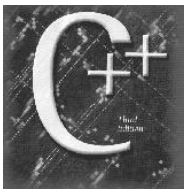

Modeling Data with Arrays, Structs, & Classes

Chapter 11

11.1 Multidimensional Arrays

- Complex arrangement of data
- Can have many dimensions (2x2)
- Syntax

  element-type arrayName [size 1] [size2];

  double table[NROWS] [NCOLS];

Declaring 2 Dimensional Arrays

- Declaration

  char ticTacToe [3][3];
- Storage location

  ticTacToe [1][2]

  Name Row Column

Example

```
const int Num_Rows = 11;
const int Seats_In_Row = 9;
string seatPlan [Num_Rows][Seats_In_Row];
```


Initialization

```
const int Num_Rows = 2;
const int Num_Cols = 3;
float matrix[Num_Rows][Num_Cols] ={{5.0,
                  4.5, 3.0}, {-16.0, -5.9, 0.0}};
```

- Nested loops with two-dimensional arrays
- SumMatrx.cpp example

SumMatrx.cpp

```
// FILE: SumMatrx.cpp
// CALCULATES THE SUM OF THE ELEMENTS IN THE
// FIRST rows ROWS OF AN ARRAY OF FLOATING POINT
// VALUES WITH num_cols (A CONSTANT) COLUMNS.
// Pre:  The type int constant num_cols is
// defined (NUM_COLS > 0) and the
// array element values are defined and rows > 0.
// Post: sum is the sum of all array element
// values.
// Returns:  Sum of all values stored in the
// array.
float sumMatrix (float table[][NUM_COLS],
                                    int rows)
```

SumMatrx.cpp

```
{
   // Local data ...
   float sum = 0.0;

   // Add each array element value to sum.
   for (int r = 0; r < rows; r++)
      for (int c = 0; c < NUM_COLS; c++)
         sum += table[r][c];

   return sum;
}
```


Bigger Arrays

const int people = 10;
const int years = 5;

money sales[people][years][12];

11.2 Array of Structs

- Declaration of an array of structs
  - employee company [10];
- Storage for 10 employee structs
- Can pass to a function
  - readEmployee (company[i]);
- Look at employee.h
- Array of structs being passed to ReadCompany.cpp

ReadCompany.cpp

```
// FILE: ReadCompany.cpp
// READS 10 EMPLOYEE RECORDS INTO ARRAY company

void readCompany (employee company[])
{
   // Read 10 employees.
   for (int i = 0; i < 10; i++)
   {
      cout << "Enter data for employee " <<
                        (i+1) << ":" << endl;
      readEmployee (company[i]);
   }
}
```


11.3 Template Classes

- Like functions use templates with classes
  - template <class T>
- Used before each class definition
- dummy.h is a definition file for a template class
- dummy.h is the header file and the definition for a template class named dummy

Dummy.h

```
// FILE: dummy.h
// HEADER FILE FOR TEMPLATE CLASS DUMMY

#ifndef DUMMY_H
#define DUMMY_H

template <class T>
class dummy
{
      public:
            // CONSTRUCTOR
            dummy ();
```

Dummy.h

```
            // STORES A VALUE OF TYPE T
            void setItem (const T&);

            // RETRIEVES A VALUE OF TYPE T
            void getItem () const;

      private:
            T value;
            bool defined;
};
#endif  // DUMMY_H
```


Template Class Syntax

Form:

```
template <class T>
class className
{
      public:

      private:

};
```


Object Definitions & Templates

- Form: **class-template-name <type> object;**
  - indexList<int> intList;
  - type can be any defined data type
  - class-template-name is name of template class
  - object is what is created of type<type>
- dummy <int> numDepend;
- dummy <string> spouseName;
  - DummyTest.cpp

DummyTest.cpp

```
// File: dummyTest.cpp
// Tests function dummy

#include "dummy.h"
#include <iostream>
#include <string>


using namespace std;
```


DummyTest.cpp

```
int main()
{
   dummy<int> numDepend;
   dummy<string> spouseName;
   int num;
   string name;

   numDepend.setItem(2);
   spouseName.setItem("Caryn");
```


