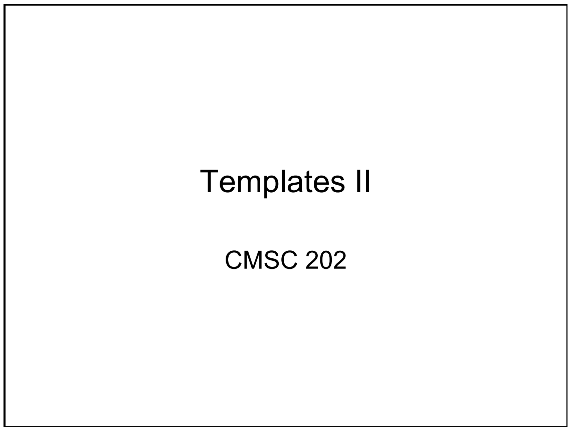
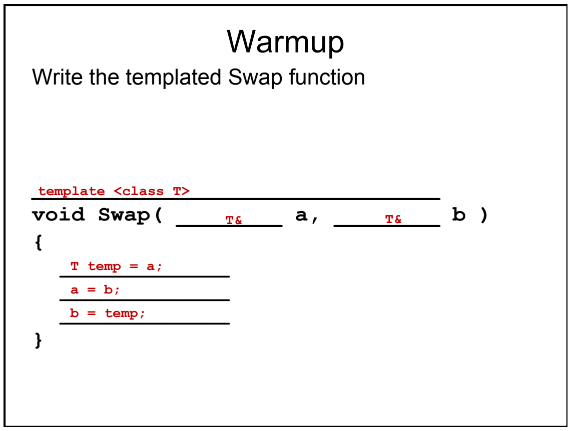


# Templates II

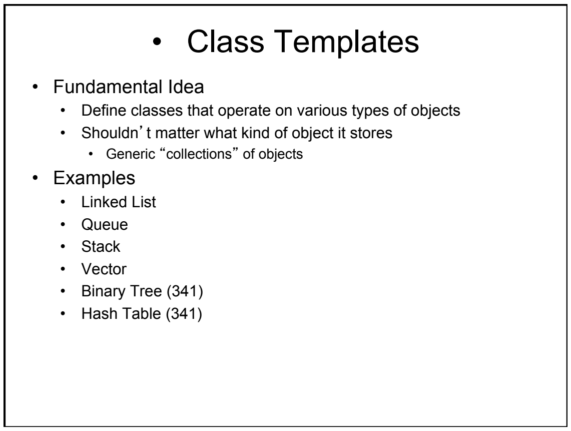
CMSC 202



## Warmup

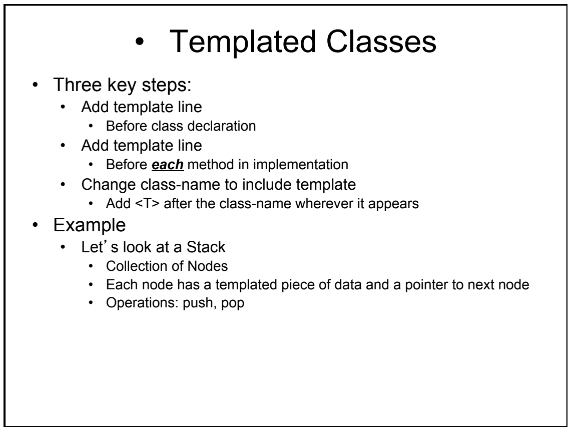
Write the templated Swap function

```
template <class T>
void Swap( T& a, T& b )
{
    T temp = a;
    a = b;
    b = temp;
}
```



