C++ Object-Oriented Prog.
Unit 6: Generic Programming

CHAPTER 22: CUSTOM-DESIGNED GENERIC DATA STRUCTURES

DR. ERIC CHOU IEEE SENIOR MEMBER

LECTURE 1

Overview

Introduction

Fixed-size data structures

- Arrays, structs

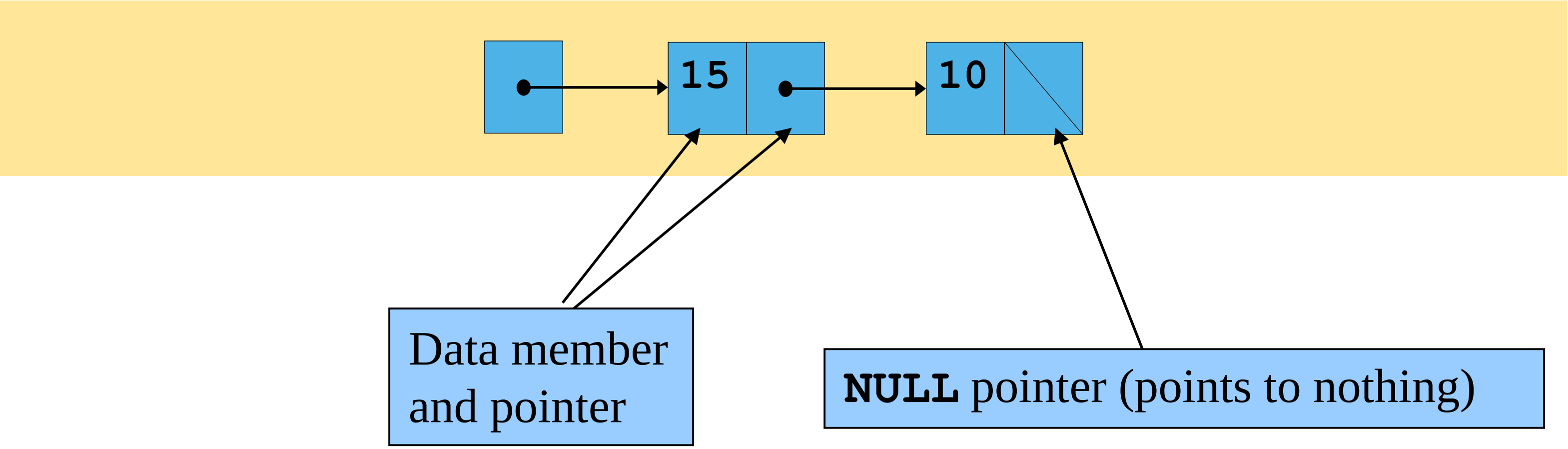
Dynamic data structures

- Grow and shrink as program runs
- Linked lists
  - Insert/remove items anywhere
- Stacks
  - Insert/remove from top of stack
- Queues
  - Like a line, insert at back, remove from front
- Binary trees
  - High-speed searching/sorting of data

Self-Referential Classes

Self-referential class

- Has pointer to object of same class
- Link together to form useful data structures
  - Lists, stacks, queues, trees
- Terminated with **NULL** pointer



Self-Referential Classes

Sample code

```
class Node {
 public:
    Node( int );
    void setData( int );
    int getData() const;
    void setNextPtr( Node * );
    const Node *getNextPtr() const;
 private:
    int data;
    Node *nextPtr;
 };
```

Pointer to object called a *link*

- `nextPtr` points to a `Node`

Dynamic Memory Allocation and Data Structures

Dynamic memory allocation

- Obtain and release memory during program execution
- Create and remove nodes

Operator `new`

- Takes type of object to create
- Returns pointer to newly created object
  - `Node *newPtr = new Node( 10 );`
  - Returns `bad_alloc` if not enough memory
  - `10` is the node's object data

Dynamic Memory Allocation and Data Structures

Operator **`delete`**

- **`delete newPtr;`**
- Deallocates memory allocated by **`new`**, calls destructor
- Memory returned to system, can be used in future
  - **`newPtr`** not deleted, only the space it points to

LECTURE 2

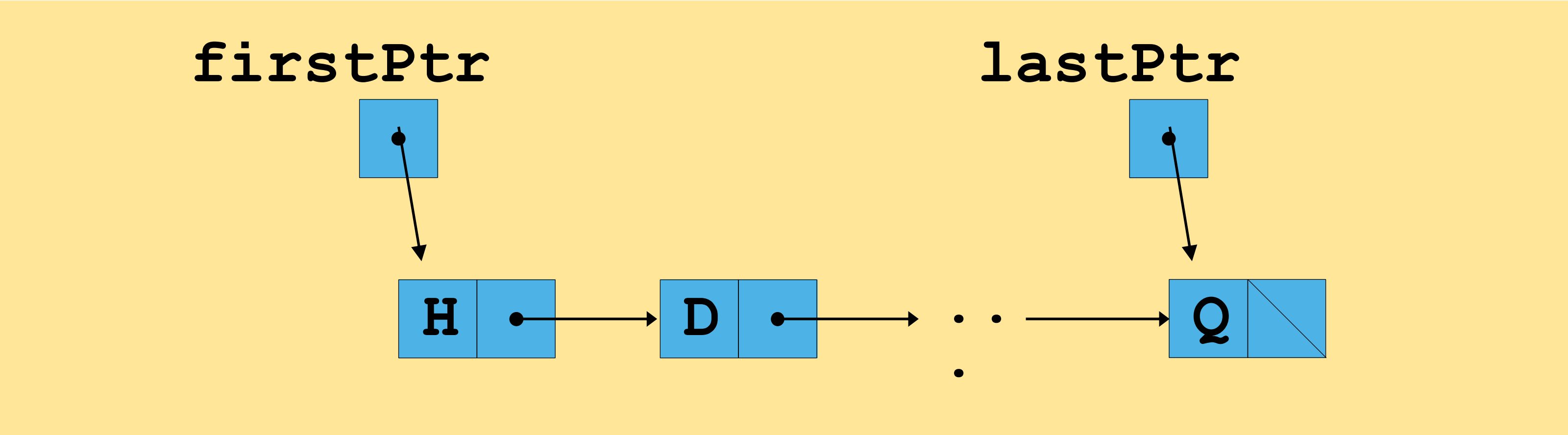
Linked List

Linked Lists

Linked list

- Collection of self-referential class objects (nodes) connected by pointers (links)
- Accessed using pointer to first node of list
  - Subsequent nodes accessed using the links in each node
- Link in last node is null (zero)
  - Indicates end of list
- Data stored dynamically
  - Nodes created as necessary
  - Node can have data of any type

Linked Lists



Linked Lists

Linked lists vs. arrays

- Arrays can become full
  - Allocating "extra" space in array wasteful, may never be used
  - Linked lists can grow/shrink as needed
  - Linked lists only become full when system runs out of memory
- Linked lists can be maintained in sorted order
  - Insert element at proper position
  - Existing elements do not need to be moved

Linked Lists

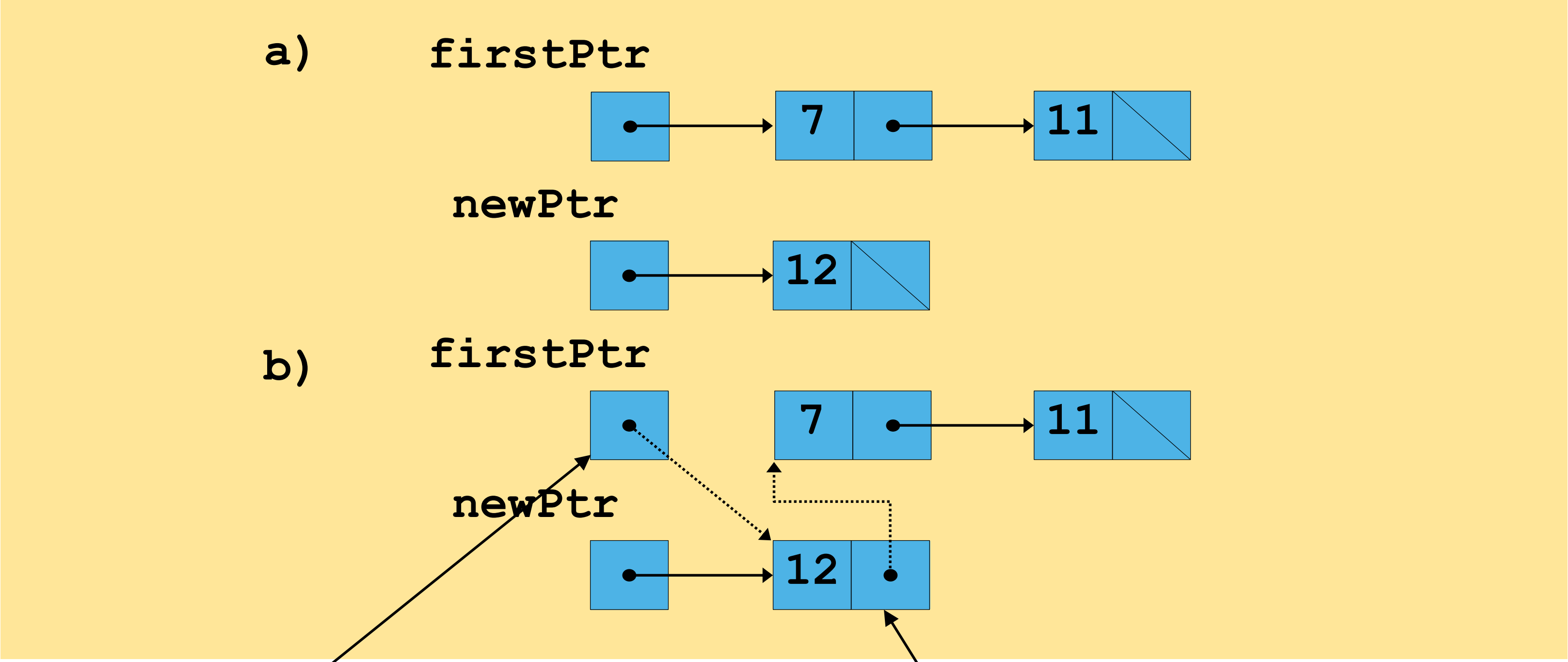
Selected linked list operations

- Insert node at front
- Insert node at back
- Remove node from front
- Remove node from back

In following illustrations

- List has **`firstPtr`** and **`lastPtr`**
- (a) is before, (b) is after

Insert at front



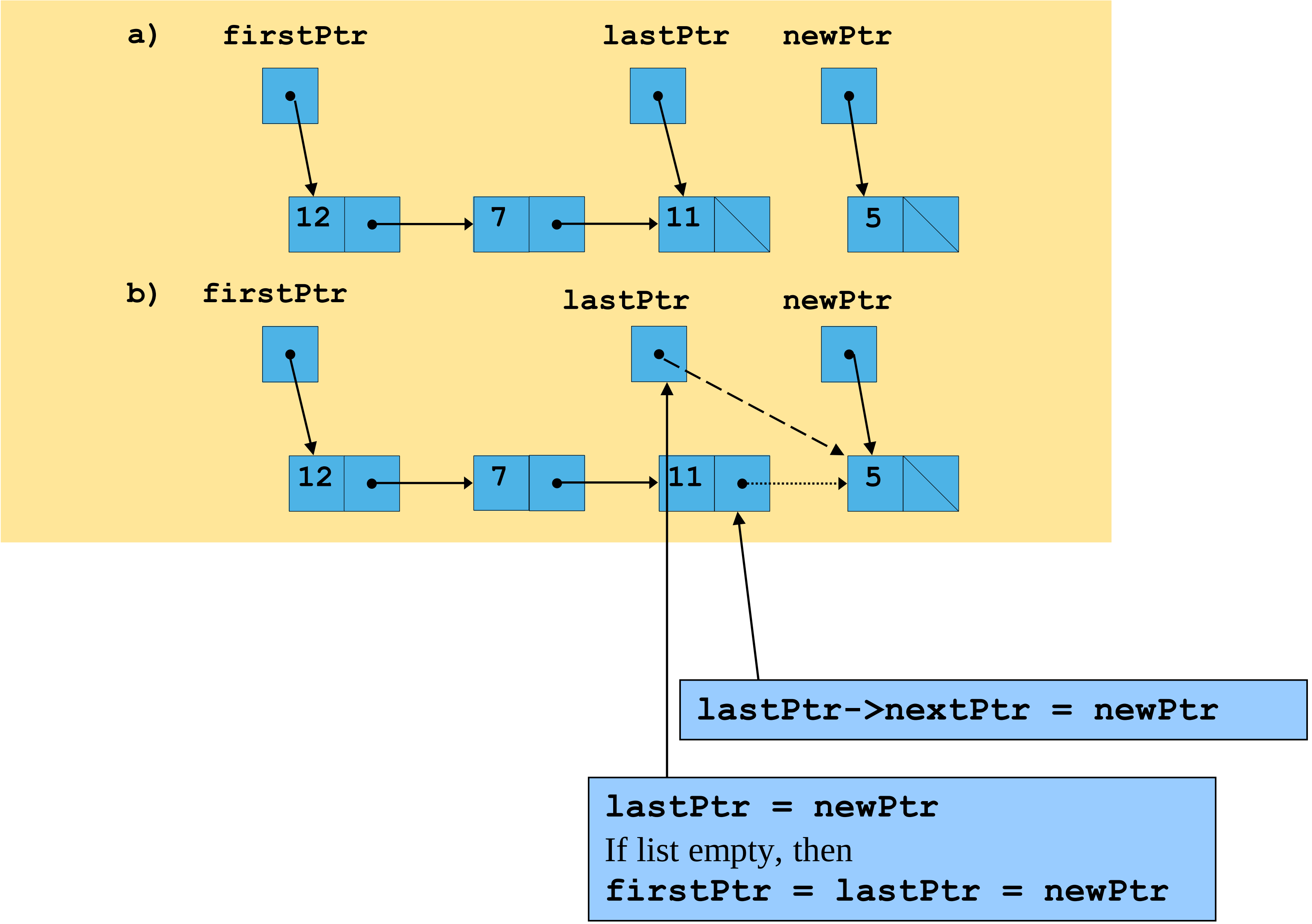
```
firstPtr = newPtr
```

If list empty, then

```
firstPtr = lastPtr = newPtr
```

```
newPtr->nextPtr = firstPtr
```