DummyTest.cpp

```
  // Retrieve and display values stored
   num = numDepend.getItem();
   name = spouseName.getItem();
   cout << num << endl;
   cout << name << endl;

   return 0;
}
```

DummyTest.cpp

Program Output

2

Caryn

Implementation of a Template Class

- Two ways to implement
- Member functions in the class definition (.h file)
  - fig 12.9 DummyFun.h is rewritten with the complete functions contained in the definition file
- Other option is to isolate details of implementation in a .cpp file (dummy.cpp)

DummyFun.h

```
// File: dummyFunctions.h
// Definition file for template class dummy with
// functions

#include <iostream>
using namespace std;

#ifndef DUMMY_FUN_H
#define DUMMY_FUN_H
```


DummyFun.h

```
template <class T>
class dummy
{
   public:
      // constructor
      dummy()
      {
         defined = false;
      }
```


DummyFun.h

```
      // Stores a value of type t
      void setItem(const T& aVal)
      {
         item = aVal;
         defined = true;
      }

      // Retrieves a value of type t
      T getItem() const
      {
         if (defined)
            return item;
```


DummyFun.h

```
         else
            cerr << "Error - no value stored!" <<
                                          endl;
      }

   private:
      T item;
      bool defined;
};

#endif  // DUMMY_FUN_H
```

Key Issues

- Notice in the next implementation file a *template <class T>* is a prefix to each function heading
- Also a *dummy<T>::* is used to let the compiler know that the function being defined is a member of a template class
  - class dummy with parameter T

Dummy.cpp

```
// File: dummy.cpp
// Implementation file for template class dummy
#include "dummy.h"
#include <iostream>
using namespace std;

// constructor
template <class T>
dummy<T>::dummy()
{
   defined = false;    // No value stored yet
}
```


Dummy.cpp

```
// Stores a value of type t
template <class T>
void dummy<T>::setItem(const T& aVal)
{
   item = aVal;
   defined = true;
}
```


Dummy.cpp

```
// Retrieves a value of type t
template <class T>
T dummy<T>::getItem() const
{
   if (defined)
      return item;
}
```


11.4 The Indexed List Abstract Data Type

- Discuss a data structure called an indexed list
- Implement the structure with a template class
- Some deficiencies with arrays in C++
  - Can be overwritten
  - Must track the size of the array
    - filled or some empty spots
  - Functions to locate various things in arrays
    - find smallest, find largest etc
- Look at IndexList.h

IndexList.h

```
// File IndexList.h

#ifndef INDEXLIST_H
#define INDEXLIST_H
#include <iostream>
using namespace std;

template <class T, int maxSize>
class indeList
{
public:
  // Constructor
  indexList ();
```

IndexList.h

```
// Add an item to the end of an indexed list
bool append (const T&);

// Replace an element at a specified index
bool replace (int, const T&);

// Insert an element at a specified index
bool insert (int, const T&);

// Retrieve an element at a specified index
bool retrieve (int, T&) const;
```


IndexList.h

```
// Delete an element at a specified index
bool remove (int);

// Find index of smallest value in a sublist
int findMin (int, int) const;

// Find index of largest value in a sublist
int findMax (int, int) const;

// Find index of a target item
// Returns -1 if target item not found
int search (const T&) const;
```


IndexList.h

```
// Sort an indexed list
void selSort ();

// Read data into the list
 void read (istream &);

// Display the list contents
void display () const;

// Get the current size
int getSize () const;
```


IndexList.h

```
 //Data Members
  T elements[maxSize];
  int size;
};

#endif  // INDEXLIST_H
```


Using the Index List

- User friendly array
- Create a list
  - indexList<int, 10> myIntData;
  - indexList<string, 5> myStringData;
- Find something in the list
  - myIntData.retrieve (0, anInt);
- Other operations

IndexListTest.cpp

```
// FILE: indexListTest.cpp
// Tests member functions of indexed list class

#include "indexList.h"
#include "indexList.cpp"
#include <iostream>
#include <string>

using namespace std;
```

IndexListTest.cpp

```
int main()
{
  indexList<int, 10>   myIntData;
  indexList<string, 5> myStringData;
  string aString;
  int anInt;
  bool aBool;

  // Store the integer data.
  myIntData.append(5);
  myIntData.append(0);
  myIntData.append(-5);
  myIntData.append(-10);
```


