IS 0020
Program Design and Software Tools

Templates

Lecture 10

March 23, 2004

Introduction

- Templates
  - Function templates
    - Specify entire range of related (overloaded) functions
    - Function-template specializations
  - Class templates
    - Specify entire range of related classes
      - Class-template specializations

Function Templates

- Overloaded functions
  - Similar operations
    - Different types of data
- Function templates
  - Identical operations
    - Different types of data
  - Single function template
    - Compiler generates separate object-code functions
  - Unlike Macros they allow Type checking

Function Templates

- Function-template definitions
  - Keyword **`template`**
  - List formal type parameters in angle brackets (**`<`** and **`>`**)
    - Each parameter preceded by keyword **`class`** or **`typename`**
      - **`class`** and **`typename`** interchangeable

```
template< class T >
template< typename ElementType >
template< class BorderType, class FillType >
```

    - Specify types of
      - Arguments to function
      - Return type of function
      - Variables within function

```
// Fig. 11.1: fig11_01.cpp
// Using template functions.
#include <iostream>

using std::cout;
using std::endl;

// function template printArray defi
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < co
      cout << array[ i ] <

   cout << endl;

} // end function print

int main()
{
   const int aCount = 5;
   const int bCount = 7;
   const int cCount = 6;

```

Function template definition; declare single formal type parameter **T**.

T is type parameter; use any valid identifier.

If **T** is user-defined type, stream-insertion operator must be overloaded for class **T**.

fig11_01.cpp
(2 of 2)

```
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, aCount );

   cout << "Array b contains

   // call double funct
   printArray( b, bCoun

   cout << "Array c con

   // call character fu
   printArray( c, cCoun

   return 0;

} // end main
```

Crea… pecialization for printing array…

```
void …, const int count )
{
   f…+ )

   c…
} // end function printArray
```

Compiler infers **`T`** is **`double`**; instantiates function-template specialization where **`T`** is

Compiler infers **`T`** is **`char`**; instantiates function-template specialization where **`T`** is **`char`**.

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```


fig11_01.cpp
output (1 of 1)

Overloading Function Templates

- Related function-template specializations
  - Same name
    - Compiler uses overloading resolution
- Function template overloading
  - Other function templates with same name
    - Different parameters
  - Non-template functions with same name
    - Different function arguments
  - Compiler performs matching process
    - Tries to find precise match of function name and argument types
    - If fails, function template
      - Generate function-template specialization with precise match

Class Templates

- Stack
  - LIFO (last-in-first-out) structure
- Class templates
  - Generic programming
  - Describe notion of stack generically
    - Instantiate type-specific version
  - Parameterized types
    - Require one or more type parameters
      - Customize “generic class” template to form class-template specialization

```
// Fig. 11.2: tstack1.h
// Stack class template.
#ifndef TSTACK1_H
#define TSTACK1_H

template< class T >
class Stack {

public:
   Stack( int = 10 );  // default constructor (stack size 10)

   // destructor
   ~Stack()
   {
      delete [] stackPtr;

   } // end ~Stack destructor

   bool push( const T& );  // push an element onto the stack
   bool pop( T& );         // pop an element off the stack

```

Specify class-template definition; type parameter **T** indicates type of **Stack** class to be created.

Function parameters of type **T**.

```
   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;

   } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {
      return top == size - 1;

   } // end function isFull

private:
   int size;     // # of elements in the stack
   int top;      // location of the top element
   T *stackPtr;  // pointer to the stack

}; // end class Stack

```

Array of elements of type **T**.

```
// constructor
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;  // Stack initially empt
   stackPtr = new T[ size ]; // alloc

} // end Stack constructor

// push element onto stack;
// if successful, return true; ot
template< class T >
bool Stack< T >::push( const T &p
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue;  // place item on Stack
      return true;  // push successful

   } // end if

   return false;  // push unsuccessful

} // end function push

```

Constructor creates array of type **`T`**. For example, compiler generates

`stackPtr = new T[ size ];`

for class-template specialization **`Stack< double >`**.

Use binary scope resolution operator (**`::`**) with class-template name (**`Stack< T >`**) to tie definition to class template's scope.

`T >`

```
// pop element off stack;
// if successful, return true; otherwise, return false
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // r
      return true;  // pop successful

   } // end if

   return false;  // pop unsuccessful

} // end function pop

#endif
```

Member function preceded with header

Use binary scope resolution operator (`::`) with class-template name (`Stack< T >`) to tie definition to class template's scope.

```cpp
// Fig. 11.3: fig11_03.cpp
// Stack-class-template test program.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"  // Stack class template definition

int main()
{
   Stack< double > doubleStack( 5 );
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleSt

   while ( doubleStack.push( doubleValue ) ) {
      cout << doubleValue << ' ';
      doubleValue += 1.1;

   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
        << "\n\nPopping elements from doubleStack\n";
```

Link to class template definition.

Instantiate object of class **`Stack< double >`**.

