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

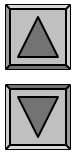
- Overloaded functions
  - Similar operations
    - Different types of data
- Function templates
  - Identical operations
    - Different types of data
  - Single function template
    - Compiler generates separate object-code functions
  - Unlike Macros they allow Type checking

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



Outline

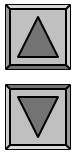
fig11_01.cpp (1 of 2)

```
1  // Fig. 11.1: fig11_01.cpp
2  // Using template functions.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function template printArray definiti
9  template< class T >
10 void printArray( const T *array, const int count )
11 {
12    for ( int i = 0; i < coun
13       cout << array[ i ] <<
14
15    cout << endl;
16
17 } // end function printAr
18
19 int main()
20 {
21    const int aCount = 5;
22    const int bCount = 7;
23    const int cCount = 6;
24
```

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**.



Outline

fig11_01.cpp (2 of 2)

```
25    int a[ aCount ] = { 1, 2, 3, 4, 5 };
26    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
27    char c[ cCount ] = "HELLO";  // 6th position for null
28
29    cout << "Array a contains:" << endl;
30
31    // call integer function-template specialization
32    printArray( a, aCount );
33
34    cout << "Array b contains
35
36    // call double funct
37    printArray( b, bCoun
38
39    cout << "Array c con
40
41    // call character fu
42    printArray( c, cCoun
43
44    return 0;
45
46 } // end main
```

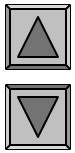
Compiler infers **T** is **double**; instantiates function-template specialization where **T** is **double**.

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

Crea ecialization for printing array

```
void                                const int count )
{
   fo                              )
   cc
} // end function printArray
```

```
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
```



Outline

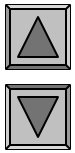
fig11_01.cpp
output (1 of 1)

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



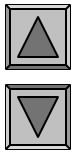
Outline

tstack1.h (1 of 4)

```
1  // Fig. 11.2: tstack1.h
2  // Stack class template.
3  #ifndef TSTACK1_H
4  #define TSTACK1_H
5
6  template< class T >
7  class Stack {
8
9  public:
10    Stack( int = 10 );  // default constructor (stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr;
16
17    } // end ~Stack destructor
18
19    bool push( const T& );  // push an element onto the stack
20    bool pop( T& );         // pop an element off the stack
21
```

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

Function parameters of type **T**.

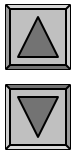


Outline

tstack1.h (2 of 4)

```
22    // determine whether Stack is empty
23    bool isEmpty() const
24    {
25       return top == -1;
26
27    } // end function isEmpty
28
29    // determine whether Stack is full
30    bool isFull() const
31    {
32       return top == size - 1;
33
34    } // end function isFull
35
36 private:
37    int size;      // # of elements in the stack
38    int top;       // location of the top element
39    T *stackPtr;   // pointer to the stack
40
41 }; // end class Stack
42
```

Array of elements of type **T**.



Outline

tstack1.h (3 of 4)

```
43 // constructor
44 template< class T >
45 Stack< T >::Stack( int s )
46 {
47    size = s > 0 ? s : 10;
48    top = -1;  // Stack initially empty
49    stackPtr = new T[ size ]; // allocate
50
51 } // end Stack constructor
52
53 // push element onto stack;
54 // if successful, return true; other
55 template< class T >
56 bool Stack< T >::push( const T &push
57 {
58    if ( !isFull() ) {
59       stackPtr[ ++top ] = pushValue;  // place item on Stack
60       return true;  // push successful
61
62    } // end if
63
64    return false;  // push unsuccessful
65
66 } // end function push
67
```

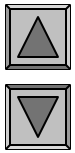
Constructor creates array of type **T**. For example, compiler generates

```
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 >



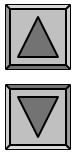
Outline

tstack1.h (4 of 4)

```
68 // pop element off stack;
69 // if successful, return true; otherwise, return false
70 template< class T >
71 bool Stack< T >::pop( T &popValue )
72 {
73    if ( !isEmpty() ) {
74       popValue = stackPtr[ top-- ];  // r
75       return true;  // pop successful
76
77    } // end if
78
79    return false;  // pop unsuccessful
80
81 } // end function pop
82
83 #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.



Outline

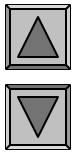
fig11_03.cpp (1 of 3)

```
1  // Fig. 11.3: fig11_03.cpp
2  // Stack-class-template test program.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "tstack1.h"  // Stack class template definition
10
11 int main()
12 {
13    Stack< double > doubleStack( 5 );
14    double doubleValue = 1.1;
15
16    cout << "Pushing elements onto doubleStack"
17
18    while ( doubleStack.push( doubleValue ) ) {
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21
22    } // end while
23
24    cout << "\nStack is full. Cannot push " << doubleValue
25         << "\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 >`**.



Outline

fig11_03.cpp (2 of 3)

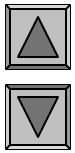
```
26
27    while ( doubleStack.pop( doubleValue ) )
28       cout << doubleValue << ' ';
29
30    cout << "\nStack is empty. Cannot pop\n";
31
32    Stack< int > intStack;
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    while ( intStack.push( intValue ) ) {
37       cout << intValue << ' ';
38       ++intValue;
39
40    } // end while
41
42    cout << "\nStack is full. Cannot push " << intValue
43         << "\n\nPopping elements from intStack\n";
44
45    while ( intStack.pop( intValue ) )
46       cout << intValue << ' ';
47
48    cout << "\nStack is empty. Cannot pop\n";
49
50    return 0;
```

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

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

```
51
52  } // end main
```

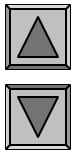
```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

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
```



Outline

fig11_03.cpp (3 of 3)

fig11_03.cpp output (1 of 1)

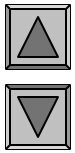


Outline

fig11_04.cpp
(1 of 2)

```
1  // Fig. 11.4: fig11_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 #include "tstack1.h"  // Stack class template definition
11
12 // function template to manipulate Stack< T >
13 template< class T >
14 void testStack(
15    Stack< T > &theStack,   // reference to Stack< T >
16    T value,                // initial value to push
17    T increment,            // increment for subsequent values
18    const char *stackName ) // name of the Stack < T > object
19 {
20    cout << "\nPushing elements onto " << stackName << '\n';
21
22    while ( theStack.push( value ) ) {
23       cout << value << ' ';
24       value += increment;
25
26    } // end while
```

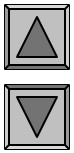
Function template to manipulate **`Stack< T >`** eliminates similar code from previous file for **`Stack< double >`** and **`Stack< int >`**.



Outline

fig11_04.cpp
(2 of 2)

