CSC224: Data Structures: Tutorial2

Arrays

An array is a number of data items of the same type arranged contiguously in memory. The array is the most commonly used data storage structure; it's built into most programming languages. Because they are so well-known, arrays offer a convenient jumping-off place for introducing data structures and for seeing how object-oriented programming and data structures relate to each other

An Array Example

Let's look at some sample programs that show how an array can be used. In case you're making the transition to OOP, we'll start with an old-fashioned procedural version, and then show the equivalent object-oriented approach. Listing 2.1 shows the old-fashioned version, called array.cpp.

```
//array.cpp
//demonstrates arrays
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
int main()
{
int arr[100]; //array
int nElems = 0; //number of items
int j; //loop counter
int searchKey; //key of item to search for
//-----------------------------------------------------------
arr[0] = 77; //insert 10 items
arr[1] = 99;
arr[2] = 44;
arr[3] = 55;
arr[4] = 22;
arr[5] = 88;
arr[6] = 11;
arr[7] = 00;
arr[8] = 66;
arr[9] = 33;
nElems = 10; //now 10 items in array
//-----------------------------------------------------------
for(j=0; j<nElems; j++) //display items
cout << arr[j] << " ";
cout << endl;
//-----------------------------------------------------------
searchKey = 66; //find item with key 66
```

```
for(j=0; j<nElems; j++) //for each element,
if(arr[j] == searchKey) //found item?
break; //yes, exit before end
if(j == nElems) //at the end?
cout << "Can't find " << searchKey << endl; //yes
else
cout << "Found " << searchKey << endl; //no
//-------------------------------------------------------------
searchKey = 55; //delete item with key 55
cout << "Deleting " << searchKey << endl;
for(j=0; j<nElems; j++) //look for it
if(arr[j] == searchKey)
break;
for(int k=j; k<nElems; k++) //move higher ones down
arr[k] = arr[k+1];
nElems--; //decrement size
//-------------------------------------------------------------
for(j=0; j<nElems; j++) //display items
cout << arr[j] << " ";
cout << endl;
return 0;
} //end main()
```

In this program, we create an array called arr, place 10 data items (kids' numbers) in it, search for the item with value 66 (the shortstop, Louisa), display all the items, remove the item with value 55 (Freddy, who had a dentist appointment), and then display the remaining nine items. The output of the program looks like this:

```
77 99 44 55 22 88 11 0 66 33
Found 66
77 99 44 22 88 11 0 66 33
```

The data we're storing in this array is type int. We've chosen a basic type to simplify the coding. Generally the items stored in a data structure consist of several data members, so they are represented by objects rather than basic types. We'll see an example of this in the next hour.

Inserting a New Item

Inserting an item into the array is easy; we use the normal array syntax arr[0] = 77;

We also keep track of how many items we've inserted into the array with the nElems variable.

Searching for an Item

The searchKey variable holds the value we're looking for. To search for an item, step through the array, comparing searchKey with each element. If the loop variable j reaches the last occupied cell with no match being found, the value isn't in the array. Appropriate messages are displayed: Found 66 or Can't find 27.

Deleting an Item

Deletion begins with a search for the specified item. For simplicity we assume (perhaps rashly) that the item is present. When we find it, we move all the items with higher index values down one element to fill in the "hole" left by the deleted element, and we decrement nElems. In a real program, we would also take appropriate action if the item to be deleted could not be found.

Displaying the Array Contents

Displaying all the elements is straightforward: we step through the array, accessing each one with arr[j] and displaying it.

Program Organization

The organization of array.cpp leaves something to be desired. There are no classes; it's just an old-fashioned procedural program. Let's see if we can make it easier to understand (among other benefits) by making it more object-oriented.

We're going to provide a gradual introduction to an object-oriented approach, using two steps. In the first step, we'll separate the data storage structure (the array) from the rest of the program by making it into a separate class. The remaining part of the program (the main() function) will become a user of the structure. In the second step, we'll improve the communication between the storage structure and its user. We'll look at these steps in the next two sections.

Dividing a Program into Classes

The array.cpp program consisted of one big function. We can reap many benefits by dividing the program into several parts: The data storage structure itself is one candidate, and the part of the program that uses this data structure is another. The first part can be represented as a class, and the second by the special function main(). By dividing the program into these two parts we can clarify the functionality of the program, making it easier to design and understand (and, in real programs, easier to modify and maintain).

In array.cpp we used an array as a data storage structure, but we treated it simply as a language element. Now we'll encapsulate the array in a class, called LowArray. We'll also provide class member functions by which statements in main() can access the array. These member functions allow communication between LowArray and main().
Our first design of the LowArray class won't be entirely successful. It's an improvement on array.cpp, but it nevertheless demonstrates the need for an even better approach. The lowArray.cpp program in Listing 2.2 shows how it looks.