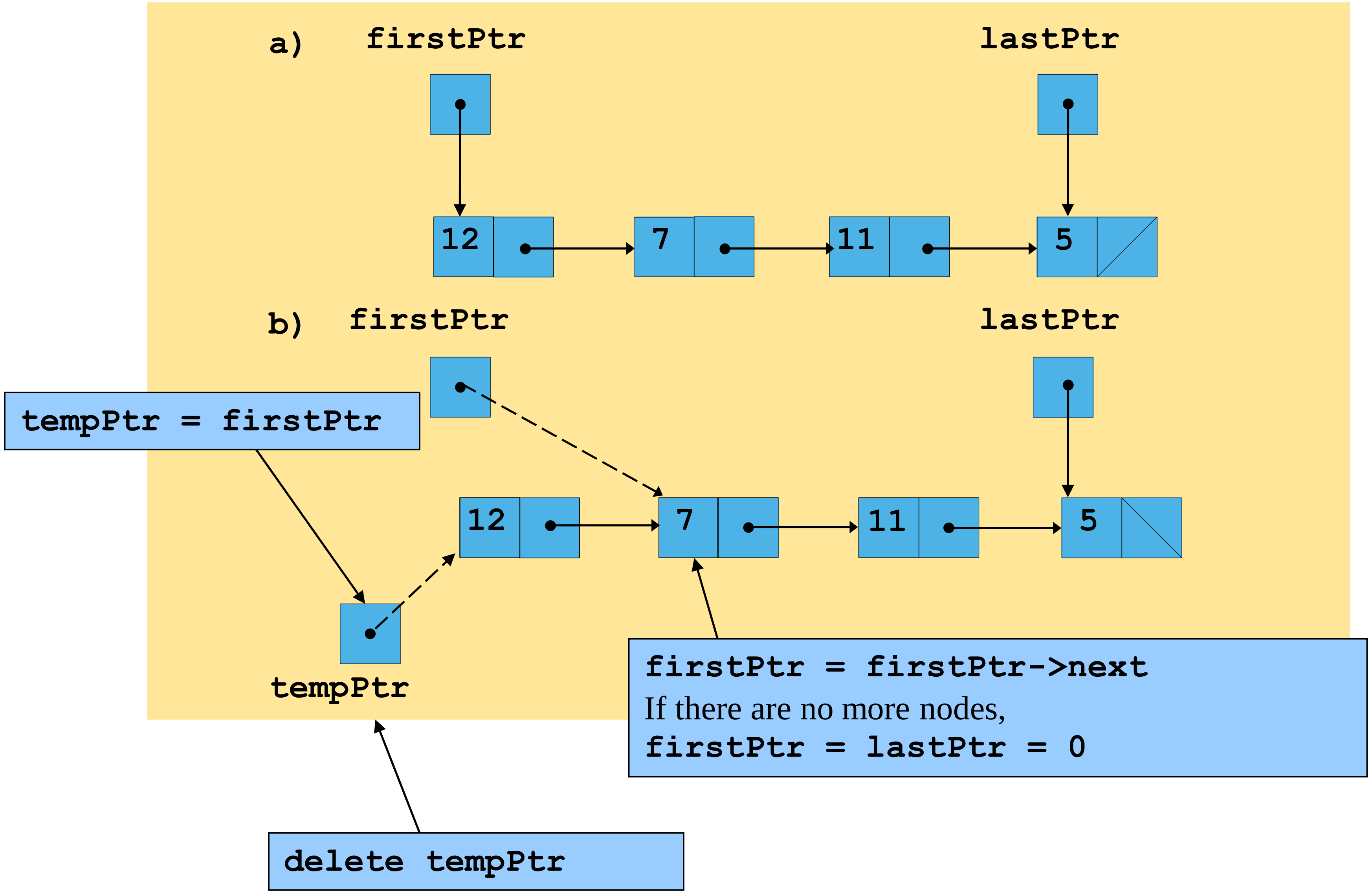
Insert at back



lastPtr->nextPtr = newPtr

lastPtr = newPtr
If list empty, then
firstPtr = lastPtr = newPtr

Remove from front



`tempPtr = firstPtr`

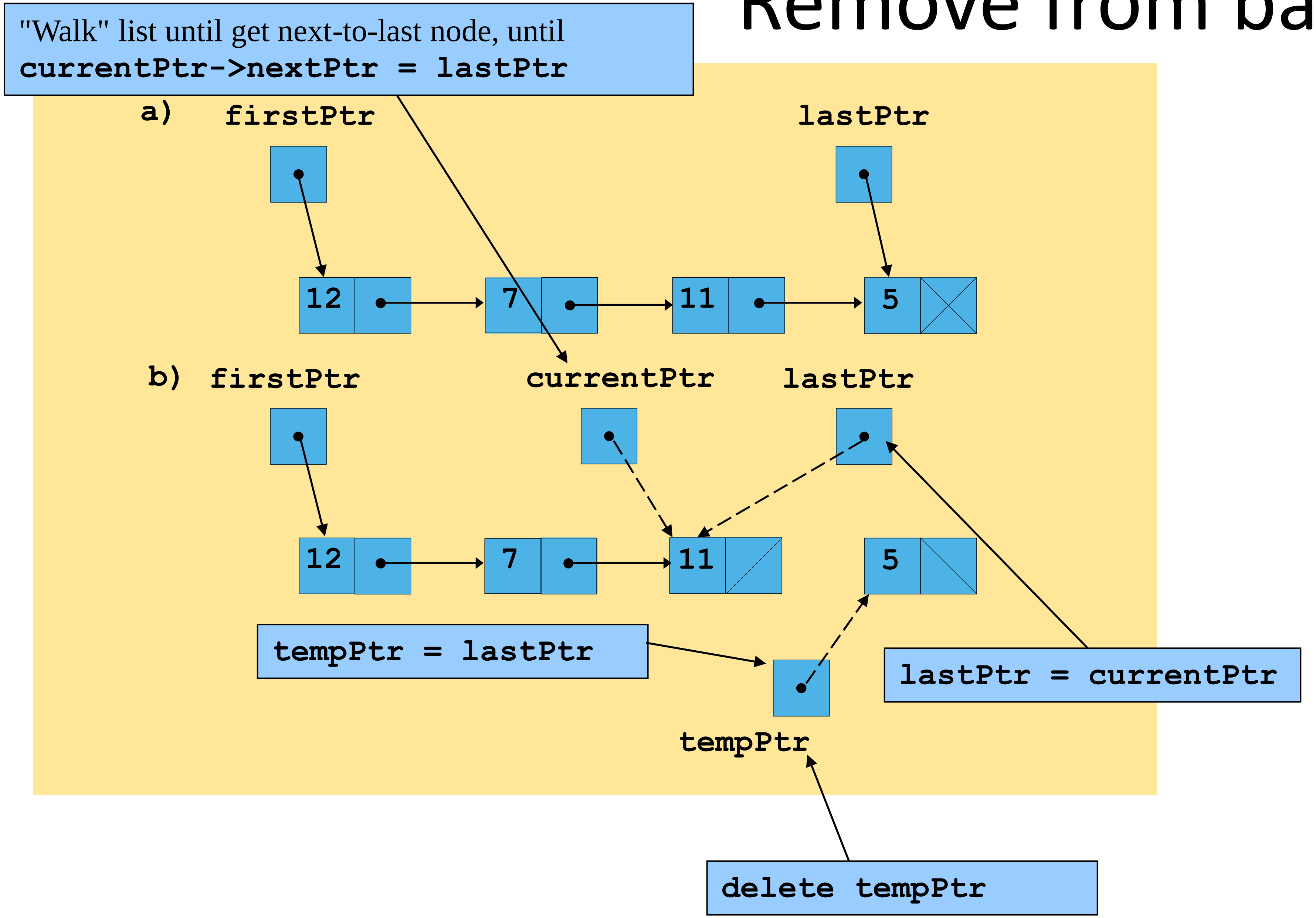
`firstPtr = firstPtr->next`

If there are no more nodes,

`firstPtr = lastPtr = 0`

`delete tempPtr`

Remove from back



"Walk" list until get next-to-last node, until
`currentPtr->nextPtr = lastPtr`

`tempPtr = lastPtr`

`lastPtr = currentPtr`

`delete tempPtr`

Linked Lists

Upcoming program has two class templates

- Create two class templates
- **`ListNode`**
  - **`data`** (type depends on class template)
  - **`nextPtr`**
- **`List`**
  - Linked list of **`ListNode`** objects
  - List manipulation functions
    - **`insertAtFront`**
    - **`insertAtBack`**
    - **`removeFromFront`**
    - **`removeFromBack`**

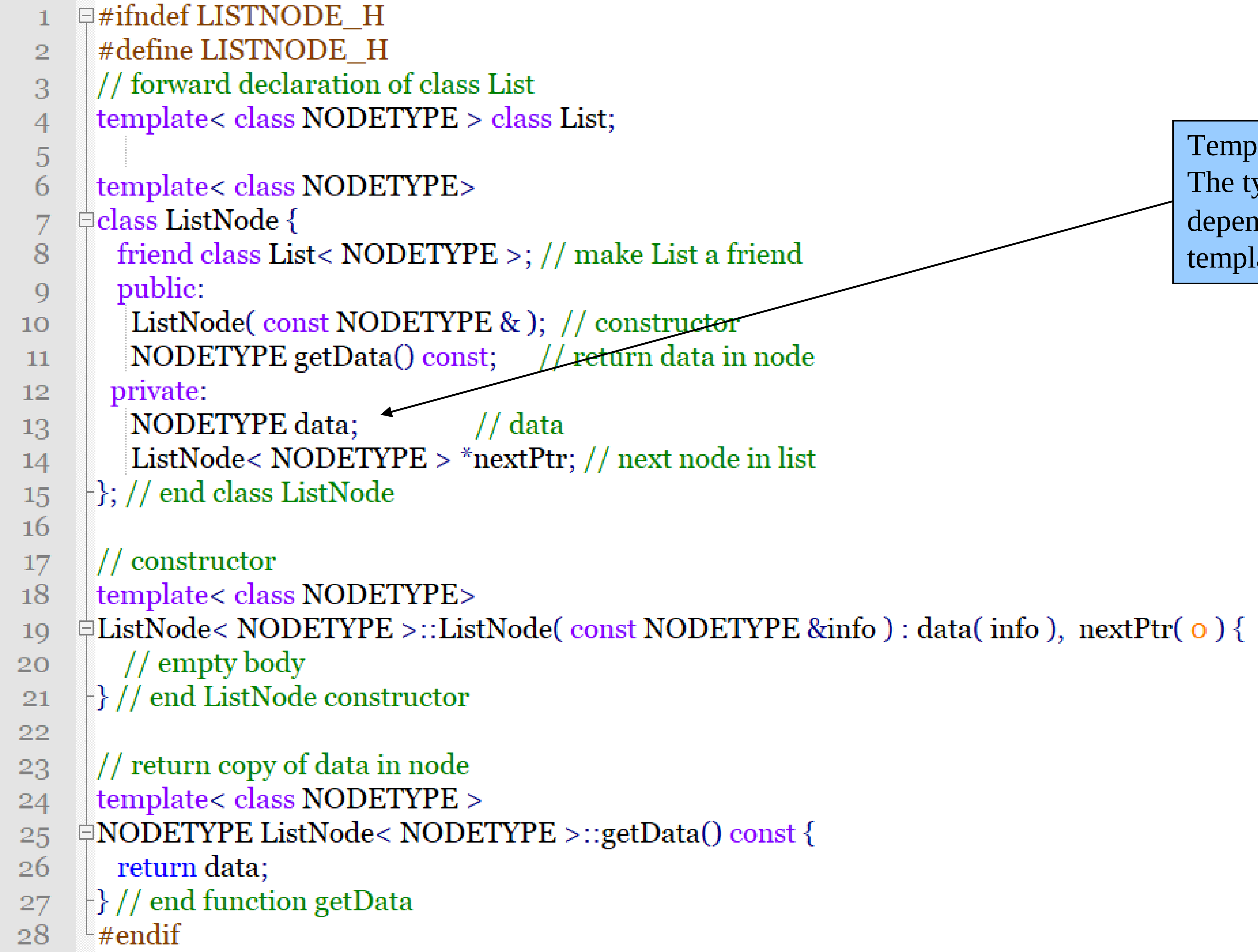
Custom Design Linked-List

Demo Program: testlist.cpp+listnode.h+list.h

Go Notepad++!!!

1. Custom-designed generic list node.
2. Generic linked-list.
3. Menu-driven linked-list test fixture.

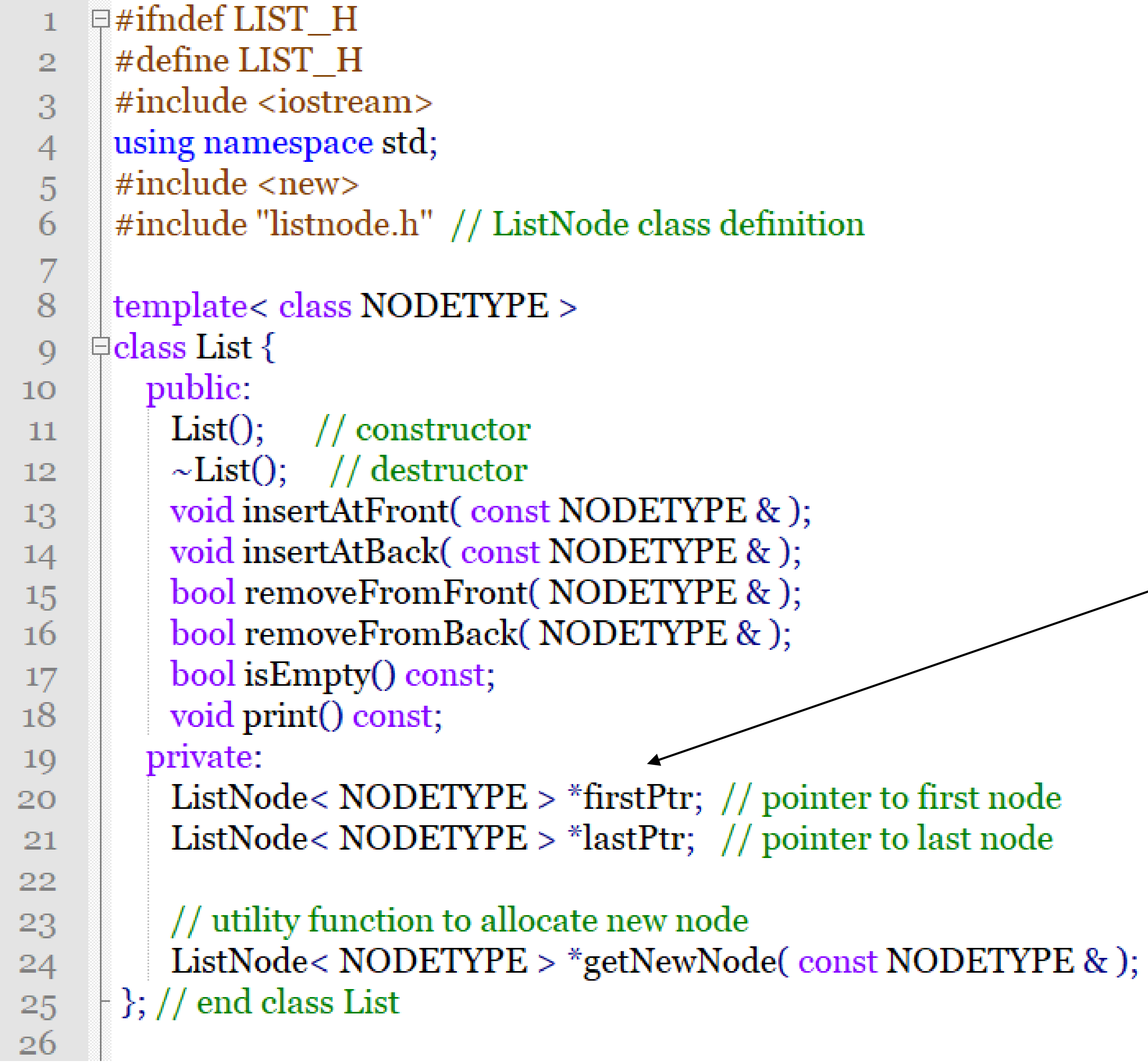
listnode.h



```
#ifndef LISTNODE_H
#define LISTNODE_H
// forward declaration of class List
template< class NODETYPE > class List;

template< class NODETYPE>
class ListNode {
  friend class List< NODETYPE >; // make List a friend
  public:
    ListNode( const NODETYPE & );  // constructor
    NODETYPE getData() const;      // return data in node
  private:
    NODETYPE data;                 // data
    ListNode< NODETYPE > *nextPtr; // next node in list
}; // end class ListNode

// constructor
template< class NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info ) : data( info ),  nextPtr( 0 ) {
   // empty body
} // end ListNode constructor

// return copy of data in node
template< class NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const {
  return data;
} // end function getData
#endif
```

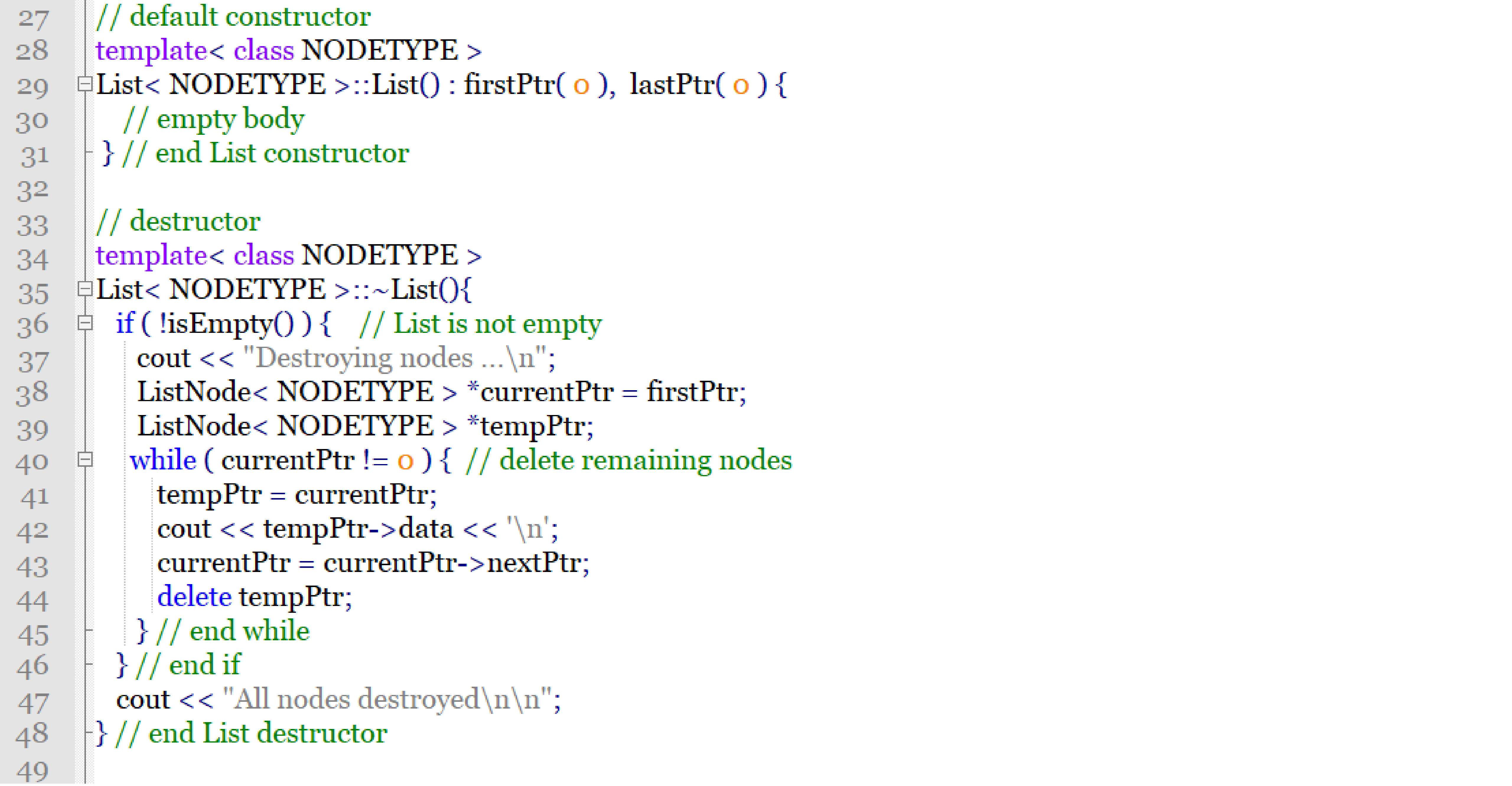
Template class **ListNode**. The type of member **data** depends on how the class template is used.

