THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

C++ Templates

CS 124 / Department of Computer Science

Motivation

Why templates?

Let's say we wanted to create a node class for singly-linked lists. This could be a list of strings, a list of integers, a list of doubles, a list of some structures or objects — we could make a linked list of almost any data type.

But what if we don't know what kind of data our node class will take? We could write one class for strings, StringNode, one for integers, IntNode, one for doubles, DoubleNode, and so on. This is cumbersome, laborious, long-winded, and error prone.

What we'd like is to be able to create a class that can handle any type of data.

Motivation

Templates to the rescue!

C++ provides a means for specifying a data type when a class is instantiated or a function is required, not when it is defined. This is done using templates.

How does C++ do this?

The C++ compiler will scan your code and find all the kinds of data you specify when instantiating your class or function, and will actually compile different versions for each data type you specify.

This is where we get the name "template". You, the programmer, write what is in effect a template for a multitude of classes (or functions), and C++ uses this template and "fills in" the data type for each version it generates.

So you only need to write the templated class or function, and C++ does the rest.

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
         item = newItem;
         ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename Object>
class Node {
private:
    Object item;
    ...
public:
   Node(Object newItem) {
        item = newItem;
        ...
}

...
```

USE

```
Node<int> n1 = Node<int>(42);

Node<std::string> n2 =
Node<std::string>("hello");

Node<double> n3 = Node<double>(3.14)

...
```

How To Use Templates

DEFINITION

```
template<typename T>
T myMax(T a, T b) {
    return (a > b) ? a : b;
}
```

USE

```
int a1 = 5, b1 = 10;

double a2 = 3.1415, b2 = 1.6666;

result = myMax<int>(a1, b1);

// returns 10

result = myMax<double>(a2, b2);

// returns 3.1415
```

This example based on http://www.cplusplus.com/doc/oldtutorial/templates/, © cplusplus.com

How To Use Templates

DEFINITION

```
template<typename T>
T myMax(T a, T b) {
    return (a > b) ? a : b;
}
```

USE

```
int a1 = 5, b1 = 10;

double a2 = 3.1415, b2 = 1.6666;

result = myMax<int>(a1, b1);

// returns 10

result = myMax<double>(a2, b2);

// returns 3.1415
```

This example based on http://www.cplusplus.com/doc/oldtutorial/templates/, © cplusplus.com

How To Use Templates

DEFINITION

```
template<typename T>
T myMax(T a, T b) {
    return (a > b) ? a : b;
}
```

USE

```
int a1 = 5, b1 = 10;

double a2 = 3.1415, b2 = 1.6666;

result = myMax<int>(a1, b1);

// returns 10

result = myMax<double>(a2, b2);

// returns 3.1415
```

This example based on http://www.cplusplus.com/doc/oldtutorial/templates/, © cplusplus.com

Templates

Again, when compiling, C++ will scan your code for all instantiations of a given templated class or function and will compile a separate version for each different datatype used in the instantiation. The name provided for the typename parameter will be replaced with the appropriate data typename in each version.

Side note

We mentioned during our introduction to C++ that it's common practice to separate declarations in a header file from implementation, in a corresponding `.cpp` file.

We can't do that with templates. This is because the compiler doesn't generate the templated class or function until it sees its instantiation or use. Accordingly, it refers back to the template to generate the code needed.

This is why templated code resides in the header file, and why for this course we'll most often be declaring *and* implementing classes and functions within the header file.