```
27
28    cout << "\nStack is full. Cannot push " << value
29         << "\n\nPopping elements from " << stackName << '\n';
30
31    while ( theStack.pop( value ) )
32       cout << value << ' ';
33
34    cout << "\nStack is empty. Cannot pop\n";
35
36 } // end function testStack
37
38 int main()
39 {
40    Stack< double > doubleStack( 5 );
41    Stack< int > intStack;
42
43    testStack( doubleStack, 1.1, 1.1, "doubleStack" );
44    testStack( intStack, 1, 1, "intStack" );
45
46    return 0;
47
48 } // end main
```



Outline

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

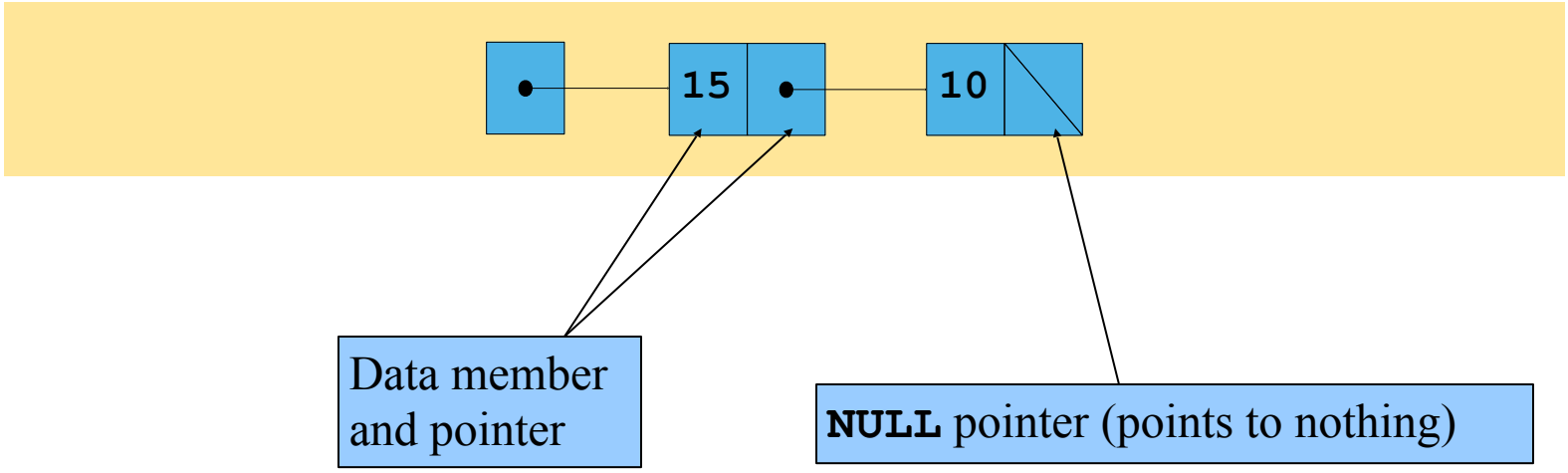
March 23, 2004

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

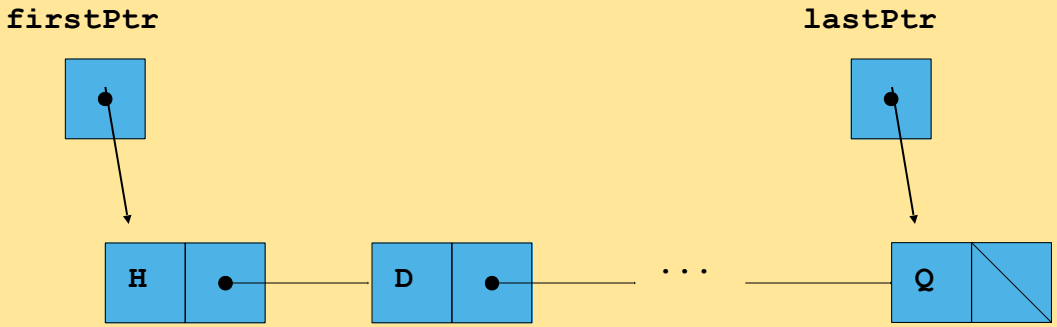
Dynamic Memory Allocation and Data Structures