Invoke function **`push`** of class-template specialization **`Stack< double >`**.

fig11_03.cpp (2 of 3)

```

   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';

   cout << "\nStack is empty. Cannot pop\n

   Stack< int > intStack;
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( intValue ) ) {
      cout << intValue << ' ';
      ++intValue;

   } // end while

   cout << "\nStack is full. Cannot push " << intValue
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   return 0;
```

Invoke function **`pop`** of class-template specialization **`Stack< double >`**.

Note similarity of code for **`Stack< int >`** to code for **`Stack< double >`**.

```

} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

fig11_03.cpp
(3 of 3)

fig11_03.cpp
output (1 of 1)

```
// Fig. 11.4: fig11_04.cpp
// Stack class template test program. Function main uses a
// function template to manipulate objects of type Stack< T >.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"  // Stack class template definition

// function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,   // reference to Stack< T >
   T value,                // initial value to push
   T increment,            // increment for subsequent values
   const char *stackName ) // name of the Stack < T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) {
      cout << value << ' ';
      value += increment;

   } // end while
```

fig11_04.cpp
(1 of 2)

Function template to manipulate **`Stack< T >`** eliminates similar code from previous file for **`Stack< double >`** and **`Stack< int >`**.

```
   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";

} // end function testStack

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;

} // end main
```

fig11_04.cpp
(2 of 2)

fig11_04.cpp
output (1 of 1)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Note output identical to that of `fig11_03.cpp`.

Class Templates and Nontype Parameters

- Class templates
  - Nontype parameters
    - Default arguments
    - Treated as consts
    - Example:

      ```
      template< class T, int elements >
      Stack< double, 100 > mostRecentSalesFigures;
      ```

      - Declares object of type `Stack< double, 100>`
  - Type parameter
    - Default type
      - Example:

        ```
        template< class T = string >
        ```

Class Templates and Nontype Parameters

- Overriding class templates
  - Class for specific type
    - Does not match common class template
  - Example:

