Lab 5: Templates

CSCI104
Why Templates???

- 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

```cpp
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

```cpp
int studentId = student.first; // returns an int
std::string answer = question.first; // returns a string
```
How to Declare Template:

- Use `template < >` tag before class declaration AND before each implementation of class’s functions
template <typename FirstType, typename SecondType>

class Pair {
public:
    Pair(FirstType f, SecondType s);
    FirstType getFirst();
    SecondType getSecond();

private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
    Pair<FirstType, SecondType>::Pair(FirstType f, SecondType s) : first(f), second(s) {
}

template <typename FirstType, typename SecondType>
    FirstType Pair<FirstType, SecondType>::getFirst() {
        return first;
    }

template <typename FirstType, typename SecondType>
    SecondType Pair<FirstType, SecondType>::getSecond() {
        return second;
    }

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

type name
int counter
string myString
typename FirstType
typename SecondType
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!

```cpp
template <typename FirstType, typename SecondType>
class Pair {
public:
    Pair(FirstType f, SecondType s);
    FirstType getFirst();
    SecondType getSecond();
private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
Pair<FirstType, SecondType>::Pair(FirstType f, SecondType s) :
    first(f), second(s) {}

template <typename FirstType, typename SecondType>
FirstType Pair<FirstType, SecondType>::getFirst() {
    return first;
}

template <typename FirstType, typename SecondType>
SecondType Pair<FirstType, SecondType>::getSecond() {
    return second;
}
```
Using Inner Class of Templated Class

- In homework, you’ve seen use of the inner struct `Item` in `TokenList`
- Inner classes work same way as templated classes
- Inner classes share their outer class’s 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
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 appropriate 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:

- **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:

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

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