IndexListTest.cpp

```
// Store the string data.
cout << "Read a list of strings:" << endl;
myStringData.read();

// Sort the indexed lists.
//myIntData.selSort();
//myStringData.selSort();

// Retrieve and display the first value in
// each list.
aBool = myIntData.retrieve(0, anInt);
      int bInt;
```


IndexListTest.cpp

```
aBool = myIntData.retrieve(1, bInt);
if (anInt == bInt)
  cout << "Equal" << endl;
else if (anInt < bInt)
  cout << "Less than"  << endl;
else
  cout << "not equal" << endl;

int index;
index =  myIntData.search(bInt);
cout << "index = " << index << endl;
```


IndexListTest.cpp

```
if (aBool)
   cout << "First integer value after sorting
           is "<< anInt << endl;

aBool = myStringData.retrieve(0, aString);
if (aBool)
   cout << "First string value after sorting
    is "<< aString << endl << endl;

// Display each list size and contents
cout << "The indexed list of integers
    contains "<< myIntData.getSize() <<
                 " values." << endl;
```


IndexListTest.cpp

```
  cout << "Its contents follows:" << endl;
  myIntData.display();

  cout << endl <<
 "The indexed list of strings contains "  <<
                 myStringData.getSize () <<
                       " values." << endl;
  cout << "Its contents follows:" << endl;
  myStringData.display();

  return 0;
}
```


IndexListTest.cpp

Read a list of strings:
Enter number of list items to read: **3**
Enter next item - **Robin**
Enter next item - **Beth**
Enter next item - **Koffman**
First integer value after sorting is -10
First string value after sorting is Beth
etc etc etc
See page 568 for remaining output

11.5 Implementing the Indexed List Class

- Details of the indexed list class
- Notice that each function begins with
  - *template <class T, int maxSize>*
- Notice that each function contains the class name and scope resolution operator
  - *indexList<T, maxSize>::*

IndexList.cpp

```
// File: indexList.cpp
// Indexed list class implementation

#include "indexList.h"
#include <iostream>
using namespace std;

template <class T, int maxSize>
indexList<T, maxSize>::indexList()
{
  size = 0;       // list is empty
}
```


IndexList.cpp

```
// Add an item to the end of an indexed list
template <class T, int maxSize>
bool indexList<T, maxSize>::append(const T& item)

// Pre:  item is defined
// Post: if size < maxSize, item is appended to
// list
// Returns: true if item was appended; otherwise,
// false
{
  bool result;
```


IndexList.cpp

```
// Add item to the end of the list if not full.
  if (size < maxSize)
  {
      elements[size] = item;
      size++;
      result = true;
  }
  else
  {
      cerr << "Array is filled - can't append!"
                          << endl;
      result = false;
```


IndexList.cpp

```
  }
  return result;
}
// Replace an item at a specified index with a
// new one.
// Pre:  item and index are defined
// Post: item overwrites old item at position
// index if valid
// Returns: true if item was inserted; otherwise,
// false
```


IndexList.cpp

```
template <class T, int maxSize>
bool indexList<T, maxSize>::replace(int index,
                          const T& item)
{
  bool result;

  // Overwrite a list element if index is valid.
  if (index >= 0  &&  index < size)
  {
      elements[index] = item;
      result = true;
  }
```

IndexList.cpp

```
    else
    {
        cerr << "Index " << index << " not in
              filled part" << " - can't insert!" <<
                                              endl;
        result = false;
    }
    return result;
}
```


IndexList.cpp

```
template <class T, int maxSize>
bool indexList<T, maxSize>::insert(int index,
                        const T& item)
{
  bool result;
       int i;

  // Insert a list element if index is valid.
  if (index >= 0  &&  index < size && size <
                        maxSize)
  {
```


IndexList.cpp

```
            for (i = size - 1; i >= index; i--)
                elements[i + 1] = elements[i];
        elements[index] = item;
                size++;
        result = true;
    }
    else
    {
        cerr << "Index " << index << " not in
              filled part" << " - can't insert!" <<
                             endl;
        result = false;
    }
```


