;
	size_t size( void ) const;
private:
	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

<u>Note</u>:

When (separately) **defining** methods of a templated class:

- 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;
	size_t size( void ) const;
private:
	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:

When (separately) **defining** methods of a templated class:

  - Need a template statement before the definition of each method
  - Need to specify the template parameter 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;
	size_t size( void ) const;
private:
	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:

When (separately) **defining** methods of a templated class:

- Need a template statement before the definition of each method
- Need to specify the template parameter 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;
	size_t size( void ) const;
private:
	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

Note:
Even though there is no stream insertion operator for the class Foo, the code compiles and runs just fine
- As long as we don't call the toString member function.

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

struct Foo{};

int main( void )
{
        ListNode< Foo > f2( Foo() , nullptr );
        ListNode< Foo > f1( Foo(), &f2 );
        cout << f1.size() << endl;
        return 0;
}
```

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

templatedList.h

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

template< typename T >
class ListNode
```

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

Template Functions

In C++ can have non-templated functions:

- Stand-alone functions
- Non-templated member functions of non-templated classes

Or templated functions:

- Stand-alone functions
- Member functions of templated classes
- Templated member functions of non-templated classes

Where should the declarations/definitions go?

Outline

- Exercise 30
- Template functions
- Template classes
- What goes where?
- Review questions

What goes where?

- For non-templated functionality, the compiler only needs to know the **declarations** and the linker can later connect things together.
- In contrast, when instantiating templated functionality, the compiler needs to know the **definitions** of the templated functionality.

⇒ We need to include the definition of the templated functionality in whatever source files use them.

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 << std::endl; }
```

foo.cpp

```
#include "foo.h"
Foo::Foo( void ){ std::cout << "constructing foo" << std::endl; }
```

main.cpp

```
#include <iostream>
#include "foo.h"
int main( void )
{
	Foo foo;
	foo.bar( 5 );
	foo.bar( "hello world" );
}
```

What goes where?

- For non-templated functionality, the compiler only needs to know the **declarations** and the linker can later connect things together.
- In contrast, when instantiating templated functionality, the compiler needs to know the **definitions** of the templated

```
>> g++ -std=c++11 -Wall -Wextra main.cpp foo.cpp
>> ./a.out
constructing foo
5
hello world
>>
```

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 << std::endl; }
```

foo.cpp

```
#include "foo.h"
Foo::Foo( void ){ std::cout << "constructing foo" << std::endl; }
```

main.cpp

```
#include <iostream>
#include "foo.h"
int main( void )
{
	Foo foo;
	foo.bar( 5 );
	foo.bar( "hello world" );
}
```

What goes where?

- Typically, the definitions of the templated functions get put into files with extensions:
  - `.inc`: “include” files, or
  - `.inl`: “inline” files

  or are kept in the header file (if they are short)

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 << std::endl; }
```

foo.cpp

```
#include "foo.h"
Foo::Foo( void ){ std::cout << "constructing foo" << std::endl; }
```

main.cpp

```
#include <iostream>
#include "foo.h"
int main( void )
{
	Foo foo;
	foo.bar( 5 );
	foo.bar( “hello world” );
}
```

Outline

- Exercise 30
- Template functions
- Template classes
- What goes where?
- Review questions

Review questions

1. How do we declare a template class?

```
template< typename T > class MyClass{ ... };
```

Review questions

2. When would you consider making a function templated?

When you find yourself writing functions with essentially the same body, but different types

Review questions

3. What is template instantiation?

Generation of a concrete class or function for a particular combination of template arguments.

Review questions

4. Can we separate declaration and definition when using templates?

Yes

Review questions

5. Why shouldn't template definitions be in `.cpp` files?

The compiler (not linker) needs to have access to the implementation of templated functions to instantiate them with the template argument.

Exercise 31

- Website -> Course Materials -> 31