list.h

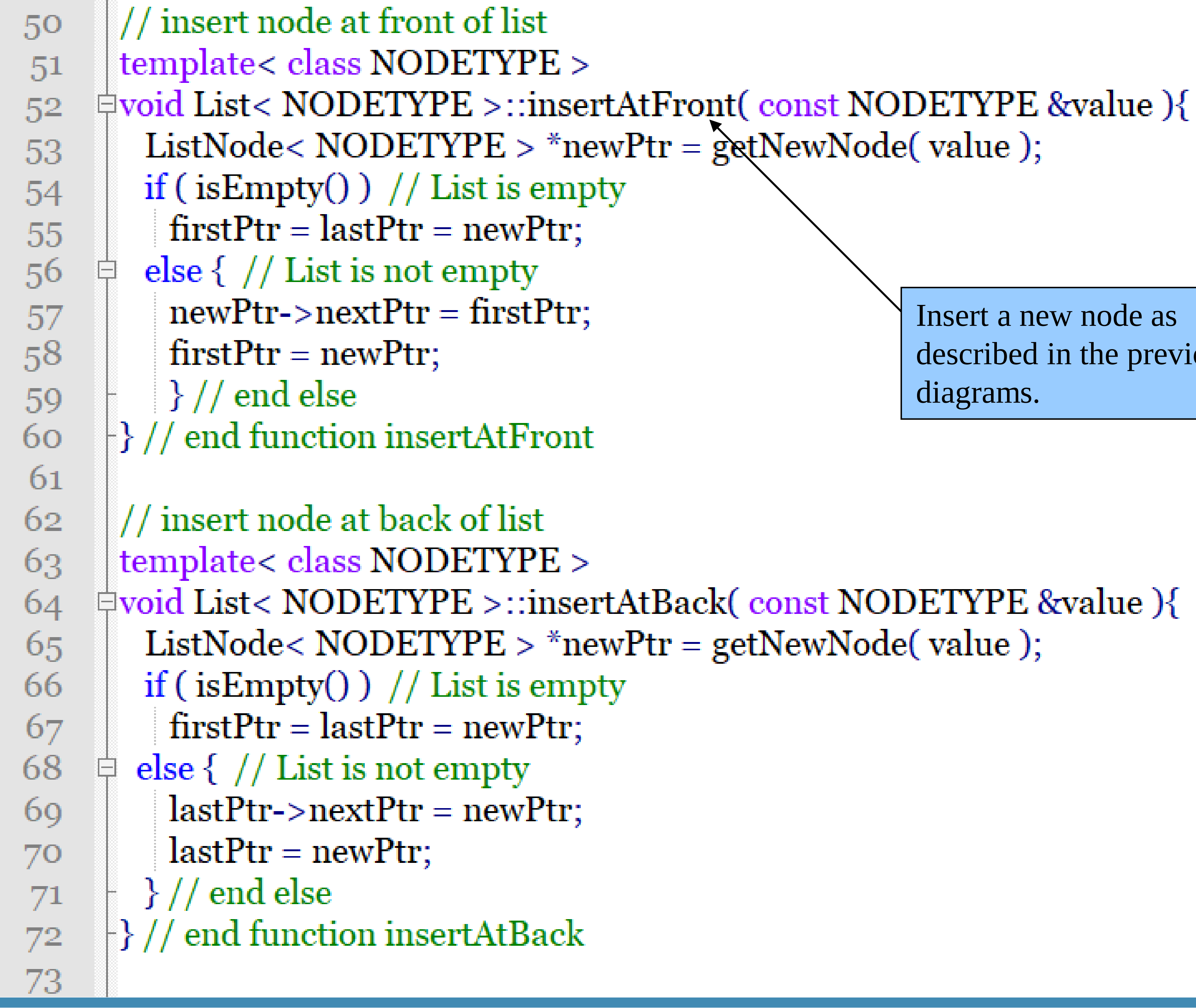


```cpp
#ifndef LIST_H
#define LIST_H
#include <iostream>
using namespace std;
#include <new>
#include "listnode.h"  // ListNode class definition

template< class NODETYPE >
class List {
   public:
     List();      // constructor
     ~List();     // destructor
     void insertAtFront( const NODETYPE & );
     void insertAtBack( const NODETYPE & );
     bool removeFromFront( NODETYPE & );
     bool removeFromBack( NODETYPE & );
     bool isEmpty() const;
     void print() const;
   private:
     ListNode< NODETYPE > *firstPtr;  // pointer to first node
     ListNode< NODETYPE > *lastPtr;   // pointer to last node

     // utility function to allocate new node
     ListNode< NODETYPE > *getNewNode( const NODETYPE & );
 }; // end class List

```

Each **`List`** has a **`firstPtr`** and **`lastPtr`**.

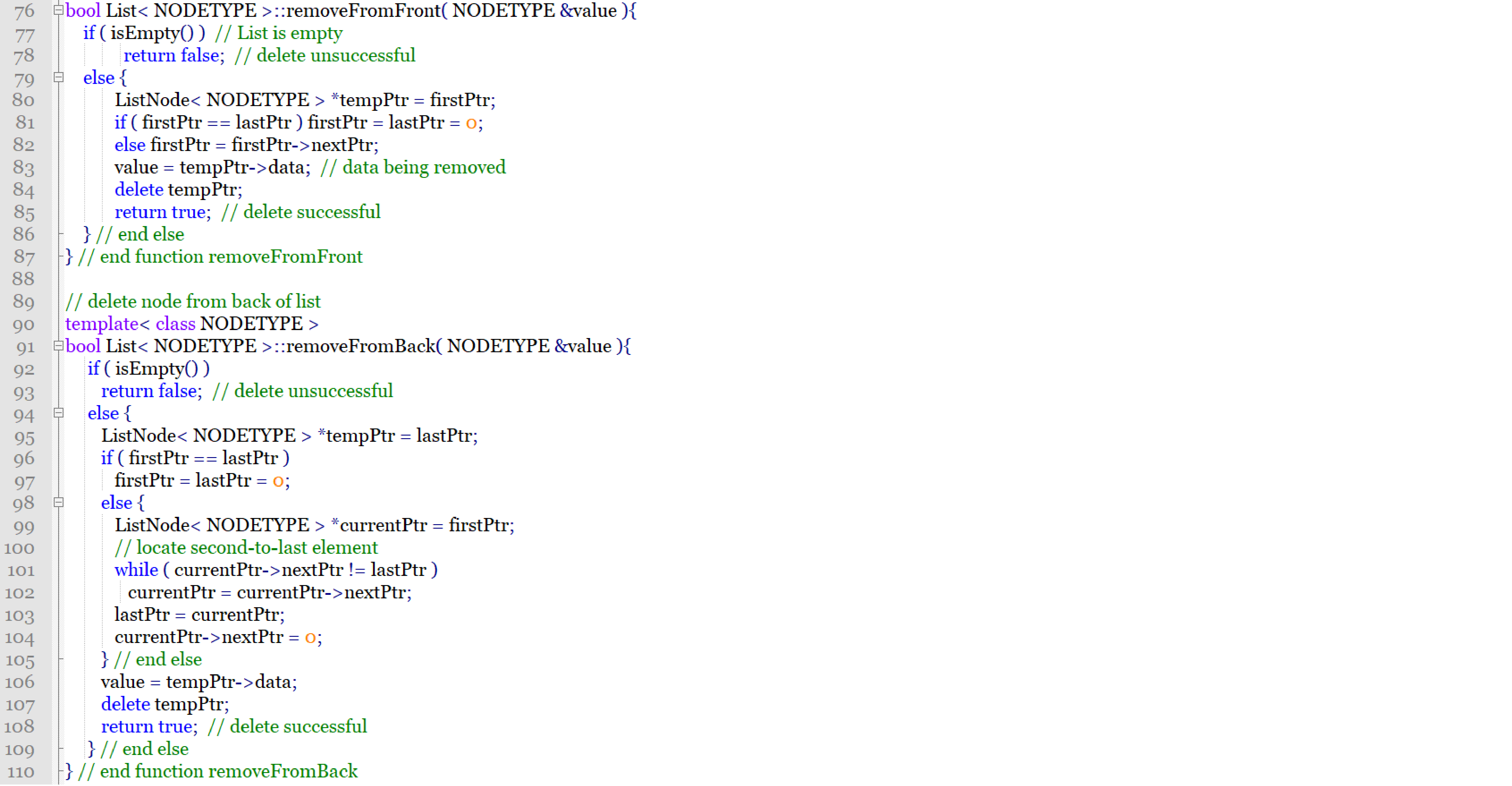


```
// default constructor
template< class NODETYPE >
List< NODETYPE >::List() : firstPtr( 0 ),  lastPtr( 0 ) {
   // empty body
 } // end List constructor

// destructor
template< class NODETYPE >
List< NODETYPE >::~List(){
  if ( !isEmpty() ) {   // List is not empty
    cout << "Destroying nodes ...\n";
    ListNode< NODETYPE > *currentPtr = firstPtr;
    ListNode< NODETYPE > *tempPtr;
   while ( currentPtr != 0 ) {  // delete remaining nodes
      tempPtr = currentPtr;
      cout << tempPtr->data << '\n';
      currentPtr = currentPtr->nextPtr;
      delete tempPtr;
    } // end while
  } // end if
  cout << "All nodes destroyed\n\n";
} // end List destructor

```

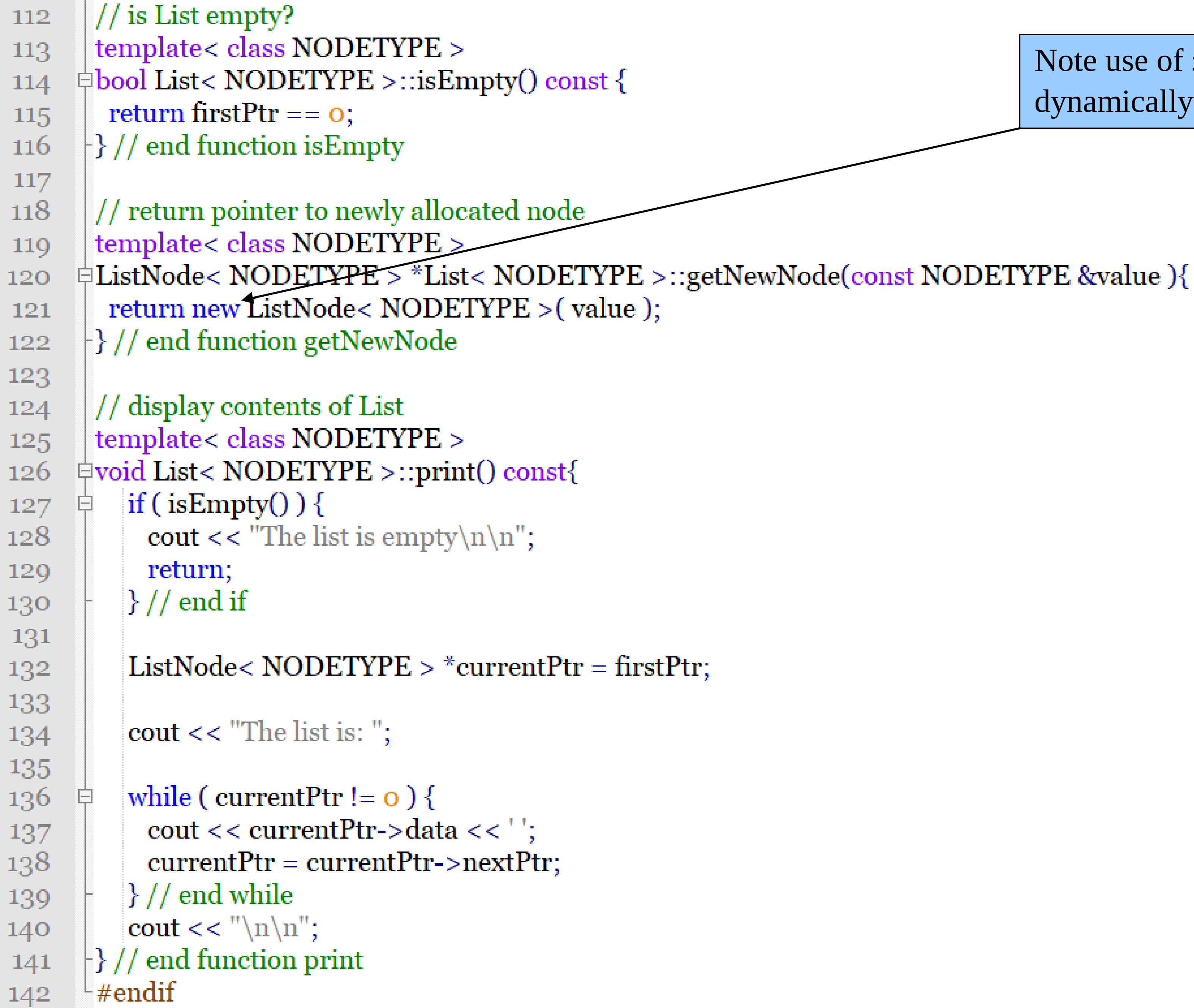


```
// insert node at front of list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value ){
  ListNode< NODETYPE > *newPtr = getNewNode( value );
  if ( isEmpty() )  // List is empty
    firstPtr = lastPtr = newPtr;
  else {  // List is not empty
    newPtr->nextPtr = firstPtr;
    firstPtr = newPtr;
    } // end else
} // end function insertAtFront

// insert node at back of list
template< class NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value ){
  ListNode< NODETYPE > *newPtr = getNewNode( value );
  if ( isEmpty() )  // List is empty
    firstPtr = lastPtr = newPtr;
 else {  // List is not empty
    lastPtr->nextPtr = newPtr;
    lastPtr = newPtr;
  } // end else
} // end function insertAtBack
```

Insert a new node as described in the previous diagrams.



```
bool List< NODETYPE >::removeFromFront( NODETYPE &value ){
  if ( isEmpty() )  // List is empty
      return false;  // delete unsuccessful
  else {
      ListNode< NODETYPE > *tempPtr = firstPtr;
      if ( firstPtr == lastPtr ) firstPtr = lastPtr = 0;
      else firstPtr = firstPtr->nextPtr;
      value = tempPtr->data;  // data being removed
      delete tempPtr;
      return true;  // delete successful
  } // end else
} // end function removeFromFront

// delete node from back of list
template< class NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value ){
   if ( isEmpty() )
     return false;  // delete unsuccessful
   else {
     ListNode< NODETYPE > *tempPtr = lastPtr;
     if ( firstPtr == lastPtr )
       firstPtr = lastPtr = 0;
     else {
       ListNode< NODETYPE > *currentPtr = firstPtr;
       // locate second-to-last element
       while ( currentPtr->nextPtr != lastPtr )
         currentPtr = currentPtr->nextPtr;
       lastPtr = currentPtr;
       currentPtr->nextPtr = 0;
     } // end else
     value = tempPtr->data;
     delete tempPtr;
     return true;  // delete successful
   } // end else
} // end function removeFromBack
```

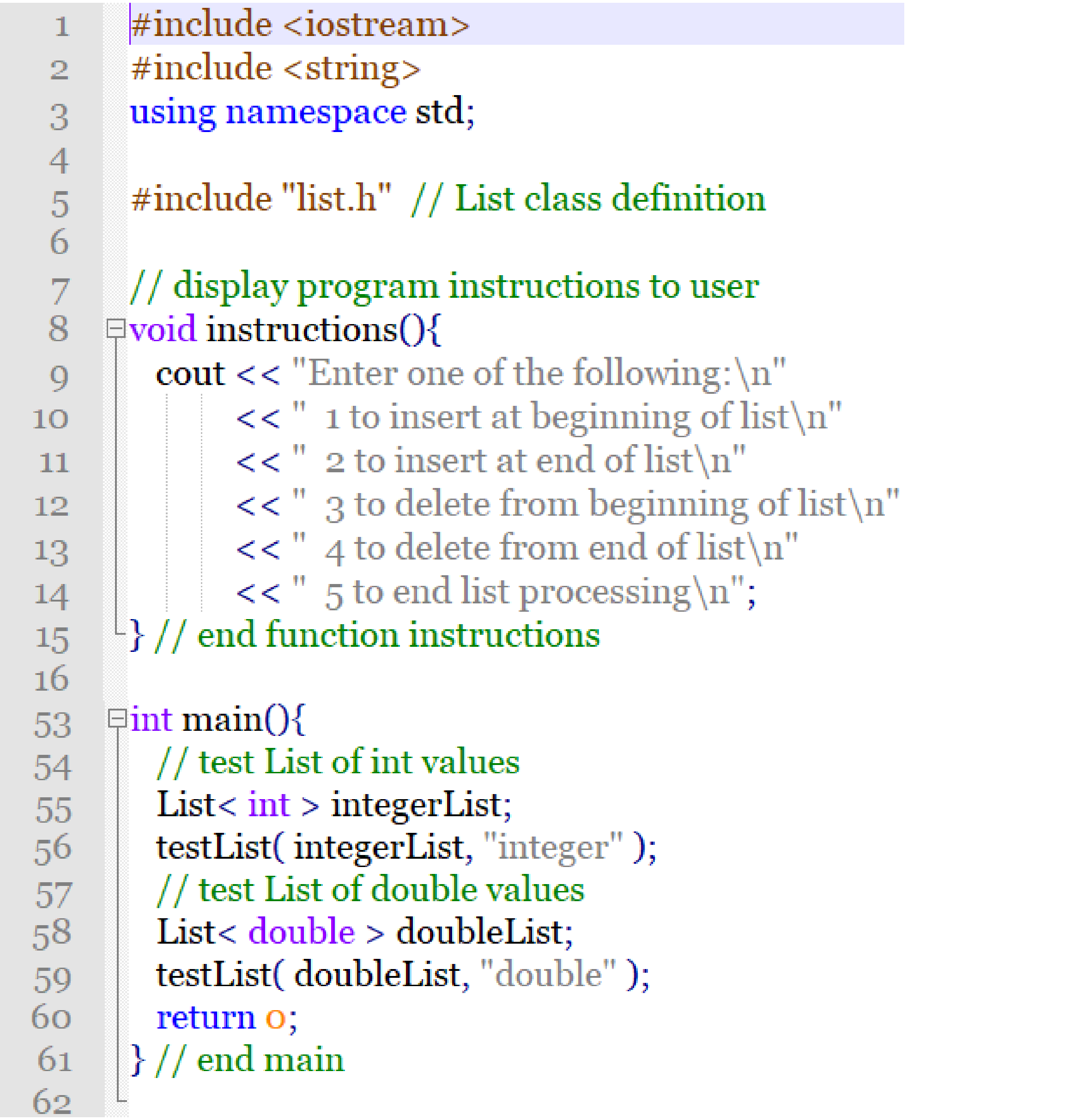


```
// is List empty?
template< class NODETYPE >
bool List< NODETYPE >::isEmpty() const {
  return firstPtr == 0;
} // end function isEmpty

// return pointer to newly allocated node
template< class NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(const NODETYPE &value ){
  return new ListNode< NODETYPE >( value );
} // end function getNewNode

// display contents of List
template< class NODETYPE >
void List< NODETYPE >::print() const{
    if ( isEmpty() ) {
      cout << "The list is empty\n\n";
      return;
    } // end if

    ListNode< NODETYPE > *currentPtr = firstPtr;

    cout << "The list is: ";

    while ( currentPtr != 0 ) {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
    } // end while
    cout << "\n\n";
} // end function print
#endif
```

