Helpful C++ Transitions

A Few Helpful Slides

#ifndef

Objects using Classes

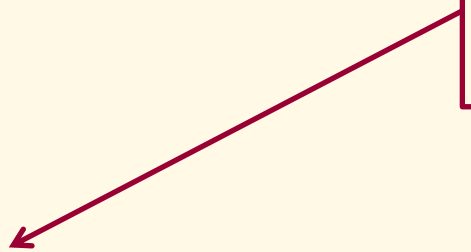
Avoiding Duplicate Headers

```
#ifndef PRODUCT_H
#define PRODUCT_H

...

#endif
```

A Node Class

```
Class Node
{
public:
    Node (string s);
private:
    string data;
    Node *link;
}
```

Constructor used when
Node is declared in public



new – Creating new objects

```
Node* newnode = new Node(s);
```

```
left = new Node ( s, item2);
```

ListNode Header File in C++

```
1  // Fig. 20.3: Listnode.h
2  // Template ListNode class definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List required to announce that class
7  // List exists so it can be used in the friend declaration at line 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE>
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    ListNode( const NODETYPE & ); // constructor
17    NODETYPE getData() const; // return data in node
18 private:
19    NODETYPE data; // data
20    ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
23 // constructor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // empty body
29 } // end ListNode constructor
```

declare class List<NODETYPE> as a friend

member data stores a value of type parameter NODETYPE

member nextPtr stores a pointer to the next ListNode object in the linked list

ListNode Header File in C++

```
30
31 // return copy of data in node
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35    return data;
36 } // end function getData
37
38 #endif
```

Stack Header File in C++


```
1  // Fig. 20.13: Stack.h
2  // Template Stack class definition derived from class List.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
11 public:
12    // push calls the List function insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16    } // end function push
17
18    // pop calls the List function removeFromFront
19    bool pop( STACKTYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function pop
23
```

create a Stack class template through private inheritance of the List class template

perform push and pop by delegating to base-class member functions insertAtFront and removeFromFront, respectively

Stack Header File in C++

```
24    // isStackEmpty calls the List function isEmpty
25    bool isStackEmpty() const
26    {
27       return isEmpty();
28    } // end function isStackEmpty
29
30    // printStack calls the List function print
31    void printStack() const
32    {
33       print();
34    } // end function print
35 }; // end class Stack
36
37 #endif
```

member functions isStackEmpty and printStack delegate to base-class member functions isEmpty and print, respectively

Stack Program in C++

```
1  // Fig. 20.14: Fig20_14.cpp
2  // Template Stack class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // Stack class definition
8
9  int main()
10 {
11    Stack< int > intStack; // create Stack of ints
12
13    cout << "processing an integer Stack" << endl;
14
15    // push integers onto intStack
16    for ( int i = 0; I < 3; i++ )
17    {
18       intStack.push( i );
19       intStack.printStack();
20    } // end for
21
22    int popInteger; // store int popped from stack
23
24    // pop integers from intStack
25    while ( !intStack.isStackEmpty() )
26    {
27       intStack.pop( popInteger );
28       cout << popInteger << " popped from stack" << endl;
29       intStack.printStack();
30    } // end while
```

push integers 0 through 2 onto intStack

pop integers 2 through 0 off intStack

Stack Program in C++

```
31
32    Stack< double > doubleStack; // create Stack of doubles
33    double value = 1.1;
34
35    cout << "processing a double Stack" << endl;
36
37    // push floating-point values onto doubleStack
38    for ( int j = 0; j < 3; j++ )
39    {
40       doubleStack.push( value );
41       doubleStack.printStack();
42       value += 1.1;
43    } // end for
44
45    double popDouble; // store double popped from stack
46
47    // pop floating-point values from doubleStack
48    while ( !doubleStack.isStackEmpty() )
49    {
50       doubleStack.pop( popDouble );
51       cout << popDouble << " popped from sta
52       doubleStack.printStack();
53    } // end while
54
55    return 0;
56 } // end main
```

push values 1.1, 2.2 and 3.3 onto doubleStack

pop values 3.3, 2.2 and 1.1 off doubleStack