Lab 4: Templates

CSCI104

Why Templates???

`std::vector<int>` `std::vector<std::string>` `std::vector<MsgNode*>`

- Code reuse!!
- Treat type as a variable
- Can accommodate all types
  - ex) MsgNode*, Cat, StudentRecord, int, string
- Compiler will substitute user-specified type
- Generates specific versions of your implementation with the type you want

Template Examples

- std::pair
  - Programmers declare with two “types”
  - Values of the types are passed into constructor

```
std::pair<int, std::string> student(1234567890, "Tommy Trojan");
std::pair<std::string, int> question("What is the answer to life, universe, and everything
```

- Return values of functions
  - Can be defined “programmatically” too

```
int studentId = student.first; // returns an int
std::string answer = question.first; // returns a string
```

How to Declare Template:

```
template < >
```

- Use **template < >** tag before class declaration AND before each implementation of class's functions

pair.h

27 lines (22 sloc) | 593 Bytes

```
1  template <typename FirstType, typename SecondType>
2  class Pair {
3  public:
4          Pair(FirstType f, SecondType s);
5
6          FirstType getFirst();
7          SecondType getSecond();
8
9  private:
10         FirstType first;
11         SecondType second;
12 };
13
14 template <typename FirstType, typename SecondType>
15 Pair<FirstType, SecondType>::Pair(FirstType f, SecondType s)
16                 : first(f), second(s) {
17 }
18
19 template <typename FirstType, typename SecondType>
20 FirstType Pair<FirstType, SecondType>::getFirst() {
21         return first;
22 }
23
24 template <typename FirstType, typename SecondType>
25 SecondType Pair<FirstType, SecondType>::getSecond() {
26         return second;
27 }
```

type name

int counter

string myString

typename FirstType

typename SecondType

FirstType and SecondType refer to the specific types that the user of the templated class specified in declaration.

THE HEADER FILE

- Implementation for all methods go in the header file
- This is required because templated classes cannot be pre-compiled
- DO NOT DO THIS FOR NON TEMPLATED CLASSES

This is all in pair.h !

27 lines (22 sloc) | 593 Bytes

```
1  template <typename FirstType, typename SecondType>
2  class Pair {
3  public:
4          Pair(FirstType f, SecondType s);
5
6          FirstType getFirst();
7          SecondType getSecond();
8
9  private:
10         FirstType first;
11         SecondType second;
12 };
13
14 template <typename FirstType, typename SecondType>
15 Pair<FirstType, SecondType>::Pair(FirstType f, SecondType s)
16                 : first(f), second(s) {
17 }
18
19 template <typename FirstType, typename SecondType>
20 FirstType Pair<FirstType, SecondType>::getFirst() {
21         return first;
22 }
23
24 template <typename FirstType, typename SecondType>
25 SecondType Pair<FirstType, SecondType>::getSecond() {
26         return second;
27 }
```

Using Inner Class of Templated Class

- Inner classes work same way as templated classes
- Inner classes share their outer classes templated type variables
- Whenever you refer to the inner class outside of your class definition, you must append **typename** to the front of the type

```
template<typename T>
class Outer
{
private:
        // We don't need template<typename T> here. Inner will get it from Outer.
        struct Inner
        {
                T val; // Inner class will share outer class's template variable name
        };

public:
        T GetValue();
private:
        Inner GetInner(); // We are in class definition, so we can refer to the inner class without I

private:
        Inner mInner;
};

// The first template<typename T> tells the compiler that we need to use T as a type variable.
// Outer<T>::GetValue is the function name. Since Outer is templated, Outer<int>::GetValue is
// very different from Outer<double>::GetValue, so must include <T> after Outer.

template<typename T>
T Outer<T>::GetValue()
{
        return mInner.val;
}

// The typename in second line at the front of function signature tells the compiler Outer<T>::Inner
// is a class or struct name, not a static variable name and Outer<T>::Inner is the return type. Agai
// since Outer is templated, we must include <T> after Outer.

template<typename T>
typename Outer<T>::Inner Outer<T>::GetInner()
{
        return mInner;
}
```

The Lab

- Template **LList**
  - So you can use it with any class, not just ints

- [ ] Template the LList class. Include `template < >` tags wherever the class is mentioned. Since there is only one generic type - convention the name is `T` (instead of `FirstType`, `SecondType`).
- [ ] Fix the inner classes Item. Item is setup to store an int variable.
- [ ] Change approriate mentions of `int` to `T`. References to inner classes need to be changed as well - remember that they are now templated.
- [ ] Copy the contents from `llist.cpp` into the bottom of `llist.h`, and fix these functions.
- [ ] Make and run the program using `make`. It should produce the following output without valgrind errors:

- Checkoff
  - Show results after running make
  - OR be working the entire time of lab

- Things to think about
  - After templating, where should your implementation go? In llist.cpp or llist.h?
  - If you would like to implement the constructor for an inner type, use the fully qualified name like this:

```
template <typename T>
LList<T>::Item::Item(const T& v, Item* p, Item* n)
```