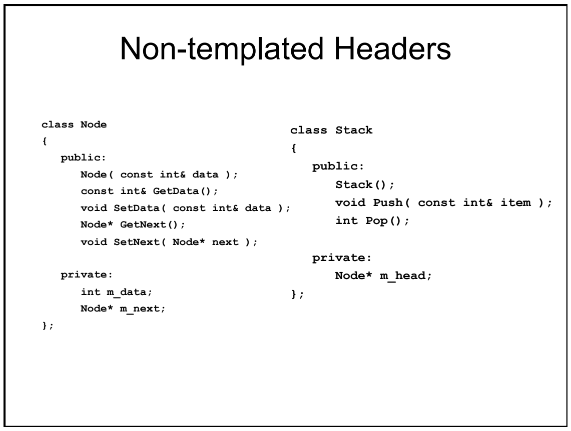
## • Class Templates

- Fundamental Idea
  - Define classes that operate on various types of objects
  - Shouldn't matter what kind of object it stores
    - Generic "collections" of objects
- Examples
  - Linked List
  - Queue
  - Stack
  - Vector
  - Binary Tree (341)
  - Hash Table (341)



# • Templated Classes

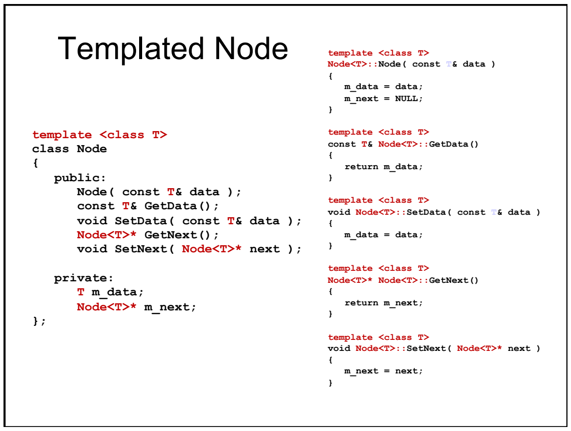
- Three key steps:
  - Add template line
    - Before class declaration
  - Add template line
    - Before ***each*** method in implementation
  - Change class-name to include template
    - Add <T> after the class-name wherever it appears
- Example
  - Let's look at a Stack
    - Collection of Nodes
    - Each node has a templated piece of data and a pointer to next node
    - Operations: push, pop



# Non-templated Headers

```
class Node
{
   public:
      Node( const int& data );
      const int& GetData();
      void SetData( const int& data );
      Node* GetNext();
      void SetNext( Node* next );

   private:
      int m_data;
      Node* m_next;
};
```

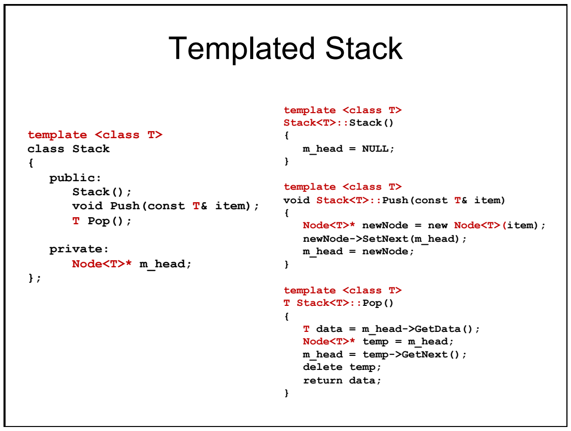
```
class Stack
{
   public:
      Stack();
      void Push( const int& item );
      int Pop();

   private:
      Node* m_head;
};
```



# Templated Node

```
template <class T>
class Node
{
   public:
      Node( const T& data );
      const T& GetData();
      void SetData( const T& data );
      Node<T>* GetNext();
      void SetNext( Node<T>* next );

   private:
      T m_data;
      Node<T>* m_next;
};
```