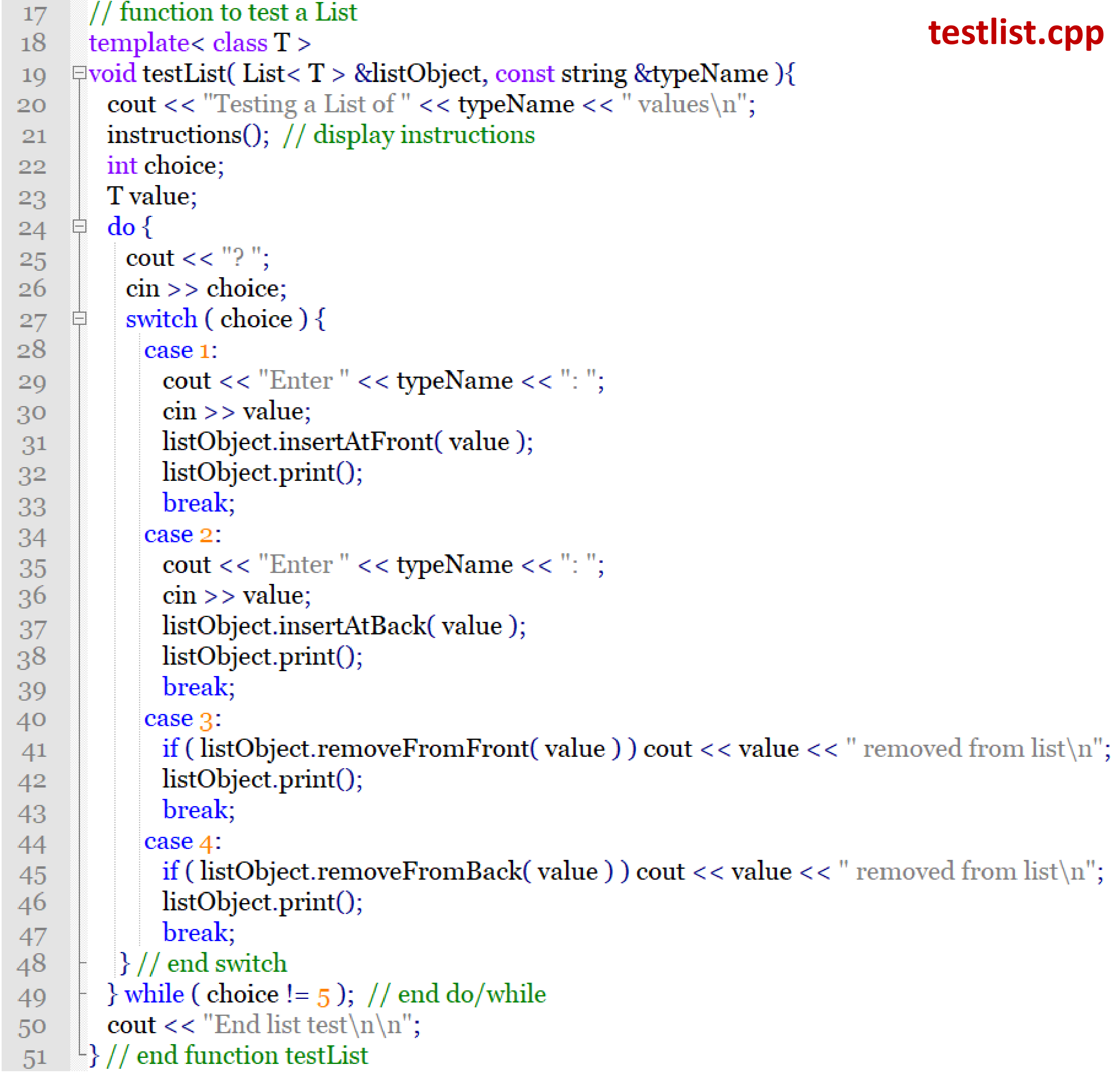
Note use of **new** operator to dynamically allocate a node.

Program to give user a menu to add/remove nodes from a list.

testlist.cpp

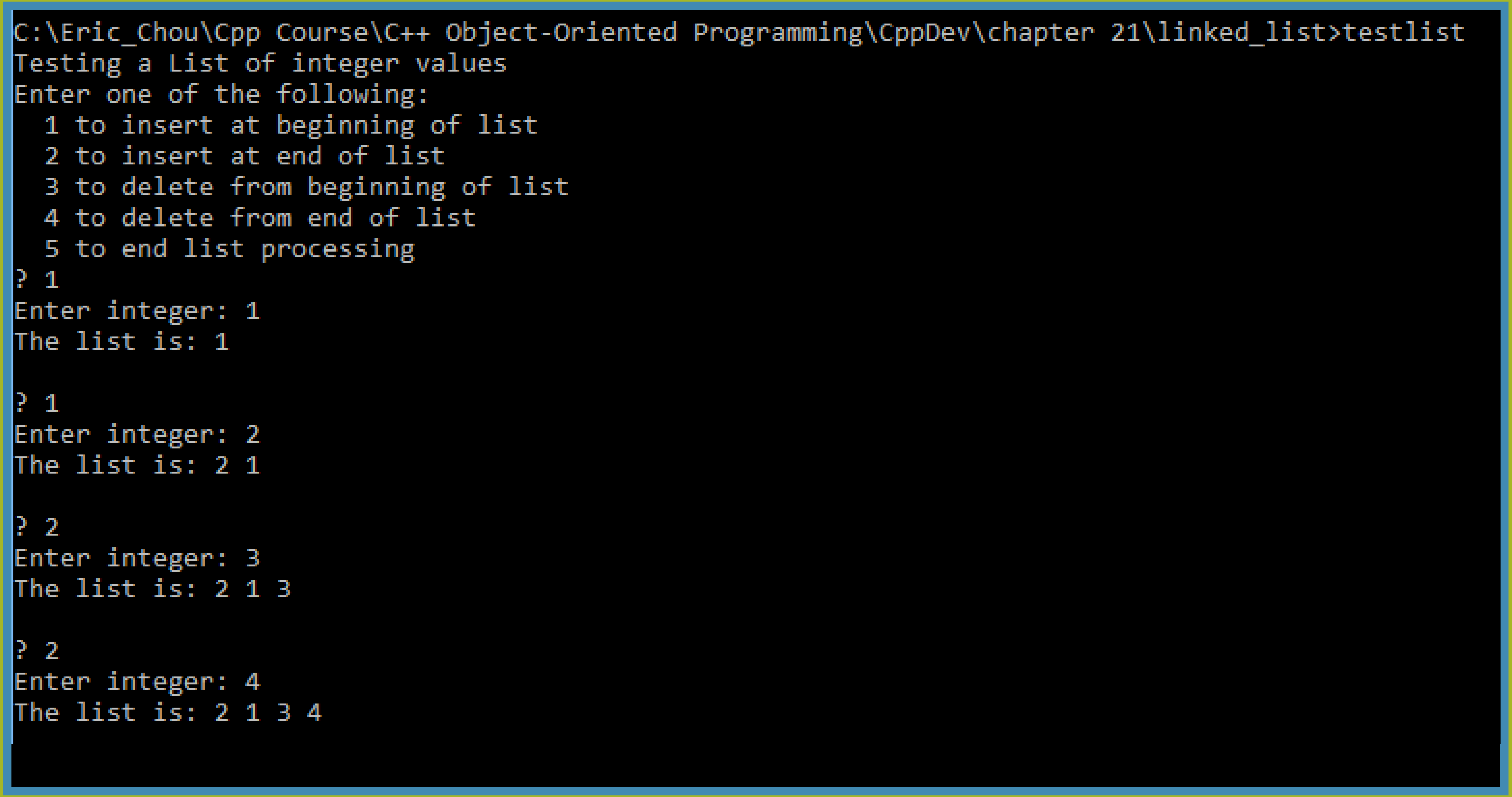


```cpp
#include <iostream>
#include <string>
using namespace std;

#include "list.h"  // List class definition

// display program instructions to user
void instructions(){
   cout << "Enter one of the following:\n"
         << "  1 to insert at beginning of list\n"
         << "  2 to insert at end of list\n"
         << "  3 to delete from beginning of list\n"
         << "  4 to delete from end of list\n"
         << "  5 to end list processing\n";
} // end function instructions

int main(){
   // test List of int values
   List< int > integerList;
   testList( integerList, "integer" );
   // test List of double values
   List< double > doubleList;
   testList( doubleList, "double" );
   return 0;
} // end main
```

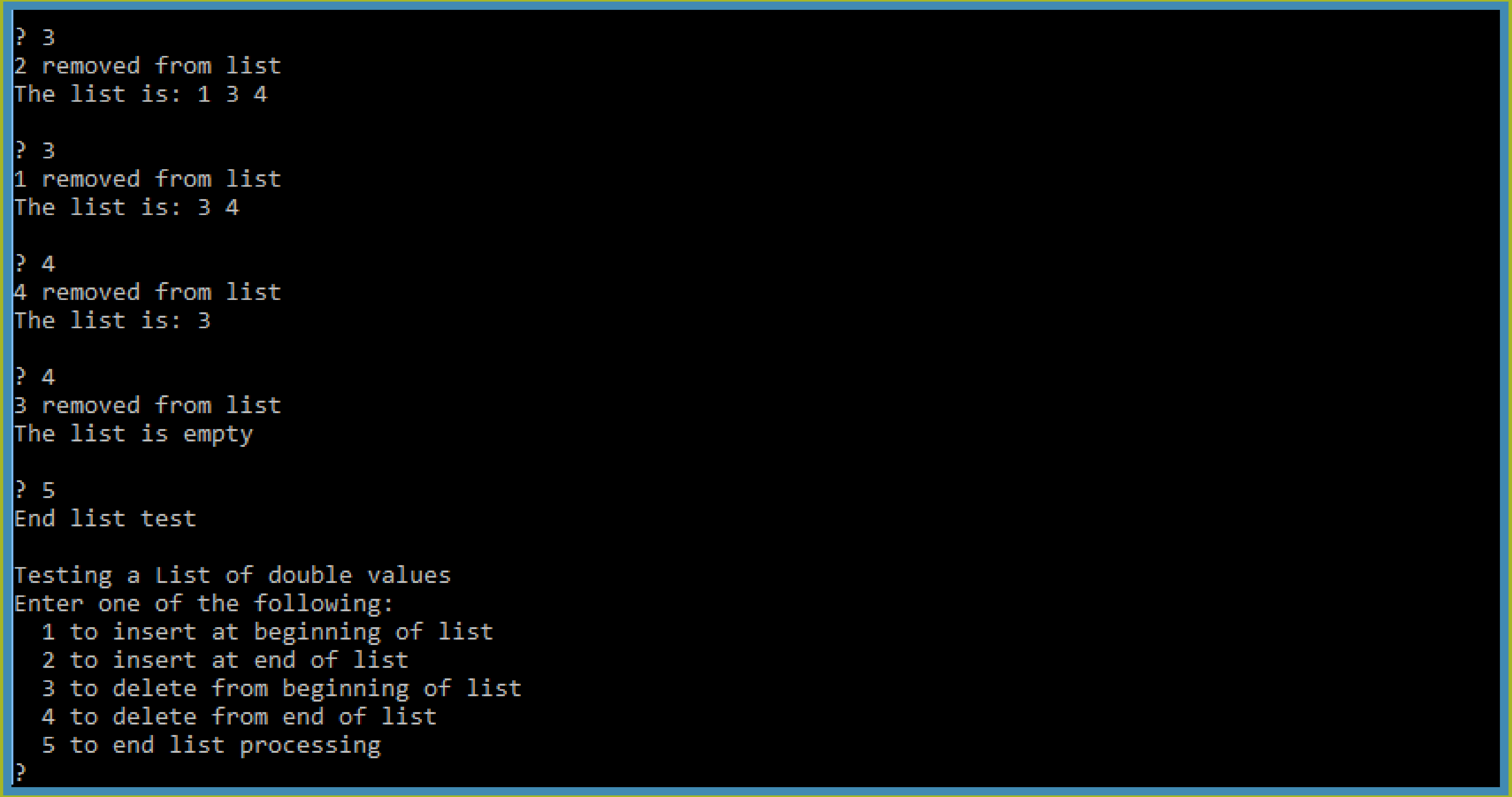


```cpp
// function to test a List
template< class T >
void testList( List< T > &listObject, const string &typeName ){
   cout << "Testing a List of " << typeName << " values\n";
   instructions();  // display instructions
   int choice;
   T value;
   do {
      cout << "? ";
      cin >> choice;
      switch ( choice ) {
         case 1:
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
            break;
         case 2:
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3:
            if ( listObject.removeFromFront( value ) ) cout << value << " removed from list\n";
            listObject.print();
            break;
         case 4:
            if ( listObject.removeFromBack( value ) ) cout << value << " removed from list\n";
            listObject.print();
            break;
      } // end switch
   } while ( choice != 5 );  // end do/while
   cout << "End list test\n\n";
} // end function testList
```

