COMP6771 Week 7.1

Templates, Part 2

Inclusion compilation model

- What is wrong with this?
- g++ min.cpp main.cpp -o main

min.h

```
1 template <typename T>
2 T min(T a, T b);
```

min.cpp

```
1 template <typename T>
2 T min(T a, T b) {
3   return a < b ? a : b;
4 }
```

main.cpp

```
1 #include <iostream>
2
3 int main() {
4   std::cout << min(1, 2) << "\n";
5 }
```

Inclusion compilation model

- When it comes to templates, we include definitions (i.e. implementation) in the .h file
  - This is because template definitions need to be known at **compile time** (template definitions can't be instantiated at link time because that would require an instantiation for all types)
- Will expose implementation details in the .h file
- Can cause slowdown in compilation as every file using min.h will have to instantiate the template, then it's up the linker to ensure there is only 1 instantiation.

min.h

```
1 template <typename T>
2 T min(T a, T b) {
3   return a < b ? a : b;
4 }
```

main.cpp

```
1 #include <iostream>
2
3 int main() {
4   std::cout << min(1, 2) << "\n";
5 }
```

Inclusion Compilation Model

- Alternative: Explicit instantiations
- Generally a bad idea

min.h

```
1 template <typename T>
2 T min(T a, T b);
```

min.cpp

```
1 template <typename T>
2 T min(T a, T b) {
3   return a < b ? a : b;
4 }
5
6 template int min<int>(int, int);
7 template double min<double>(double, double);
```

main.cpp

```
1 #include <iostream>
2
3 int main() {
4   std::cout << min(1, 2) << "\n";
5   std::cout << min(1.0, 2.0) << "\n";
6 }
```

Inclusion Compilation Model

- Exact same principles will apply for classes
- Implementations must be in header file, and compiler should only behave as if one Stack<int> was instantiated

Stack.h

```h
#include <vector>

template <typename T>
class Stack {
 public:
  Stack() {}
  void pop();
  void push(const T& i);
 private:
  std::vector<T> items_;
}

#include "stack.tpp"
```

Stack.tpp

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

template <typename T>
void Stack<T>::push(const T& i)
  items_.push_back(i);
}
```

main.cpp

```cpp
int main() {
  Stack<int> s;
}
```

Inclusion Compilation Model

- Lazy instantiation: Only members functions that are called are instantiated
  - In this case, pop() will not be instantiated

Stack.h

```
#include <vector>

template <typename T>
class Stack {
 public:
  Stack() {}
  void pop();
  void push(const T& i);
 private:
  std::vector<T> items_;
}

#include "stack.tpp"
```

Stack.tpp

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

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

main.cpp

```
int main() {
  Stack<int> s;
  s.push(5);
}
```

Static Members

```
#include <vector>

template <typename T>
class Stack {
 public:
  Stack();
  ~Stack();
  void push(T&);
  void pop();
  T& top();
  const T& top() const;
  static int numStacks_;
 private:
  std::vector<T> stack_;
};

template <typename T>
int Stack<T>::numStacks_ = 0;

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

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

- Each template instantiation has it's own set of static members

```
#include <iostream>

#include "lectures/week7/stack.h"

int main() {
  Stack<float> fs;
  Stack<int> is1, is2, is3;
  std::cout << Stack<float>::numStacks_ << "\n";
  std::cout << Stack<int>::numStacks_ << "\n";
}
```

Friends

- Each stack instantiation has one unique instantiation of the friend

```
1  #include <vector>
2
3  template <typename T>
4  class Stack {
5   public:
6    Stack();
7    ~Stack();
8    void push(T&);
9    void pop();
10   friend std::ostream& operator<<(std::ostream& os, const Stack& s);
11  private:
12   std::vector<T> stack_;
13 };
14
15 template <typename T>
16 void push(T& t) {
17   stack_.push_back(t);
18 }
19
20 template <typename T>
21 std::ostream& operator<<(std::ostream& os, const Stack<T>& s) {
22   std::cout << "My top item is " << s.stack_.back() << \n";
23 }
```

```
1  #include <iostream>
2  #include <string>
3
4  #include "lectures/week7/stack.h"
5
6  int main() {
7    Stack<std::string> ss;
8    ss.push("Hello");
9    std::cout << ss << "\n":
10
11   Stack<int> is;
12   is.push(5);
13   std::cout << is << "\n":
14 }
```

Default Members

```
#include <vector>

template <typename T, typename CONT = std::vector<T>>
class Stack {
 public:
  Stack();
  ~Stack();
  void push(T&);
  void pop();
  T& top();
  const T& top() const;
  static int numStacks_;
 private:
  CONT stack_;
};

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

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

template <typename T, typename CONT>
Stack<T, CONT>:: ~Stack() { numStacks_--; }
```

- We can provide default arguments to template types (where the defaults themselves are types)
- It means we have to update all of our template parameter lists

```
#include <iostream>

#include "lectures/week7/stack.h"

int main() {
  Stack<float> fs;
  Stack<int> is1, is2, is3;
  std::cout << Stack<float>::numStacks_ << "\n";
  std::cout << Stack<int>::numStacks_ << "\n";
}
```