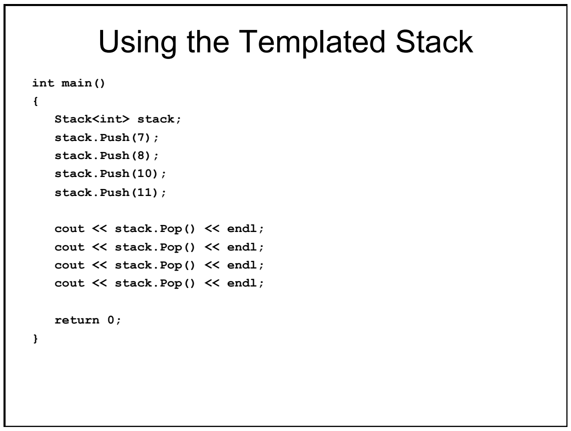
```
template <class T>
Node<T>::Node( const T& data )
{
   m_data = data;
   m_next = NULL;
}

template <class T>
const T& Node<T>::GetData()
{
   return m_data;
}

template <class T>
void Node<T>::SetData( const T& data )
{
   m_data = data;
}

template <class T>
Node<T>* Node<T>::GetNext()
{
   return m_next;
}

template <class T>
void Node<T>::SetNext( Node<T>* next )
{
   m_next = next;
}
```



## Templated Stack

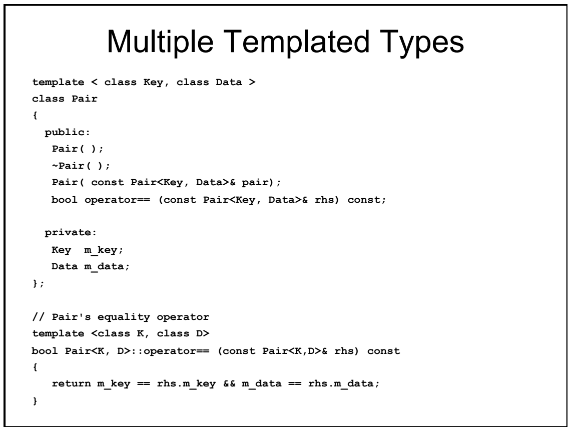
```
template <class T>
class Stack
{
   public:
      Stack();
      void Push(const T& item);
      T Pop();

   private:
      Node<T>* m_head;
};
```

```
template <class T>
Stack<T>::Stack()
{
   m_head = NULL;
}

template <class T>
void Stack<T>::Push(const T& item)
{
   Node<T>* newNode = new Node<T>(item);
   newNode->SetNext(m_head);
   m_head = newNode;
}

template <class T>
T Stack<T>::Pop()
{
   T data = m_head->GetData();
   Node<T>* temp = m_head;
   m_head = temp->GetNext();
   delete temp;
   return data;
}
```



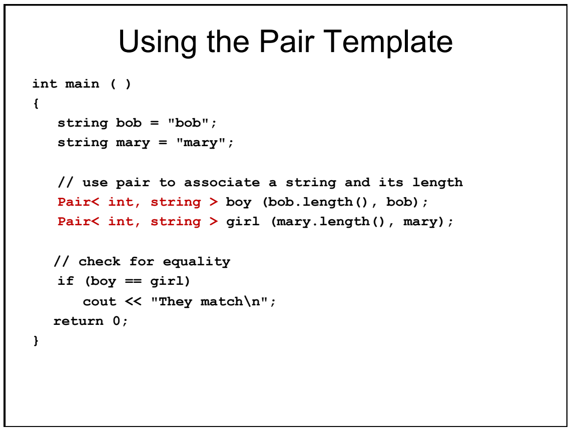
## Using the Templated Stack

```
int main()
{
   Stack<int> stack;
   stack.Push(7);
   stack.Push(8);
   stack.Push(10);
   stack.Push(11);

   cout << stack.Pop() << endl;
   cout << stack.Pop() << endl;
   cout << stack.Pop() << endl;
   cout << stack.Pop() << endl;

   return 0;
}
```