```
template<>
Class Array< Martian > {
  // body of class definition
};
```

Templates and Inheritance

- Several ways of relating templates and inheritance
  - Class template derived from class-template specialization
  - Class template derived from non-template class
  - Class-template specialization derived from class-template specialization
  - Non-template class derived from class-template specialization

Templates and Friends

- Friendships between class template and
  - Global function
  - Member function of another class
  - Entire class

Templates and Friends

- **`friend`** functions
  - Inside definition of **`template< class T > class X`**
    - **`friend void f1();`**
      - **`f1()`** **`friend`** of all class-template specializations
    - **`friend void f2( X< T > & );`**
      - **`f2( X< float > & )`** **`friend`** of **`X< float >`** only,
        `f2( X< double > & )` **`friend`** of **`X< double >`** only,
        `f2( X< int > & )` **`friend`** of **`X< int >`** only,
        …
    - **`friend void A::f4();`**
      - Member function **`f4`** of class **`A`** **`friend`** of all class-template specializations

Templates and Friends

- **friend** functions
  - Inside definition of **template< class T > class X**
    - **friend void C< T >::f5(X< T > &);**
      - Member function **C<float>::f5(X< float> &)** **friend** of **class X<float>** only
- **friend** classes
  - Inside definition of **template< class T > class X**
    - **friend class Y;**
      - Every member function of **Y** friend of every class-template specialization
    - **friend class Z<T>;**
      - **class Z<float> friend** of class-template specialization **X<float>,** etc.

Templates and static Members

- Non-template class
  - **`static`** data members shared between all objects
- Class-template specialization
  - Each has own copy of **`static`** data members
  - **`static`** variables initialized at file scope
  - Each has own copy of **`static`** member functions

IS 0020
Program Design and Software Tools

Data Structures

Lecture 10

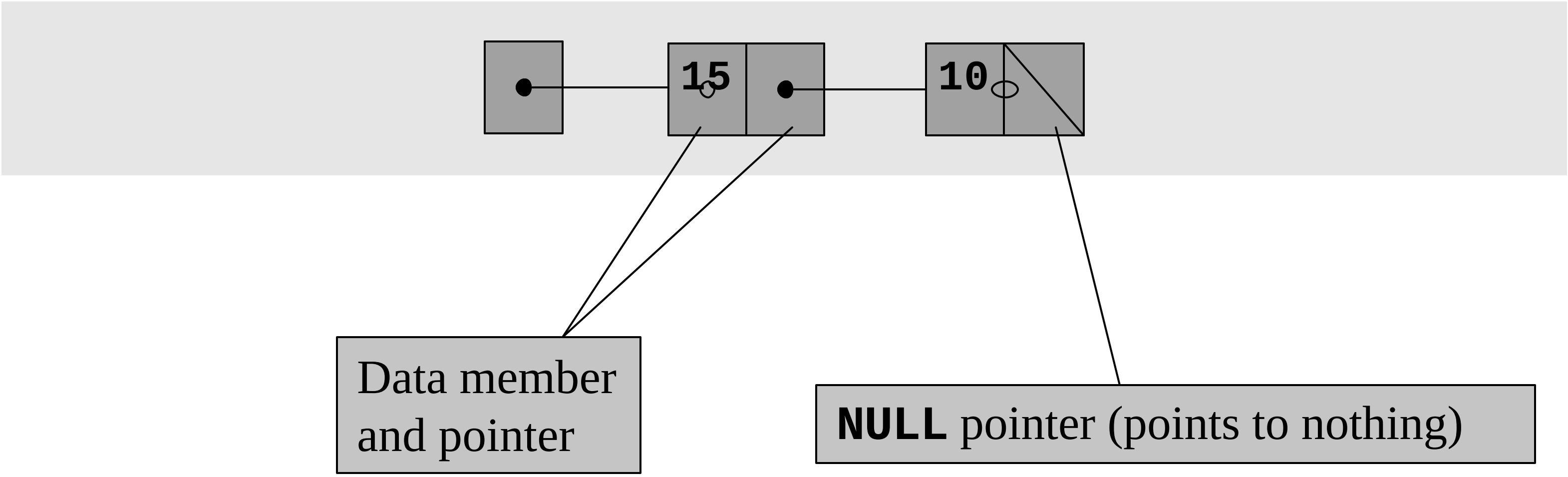
March 23, 2004

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

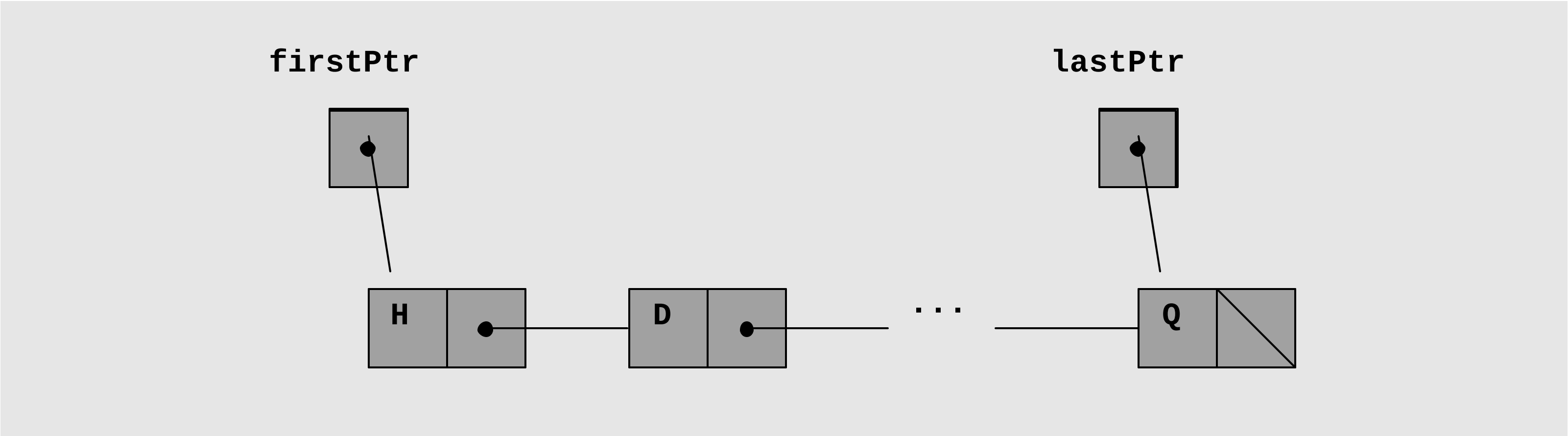
Dynamic Memory Allocation and Data Structures

- 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

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



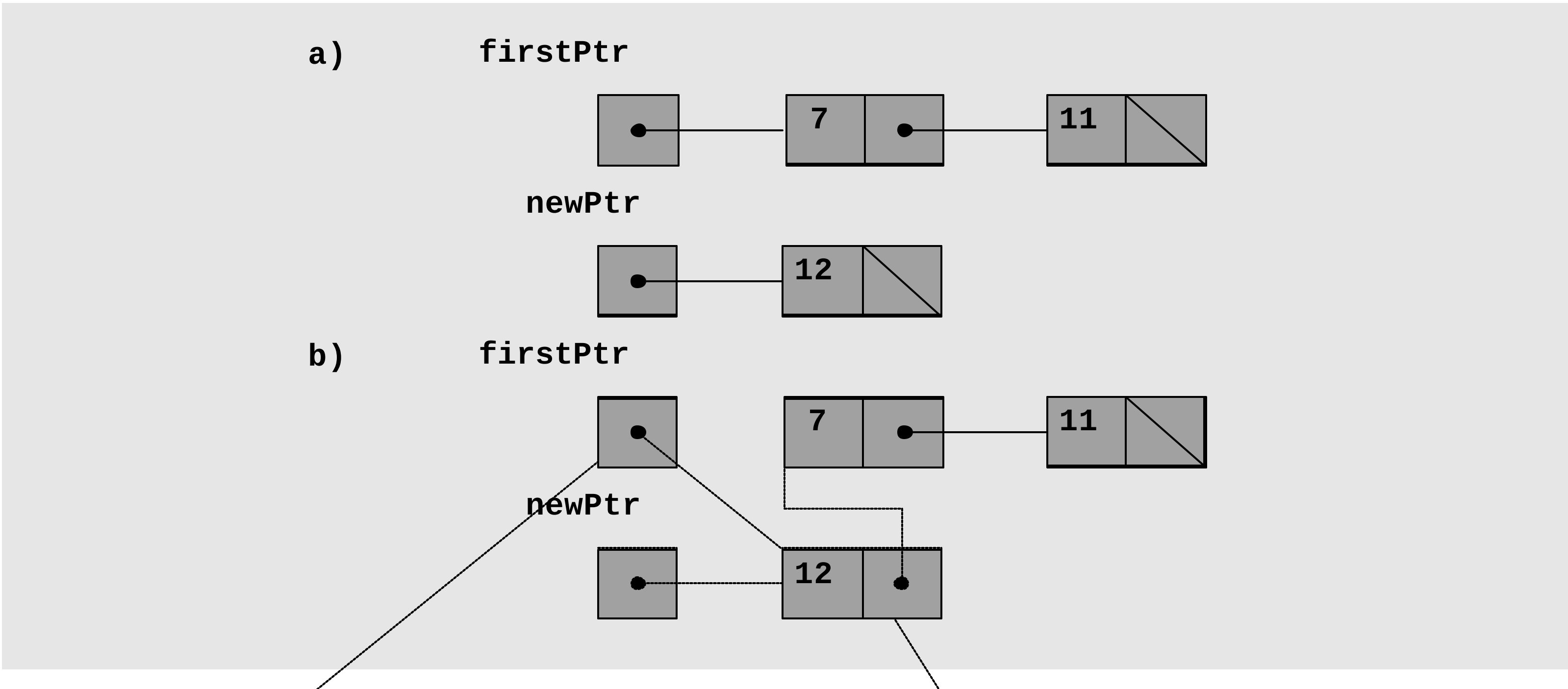
Linked Lists

- 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



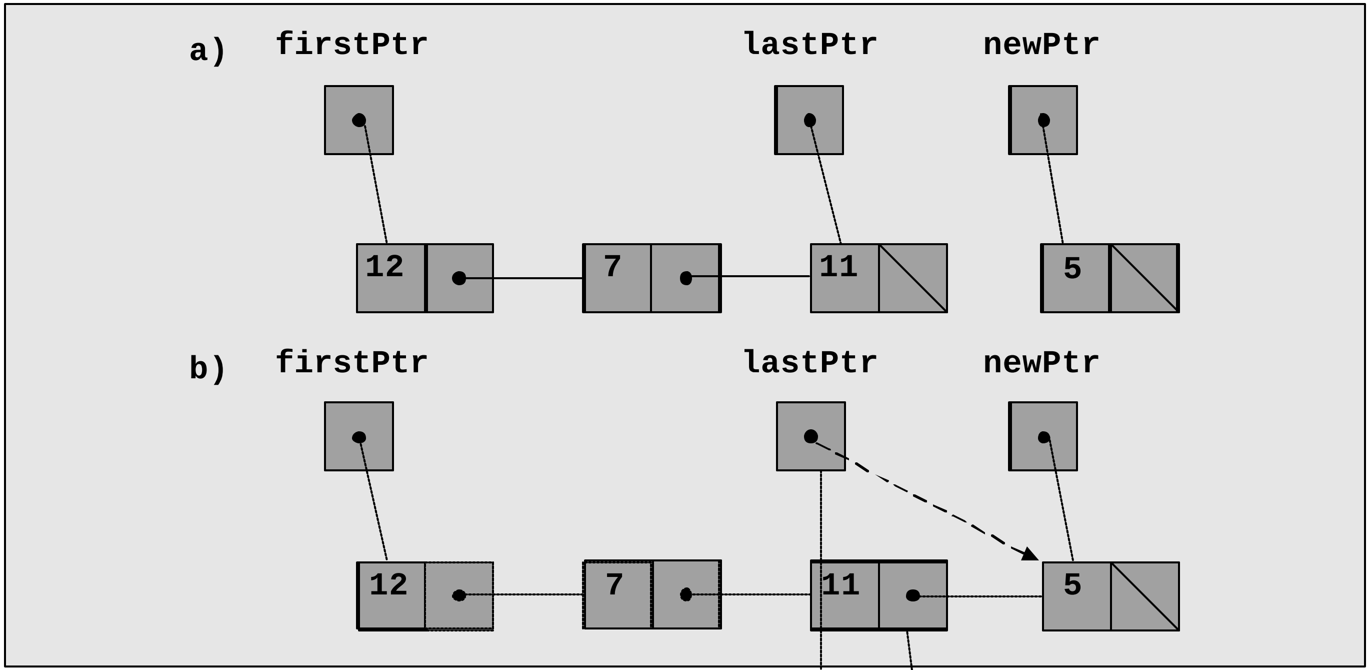