IndexList.cpp

```
  return result;
}
// Retrieve an item at a specified index
// Pre:  item and index are defined
// Post: if index is valid, elements[index] is
// returned
// Returns: true if item was returned; otherwise,
// false
template <class T, int maxSize>
bool indexList<T, maxSize>::retrieve(int index,
                               T& item) const
{
```


IndexList.cpp

```
    bool result;

    // Return a list element through item if
    // index is valid.
    if (index >= 0  &&  index < size)
    {
        item = elements[index];
        result = true;
    }
    else
    {
```


IndexList.cpp

```
        cerr << "Index " << index << " not in
              filled part" << " - can't retrieve!"
                                    << endl;
        result = false;
    }
    return result;
}
```

IndexList.cpp

```
// Delete an element at a specified index
// Pre:  index is defined
// Post: if index is valid, elements[index] is
// replaced with elements[size] and size is
// decremented.
// Returns: true if item was deleted; otherwise,
// false

template <class T, int maxSize>
bool indexList<T, maxSize>::remove(int index)
{
  bool result;
```


IndexList.cpp

```
    int i;

  // Delete element at index i by moving
  // elements up
  if (index >= 0  &&  index < size)
  {
      // Shift each element up 1 position
      for (i = index + 1; i < size; i++)
                elements[i-1] = elements[i];
      size--;           // Decrement size
      result = true;
  }
```


IndexList.cpp

```
  else
  {
      cerr << "Index " << index << " not in
  filled part"
           << " - can't delete!" << endl;
      result = false;
  }
  return result;
}
```


IndexList.cpp

```
// Read data into the list
// Pre:  none
// Post: All data items are stored in array
// elements and size is the count of items
template <class T, int maxSize>
void indexList<T, maxSize>::read(istream& ins)
{
  T nextItem;
  size = 0;
  ins >> nextItem;
  while (!ins.eof() && size < maxSize)
  {
```


IndexList.cpp

```
      elements[size] = nextItem;
           size++;
           cout << "Enter next item - ";
           ins >> nextItem;
  }
}

// Read data into the list
// Pre:  none
// Post: All data items are stored in array
// elements and size is the count of items
```


IndexList.cpp

```
template <class T, int maxSize>
void indexList<T, maxSize>::read(istream& ins)
{
  int numItems;
  T nextItem;

  cout << "Enter number of list items to read: ";
  ins >> numItems;
  ins.ignore(80, '\n');

  size = 0;
```

IndexList.cpp

```
   if (numItems >= 0  &&  numItems <= maxSize)
      while (size < numItems)
      {
         cout << "Enter next item - ";
         ins >> nextItem;
         elements[size] = nextItem;
         size++;
      }
   else
```


IndexList.cpp

```
         cerr << "Number of items " << numItems
              << " is invalid"
              << " - data entry is cancelled!" <<
                                          endl;
}

// Display the list contents
// Pre:  none
// Post: Displays each item stored in the list
template <class T, int maxSize>
void indexList<T, maxSize>::display() const
{
```


IndexList.cpp

```
   // Display each list element.
   for (int i = 0; i < size; i++)
       cout << elements[i] << endl;
}
```


IndexList.cpp

```
// Find index of a target item
// Pre:  none
// Post: Returns the index of target if found;
// otherwise, return -1.

template <class T, int maxSize>
int indexList<T, maxSize>::search(const T&
                           target) const
{
  for (int i = 0; i < maxSize; i++)
      if (elements[i] == target)
            return i;
```


IndexList.cpp

```
   // target not found
   return -1;
}


// Get the current size
template <class T, int maxSize>
int indexList<T, maxSize>::getSize() const
{
  return size;
}
```


IndexList.cpp

```
// Sort the indexed list
template <class T, int maxSize>
void indexList<T, maxSize>::selSort()
{
  // Selection sort stub - do nothing
}
```

11.6 Illustrating Object Oriented Design

Telephone Case Study

- Identify objects & define services
- Identify interactions among objects
- Determine the specification for each object
- Implement each object

- As the author states OO Design is outside the scope of this course. Provide an idea of what is required

Specification

- Attributes
  - name & number
- Member Functions
  - entry, setEntry, getName, getNumber
- Operations
  - ==, <, >, <<, >>