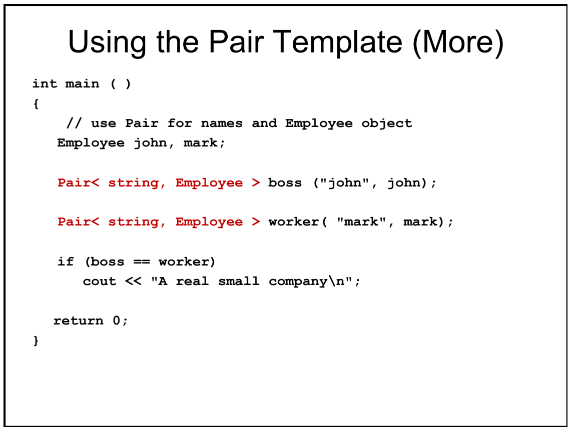
## Multiple Templated Types

```
template < class Key, class Data >
class Pair
{
  public:
   Pair( );
   ~Pair( );
   Pair( const Pair<Key, Data>& pair);
   bool operator== (const Pair<Key, Data>& rhs) const;

  private:
   Key  m_key;
   Data m_data;
};

// Pair's equality operator
template <class K, class D>
bool Pair<K, D>::operator== (const Pair<K,D>& rhs) const
{
   return m_key == rhs.m_key && m_data == rhs.m_data;
}
```



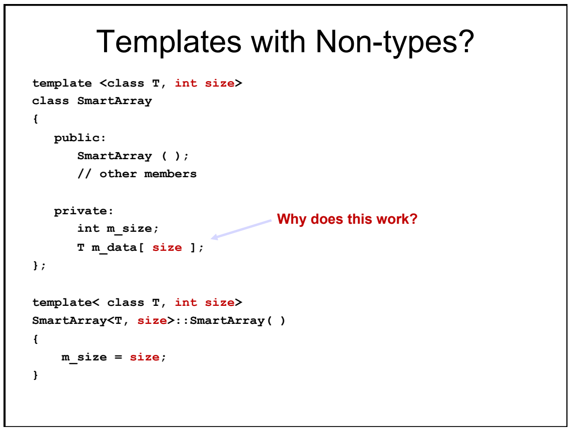
## Using the Pair Template

```
int main ( )
{
   string bob = "bob";
   string mary = "mary";

   // use pair to associate a string and its length
   Pair< int, string > boy (bob.length(), bob);
   Pair< int, string > girl (mary.length(), mary);

  // check for equality
   if (boy == girl)
      cout << "They match\n";
  return 0;
}
```



## Using the Pair Template (More)

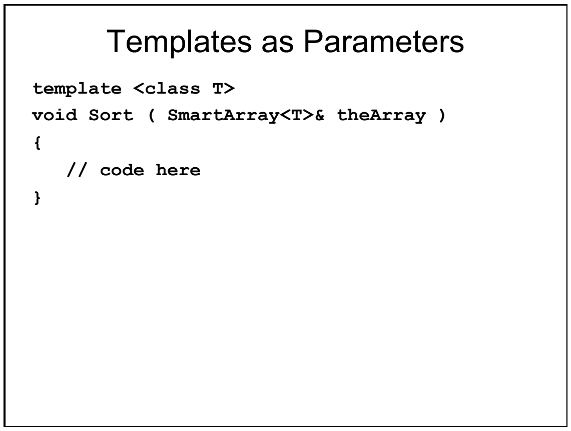
```
int main ( )
{
    // use Pair for names and Employee object
   Employee john, mark;

   Pair< string, Employee > boss ("john", john);

   Pair< string, Employee > worker( "mark", mark);

   if (boss == worker)
      cout << "A real small company\n";

  return 0;
}
```