```
firstPtr = newPtr
```

If list empty, then

```
firstPtr = lastPtr = newPtr
```

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

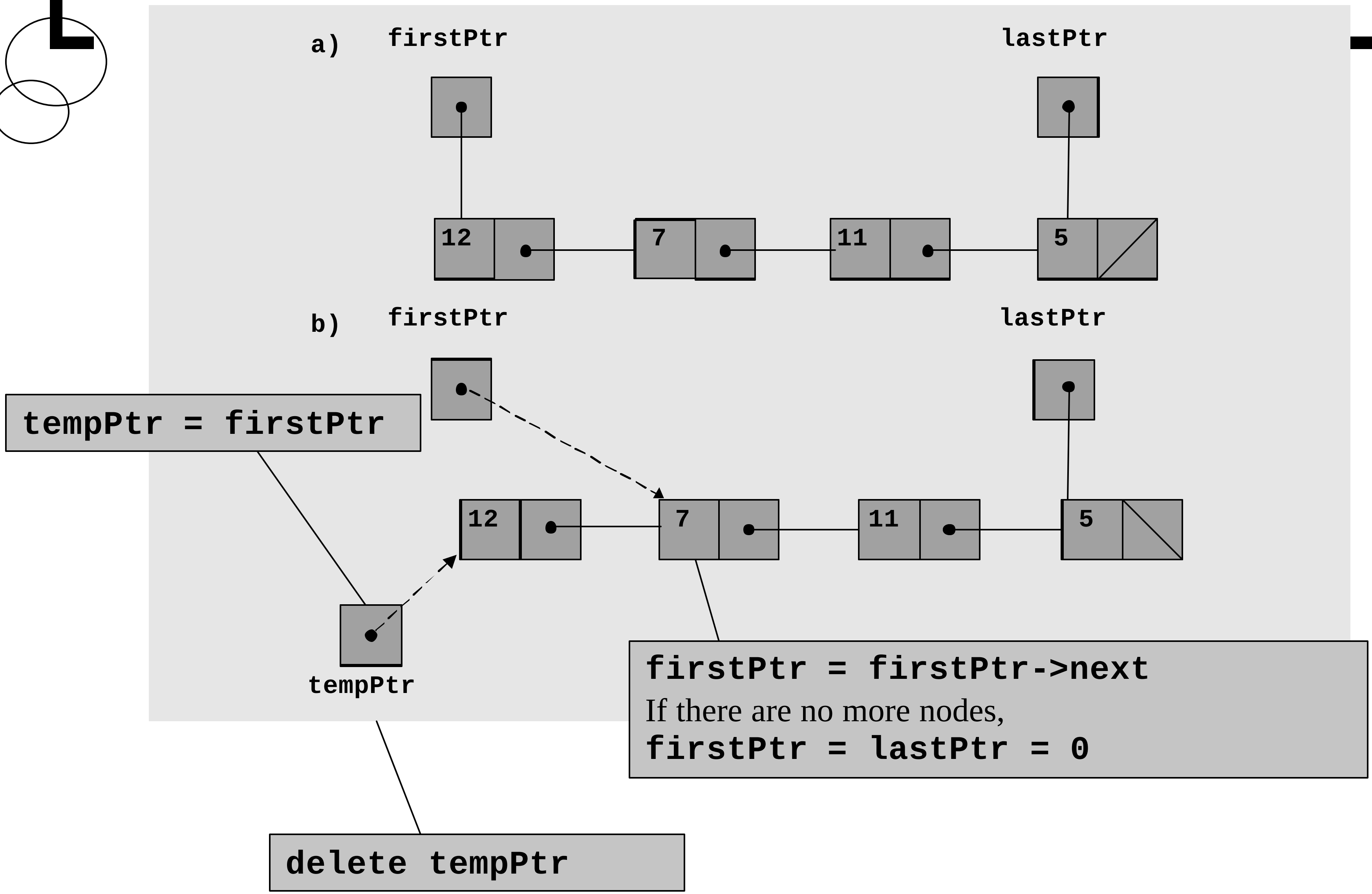
Insert at back