- 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



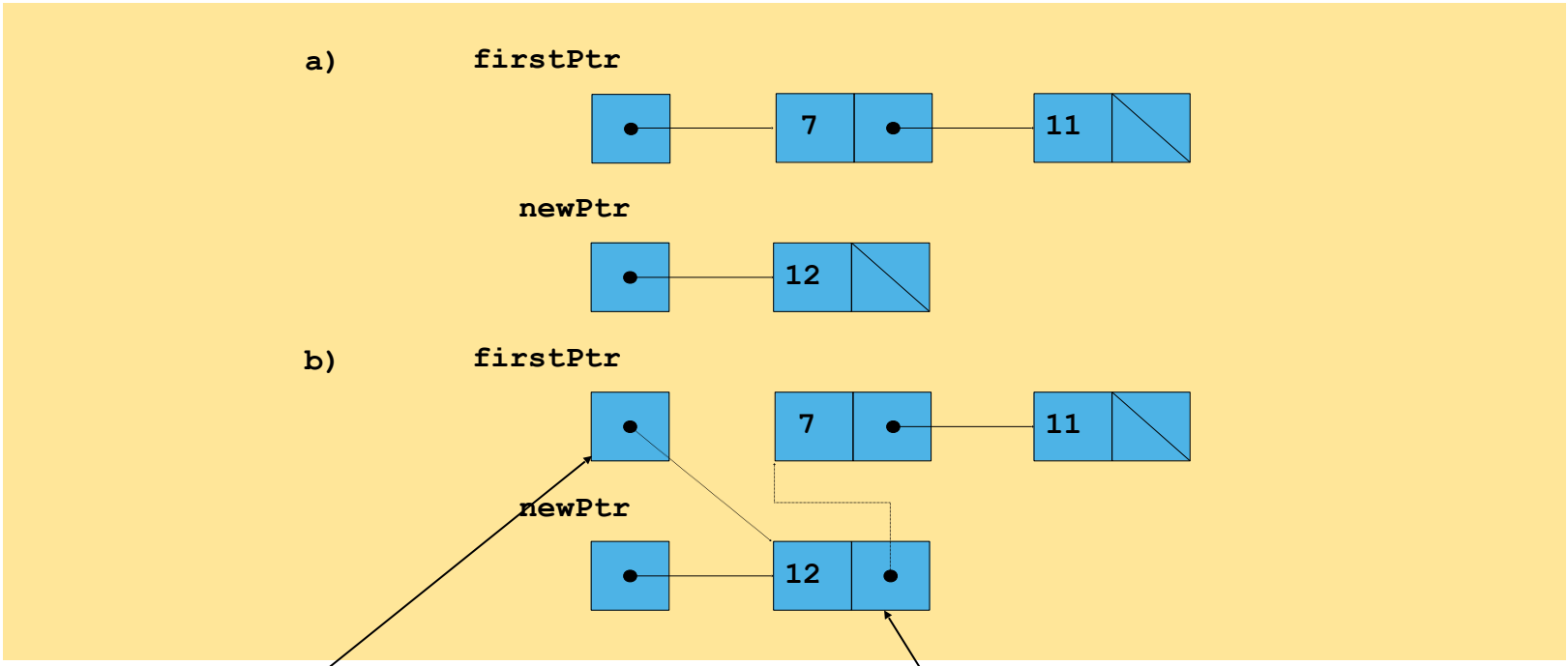
Linked Lists

- 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

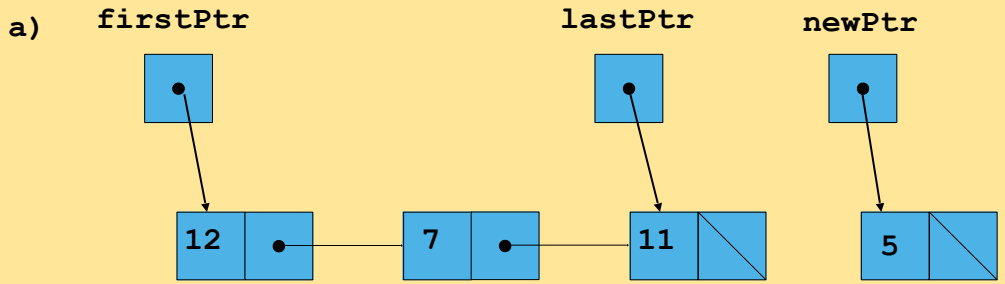
Insert at front



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

newPtr->nextPtr = firstPtr

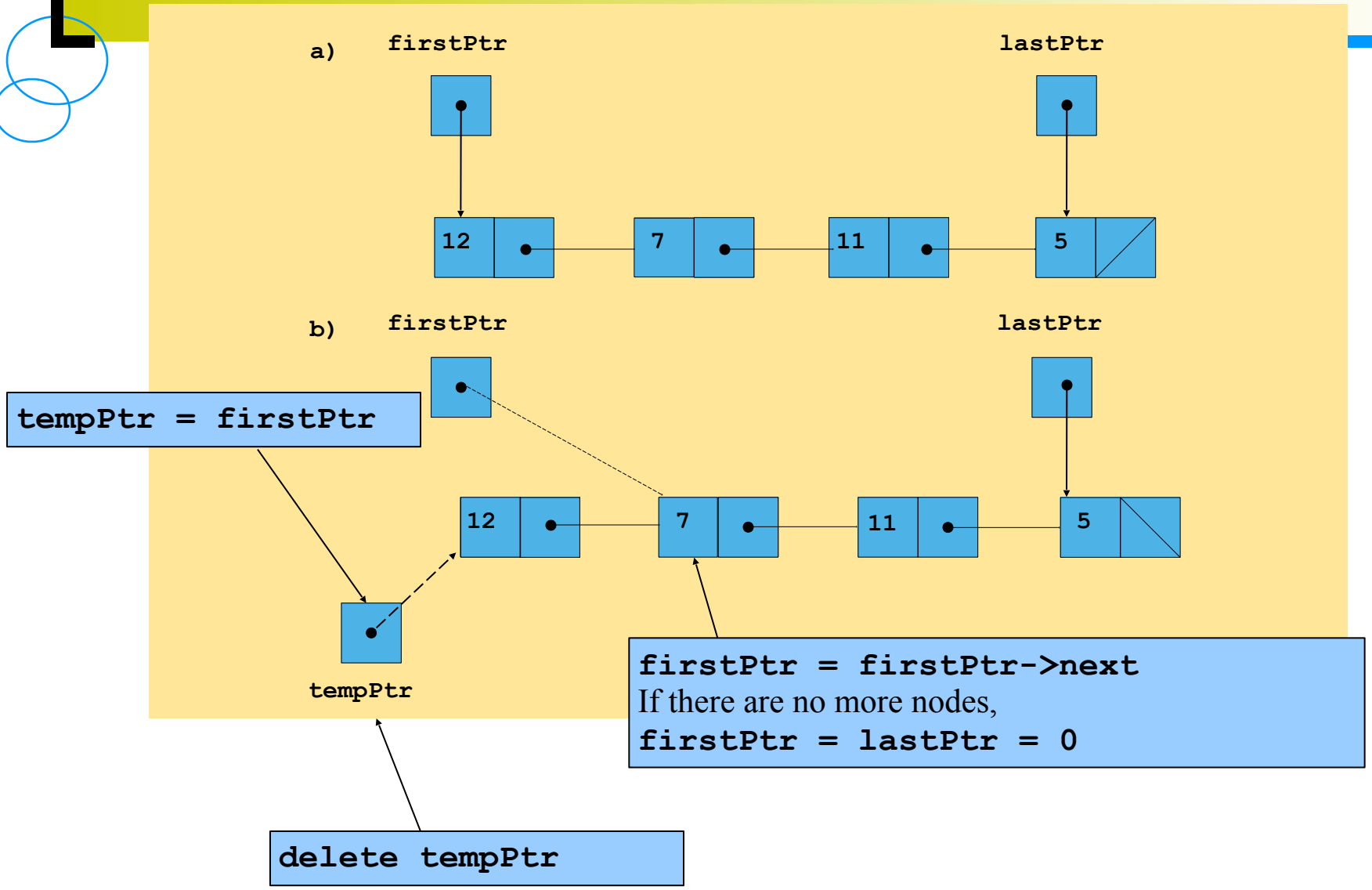
Insert at back




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

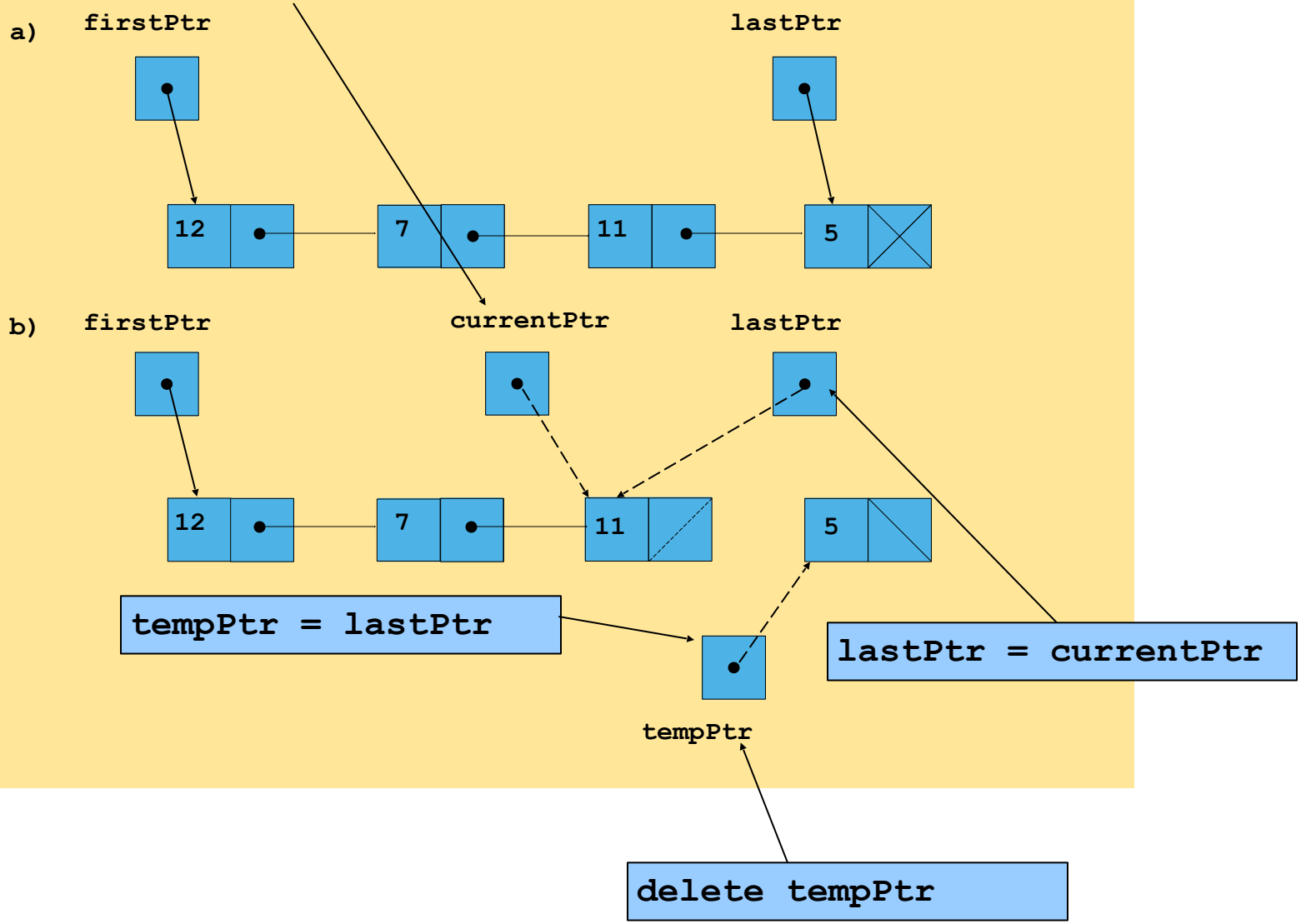
```
lastPtr = newPtr
```
If list empty, then
```
firstPtr = lastPtr = newPtr
```

Remove from front



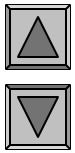
Remove from back

"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`**

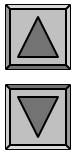


Outline

listnode.h (1 of 2)