## Templates with Non-types?

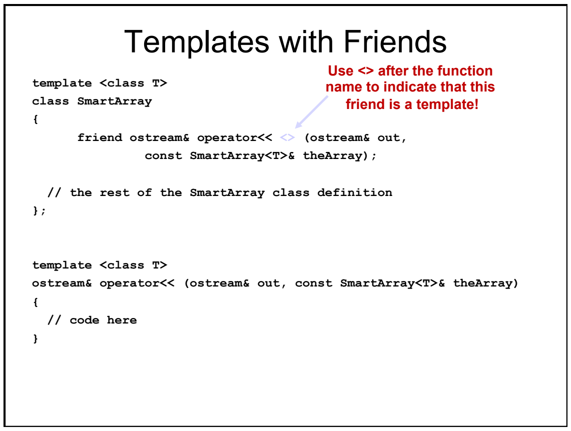
```
template <class T, int size>
class SmartArray
{
   public:
      SmartArray ( );
      // other members

   private:
      int m_size;
      T m_data[ size ];
};

template< class T, int size>
SmartArray<T, size>::SmartArray( )
{
    m_size = size;
}
```

Why does this work?



## Templates as Parameters

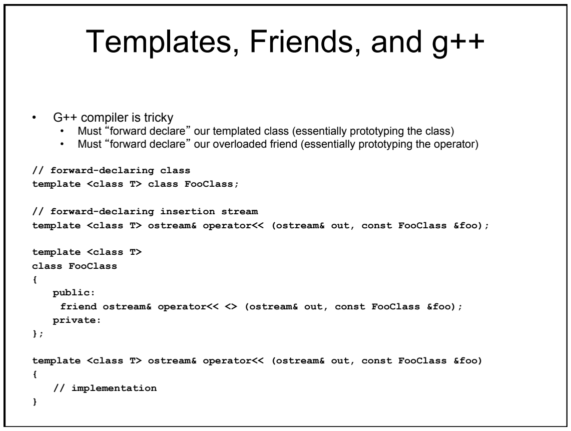
```
template <class T>
void Sort ( SmartArray<T>& theArray )
{
   // code here
}
```



## Templates with Friends

```
template <class T>
class SmartArray
{
      friend ostream& operator<< <> (ostream& out,
               const SmartArray<T>& theArray);

  // the rest of the SmartArray class definition
};


template <class T>
ostream& operator<< (ostream& out, const SmartArray<T>& theArray)
{
  // code here
}
```

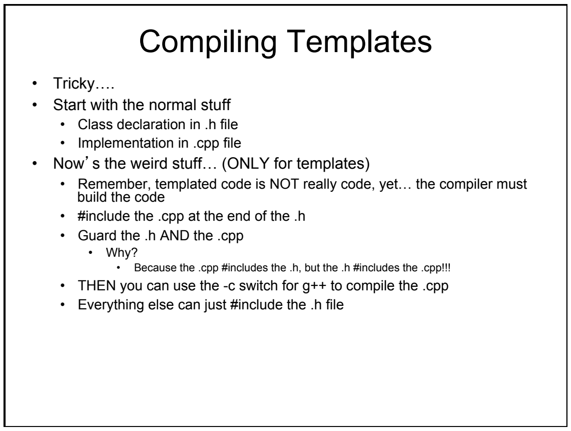
Use <> after the function name to indicate that this friend is a template!



## Templates, Friends, and g++

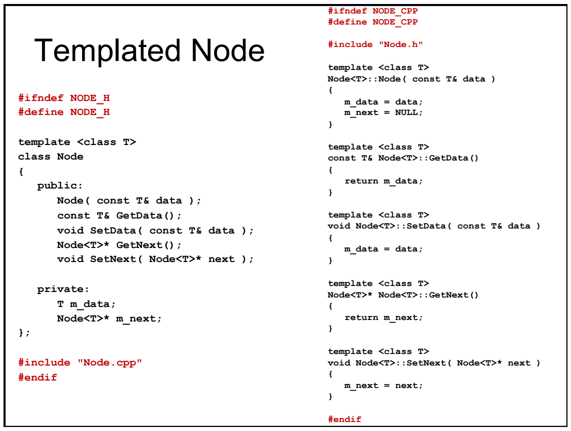
- G++ compiler is tricky
  - Must “forward declare” our templated class (essentially prototyping the class)
  - Must “forward declare” our overloaded friend (essentially prototyping the operator)

```
// forward-declaring class
template <class T> class FooClass;

// forward-declaring insertion stream
template <class T> ostream& operator<< (ostream& out, const FooClass &foo);

template <class T>
class FooClass
{
   public:
    friend ostream& operator<< <> (ostream& out, const FooClass &foo);
   private:
};

template <class T> ostream& operator<< (ostream& out, const FooClass &foo)
{
   // implementation
}
```