```
lastPtr->nextPtr = newPtr
```

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

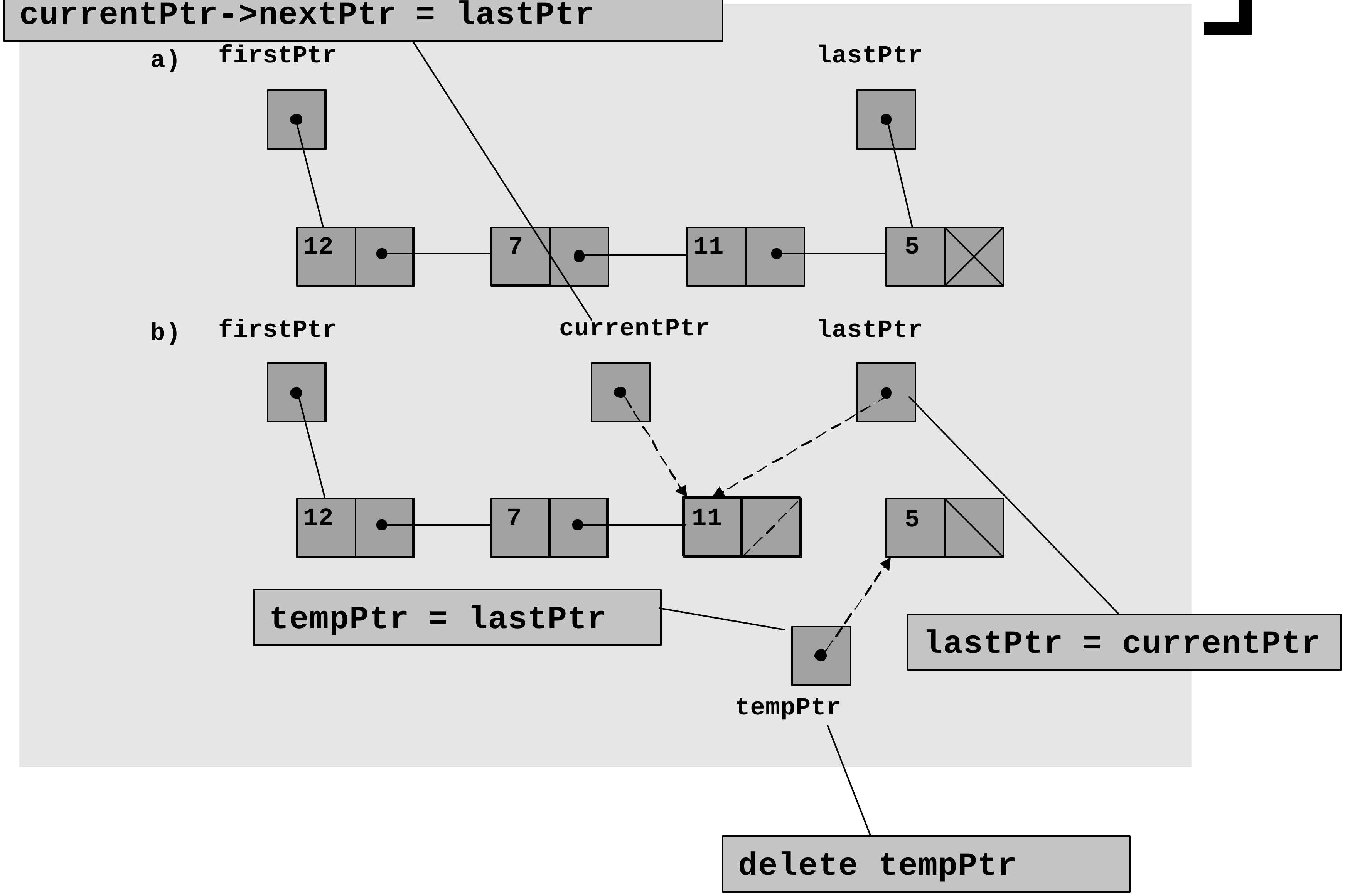
Remove from front



Remove from back

"Walk" list until get next-to-last node, until

`currentPtr->nextPtr = lastPtr`



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`**

```
// Fig. 17.3: listnode.h
// Template ListNode class definition.
#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

```

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

```
// 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
```

Outline

list.h (1 of 9)

```
// Fig. 17.4: list.h
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>

using std::cout;

#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

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

} // end List constructor

```

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

```
// 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

```

list.h (3 of 9)

list.h (4 of 9)

```
// 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 a new node as described in the previous diagrams.

Outline

list.h (5 of 9)

```
// 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

```

Outline

list.h (6 of 9)

```
// delete node from front of list
template< class NODETYPE >
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

```

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

Outline

list.h (9 of 9)

```
// 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
```

fig17_05.cpp
(1 of 4)

```
// Fig. 17.5: fig17_05.cpp
// List class test program.
#include <iostream>

using std::cin;
using std::endl;

#include <string>

using std::string;

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

// 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;

```

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

```
   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;

```

Outline

fig17_05.cpp
(3 of 4)

```
         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

// 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
```

Outline

fig17_05.cpp
(4 of 4)

```
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
```

fig17_05.cpp
output (1 of 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
```


fig17_05.cpp
output (2 of 4)

```
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: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 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
```

Linked Lists

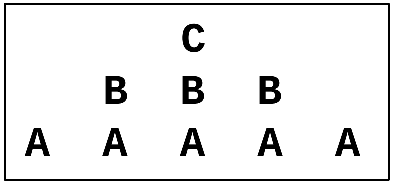
- Types of linked lists
  - Singly linked list (used in example)
    - Pointer to first node
    - Travel in one direction (null-terminated)
  - Circular, singly-linked
    - As above, but last node points to first
  - Doubly-linked list
    - Each node has a forward and backwards pointer
    - Travel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

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

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