Entry.h

```
// FILE: entry.h
// DEFINITION FOR A DIRECTORY ENTRY CLASS
#include <iostream>
#include <string>
using namespace std;
#ifndef ENTRY_H
#define ENTRY_H
class entry
{
public:
        entry ();
        void setEntry (const string&,
                  const string& nr = "");
```


Entry.h

```
// ACCESSOR FUNCTIONS
string getName () const;
string getNumber () const;

// Operators
friend bool operator == (const entry&);
friend bool operator <  (const entry&);
friend bool operator >  (const entry&);
```


Entry.h

```
   friend ostream& operator << (ostream&,
                                const entry&);
   friend istream& operator >> (istream&,
                                      entry&);

private:
   string name;      // Person's name
   string number;    // and phone number
};

#endif  // ENTRY_H
```


TelDirecMenu.cpp

```
// FILE: TelDirecMenu.cpp
// MENU DRIVEN TELEPHONE DIRECTORY UPDATE PROGRAM

#include <iostream>
#include <string>
#include <cctype>         // For toupper
#include "indexList.h"
#include "indexList.cpp"
#include "entry.h"
#include "entry.cpp"

typedef indexList<entry, 100> telIndexList;
```

TelDirecMenu.cpp

```
// Function prototype
// PERFORMS USER SELECTION
void select (telIndexList&, char);

int main ()
{
   // Local data
   telIndexList telDirec;
   char choice;
   // Read the initial directory.
   cout << "Enter the initial directory
                        entries - ";
   telDirec.read ();
```


TelDirecMenu.cpp

```
// Keep reading and performing operations
// until user enters Q
 do
 {
    // Display the menu.
    cout << "Choose an operation from the list
                        below:" << endl;
    cout << "A(Add), C(Change), D(Delete)," <<
                endl << "G (Get), S(Sort),
            P(Print),  Q(Quit): " << endl;
    cin >> choice;
```


TelDirecMenu.cpp

```
      // Perform the operation selected.
      select (telDirec, choice);
   }
   while (toupper (choice) != 'Q');
  return 0;
}
// PERFORMS USER SELECTION
void select (telIndexList& telDirec,
                              char choice)
{
   // Local data
   entry anEntry;      // one entry
   string aName;       // input - entry name
   int index;          // index of name
```


TelDirecMenu.cpp

```
char answer;
switch (toupper (choice))
{
   case 'A':   // Add an entry
      cout << "Enter entry to add - ";
      cin >> anEntry;
      telDirec.append (anEntry);
      break;

   case 'C':   // Change an entry
      cout << "Enter entry to change - ";
      cin >> anEntry;
```


TelDirecMenu.cpp

```
      index = telDirec.search (anEntry);
      if (index >= 0)
         telDirec.insert (index, anEntry);
      else
      {
         cout << "Name not in directory. " <<
         "Do you wish to add it (Y or N): ";
         cin >> answer;
         if (toupper (answer) == 'Y')
            telDirec.append (anEntry);
      }
      break;
```


TelDirecMenu.cpp

```
   case 'D':   // Delete an entry
      cout <<
         "Enter name of entry to delete: ";
      cin.ignore (1, '\n');
      getline (cin, aName, '\n');
      anEntry.setEntry (aName);
      index = telDirec.search (anEntry);
      if (index >= 0)
         telDirec.remove (index);
      else
         cout <<
         "Entry not found - no deletion" <<
                                       endl;
      break;
```

TelDirecMenu.cpp

```
      case 'G':   // Get a number
         cout << "Enter name of entry to get: ";
         cin.ignore (1, '\n');
         getline (cin, aName, '\n');
         anEntry.set_entry (aName);
         index = telDirec.search (anEntry);
         if (index >= 0)
         {
            telDirec.retrieve (index, anEntry);
            cout <<
             "The number you requested is " <<
                  anEntry.getNumber () << endl;
         }
```


TelDirecMenu.cpp

```
         else
            cout << "Unlisted number" << endl;
         break;

      case 'S':   // Sort directory
         telDirec.selSort ();
         break;

      case 'P':   // Print directory
         telDirec.display ();
         break;
```


TelDirecMenu.cpp

```
      case 'Q':   // Quit directory
         cout << "Exiting program" << endl;
         break;

      default:
         cout << "Choice is invalid - try again"
                                    << endl;
         cin.ignore (80, '\n');
   }
}  // end select
```