```
1  // Fig. 17.3: listnode.h
2  // Template ListNode class definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List
7  template< class NODETYPE > class List;
8
9  template< class NODETYPE>
10 class ListNode {
11    friend class List< NODETYPE >; // make List a friend
12
13 public:
14    ListNode( const NODETYPE & );  // constructor
15    NODETYPE getData() const;      // return data in node
16
17 private:
18    NODETYPE data;                 // data
19    ListNode< NODETYPE > *nextPtr; // next node in list
20
21 }; // end class ListNode
22
```

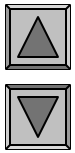
Template class **ListNode**. The type of member **data** depends on how the class template is used.



Outline

listnode.h (2 of 2)

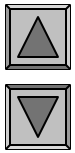
```
23 // constructor
24 template< class NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ),
27      nextPtr( 0 )
28 {
29    // empty body
30
31 } // end ListNode constructor
32
33 // return copy of data in node
34 template< class NODETYPE >
35 NODETYPE ListNode< NODETYPE >::getData() const
36 {
37    return data;
38
39 } // end function getData
40
41 #endif
```



Outline

list.h (1 of 9)

```
1  // Fig. 17.4: list.h
2  // Template List class definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7
8  using std::cout;
9
10 #include <new>
11 #include "listnode.h"  // ListNode class definition
12
13 template< class NODETYPE >
14 class List {
15
16 public:
17    List();      // constructor
18    ~List();     // destructor
19    void insertAtFront( const NODETYPE & );
20    void insertAtBack( const NODETYPE & );
21    bool removeFromFront( NODETYPE & );
22    bool removeFromBack( NODETYPE & );
23    bool isEmpty() const;
24    void print() const;
25
```

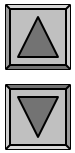


Outline

list.h (2 of 9)

```
26 private:
27    ListNode< NODETYPE > *firstPtr;  // pointer to first node
28    ListNode< NODETYPE > *lastPtr;   // pointer to last node
29
30    // utility function to allocate new node
31    ListNode< NODETYPE > *getNewNode( const NODETYPE & );
32
33 }; // end class List
34
35 // default constructor
36 template< class NODETYPE >
37 List< NODETYPE >::List()
38    : firstPtr( 0 ),
39      lastPtr( 0 )
40 {
41    // empty body
42
43 } // end List constructor
44
```

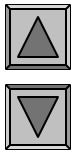
Each **List** has a **firstPtr** and **lastPtr**.



Outline

list.h (3 of 9)