```
// Fig. 17.10: stack.h
// Template Stack class definition derived from class List.
#ifndef STACK_H
#define STACK_H

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

template< class STACKTYPE >
class Stack : private List< STACKTYPE > {

public:
   // push calls List function insertAtFront
   void push( const STACKTYPE &data )
   {
      insertAtFront( data );

   } // end function push

   // pop calls List function removeFromFront
   bool pop( STACKTYPE &data )
   {
      return removeFromFront( data );

   } // end function pop

```

Stack inherits from **List**.

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

Outline

stack.h (2 of 2)

```
   // isStackEmpty calls List function isEmpty
   bool isStackEmpty() const
   {
      return isEmpty();

   } // end function isStackEmpty

   // printStack calls List function print
   void printStack() const
   {
      print();

   } // end function print

}; // end class Stack

#endif
```

Outline

fig17_11.cpp
(1 of 3)

```
// Fig. 17.11: fig17_11.cpp
// Template Stack class test program.
#include <iostream>

using std::endl;

#include "stack.h"  // Stack class definition

int main()
{
   Stack< int > intStack;  // create Stack of ints

   cout << "processing an integer Stack" << endl;

   // push integers onto intStack
   for ( int i = 0; i < 4; i++ ) {
      intStack.push( i );
      intStack.printStack();

   } // end for

   // pop integers from intStack
   int popInteger;

```

```
   while ( !intStack.isStackEmpty() ) {
      intStack.pop( popInteger );
      cout << popInteger << " popped from stack" << endl;
      intStack.printStack();

   } // end while

   Stack< double > doubleStack;  // create Stack of doubles
   double value = 1.1;

   cout << "processing a double Stack" << endl;

   // push floating-point values onto doubleStack
   for ( int j = 0; j< 4; j++ ) {
      doubleStack.push( value );
      doubleStack.printStack();
      value += 1.1;

   } // end for

```

```
   // pop floating-point values from doubleStack
   double popDouble;

   while ( !doubleStack.isStackEmpty() ) {
      doubleStack.pop( popDouble );
      cout << popDouble << " popped from stack" << endl;
      doubleStack.printStack();

   } // end while

   return 0;

} // end main
```

Outline

fig17_11.cpp
output (1 of 2)

```
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
```

Outline

fig17_11.cpp
output (2 of 2)

```
// Fig. 17.12: stackcomposition.h
// Template Stack class definition with composed List object.
#ifndef STACKCOMPOSITION
#define STACKCOMPOSITION

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

template< class STACKTYPE >
class Stack {

public:
   // no constructor; List constructor 

   // push calls stackList object's insertAtFront function
   void push( const STACKTYPE &data )
   {
      stackList.insertAtFront( data );

   } // end function push

   // pop calls stackList object's removeFromFront function
   bool pop( STACKTYPE &data )
   {
      return stackList.removeFromFront( data );

   } // end function pop

```

Alternative implementation of **`stack.h`**, using composition.

Declare a private **`List`** member, use to manipulate stack.

```
   // isStackEmpty calls stackList object's isEmpty function
   bool isStackEmpty() const
   {
      return stackList.isEmpty();

   } // end function isStackEmpty

   // printStack calls stackList object's print function
   void printStack() const
   {
      stackList.print();

   } // end function printStack

private:
   List< STACKTYPE > stackList;  // composed List object

}; // end class Stack

#endif
```

stackcomposition.h
(2 of 2)

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`**)

```
// Fig. 17.13: queue.h
// Template Queue class definition derived from class List.
#ifndef QUEUE_H
#define QUEUE_H

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

template< class QUEUETYPE >
class Queue : private List< QUEUETYPE > {

public:
   // enqueue calls List function insertAtBack
   void enqueue( const QUEUETYPE &data )
   {
      insertAtBack( data );

   } // end function enqueue

   // dequeue calls List function removeFromFront
   bool dequeue( QUEUETYPE &data )
   {
      return removeFromFront( data );

   } // end function dequeue

```

Inherit from template class `List`.

Reuse the appropriate `List` functions.

queue.h (1 of 2)

```
   // isQueueEmpty calls List function isEmpty
   bool isQueueEmpty() const
   {
      return isEmpty();

   } // end function isQueueEmpty

   // printQueue calls List function print
   void printQueue() const
   {
      print();

   } // end function printQueue

}; // end class Queue

#endif
```

Outline

fig17_14.cpp
(1 of 3)

```
// Fig. 17.14: fig17_14.cpp
// Template Queue class test program.
#include <iostream>

using std::endl;

#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;

   // 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
```

fig17_14.cpp
(3 of 3)

fig17_14.cpp
output (1 of 2)