TelDirecMenu.cpp

Program Output

Enter the initial directory entries -
Enter the number of list items to read: **1**
Enter the next item - Enter name: **Maria Sanchez**
Enter number: **215-555-1234**
Enter your choice -
A(add), C(change), D(delete)
G(get), S(sort), P(print), Q(quit)

See page 583 for additional output

11.7 Operator Overloading and Friends

- Ways to call some class functions
- a.lessThan (b)
  - A and b are objects that are compared by passing one object to the other objects member function lessThan
- lessThan2 (a, b)
  - Both objects are treated the same
  - Must use a friend function to do this

Function lessThan

```
bool entry::lessThan (const entry& anEntry) const
{
      return (name<anEntry.name);
}
```

Function lessThan2

bool lessThan2 (const entry& entry1, const entry$ entry2)

{

return (entry1.name < entry2.name);

}

- Notice no **::** Because function lseethan2 is a friend function not a member function of the class

Friend Function Declaration

- Form:
  - ***friend result-type function-name (arg list);***
  - ***friend result-type operator opSymbol (arg list);***
- Example:
  - ***friend bool equals (const entry& , const entry&);***
  - ***friend bool operator == (const entry& , const entry&);***

Entry.cpp

```
// File: entry.cpp
// Implementation file for class entry

#include "entry.h"
#include <iostream>
#include <string>
using namespace std;
// constructor
entry::entry()
{
   name = "";
   number = "";
}
```


Entry.cpp

```
// Store data in an entry
void entry::setEntry(const string& na,
                     const string& nr)
{
   name = na;
   number = nr;
}
// Get name
string entry::getName() const
{
   return name;
}
```


Entry.cpp

```
// Get number
string entry::getNumber() const
{
   return number;
}
// Operators
/*
bool entry::operator == (const entry& dE)
{
   return (name == dE.name);
}
*/
```


Entry.cpp

```
bool operator == (const entry& dE1,
                  const entry&dE)
{
   return (dE1.name == dE.name);
}

bool entry::operator < (const entry& dE)
{
   return (name < dE.name);
}
```

Entry.cpp

```
bool entry::operator > (const entry& dE)
{
   return (name > dE.name);
}
// friends
ostream& operator <<
            (ostream& outs, const entry& dE)
{
   outs << "Name   is " << dE.name << endl;
   outs << "Number is " << dE.number << endl;

   return outs;
}
```


Entry.cpp

```
istream& operator >>
                  (istream& ins, entry& dE)
{
   cout << "Enter name: ";
   getline(ins, dE.name, '\n');
   cout << "Enter number: ";
   ins >> dE.number;

   return ins;
}
```


11.8 The Stack Abstract Data Type

- Stack ADT is a structure where only the top element can be accessed
  - Stack of plates in a cafeteria
- Pushing and popping a stack
- Compilers use stacks to store procedures and function arguments
- Keep track of things in the program
- Implement a stack with a template class

Stack.h

```
//FILE: stack.h
#ifndef STACK_H
#define STACK_H

template <class stackElType,
                        int maxSize = 100>
class stack
{
   public:
      // Member functions ...
      // CONSTRUCTOR TO CREATE AN EMPTY STACK
      stack ();
```


Stack.h

```
      bool push (const stackElType& x);
      bool pop (stackElType& x);
      bool peek (stackElType& x) const;
      bool isEmpty () const;
      bool isFull () const;
private:
      int top;
      stackElType items[maxSize];
};

#endif
```


StackText.cpp

```
// File: StackText.cpp
// Use a stack to store strings and display
// them in reverse order

#include "stack.h"
#include <string>
#include <iostream>
using namespace std;
typedef stack<string, 20> stringStack;
int fillStack(stringStack& s);
void displayStack(stringStack& s);
```

StackText.cpp

```
int main()
{
   // Local data
   stringStack s;

   // Read data into the stack.
   fillStack(s);

   // Display the stack contents
   displayStack(s);

   return 0;
}
```