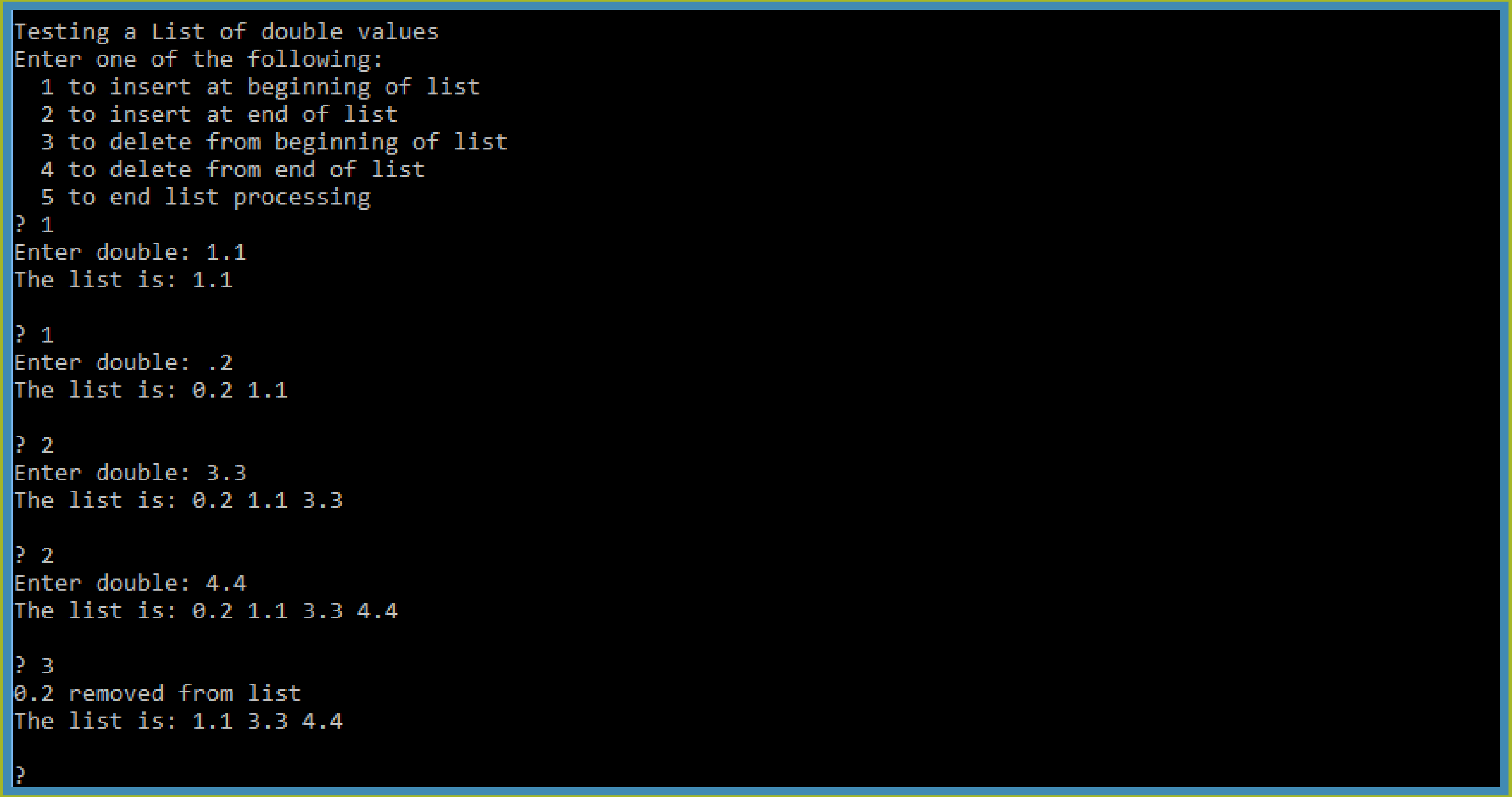
eC Learning Channel



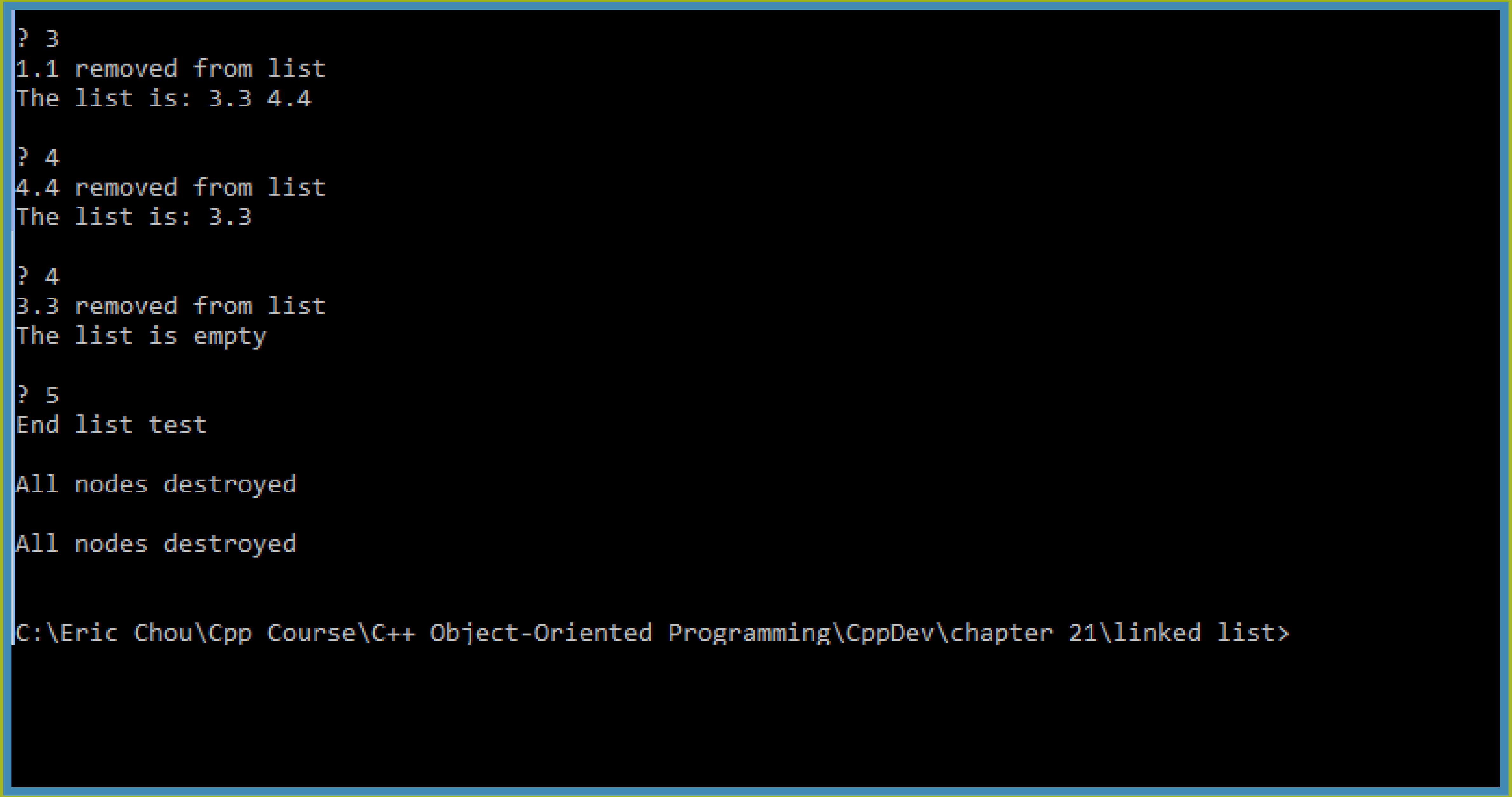
```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 21\linked_list>testlist
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4
```



```
? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
?
```



```
Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: .2
The list is: 0.2 1.1

? 2
Enter double: 3.3
The list is: 0.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 0.2 1.1 3.3 4.4

? 3
0.2 removed from list
The list is: 1.1 3.3 4.4

?
```



```
? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed


C:\Eric Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 21\linked list>
```

Linked Lists

Types of linked lists

- Singly linked list (used in example)
  - Pointer to first node
  - Travel in one direction (null-terminated)
- Circular, singly-linked
  - As above, but last node points to first
- Doubly-linked list
  - Each node has a forward and backwards pointer
  - Travel forward or backward
  - Last node null-terminated
- Circular, double-linked
  - As above, but first and last node joined

LECTURE 1

Stack

Stacks

Stack

- Nodes can be added/removed from top
  - Constrained version of linked list
  - Like a stack of plates
- Last-in, first-out (LIFO) data structure
- Bottom of stack has null link

Stack operations

- Push: add node to top
- Pop: remove node from top
  - Stores value in reference variable

Stacks

Stack applications

- Function calls: know how to return to caller
  - Return address pushed on stack
  - Most recent function call on top
  - If function A calls B which calls C:
- Used to store automatic variables
  - Popped of stack when no longer needed
- Used by compilers
  - Example in the exercises in book

		C		
	B	B	B	
A	A	A	A	A

Stacks

Upcoming program

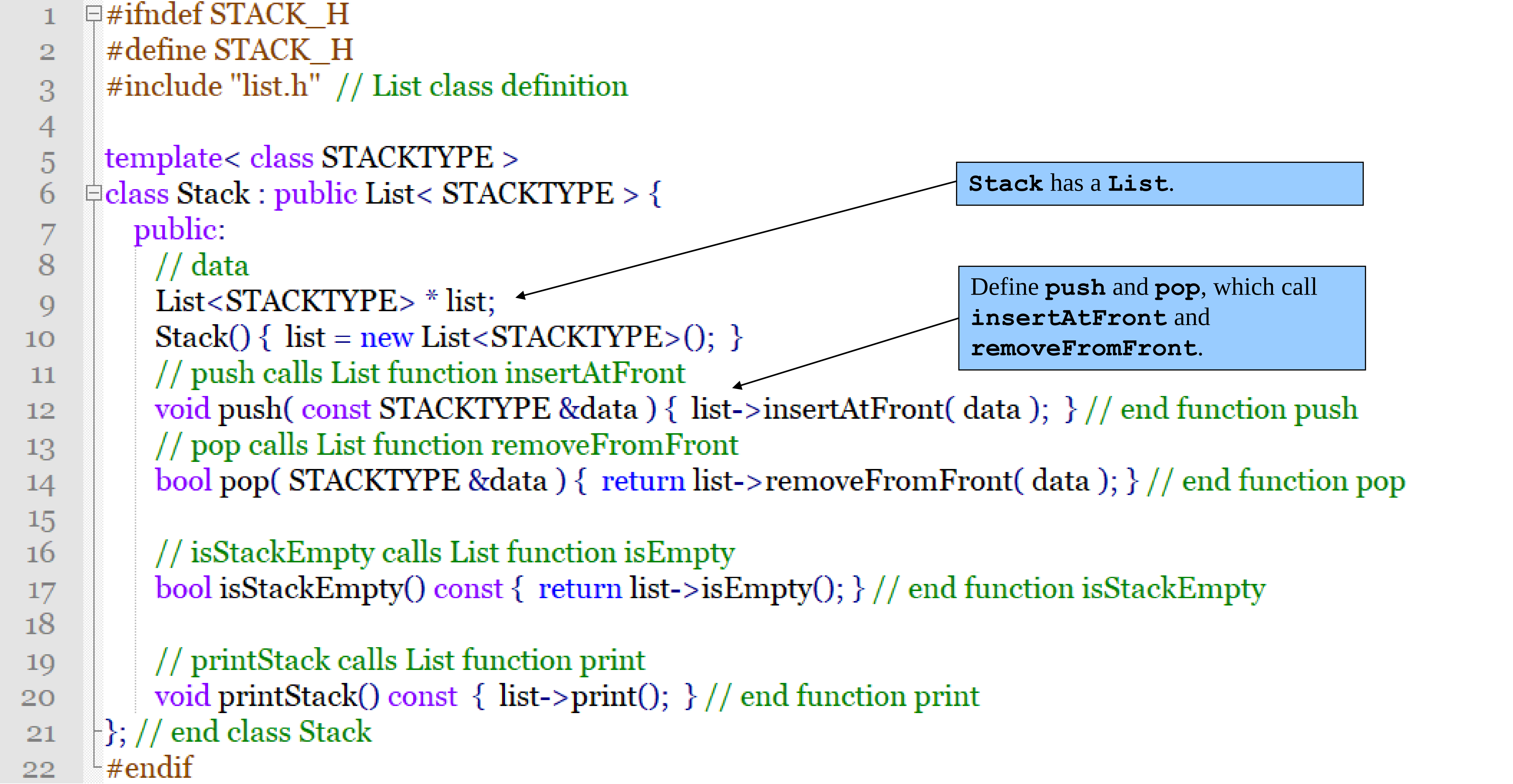
- Create stack from list
  - **`insertAtFront`**, **`removeFromFront`**
- Software reusability
  - Inheritance
    - Stack inherits from **`List`**
  - Composition
    - Stack contains a private **`List`** object
    - Performs operations on that object
- Makes stack implementation simple

Custom Design Stack using Linked-List

Demo Program: teststack.cpp+stack.h+listnode.h+list.h

Go Notepad++!!!

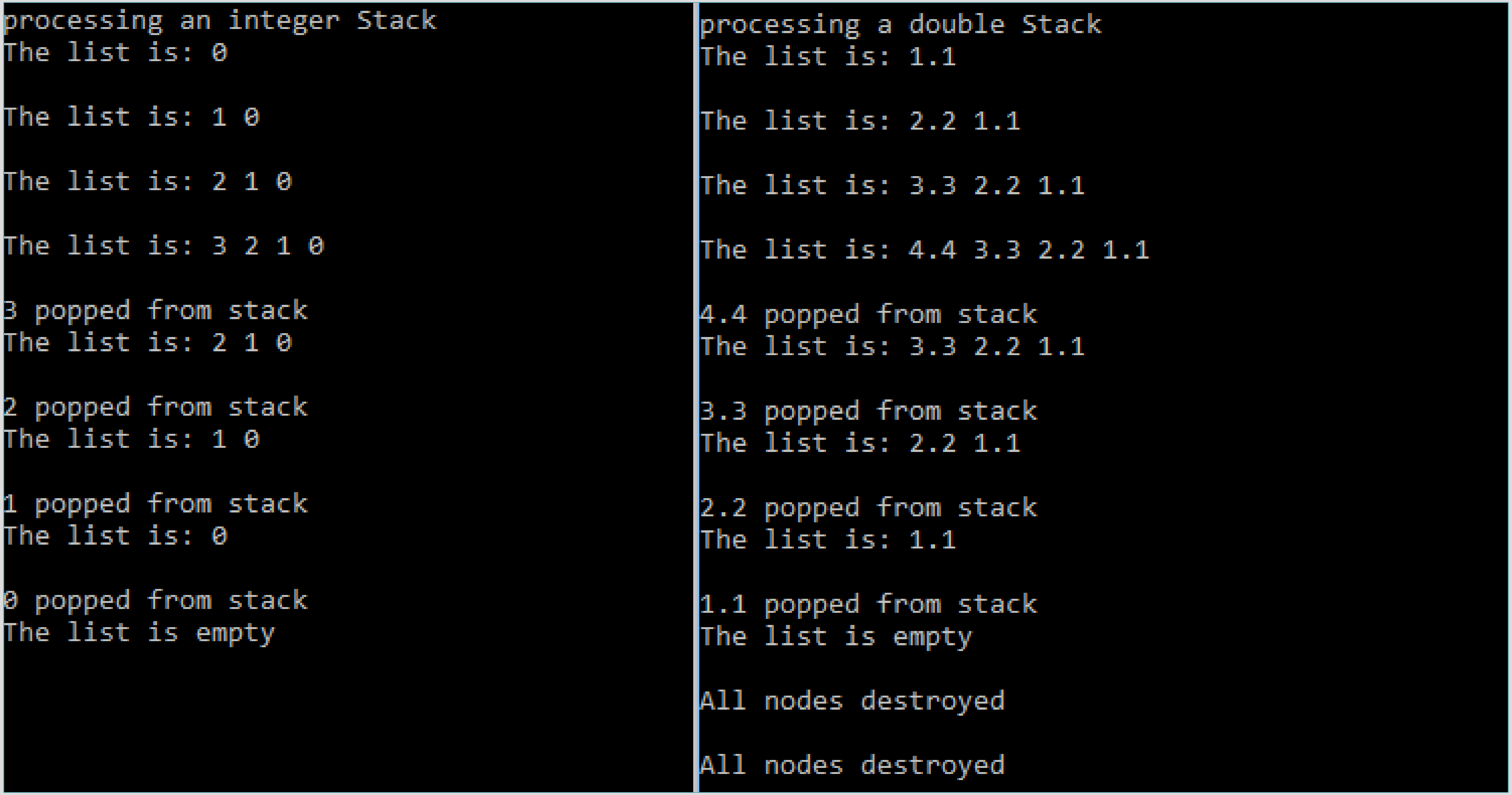
1. Custom-designed generic list node.
2. Generic linked-list.
3. Generic stack class has a List object (using **has_A** relationship instead of inheritance).
4. Menu-driven stack test fixture.



```
#ifndef STACK_H
#define STACK_H
#include "list.h"  // List class definition

template< class STACKTYPE >
class Stack : public List< STACKTYPE > {
  public:
    // data
    List<STACKTYPE> * list;
    Stack() {  list = new List<STACKTYPE>();  }
    // push calls List function insertAtFront
    void push( const STACKTYPE &data ) {  list->insertAtFront( data );  } // end function push
    // pop calls List function removeFromFront
    bool pop( STACKTYPE &data ) {  return list->removeFromFront( data ); } // end function pop

    // isStackEmpty calls List function isEmpty
    bool isStackEmpty() const {  return list->isEmpty(); } // end function isStackEmpty

    // printStack calls List function print
    void printStack() const  {  list->print();  } // end function print
}; // end class Stack
#endif
```

Stack has a **List**.