```
45 // destructor
46 template< class NODETYPE >
47 List< NODETYPE >::~List()
48 {
49    if ( !isEmpty() ) {    // List is not empty
50       cout << "Destroying nodes ...\n";
51
52       ListNode< NODETYPE > *currentPtr = firstPtr;
53       ListNode< NODETYPE > *tempPtr;
54
55       while ( currentPtr != 0 ) {  // delete remaining nodes
56          tempPtr = currentPtr;
57          cout << tempPtr->data << '\n';
58          currentPtr = currentPtr->nextPtr;
59          delete tempPtr;
60
61       } // end while
62
63    } // end if
64
65    cout << "All nodes destroyed\n\n";
66
67 } // end List destructor
68
```

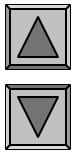


Outline

list.h (4 of 9)

```
69 // insert node at front of list
70 template< class NODETYPE >
71 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
72 {
73    ListNode< NODETYPE > *newPtr = getNewNode( value );
74
75    if ( isEmpty() )  // List is empty
76       firstPtr = lastPtr = newPtr;
77
78    else {  // List is not empty
79       newPtr->nextPtr = firstPtr;
80       firstPtr = newPtr;
81
82    } // end else
83
84 } // end function insertAtFront
85
```

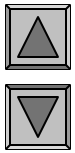
Insert a new node as described in the previous diagrams.



Outline

list.h (5 of 9)

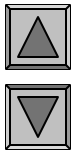
```
86 // insert node at back of list
87 template< class NODETYPE >
88 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
89 {
90    ListNode< NODETYPE > *newPtr = getNewNode( value );
91
92    if ( isEmpty() )  // List is empty
93       firstPtr = lastPtr = newPtr;
94
95    else {  // List is not empty
96       lastPtr->nextPtr = newPtr;
97       lastPtr = newPtr;
98
99    } // end else
100
101 } // end function insertAtBack
102
```



Outline

list.h (6 of 9)

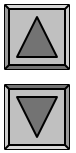
```
103 // delete node from front of list
104 template< class NODETYPE >
105 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
106 {
107    if ( isEmpty() )  // List is empty
108       return false;  // delete unsuccessful
109
110    else {
111       ListNode< NODETYPE > *tempPtr = firstPtr;
112
113       if ( firstPtr == lastPtr )
114          firstPtr = lastPtr = 0;
115       else
116          firstPtr = firstPtr->nextPtr;
117
118       value = tempPtr->data;  // data being removed
119       delete tempPtr;
120
121       return true;  // delete successful
122
123    } // end else
124
125 } // end function removeFromFront
126
```



Outline

list.h (7 of 9)

```
127 // delete node from back of list
128 template< class NODETYPE >
129 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
130 {
131    if ( isEmpty() )
132       return false;  // delete unsuccessful
133
134    else {
135       ListNode< NODETYPE > *tempPtr = lastPtr;
136
137       if ( firstPtr == lastPtr )
138          firstPtr = lastPtr = 0;
139       else {
140          ListNode< NODETYPE > *currentPtr = firstPtr;
141
142          // locate second-to-last element
143          while ( currentPtr->nextPtr != lastPtr )
144             currentPtr = currentPtr->nextPtr;
145
146          lastPtr = currentPtr;
147          currentPtr->nextPtr = 0;
148
149       } // end else
150
151       value = tempPtr->data;
152       delete tempPtr;
153
```

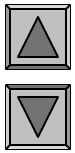


Outline

list.h (8 of 9)

```
154       return true;  // delete successful
155
156    } // end else
157
158 } // end function removeFromBack
159
160 // is List empty?
161 template< class NODETYPE >
162 bool List< NODETYPE >::isEmpty() const
163 {
164    return firstPtr == 0;
165
166 } // end function isEmpty
167
168 // return pointer to newly allocated node
169 template< class NODETYPE >
170 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
171    const NODETYPE &value )
172 {
173    return new ListNode< NODETYPE >( value );
174
175 } // end function getNewNode
176
```

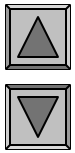
Note use of **new** operator to dynamically allocate a node.



Outline

list.h (9 of 9)

```
177 // display contents of List
178 template< class NODETYPE >
179 void List< NODETYPE >::print() const
180 {
181    if ( isEmpty() ) {
182       cout << "The list is empty\n\n";
183       return;
184
185    } // end if
186
187    ListNode< NODETYPE > *currentPtr = firstPtr;
188
189    cout << "The list is: ";
190
191    while ( currentPtr != 0 ) {
192       cout << currentPtr->data << ' ';
193       currentPtr = currentPtr->nextPtr;
194
195    } // end while
196
197    cout << "\n\n";
198
199 } // end function print
200
201 #endif
```

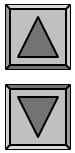


Outline

fig17_05.cpp
(1 of 4)

```
1  // Fig. 17.5: fig17_05.cpp
2  // List class test program.
3  #include <iostream>
4
5  using std::cin;
6  using std::endl;
7
8  #include <string>
9
10 using std::string;
11
12 #include "list.h"  // List class definition
13
14 // function to test a List
15 template< class T >
16 void testList( List< T > &listObject, const string &typeName )
17 {
18    cout << "Testing a List of " << typeName << " values\n";
19
20    instructions();  // display instructions
21
22    int choice;
23    T value;
24
```

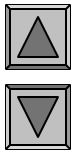
Program to give user a menu to add/remove nodes from a list.



Outline

fig17_05.cpp
(2 of 4)

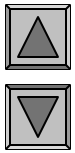
```
25    do {
26       cout << "? ";
27       cin >> choice;
28
29       switch ( choice ) {
30          case 1:
31             cout << "Enter " << typeName << ": ";
32             cin >> value;
33             listObject.insertAtFront( value );
34             listObject.print();
35             break;
36
37          case 2:
38             cout << "Enter " << typeName << ": ";
39             cin >> value;
40             listObject.insertAtBack( value );
41             listObject.print();
42             break;
43
44          case 3:
45             if ( listObject.removeFromFront( value ) )
46                cout << value << " removed from list\n";
47
48             listObject.print();
49             break;
50
```



Outline

fig17_05.cpp
(3 of 4)

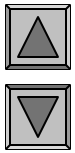
```
51          case 4:
52             if ( listObject.removeFromBack( value ) )
53                cout << value << " removed from list\n";
54
55             listObject.print();
56             break;
57
58       } // end switch
59
60    } while ( choice != 5 );  // end do/while
61
62    cout << "End list test\n\n";
63
64 } // end function testList
65
66 // display program instructions to user
67 void instructions()
68 {
69    cout << "Enter one of the following:\n"
70         << "  1 to insert at beginning of list\n"
71         << "  2 to insert at end of list\n"
72         << "  3 to delete from beginning of list\n"
73         << "  4 to delete from end of list\n"
74         << "  5 to end list processing\n";
75
76 } // end function instructions
```



Outline

fig17_05.cpp
(4 of 4)

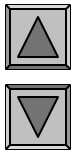
```
77
78 int main()
79 {
80    // test List of int values
81    List< int > integerList;
82    testList( integerList, "integer" );
83
84    // test List of double values
85    List< double > doubleList;
86    testList( doubleList, "double" );
87
88    return 0;
89
90 } // end main
```



Outline

fig17_05.cpp
output (1 of 4)

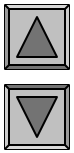
```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 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
```



Outline