## Compiling Templates

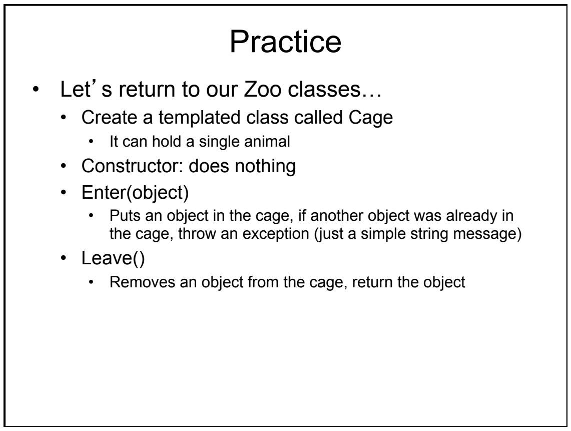
- Tricky….
- Start with the normal stuff
  - Class declaration in .h file
  - Implementation in .cpp file
- Now's the weird stuff… (ONLY for templates)
  - Remember, templated code is NOT really code, yet… the compiler must build the code
  - #include the .cpp at the end of the .h
  - Guard the .h AND the .cpp
    - Why?
      - Because the .cpp #includes the .h, but the .h #includes the .cpp!!!
  - THEN you can use the -c switch for g++ to compile the .cpp
  - Everything else can just #include the .h file



## Templated Node

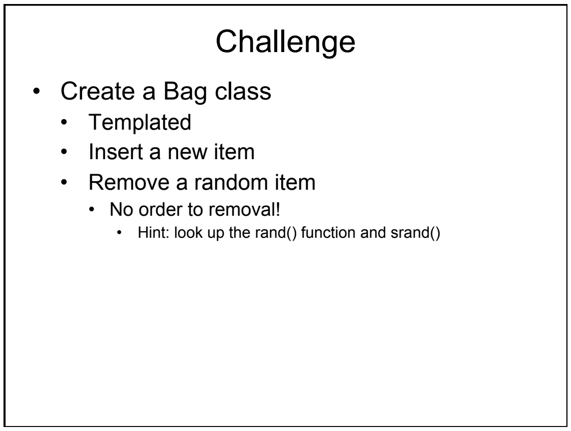
```
#ifndef NODE_H
#define NODE_H

template <class T>
class Node
{
   public:
      Node( const T& data );
      const T& GetData();
      void SetData( const T& data );
      Node<T>* GetNext();
      void SetNext( Node<T>* next );

   private:
      T m_data;
      Node<T>* m_next;
};

#include "Node.cpp"
#endif
```

```
#ifndef NODE_CPP
#define NODE_CPP

#include "Node.h"

template <class T>
Node<T>::Node( const T& data )
{
   m_data = data;
   m_next = NULL;
}

template <class T>
const T& Node<T>::GetData()
{
   return m_data;
}

template <class T>
void Node<T>::SetData( const T& data )
{
   m_data = data;
}

template <class T>
Node<T>* Node<T>::GetNext()
{
   return m_next;
}

template <class T>
void Node<T>::SetNext( Node<T>* next )
{
   m_next = next;
}

#endif
```



## Practice

- Let's return to our Zoo classes…
  - Create a templated class called Cage
    - It can hold a single animal
  - Constructor: does nothing
  - Enter(object)
    - Puts an object in the cage, if another object was already in the cage, throw an exception (just a simple string message)
  - Leave()
    - Removes an object from the cage, return the object



## Challenge

- Create a Bag class
  - Templated
  - Insert a new item
  - Remove a random item
    - No order to removal!
      - Hint: look up the rand() function and srand()