Define **push** and **pop**, which call **insertAtFront** and **removeFromFront**.



```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

The list is: 3 2 1 0

3 popped from stack
The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty
```

```
processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

The list is: 4.4 3.3 2.2 1.1

4.4 popped from stack
The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

LECTURE 1

Queue

Queues

Queue

- Like waiting in line
- Nodes added to back (*tail*), removed from front (*head*)
- First-in, first-out (FIFO) data structure
- Insert/remove called enqueue/dequeue

Applications

- Print spooling
  - Documents wait in queue until printer available
- Packets on network
- File requests from server

Queues

Upcoming program

- Queue implementation
- Reuse **`List`** as before
  - **`insertAtBack`** (**enqueue**)
  - **`removeFromFront`** (**dequeue**)

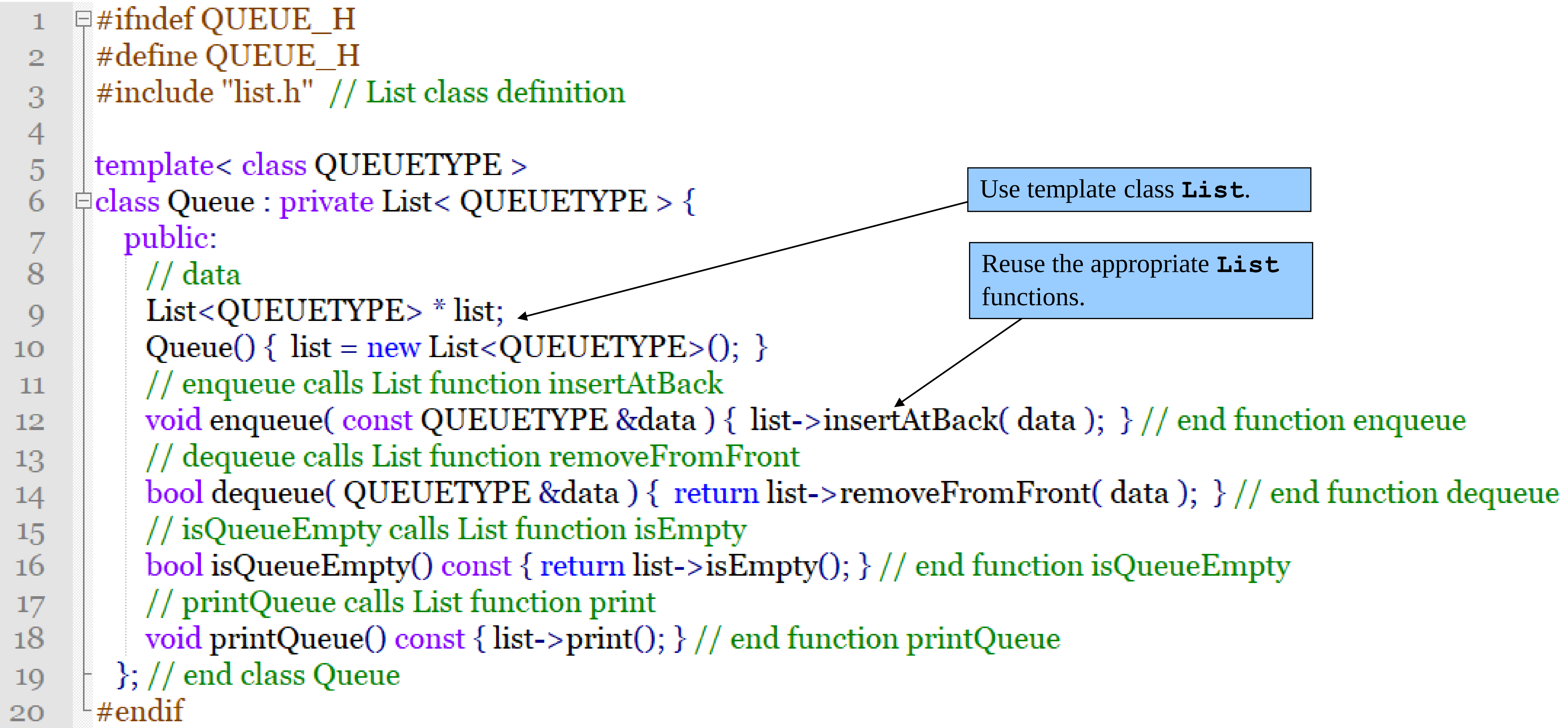
Custom Design Queue using Linked-List

Demo Program: testqueue.cpp+queue.h+listnode.h+list.h

Go Notepad++!!!

1. Custom-designed generic list node.
2. Generic linked-list.
3. Generic Queue class has a List object (using **has_A** relationship instead of inheritance).
4. Menu-driven Queue test fixture.

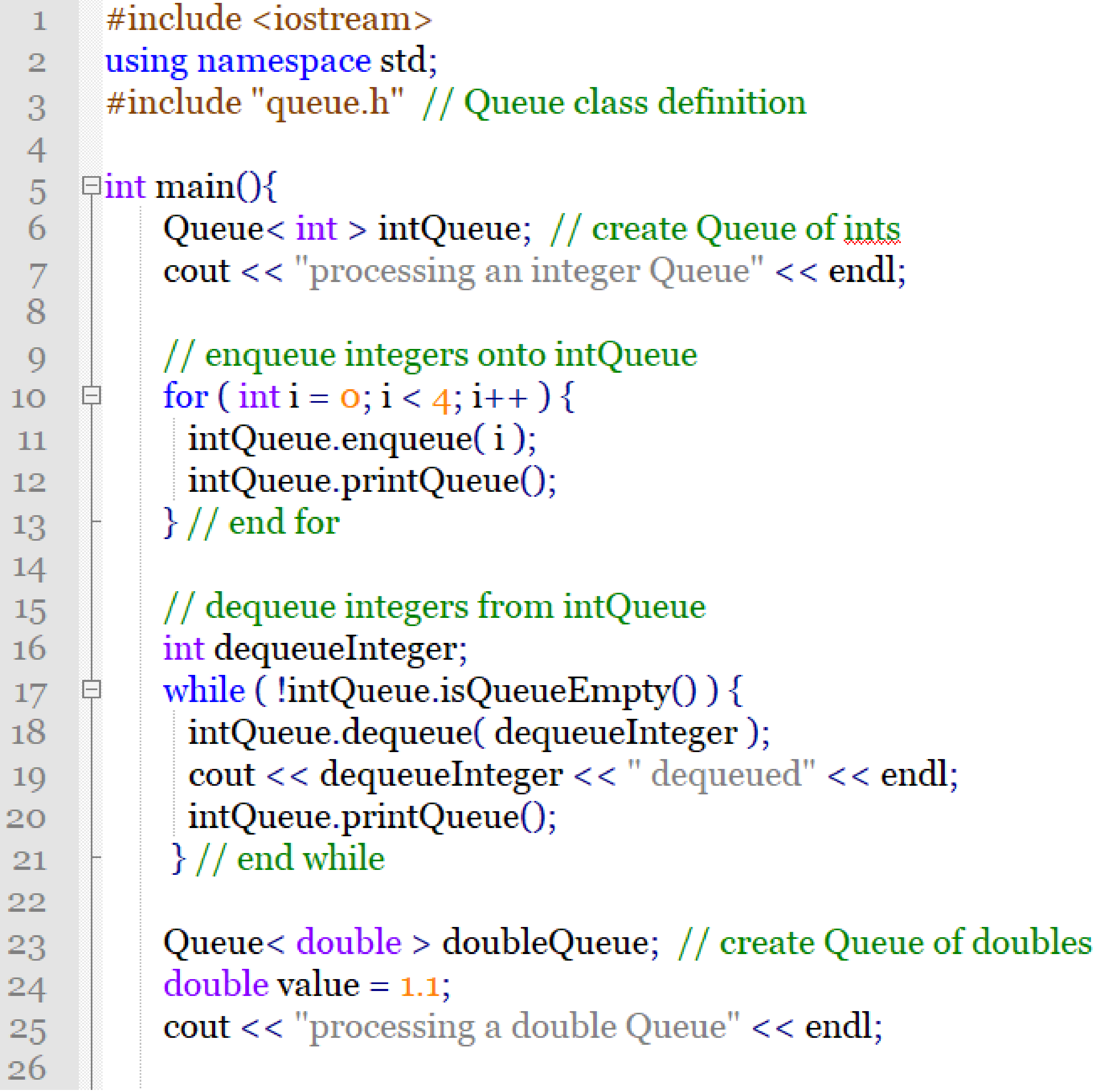
Custom Design Queue Class



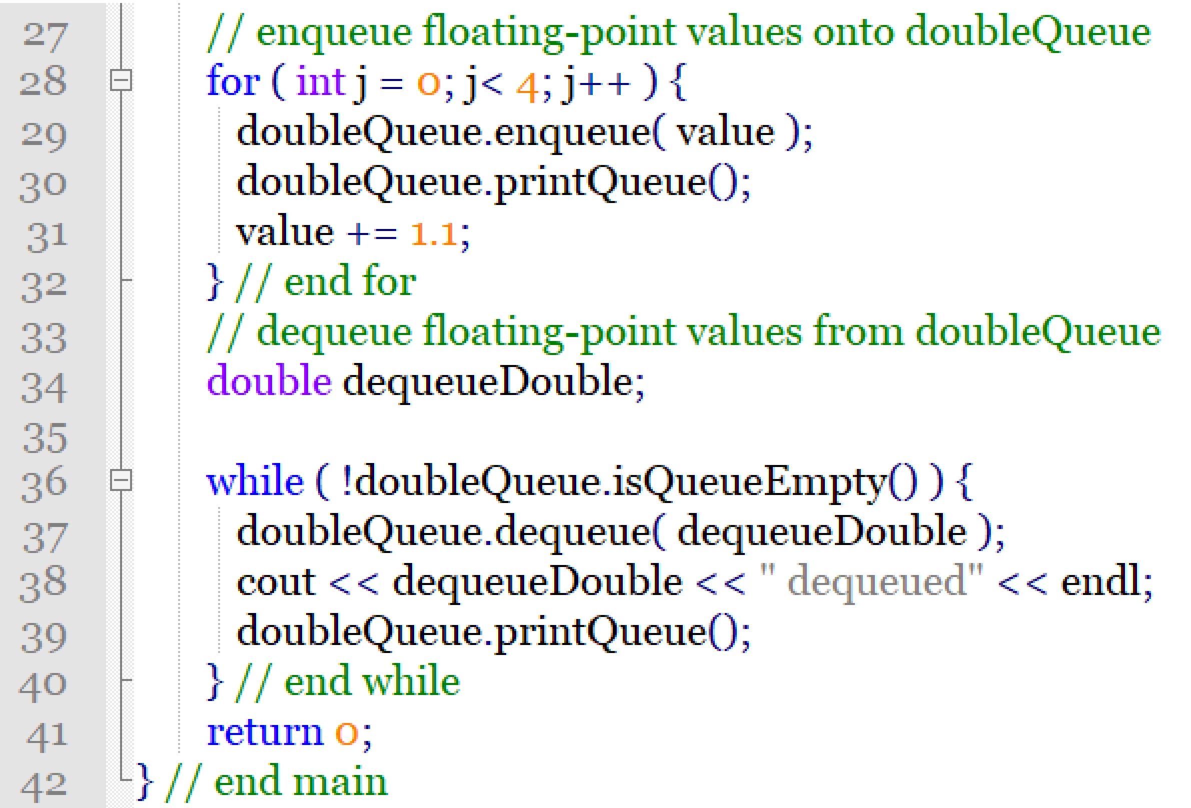
```
#ifndef QUEUE_H
#define QUEUE_H
#include "list.h"  // List class definition

template< class QUEUETYPE >
class Queue : private List< QUEUETYPE > {
  public:
    // data
    List<QUEUETYPE> * list;
    Queue() {  list = new List<QUEUETYPE>();  }
    // enqueue calls List function insertAtBack
    void enqueue( const QUEUETYPE &data ) {  list->insertAtBack( data );  } // end function enqueue
    // dequeue calls List function removeFromFront
    bool dequeue( QUEUETYPE &data ) {  return list->removeFromFront( data );  } // end function dequeue
    // isQueueEmpty calls List function isEmpty
    bool isQueueEmpty() const { return list->isEmpty(); } // end function isQueueEmpty
    // printQueue calls List function print
    void printQueue() const { list->print(); } // end function printQueue
 }; // end class Queue
#endif
```

Use template class **`List`**.

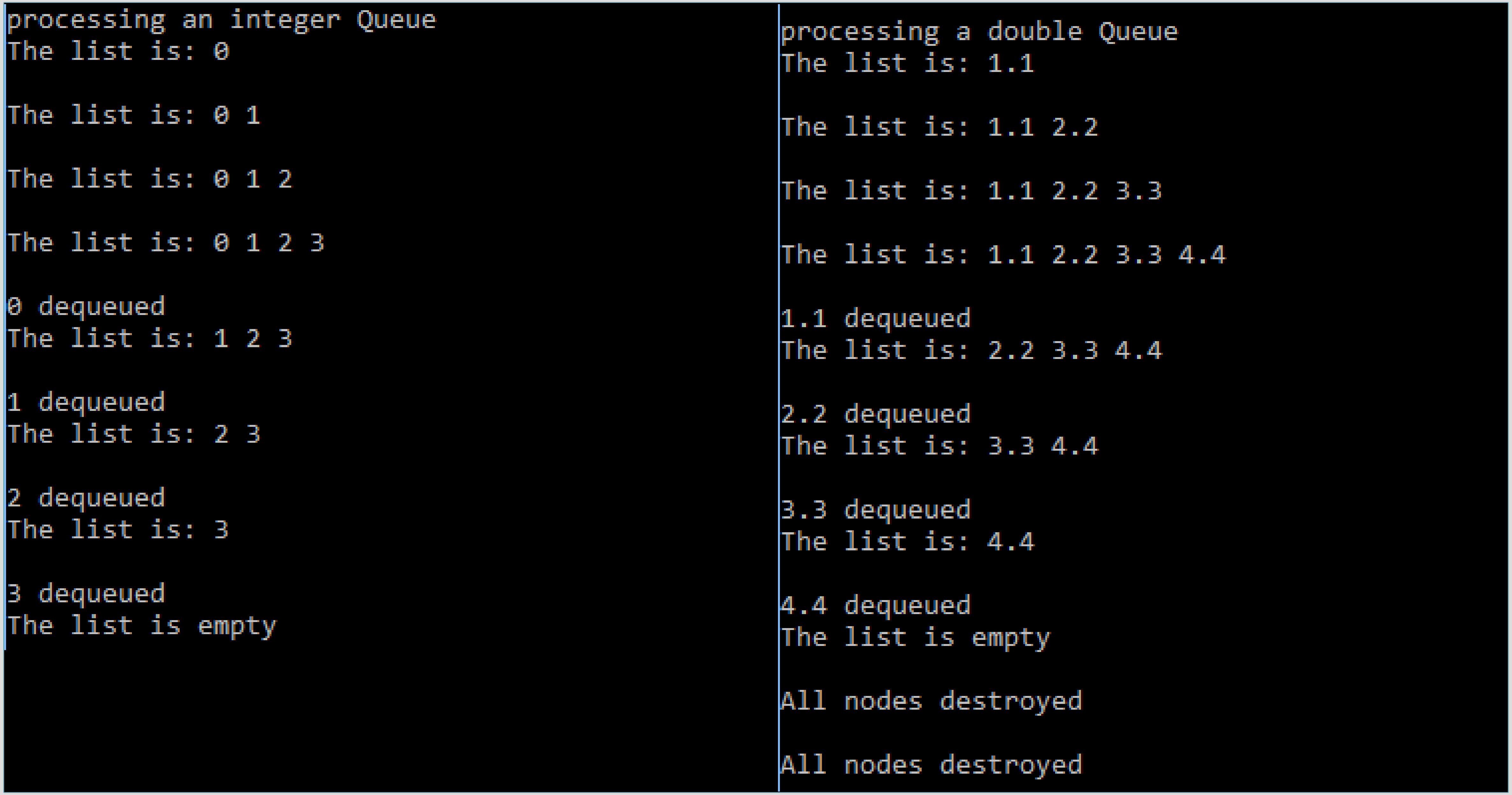
Reuse the appropriate **`List`** functions.



```cpp
#include <iostream>
using namespace std;
#include "queue.h"  // Queue class definition

int main(){
    Queue< int > intQueue;  // create Queue of ints
    cout << "processing an integer Queue" << endl;

    // enqueue integers onto intQueue
    for ( int i = 0; i < 4; i++ ) {
      intQueue.enqueue( i );
      intQueue.printQueue();
    } // end for

    // dequeue integers from intQueue
    int dequeueInteger;
    while ( !intQueue.isQueueEmpty() ) {
      intQueue.dequeue( dequeueInteger );
      cout << dequeueInteger << " dequeued" << endl;
      intQueue.printQueue();
     } // end while

    Queue< double > doubleQueue;  // create Queue of doubles
    double value = 1.1;
    cout << "processing a double Queue" << endl;

```



```cpp
    // enqueue floating-point values onto doubleQueue
    for ( int j = 0; j< 4; j++ ) {
      doubleQueue.enqueue( value );
      doubleQueue.printQueue();
      value += 1.1;
    } // end for
    // dequeue floating-point values from doubleQueue
    double dequeueDouble;

    while ( !doubleQueue.isQueueEmpty() ) {
      doubleQueue.dequeue( dequeueDouble );
      cout << dequeueDouble << " dequeued" << endl;
      doubleQueue.printQueue();
    } // end while
    return 0;
} // end main
```



```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

The list is: 0 1 2 3

0 dequeued
The list is: 1 2 3

1 dequeued
The list is: 2 3

2 dequeued
The list is: 3

3 dequeued
The list is empty
```

```
processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

The list is: 1.1 2.2 3.3 4.4

1.1 dequeued
The list is: 2.2 3.3 4.4

2.2 dequeued
The list is: 3.3 4.4

3.3 dequeued
The list is: 4.4

4.4 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

LECTURE 1

Tree

Trees

Linear data structures

- Lists, queues, stacks

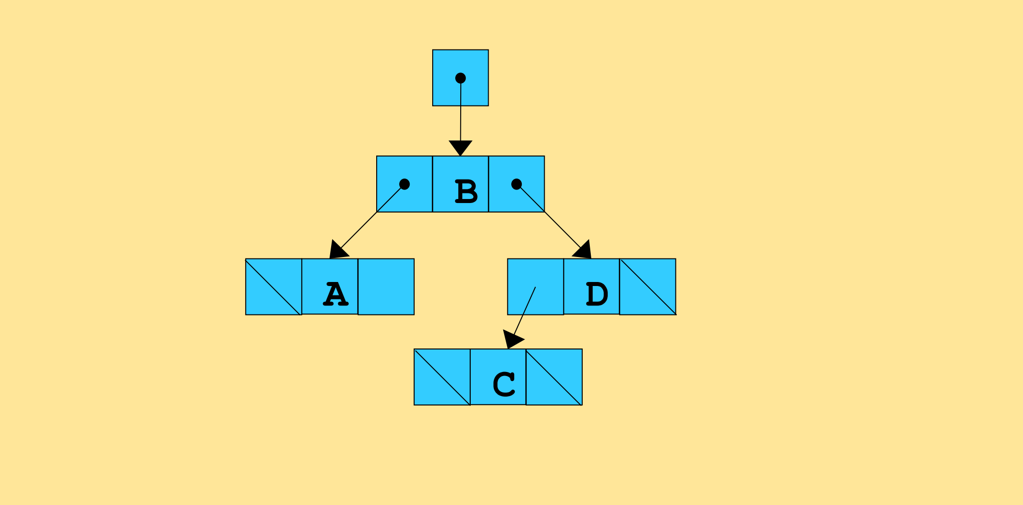
Trees

- Nonlinear, two-dimensional
- Tree nodes have 2 or more links
- Binary trees have exactly 2 links/node
  - None, both, or one link can be null