```
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
```

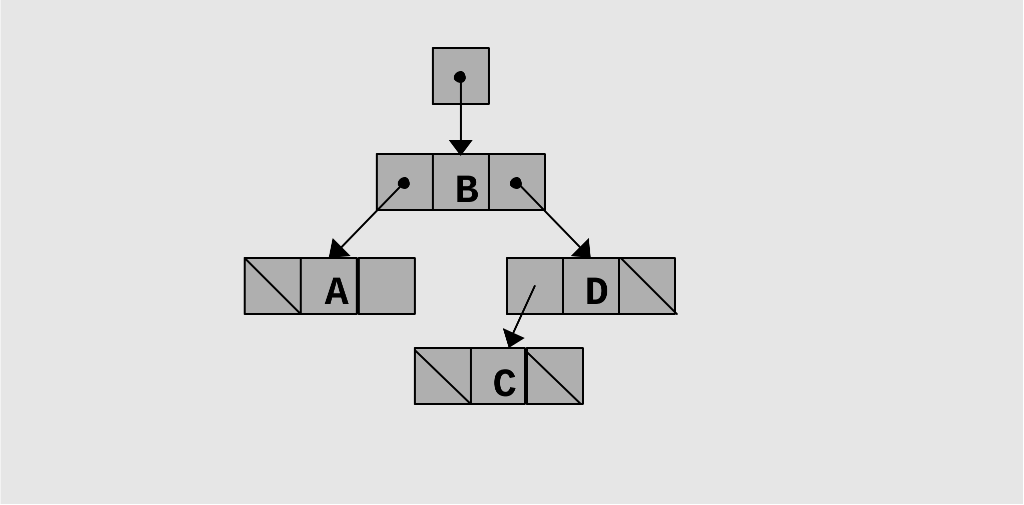
```
The list is: 1.1 2.2 3.3

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
```

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

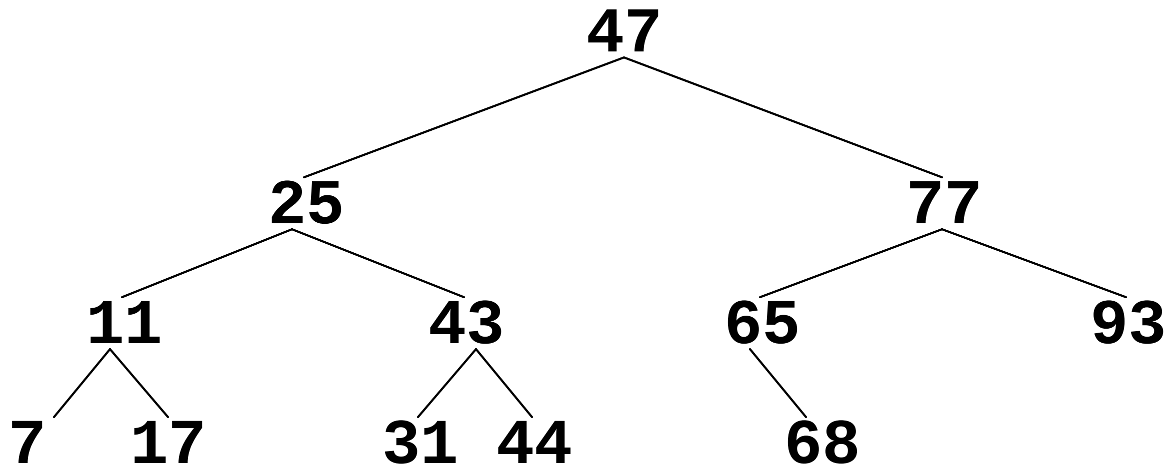
Trees

- 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`**

```
// Fig. 17.17: treenode.h
// Template TreeNode class definition.
#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 )
   {
      // empty body

   } // end TreeNode constructor

```

Binary trees have two pointers.

Outline

treenode.h (2 of 2)

```
   // 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
```

Outline

tree.h (1 of 6)

```
// Fig. 17.18: tree.h
// Template Tree class definition.
#ifndef TREE_H
#define TREE_H

#include <iostream>

using std::endl;

#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;

```

Outline

tree.h (2 of 6)

```
   // 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

// 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  // subtree is not empty

      // data to insert is less than data in current node
      if ( value < ( *ptr )->data )
         insertNodeHelper( &( ( *ptr )->leftPtr ), value );

      else

         // data to insert is greater than data in current node
         if ( value > ( *ptr )->data )
            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
```

Preorder: print, left, right

```
// 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

```

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

```cpp
// Fig. 17.19: fig17_19.cpp
// Tree class test program.
#include <iostream>

using std::cout;
using std::cin;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include "tree.h"  // Tree class definition

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
```

Outline

fig17_19.cpp
(2 of 3)

```

   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();

```

Outline

fig17_19.cpp
(3 of 3)

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

   cout << "\nPostorder traversal\n";
   doubleTree.postOrderTraversal();

   cout << endl;

   return 0;

} // end main
```

```
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
```


fig17_19.cpp
output (1 of 1)