```
? 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
```

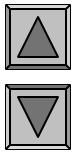
fig17_05.cpp
output (2 of 4)



Outline

fig17_05.cpp
output (3 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
```



Outline

```
? 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
```

fig17_05.cpp
output (4 of 4)

Linked Lists

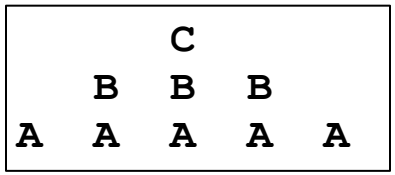
- Types of linked lists
  - Singly linked list (used in example)
    - Pointer to first node
    - Travel in one direction (null-terminated)
  - Circular, singly-linked
    - As above, but last node points to first
  - Doubly-linked list
    - Each node has a forward and backwards pointer
    - Travel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

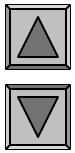
Stacks

- 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



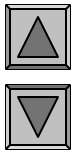
Outline

stack.h (1 of 2)

```
1  // Fig. 17.10: 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< class STACKTYPE >
9  class Stack : private List< STACKTYPE > {
10
11 public:
12    // push calls List function insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16
17    } // end function push
18
19    // pop calls List function removeFromFront
20    bool pop( STACKTYPE &data )
21    {
22       return removeFromFront( data );
23
24    } // end function pop
25
```

`Stack` inherits from **`List`**.

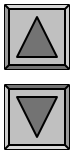
Define **`push`** and **`pop`**, which call **`insertAtFront`** and **`removeFromFront`**.



Outline

stack.h (2 of 2)

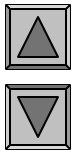
```
26    // isStackEmpty calls List function isEmpty
27    bool isStackEmpty() const
28    {
29       return isEmpty();
30
31    } // end function isStackEmpty
32
33    // printStack calls List function print
34    void printStack() const
35    {
36       print();
37
38    } // end function print
39
40 }; // end class Stack
41
42 #endif
```



Outline

fig17_11.cpp
(1 of 3)

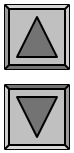
```
1  // Fig. 17.11: fig17_11.cpp
2  // Template Stack class test program.
3  #include <iostream>
4
5  using std::endl;
6
7  #include "stack.h"  // Stack class definition
8
9  int main()
10 {
11    Stack< int > intStack;  // create Stack of ints
12
13    cout << "processing an integer Stack" << endl;
14
15    // push integers onto intStack
16    for ( int i = 0; i < 4; i++ ) {
17       intStack.push( i );
18       intStack.printStack();
19
20    } // end for
21
22    // pop integers from intStack
23    int popInteger;
24
```



Outline

```
25    while ( !intStack.isStackEmpty() ) {
26       intStack.pop( popInteger );
27       cout << popInteger << " popped from stack" << endl;
28       intStack.printStack();
29
30    } // end while
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< 4; j++ ) {
39       doubleStack.push( value );
40       doubleStack.printStack();
41       value += 1.1;
42
43    } // end for
44
```

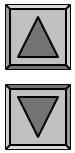
fig17_11.cpp
(2 of 3)



Outline

```
45    // pop floating-point values from doubleStack
46    double popDouble;
47
48    while ( !doubleStack.isStackEmpty() ) {
49       doubleStack.pop( popDouble );
50       cout << popDouble << " popped from stack" << endl;
51       doubleStack.printStack();
52
53    } // end while
54
55    return 0;
56
57 } // end main
```

fig17_11.cpp
(3 of 3)

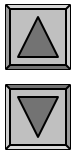


Outline

fig17_11.cpp
output (1 of 2)

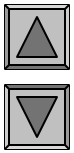
```
processing an integer Stack
The list is: 0

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)



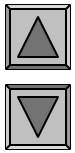
Outline

stackcomposition.h
(1 of 2)

```
1  // Fig. 17.12: stackcomposition.h
2  // Template Stack class definition with composed List object.
3  #ifndef STACKCOMPOSITION
4  #define STACKCOMPOSITION
5
6  #include "list.h"  // List class definit
7
8  template< class STACKTYPE >
9  class Stack {
10
11 public:
12    // no constructor; List constructor d
13
14    // push calls stackList object's insertAtFront function
15    void push( const STACKTYPE &data )
16    {
17       stackList.insertAtFront( data );
18
19    } // end function push
20
21    // pop calls stackList object's removeFromFront function
22    bool pop( STACKTYPE &data )
23    {
24       return stackList.removeFromFront( data );
25
26    } // end function pop
27
```

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

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



Outline

stackcomposition.h
(2 of 2)

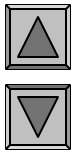
```
28    // isStackEmpty calls stackList object's isEmpty function
29    bool isStackEmpty() const
30    {
31       return stackList.isEmpty();
32
33    } // end function isStackEmpty
34
35    // printStack calls stackList object's print function
36    void printStack() const
37    {
38       stackList.print();
39
40    } // end function printStack
41
42 private:
43    List< STACKTYPE > stackList;  // composed List object
44
45 }; // end class Stack
46
47 #endif
```

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`**)



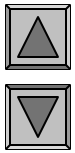
Outline

queue.h (1 of 2)

