Template classes

When we learned STL, we considered these templates:

```
template< typename T >
struct Node {
    T payload;
    Node *next;
};
template< typename T >
void print_list(Node<T> *head) {
    Node<T> *cur = head;
    while(cur != NULL) {
        std::cout << cur->payload << " ";
        cur = cur->next;
    }
    std::cout << std::endl;
}
```

One `struct`/function that works for (almost) any type `T`

Template classes

```
// template1.cpp
```

```
1  #include "node_template.h"
2
3  int main() {
4      Node<float> f3 = {95.1f, NULL}; // float payload
5      Node<float> f2 = {48.7f, &f3};  // float payload
6      Node<float> f1 = {24.3f, &f2};  // float payload
7      print_list(&f1);
8
9      Node<int> i2 = {239, NULL};     // int payload
10     Node<int> i1 = {114, &i2};      // int payload
11     print_list(&i1);
12     return 0;
13 }
```

```
$ g++ -o template1 template1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./template1
```

```
24.3 48.7 95.1
114 239
```

Now we make it a template class.

Template classes

```
// template_class1.h

#include <iostream>

template< typename T >
class Node {
public:
    Node(T pay, Node<T> *nx) : payload(pay), next(nx) { }

    void print() const {
        const Node<T> *cur = this;
        while(cur != NULL) {
            std::cout << cur->payload << ' ';
            cur = cur->next;
        }
        std::cout << std::endl;
    }

private:
    T payload;
    Node<T> *next;
};
```

Template classes

```
// template_class1.cpp

#include "template_class1.h"

int main() {
    Node<float> f3 = {95.1f, NULL};
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print();

    Node<int> i2 = {239, NULL};
    Node<int> i1 = {114, &i2};
    i1.print();
    return 0;
}
```

```
$ g++ -o template_class1 template_class1.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./template_class1

24.3 48.7 95.1
114 239
```

Template classes

Everything looks fine. Now lets separate the template class into a `.h` file and a `.cpp` file.

```
// template_class2.h
```

```
1  #ifndef NODE_TEMPLATE_H__
2  #define NODE_TEMPLATE_H__
3  template< typename T >
4  class Node {
5  public:
6      Node(T pay, Node<T> *nx) : payload(pay), next(nx) {}
7      void print() const;
8  private:
9      T payload;
10     Node<T> *next;
11 };
12 #endif // NODE_TEMPLATE_H__
```

Template classes

```
// template_class2.cpp

#include "template_class2.h"
#include <iostream>

template< typename T >
void Node<T>::print() const {
    const Node<T> *cur = this;
    while(cur != NULL) {
        std::cout << cur->payload << ' ';
        cur = cur->next;
    }
    std::cout << std::endl;
}
```

Template classes

```
// template_main2.cpp

#include "template_class2.h"
#include <iostream>

int main() {
    Node<float> f3 = {95.1f, NULL};
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print();

    Node<int> i2 = {239, NULL};
    Node<int> i1 = {114, &i2};
    i1.print();
    return 0;
}
```

```
$ g++ -o template_main2 template_main2.cpp template_class2.cpp -std=c++11 -peda

/tmp/cc4NK2Jh.o: In function ‘main’:
template_main2.cpp:(.text+0x6e): undefined reference to ‘Node<float>::print() c
template_main2.cpp:(.text+0xa5): undefined reference to ‘Node<int>::print() con
collect2: error: ld returned 1 exit status
```

Template instantiation

Compiler acts **lazily** when instantiating templates

It doesn't instantiate until **first use**

```
Node<float> f3 = {95.1f, NULL}; // OK fine, I'll instantiate Node<float>
Node<float> f2 = {48.7f, &f3};
Node<float> f1 = {24.3f, &f2};
f1.print();                     // OK fine, I'll instantiate Node<float>::print

Node<int> i2 = {239, NULL};          // OK fine, I'll instantiate Node<int>
Node<int> i1 = {114, &i2};
i1.print();                    // OK fine, I'll instantiate Node<int>::print
```

When instantiating, compiler needs the relevant template classes and functions **to be fully defined already** ...

... **in contrast to typical function or class use**, where definition can be in separate .cpp files. i.e. template classes have to be typically defined in the header files.

Template instantiation

```
$ g++ -o template_main2 template_main2.cpp template_class2.cpp -std=c++11 -peda

/tmp/ccPI9cf1.o: In function ‘main’:
template_main2.cpp:(.text+0x6e): undefined reference to ‘Node<float>::print() c
template_main2.cpp:(.text+0xa5): undefined reference to ‘Node<int>::print() con
collect2: error: ld returned 1 exit status
```

Lazily instantating `Node<float>` & `Node<int>` is fine

- Both are defined in the header

Instantating `Node<float>::print()` & `Node<int>::print()` **won't work**

- They are in a separate source file not #included
- By convention, we never #include `.cpp` files

Template instantiation

There are a couple of possible solutions:

- Move `Node<T>::print()` definition back into the header
- Put member function definitions in a separate file and include it, but **don't** give it a `.cpp` extension. i.e. a different extension such as `.inc` or `.inl`.

`// template_class2.inc`

```
1  #include <iostream>
2
3  template< typename T >
4  void Node<T>::print() const {
5      const Node<T> *cur = this;
6      while(cur != NULL) {
7          std::cout << cur->payload << ' ';
8          cur = cur->next;
9      }
10     std::cout << std::endl;
11 }
```

Template instantiation

```
// template_main2_fixed.cpp

#include "template_class2.h"   // template class definition
#include "template_class2.inc" // template class member function definitions

int main() {
    Node<float> f3 = {95.1f, NULL}; // instantiate Node<float>
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print(); // instantiate Node<float>::print  *** will this work? ***

    Node<int> i2 = {239, NULL};    // instantiate Node<float>
    Node<int> i1 = {114, &i2};
    i1.print(); // instantiate Node<int>::print    *** will this work? ***
    return 0;
}
```

```
$ g++ -o template_main2_fixed template_main2_fixed.cpp -std=c++11 -pedantic -Wa

$ ./template_main2_fixed

24.3 48.7 95.1
114 239
```