StackText.cpp

```
//    Reads data characters and pushes them
//    onto stack s.
//    Pre : s is an empty stack.
//    Post: s contains the strings read in
//    reverse order.
//    Returns the number of strings read not
//    counting the sentinel.

int fillStack(stringStack& s)
{
   // Local data
      string nextStr;
```


StackText.cpp

```
   int numStrings;
   const string sentinel = "***";

   numStrings = 0;
   cout << "Enter next string or " <<
                        sentinel << "> ";
   cin >> nextStr;
   while ((nextStr != sentinel) && (!s.isFull()))
   {
      s.push (nextStr);
      numStrings++;
      cout << "Enter next string or " <<
                        sentinel << "> ";
```


StackText.cpp

```
cin >> nextStr;
   }

   return numStrings;
}  // end fillStack


// Pops each string from stack s and displays it.
// Pre : Stack s is defined.
// Post: Stack s is empty and all strings
// are displayed.
void displayStack(stringStack& s)
```


StackText.cpp

```
{
   // Local data
   string nextStr;

   // Pop and display strings until stack
   // is empty.
   while (!s.isEmpty())
   {
      s.pop(nextStr);   // Pop next string off.
      cout << nextStr << endl;
   }
}  // end displayStack
```


StackText.cpp

Enter a phrase and press return >

Here are strings!

strings!
are
Here

11.9 Implementing the Stack Class

- Details of stack implementation
- Two private data members
  - int top // Index of stack top
  - array items; // Place were data is stored
- Look at stack.cpp

Stack.cpp

```
// File: stack.cpp
// Implementation of template class stack

#include "stack.h"

// constructor to create an empty stack

template <class stackElType, int maxSize>
stack<stackElType, maxSize>::stack ()
{
   top = -1;
}
```


Stack.cpp

```
// Push an element onto the stack
// Pre:  The element x is defined.
// Post: If the stack is not full, the item
// is pushed onto the stack and true is
// returned. Otherwise, the stack is unchanged
// and false is returned.

template <class stackElType, int maxSize>
bool stack<stackElType, maxSize>::push
   (const stackElType& x)
{
   bool success;
```


Stack.cpp

```
   if (top < maxSize - 1)     // If there is room
   {
      top++;                   // increment top
      items[top] = x;          // insert x
      success = true;
   }
   else
      success = false;   // no room, failure

   return success;
}  // end push
```


Stack.cpp

```
// Pop an element off the stack
// Pre:  none
// Post: If the stack is not empty, the value
// at the top of the stack is removed, its
// value is placed in x, and true is returned.
// If the stack is empty, x is not
// defined and false is returned.

template <class stackElType, int maxSize>
bool stack<stackElType, maxSize>::pop
   (stackElType& x)   // OUT: Element popped
{
   bool success;
```


Stack.cpp

```
    if (top >= 0)
   {
      x = items[top];
      top--;
      success = true;
   }
   else
      success = false;

   return success;
}  // end pop
```

Stack.cpp

```
// Access top element from stack without popping
// Pre:  none
// Post: If the stack is not empty, the value
// at the top is copied into x and true is
// returned. If the stack is empty, x is not
// defined and false is returned. In either
// case, the stack is not changed.

template <class stackElType, int maxSize>
bool stack<stackElType, maxSize>::peek
   (stackElType& x)const
{
   bool success;
```


Stack.cpp

```
   if (top >= 0)                // if not empty
   {
      x = items[top];           // retrieve top
      success = true;           // success
   }
   else
      success = false;          // failure

   return success;
}  // end peek
```


Stack.cpp

```
// Test to see if stack is empty
// Pre:  none
// Post: Returns true if the stack is empty;
// otherwise, returns false.

template <class stackElType, int maxSize>
bool stack<stackElType, maxSize>::isEmpty() const
{
   return top < 0;
}
```


Stack.cpp

```
// Test to see if stack is full
// Pre:  none
// Post: Returns true if the stack is full;
// otherwise, returns false.

template <class stackElType, int maxSize>
bool stack<stackElType, maxSize>::isFull() const

{
   return top >= maxSize-1;
}
```


11.10 Common Programming Errors

- Same errors to watch out for that we discussed in chapters 9 an 11
- Multidimensional arrays and subscripts
- Use of nested for loops to access data
- Template definitions
- Use of scope resolution operator
- Keyword friend when using friend functions