```
1  // Fig. 17.13: queue.h
2  // Template Queue class definition derived from class List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "list.h"  // List class definition
7
8  template< class QUEUETYPE >
9  class Queue : private List< QUEUETYPE > {
10
11 public:
12    // enqueue calls List function insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16
17    } // end function enqueue
18
19    // dequeue calls List function removeFromFront
20    bool dequeue( QUEUETYPE &data )
21    {
22       return removeFromFront( data );
23
24    } // end function dequeue
25
```

Inherit from template class **`List`**.

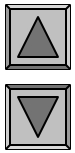
Reuse the appropriate **`List`** functions.



Outline

queue.h (2 of 2)

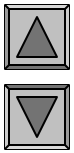
```
26    // isQueueEmpty calls List function isEmpty
27    bool isQueueEmpty() const
28    {
29       return isEmpty();
30
31    } // end function isQueueEmpty
32
33    // printQueue calls List function print
34    void printQueue() const
35    {
36       print();
37
38    } // end function printQueue
39
40 }; // end class Queue
41
42 #endif
```



Outline

fig17_14.cpp
(1 of 3)

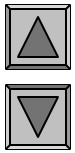
```
1  // Fig. 17.14: fig17_14.cpp
2  // Template Queue class test program.
3  #include <iostream>
4
5  using std::endl;
6
7  #include "queue.h"  // Queue class definition
8
9  int main()
10 {
11    Queue< int > intQueue;  // create Queue of ints
12
13    cout << "processing an integer Queue" << endl;
14
15    // enqueue integers onto intQueue
16    for ( int i = 0; i < 4; i++ ) {
17       intQueue.enqueue( i );
18       intQueue.printQueue();
19
20    } // end for
21
22    // dequeue integers from intQueue
23    int dequeueInteger;
24
```



Outline

```
25    while ( !intQueue.isQueueEmpty() ) {
26       intQueue.dequeue( dequeueInteger );
27       cout << dequeueInteger << " dequeued" << endl;
28       intQueue.printQueue();
29
30    } // end while
31
32    Queue< double > doubleQueue;  // create Queue of doubles
33    double value = 1.1;
34
35    cout << "processing a double Queue" << endl;
36
37    // enqueue floating-point values onto doubleQueue
38    for ( int j = 0; j< 4; j++ ) {
39       doubleQueue.enqueue( value );
40       doubleQueue.printQueue();
41       value += 1.1;
42
43    } // end for
44
```

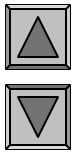
fig17_14.cpp
(2 of 3)



Outline

```
45    // dequeue floating-point values from doubleQueue
46    double dequeueDouble;
47
48    while ( !doubleQueue.isQueueEmpty() ) {
49       doubleQueue.dequeue( dequeueDouble );
50       cout << dequeueDouble << " dequeued" << endl;
51       doubleQueue.printQueue();
52
53    } // end while
54
55    return 0;
56
57 } // end main
```

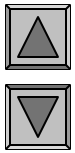
fig17_14.cpp
(3 of 3)



Outline

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
```



Outline

```
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
```

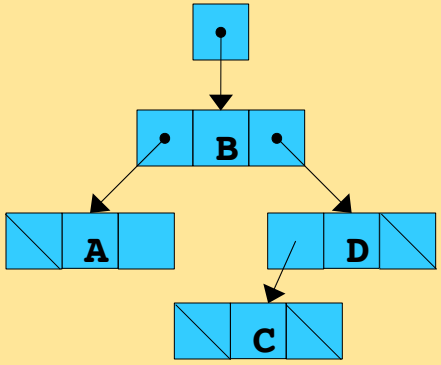
fig17_14.cpp
output (2 of 2)

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

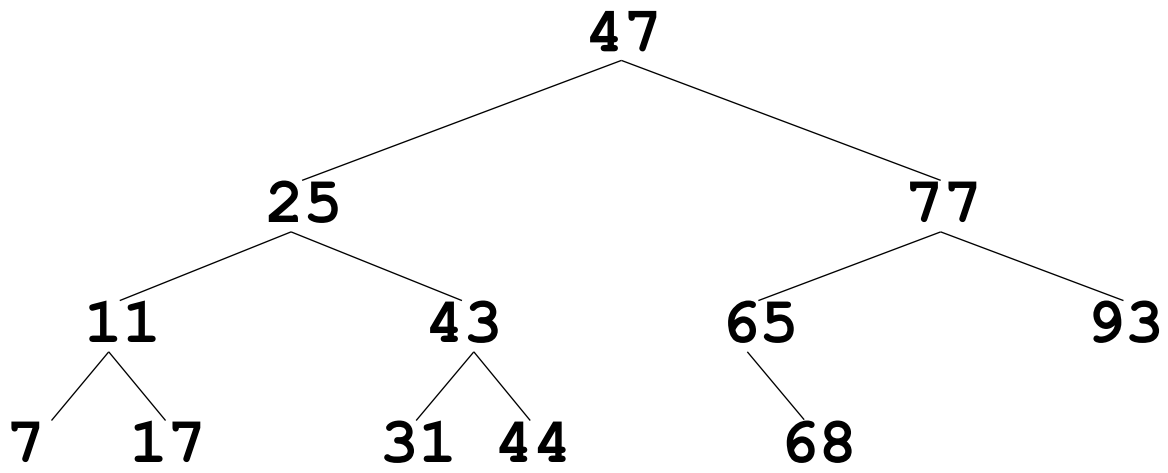
Trees

- 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

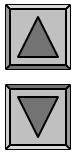
- Inserting nodes
  - Use recursive function
  - Begin at root
  - If current node empty, insert new node here (base case)
  - Otherwise,
    - If value > node, insert into right subtree
    - If value < node, insert into left subtree
    - If neither > nor <, must be =
      - Ignore duplicate

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`**

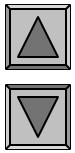


Outline

treenode.h (1 of 2)

```
1  // Fig. 17.17: treenode.h
2  // Template TreeNode class definition.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // forward declaration of class Tree
7  template< class NODETYPE > class Tree;
8
9  template< class NODETYPE >
10 class TreeNode {
11    friend class Tree< NODETYPE >;
12
13 public:
14
15    // constructor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( 0 ),
18         data( d ),
19         rightPtr( 0 )
20    {
21       // empty body
22
23    } // end TreeNode constructor
24
```

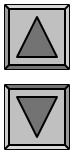
Binary trees have two pointers.