Trees

Terminology

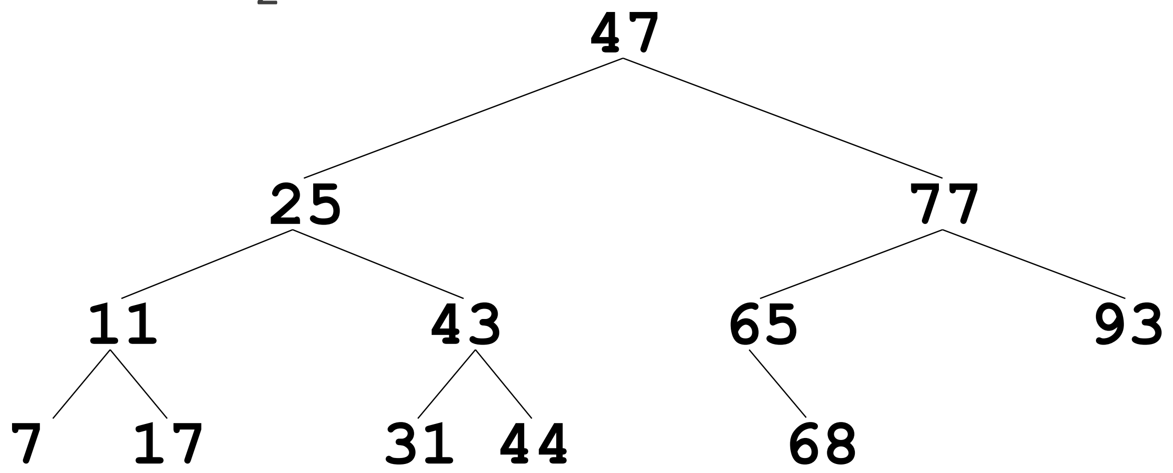
- *Root node*: first node on tree
- Link refers to *child* of node
  - Left child is root of *left subtree*
  - Right child is root of *right subtree*
- *Leaf node*: node with no children
- Trees drawn from root downwards



Trees

Binary search tree

- Values in left subtree less than parent node
- Values in right subtree greater than parent
    - Does not allow duplicate values (good way to remove them)
- Fast searches, $\log_2 n$ comparisons for a balanced tree



Trees

Inserting nodes

- Use recursive function
- Begin at root
- If current node empty, insert new node here (base case)
- Otherwise,
  - If value > node, insert into right subtree
  - If value < node, insert into left subtree
  - If neither > nor <, must be =
    - Ignore duplicate

Trees

Tree traversals

- In-order (print tree values from least to greatest)
  - Traverse left subtree (call function again)
  - Print node
  - Traverse right subtree
- Preorder
  - Print node
  - Traverse left subtree
  - Traverse right subtree
- Postorder
  - Traverse left subtree
  - Traverse rigth subtree
  - Print node

Trees

Upcoming program

- Create 2 template classes
- `TreeNode`
  - `data`
  - `leftPtr`
  - `rightPtr`
- `Tree`
  - `rootPtr`
  - Functions
    - `InsertNode`
    - `inOrderTraversal`
    - `preOrderTraversal`
    - `postOrderTraversal`

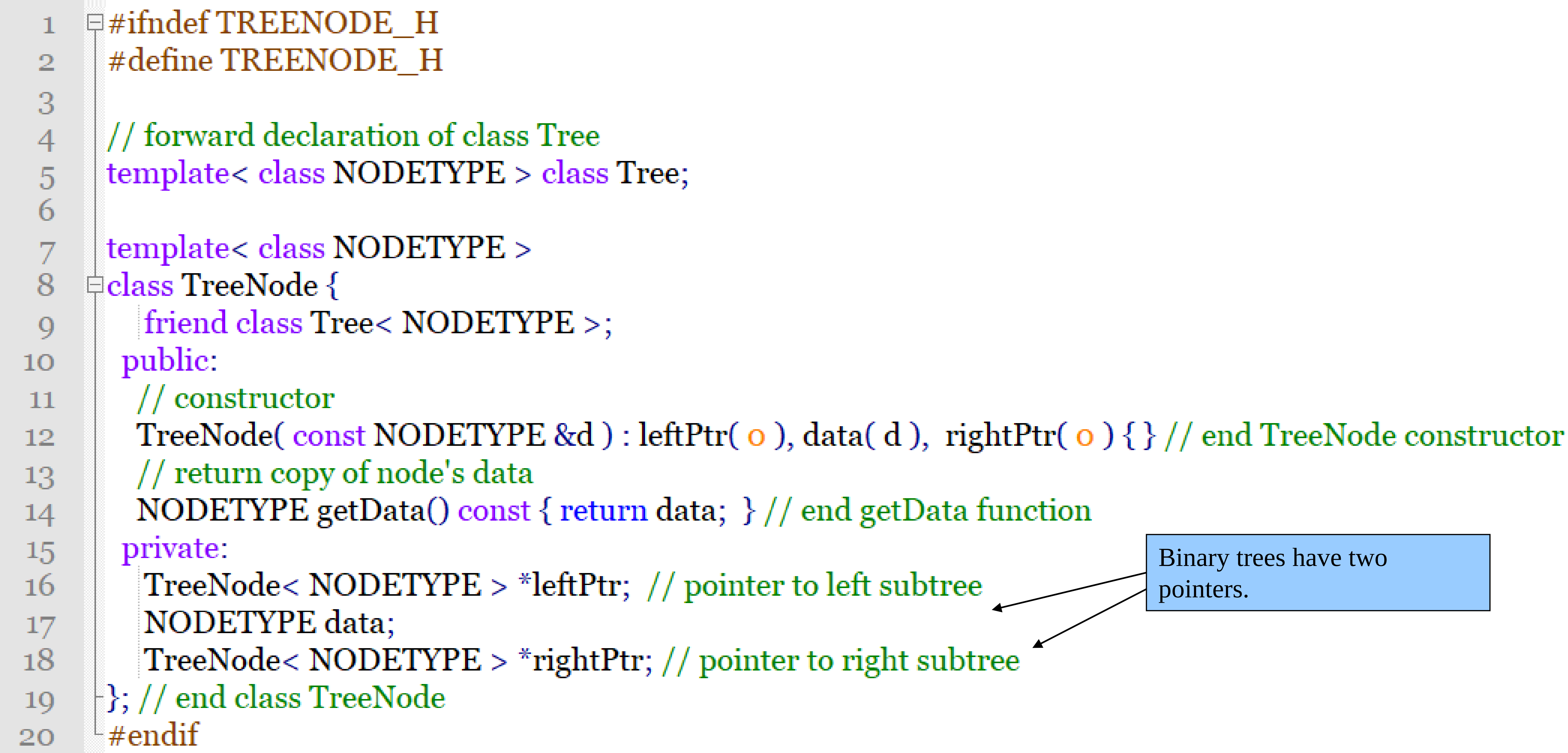
Custom Design Tree using Treenode

Demo Program: testtree.cpp+treenode.h+tree.h

Go Notepad++!!!

1. Custom-designed generic tree node.
2. Generic tree class.
3. Menu-driven Tree test fixture.

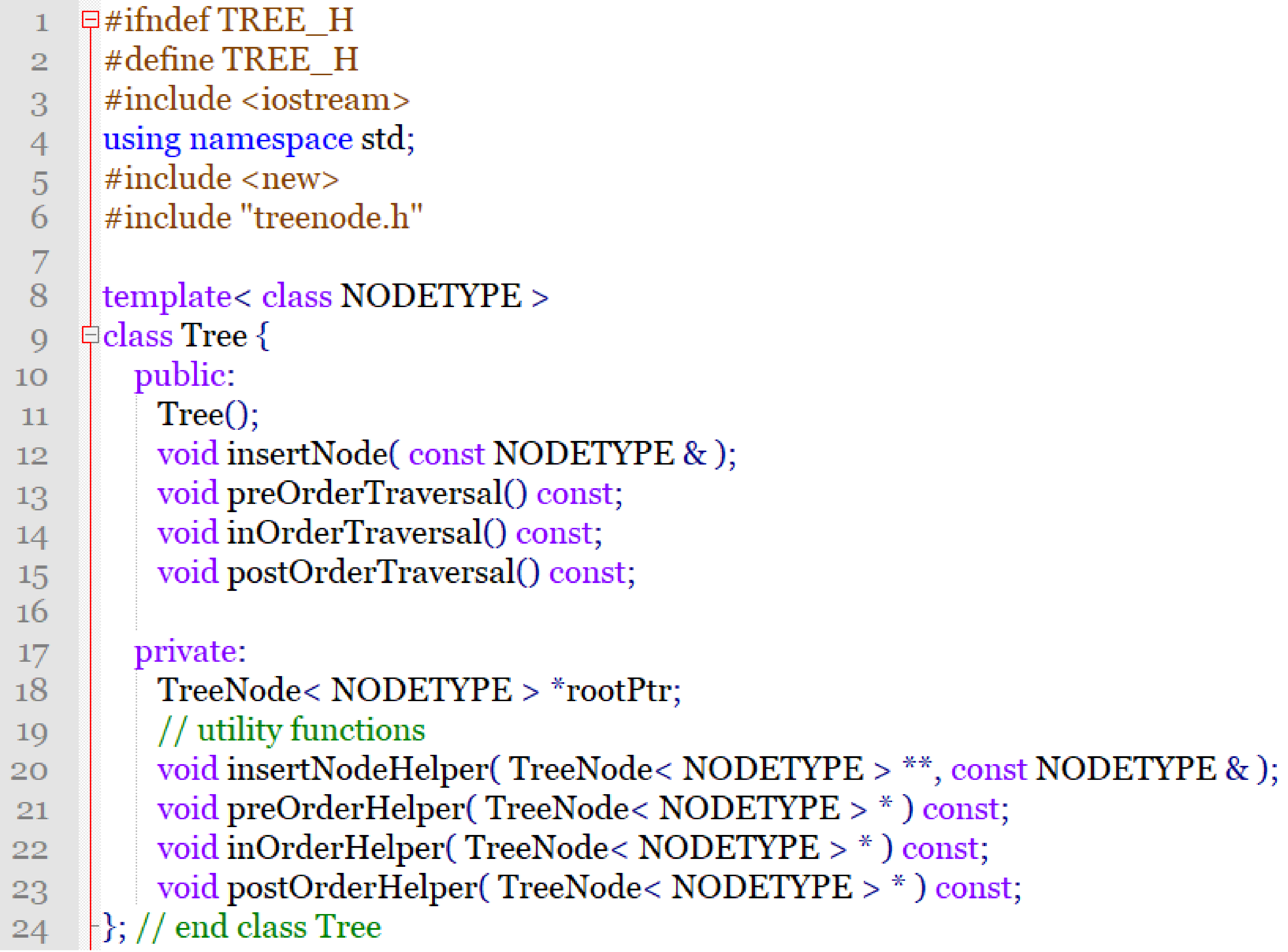
treenode.h



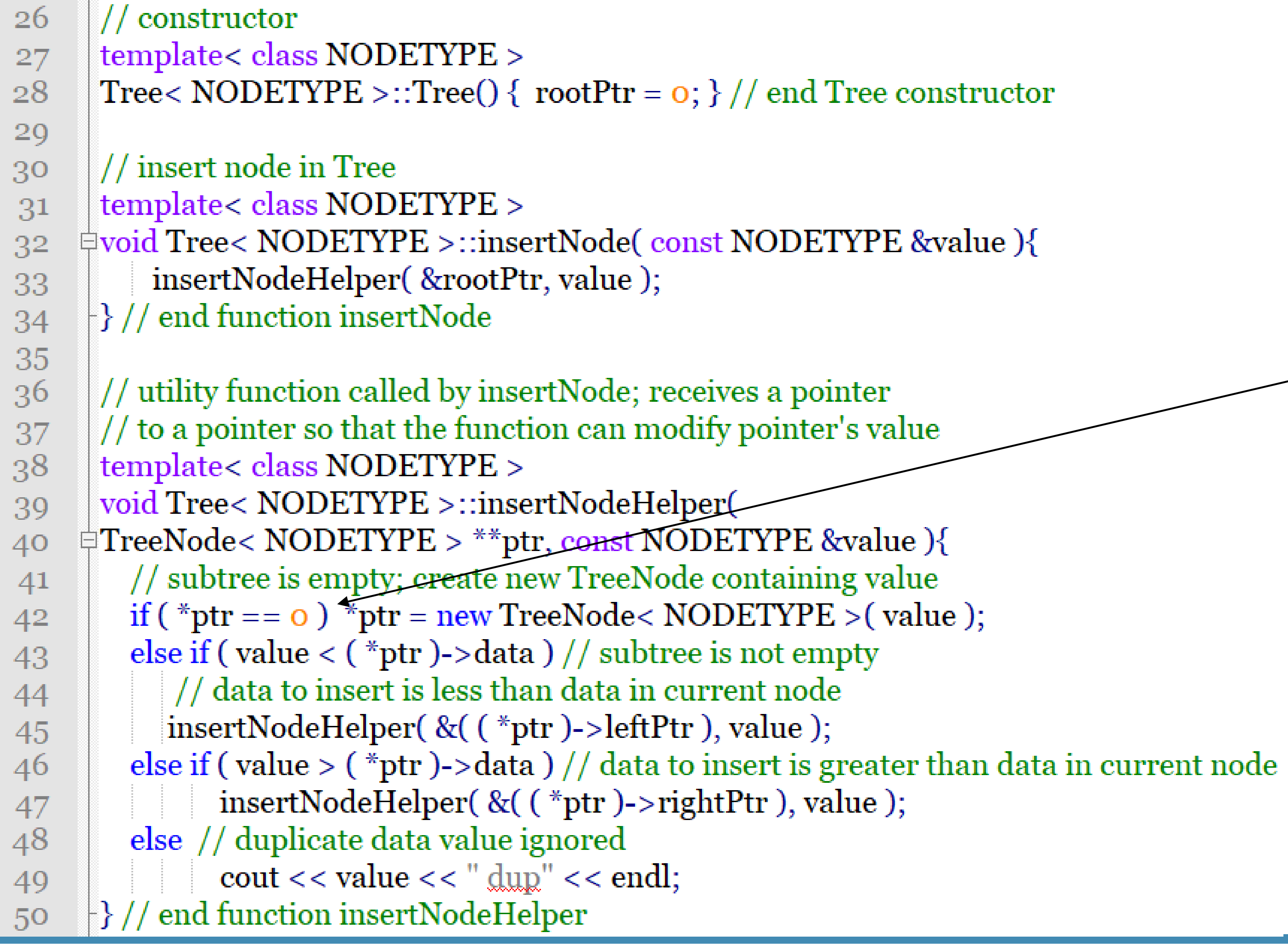
```
#ifndef TREENODE_H
#define TREENODE_H

// forward declaration of class Tree
template< class NODETYPE > class Tree;

template< class NODETYPE >
class TreeNode {
    friend class Tree< NODETYPE >;
  public:
    // constructor
    TreeNode( const NODETYPE &d ) : leftPtr( 0 ), data( d ),  rightPtr( 0 ) { } // end TreeNode constructor
    // return copy of node's data
    NODETYPE getData() const { return data;  } // end getData function
  private:
    TreeNode< NODETYPE > *leftPtr;  // pointer to left subtree
    NODETYPE data;
    TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
}; // end class TreeNode
#endif
```

Binary trees have two pointers.

tree.h



```
#ifndef TREE_H
#define TREE_H
#include <iostream>
using namespace std;
#include <new>
#include "treenode.h"

template< class NODETYPE >
class Tree {
  public:
    Tree();
    void insertNode( const NODETYPE & );
    void preOrderTraversal() const;
    void inOrderTraversal() const;
    void postOrderTraversal() const;

  private:
    TreeNode< NODETYPE > *rootPtr;
    // utility functions
    void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE & );
    void preOrderHelper( TreeNode< NODETYPE > * ) const;
    void inOrderHelper( TreeNode< NODETYPE > * ) const;
    void postOrderHelper( TreeNode< NODETYPE > * ) const;
}; // end class Tree
```