Outline

treenode.h (2 of 2)

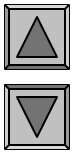
```
25    // return copy of node's data
26    NODETYPE getData() const
27    {
28       return data;
29
30    } // end getData function
31
32 private:
33    TreeNode< NODETYPE > *leftPtr;  // pointer to left subtree
34    NODETYPE data;
35    TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
36
37 }; // end class TreeNode
38
39 #endif
```



Outline

tree.h (1 of 6)

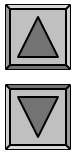
```
1  // Fig. 17.18: tree.h
2  // Template Tree class definition.
3  #ifndef TREE_H
4  #define TREE_H
5
6  #include <iostream>
7
8  using std::endl;
9
10 #include <new>
11 #include "treenode.h"
12
13 template< class NODETYPE >
14 class Tree {
15
16 public:
17    Tree();
18    void insertNode( const NODETYPE & );
19    void preOrderTraversal() const;
20    void inOrderTraversal() const;
21    void postOrderTraversal() const;
22
23 private:
24    TreeNode< NODETYPE > *rootPtr;
25
```



Outline

tree.h (2 of 6)

```
26    // utility functions
27    void insertNodeHelper(
28       TreeNode< NODETYPE > **, const NODETYPE & );
29    void preOrderHelper( TreeNode< NODETYPE > * ) const;
30    void inOrderHelper( TreeNode< NODETYPE > * ) const;
31    void postOrderHelper( TreeNode< NODETYPE > * ) const;
32
33 }; // end class Tree
34
35 // constructor
36 template< class NODETYPE >
37 Tree< NODETYPE >::Tree()
38 {
39    rootPtr = 0;
40
41 } // end Tree constructor
42
43 // insert node in Tree
44 template< class NODETYPE >
45 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
46 {
47    insertNodeHelper( &rootPtr, value );
48
49 } // end function insertNode
50
```



Outline

tree.h (3 of 6)

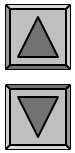
```
51 // utility function called by insertNode; receives a pointer
52 // to a pointer so that the function can modify pointer's value
53 template< class NODETYPE >
54 void Tree< NODETYPE >::insertNodeHelper(
55    TreeNode< NODETYPE > **ptr, const NODETYPE &value )
56 {
57    // subtree is empty; create new TreeNode containing value
58    if ( *ptr == 0 )
59       *ptr = new TreeNode< NODETYPE >( value );
60
61    else  // subtree is not empty
62
63       // data to insert is less than data in current node
64       if ( value < ( *ptr )->data )
65          insertNodeHelper( &( ( *ptr )->leftPtr ), value );
66
67       else
68
69          // data to insert is greater than data in current node
70          if ( value > ( *ptr )->data )
71             insertNodeHelper( &( ( *ptr )->rightPtr ), value );
72
73          else  // duplicate data value ignored
74             cout << value << " dup" << endl;
75
76 } // 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.

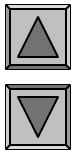


Outline

tree.h (4 of 6)

```
77
78 // begin preorder traversal of Tree
79 template< class NODETYPE >
80 void Tree< NODETYPE >::preOrderTraversal() const
81 {
82    preOrderHelper( rootPtr );
83
84 } // end function preOrderTraversal
85
86 // utility function to perform preorder traversal of Tree
87 template< class NODETYPE >
88 void Tree< NODETYPE >::preOrderHelper(
89    TreeNode< NODETYPE > *ptr ) const
90 {
91    if ( ptr != 0 ) {
92       cout << ptr->data << ' ';          // process node
93       preOrderHelper( ptr->leftPtr );    // go to left subtree
94       preOrderHelper( ptr->rightPtr );   // go to right subtree
95
96    } // end if
97
98 } // end function preOrderHelper
99
```

Preorder: print, left, right

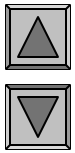


Outline

tree.h (5 of 6)

```
100 // begin inorder traversal of Tree
101 template< class NODETYPE >
102 void Tree< NODETYPE >::inOrderTraversal() const
103 {
104    inOrderHelper( rootPtr );
105
106 } // end function inOrderTraversal
107
108 // utility function to perform inorder traversal of Tree
109 template< class NODETYPE >
110 void Tree< NODETYPE >::inOrderHelper(
111    TreeNode< NODETYPE > *ptr ) const
112 {
113    if ( ptr != 0 ) {
114       inOrderHelper( ptr->leftPtr );   // go to left subtree
115       cout << ptr->data << ' ';        // process node
116       inOrderHelper( ptr->rightPtr );  // go to right subtree
117
118    } // end if
119
120 } // end function inOrderHelper
121
```

In order: left, print, right

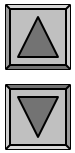


Outline

tree.h (6 of 6)

```
122 // begin postorder traversal of Tree
123 template< class NODETYPE >
124 void Tree< NODETYPE >::postOrderTraversal() const
125 {
126    postOrderHelper( rootPtr );
127
128 } // end function postOrderTraversal
129
130 // utility function to perform postorder traversal of Tree
131 template< class NODETYPE >
132 void Tree< NODETYPE >::postOrderHelper(
133    TreeNode< NODETYPE > *ptr ) const
134 {
135    if ( ptr != 0 ) {
136       postOrderHelper( ptr->leftPtr );   // go to left subtree
137       postOrderHelper( ptr->rightPtr );  // go to right subtree
138       cout << ptr->data << ' ';          // process node
139
140    } // end if
141
142 } // end function postOrderHelper
143
144 #endif
```

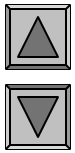
Postorder: left, right, print



Outline

fig17_19.cpp
(1 of 3)

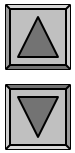
```
1  // Fig. 17.19: fig17_19.cpp
2  // Tree class test program.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include "tree.h"  // Tree class definition
13
14 int main()
15 {
16    Tree< int > intTree;  // create Tree of int values
17    int intValue;
18
19    cout << "Enter 10 integer values:\n";
20
21    for( int i = 0; i < 10; i++ ) {
22       cin >> intValue;
23       intTree.insertNode( intValue );
24
25    } // end for
```



Outline

fig17_19.cpp
(2 of 3)

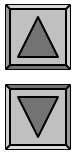
```
26
27    cout << "\nPreorder traversal\n";
28    intTree.preOrderTraversal();
29
30    cout << "\nInorder traversal\n";
31    intTree.inOrderTraversal();
32
33    cout << "\nPostorder traversal\n";
34    intTree.postOrderTraversal();
35
36    Tree< double > doubleTree;  // create Tree of double values
37    double doubleValue;
38
39    cout << fixed << setprecision( 1 )
40         << "\n\n\nEnter 10 double values:\n";
41
42    for ( int j = 0; j < 10; j++ ) {
43       cin >> doubleValue;
44       doubleTree.insertNode( doubleValue );
45
46    } // end for
47
48    cout << "\nPreorder traversal\n";
49    doubleTree.preOrderTraversal();
50
```



Outline

```
51    cout << "\nInorder traversal\n";
52    doubleTree.inOrderTraversal();
53
54    cout << "\nPostorder traversal\n";
55    doubleTree.postOrderTraversal();
56
57    cout << endl;
58
59    return 0;
60
61 } // end main
```

fig17_19.cpp
(3 of 3)



Outline

fig17_19.cpp
output (1 of 1)

```
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
```