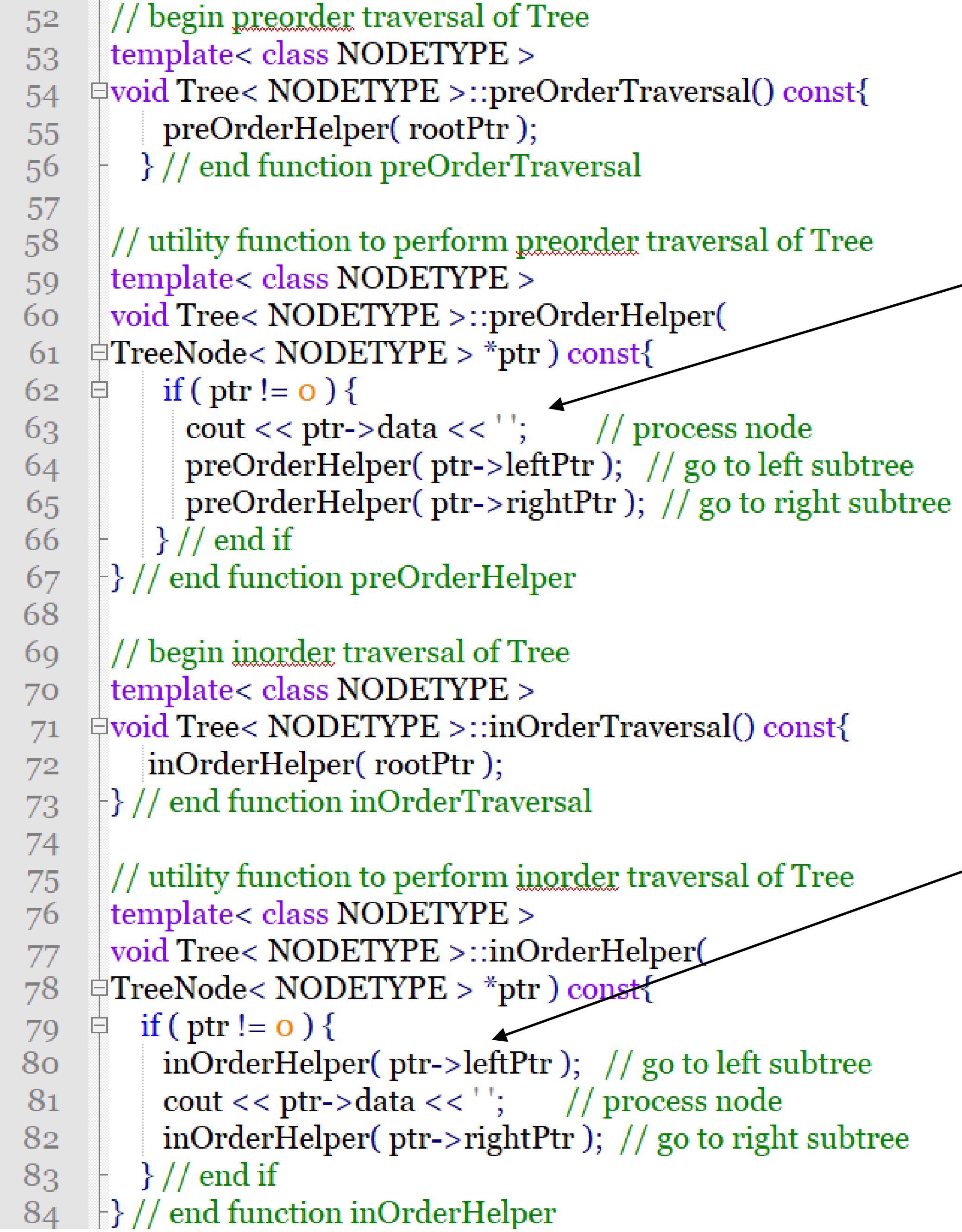
```cpp
// constructor
template< class NODETYPE >
Tree< NODETYPE >::Tree() {  rootPtr = 0; } // end Tree constructor

// insert node in Tree
template< class NODETYPE >
void Tree< NODETYPE >::insertNode( const NODETYPE &value ){
    insertNodeHelper( &rootPtr, value );
} // end function insertNode

// utility function called by insertNode; receives a pointer
// to a pointer so that the function can modify pointer's value
template< class NODETYPE >
void Tree< NODETYPE >::insertNodeHelper(
TreeNode< NODETYPE > **ptr, const NODETYPE &value ){
  // subtree is empty; create new TreeNode containing value
  if ( *ptr == 0 )  *ptr = new TreeNode< NODETYPE >( value );
  else if ( value < ( *ptr )->data ) // subtree is not empty
     // data to insert is less than data in current node
    insertNodeHelper( &( ( *ptr )->leftPtr ), value );
  else if ( value > ( *ptr )->data ) // data to insert is greater than data in current node
        insertNodeHelper( &( ( *ptr )->rightPtr ), value );
  else  // duplicate data value ignored
        cout << value << " dup" << endl;
} // end function insertNodeHelper
```

Recursive function to insert a new node. If the current node is empty, insert the new node here.

If new value greater than current node (**`ptr`**), insert into right subtree.

If less, insert into left subtree.

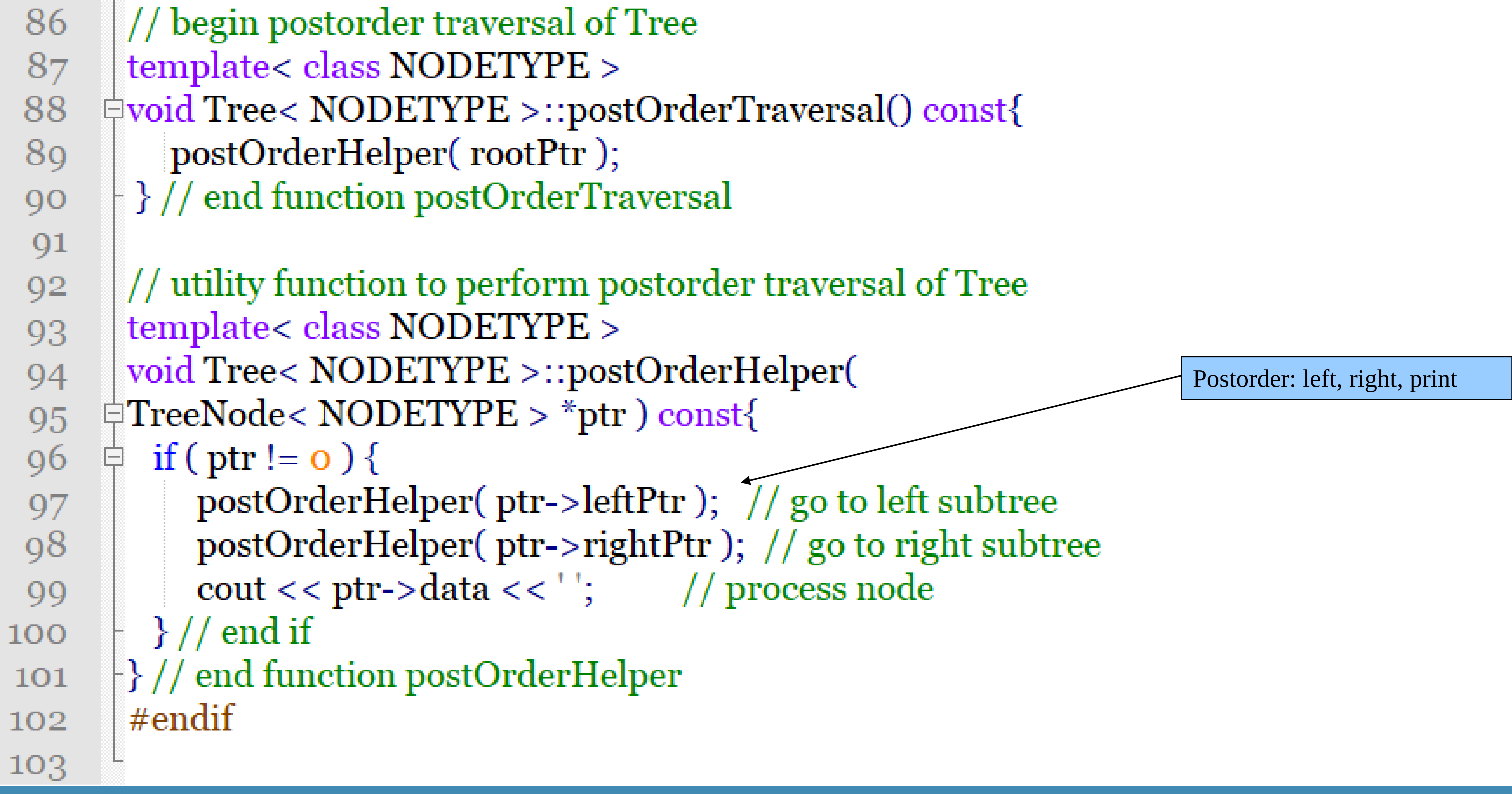
If neither case applies, node is a duplicate -- ignore.



```
// begin preorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::preOrderTraversal() const{
     preOrderHelper( rootPtr );
  } // end function preOrderTraversal

// utility function to perform preorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::preOrderHelper(
TreeNode< NODETYPE > *ptr ) const{
     if ( ptr != 0 ) {
       cout << ptr->data << ' ';        // process node
       preOrderHelper( ptr->leftPtr );  // go to left subtree
       preOrderHelper( ptr->rightPtr ); // go to right subtree
     } // end if
} // end function preOrderHelper

// begin inorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const{
    inOrderHelper( rootPtr );
} // end function inOrderTraversal

// utility function to perform inorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::inOrderHelper(
TreeNode< NODETYPE > *ptr ) const{
   if ( ptr != 0 ) {
     inOrderHelper( ptr->leftPtr );  // go to left subtree
     cout << ptr->data << ' ';       // process node
     inOrderHelper( ptr->rightPtr ); // go to right subtree
   } // end if
} // end function inOrderHelper
```

Preorder: print, left, right

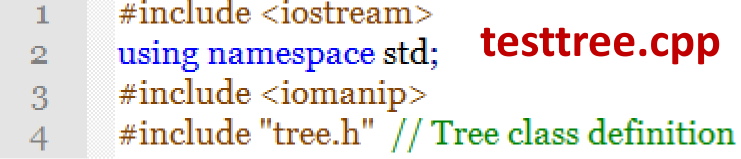
In order: left, print, right



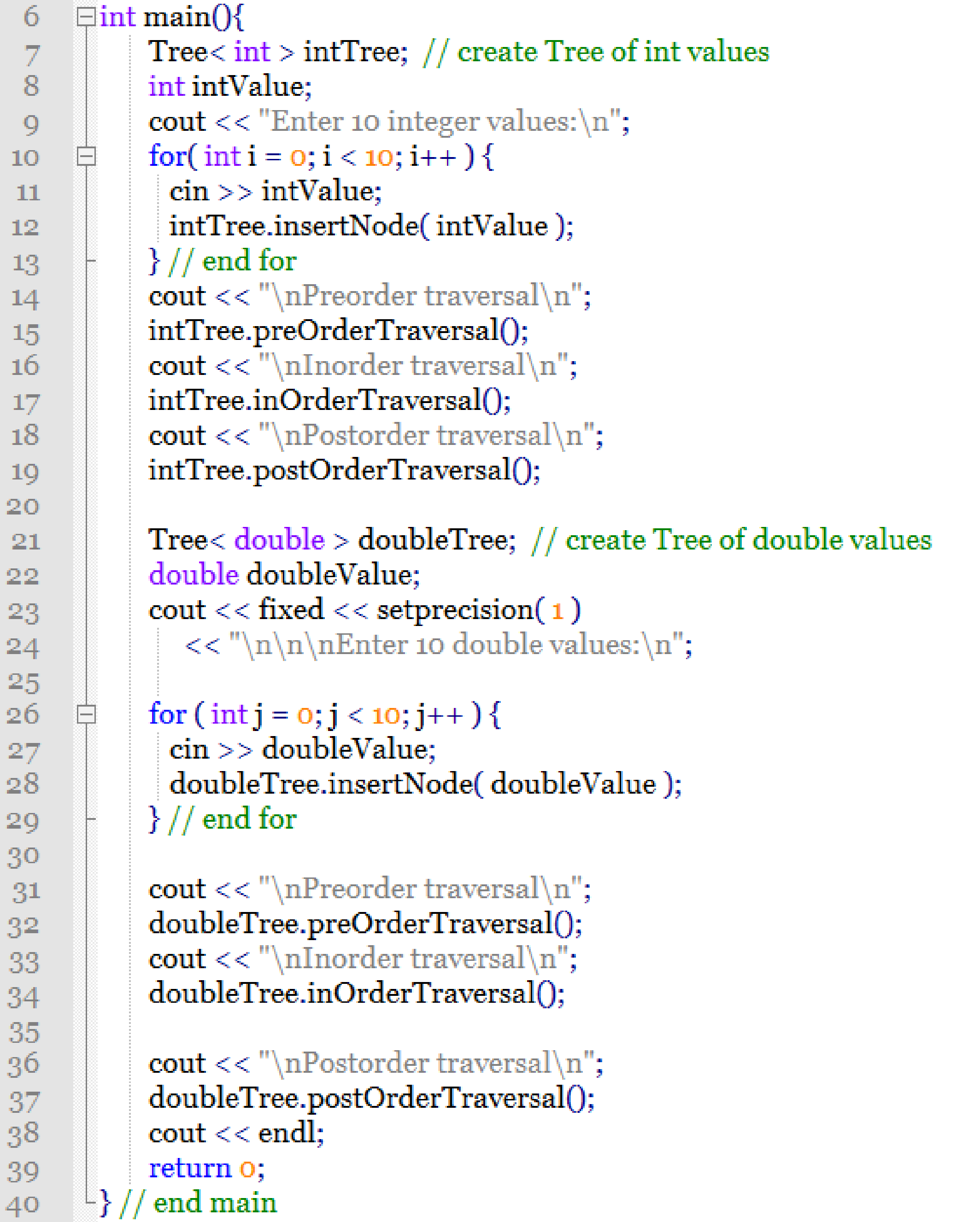
```
// begin postorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const{
    postOrderHelper( rootPtr );
 } // end function postOrderTraversal

// utility function to perform postorder traversal of Tree
template< class NODETYPE >
void Tree< NODETYPE >::postOrderHelper(
TreeNode< NODETYPE > *ptr ) const{
  if ( ptr != 0 ) {
     postOrderHelper( ptr->leftPtr );   // go to left subtree
     postOrderHelper( ptr->rightPtr );  // go to right subtree
     cout << ptr->data << ' ';         // process node
  } // end if
} // end function postOrderHelper
#endif
```

Postorder: left, right, print

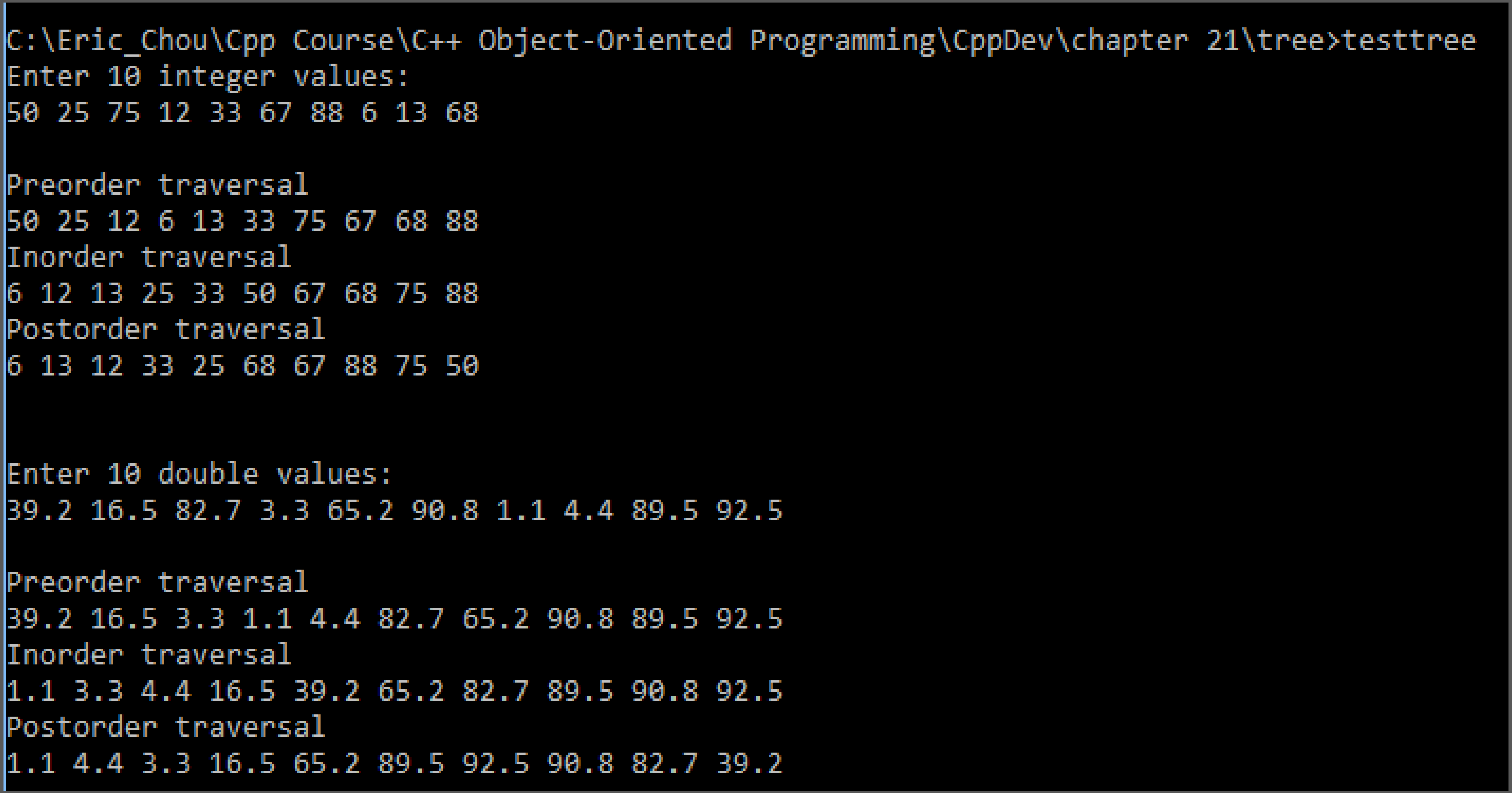
testtree.cpp



```cpp
#include <iostream>
using namespace std;
#include <iomanip>
#include "tree.h"  // Tree class definition
```



```cpp
int main(){
    Tree< int > intTree;  // create Tree of int values
    int intValue;
    cout << "Enter 10 integer values:\n";
    for( int i = 0; i < 10; i++ ) {
      cin >> intValue;
      intTree.insertNode( intValue );
    } // end for
    cout << "\nPreorder traversal\n";
    intTree.preOrderTraversal();
    cout << "\nInorder traversal\n";
    intTree.inOrderTraversal();
    cout << "\nPostorder traversal\n";
    intTree.postOrderTraversal();

    Tree< double > doubleTree;  // create Tree of double values
    double doubleValue;
    cout << fixed << setprecision( 1 )
        << "\n\n\nEnter 10 double values:\n";

    for ( int j = 0; j < 10; j++ ) {
      cin >> doubleValue;
      doubleTree.insertNode( doubleValue );
    } // end for

    cout << "\nPreorder traversal\n";
    doubleTree.preOrderTraversal();
    cout << "\nInorder traversal\n";
    doubleTree.inOrderTraversal();

    cout << "\nPostorder traversal\n";
    doubleTree.postOrderTraversal();
    cout << endl;
    return 0;
} // end main
```



```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 21\tree>testtree
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```