THE lowArray.cpp PROGRAM

```
//lowArray.cpp
//demonstrates array class with low-level interface
#include <iostream>
#include <vector>
using namespace std;
////////////////////////////////////////////////////////////////
class LowArray
{
private:
vector<double> v; //vector holds doubles
public:
//--------------------------------------------------------------
LowArray(int max) //constructor
{ v.resize(max); } //size the vector
//--------------------------------------------------------------
void setElem(int index, double value) //put element into
{ v[index] = value; } //array, at index
//--------------------------------------------------------------
double getElem(int index) //get element from
{ return v[index]; } //array, at index
//--------------------------------------------------------------
}; //end class LowArray
////////////////////////////////////////////////////////////////
int main()
{
LowArray arr(100); //create a LowArray
int nElems = 0; //number of items
int j; //loop variable
//--------------------------------------------------------------
arr.setElem(0, 77); //insert 10 items
arr.setElem(1, 99);
arr.setElem(2, 44);
arr.setElem(3, 55);
arr.setElem(4, 22);
arr.setElem(5, 88);
arr.setElem(6, 11);
arr.setElem(7, 00);
arr.setElem(8, 66);
arr.setElem(9, 33);
```

```
nElems = 10; //now 10 items in array
//-----------------------------------------------------------
for(j=0; j<nElems; j++) //display items
cout << arr.getElem(j) << " ";
cout << endl;
//-----------------------------------------------------------
int searchKey = 26; //search for item
for(j=0; j<nElems; j++) //for each element,
if(arr.getElem(j) == searchKey) //found item?
break;
if(j == nElems) //no
cout << "Can't find " << searchKey << endl;
else //yes
cout << "Found " << searchKey << endl;
//-----------------------------------------------------------
double deleteKey = 55; //delete value 55
cout << "Deleting element " << deleteKey << endl;
for(j=0; j<nElems; j++) //look for it
if(arr.getElem(j) == deleteKey)
break;
for(int k=j; k<nElems; k++) //higher ones down
arr.setElem(k, arr.getElem(k+1) );
nElems--; //decrement size
//-----------------------------------------------------------
for(j=0; j<nElems; j++) //display items
cout << arr.getElem(j) << " ";
cout << endl;
return 0;
} //end main()
```

The output from this program is similar to that from array.cpp, except that we try to find a non-existent key value (26) before deleting the item with the key value 55.

```
77 99 44 55 22 88 11 0 66 33
Can't find 26
77 99 44 22 88 11 0 66 33
```

The LowArray **Class and** main()

In lowArray.cpp, we wrap the class LowArray around an ordinary C++ array. The array is hidden from the outside world inside the class; it's private, so only LowArray class member functions can access it. There are three such functions: setElem() and getElem(), which insert and retrieve an element, respectively; and a constructor, which

creates an empty array of a specified size. Notice that we use a vector to store the data instead of an array. It operates in most ways
like an array, but allows us to specify the vector's size in the class constructor, using the resize() member function. This allows an object of class LowArray to hold any amount of data, unlike an array, which must always hold the same amount of data.

The main() function creates an object of the LowArray class and uses it to store and manipulate data. Think of LowArray as a tool, and main() as a user of the tool. We've divided the program into two parts with clearly defined roles. This is a valuable first step in making a program object-oriented. A class used to store data objects, as is LowArray in the lowArray.cpp program, is sometimes called a *container class*. Typically, a container class not only stores
the data, but also provides member functions for accessing the data, and perhaps also sorting it and performing other complex actions on it. The problem with the lowArray.cpp program is that main() must do too much work. In the next section we'll see how to fix this, and introduce the idea of interfaces.

Class Interfaces

We've seen how a program can be divided into separate parts. How do these parts interact with each other? Communication between classes and other parts of the program, and the division of responsibility between them, are important aspects of object-oriented programming. This is especially true when a class might have many different users.
Typically a class can be used over and over by different users (or the same user) for different purposes. For example, it's possible that someone might use the LowArray class in some other program to store the serial numbers of her traveler's checks. The class can handle this just as well as it can store the numbers of baseball players.
If a class is used by many different programmers, the class should be designed so that it's easy to use. The way a class user relates to the class is called the class *interface*. Because class data members are typically private, when we talk about the interface we usually mean the class member functions: what they do and what their arguments are. It's by calling these member functions that a class user interacts with an object of the class. One of the important advantages conferred by object-oriented programming is that a class interface can be designed to be as convenient and efficient as possible. Figure 2.3 is a fanciful interpretation of the LowArray interface.

THE highArray.cpp PROGRAM

```
//highArray.cpp
//demonstrates array class with high-level interface
#include <iostream>
#include <vector>
using namespace std;
////////////////////////////////////////////////////////////////
class HighArray
{
```

```
private:
vector<double> v; //vector v
int nElems; //number of data items
public:
//--------------------------------------------------------------
HighArray() : nElems(0) //default constructor
{ }
//--------------------------------------------------------------
HighArray(int max) : nElems(0) //1-arg constructor
{ v.resize(max); } //size the vector
//--------------------------------------------------------------
bool find(double searchKey) //find specified value
{
int j;
for(j=0; j<nElems; j++) //for each element,
if(v[j] == searchKey) //found item?
break; //exit loop before end
if(j == nElems) //gone to end?
return false; //yes, can't find it
else
return true; //no, found it
} //end find()
//--------------------------------------------------------------
void insert(double value) //put element into array
{
v[nElems] = value; //insert it
nElems++; //increment size
}
//--------------------------------------------------------------
bool remove(double value) //remove element from array
{
int j;
for(j=0; j<nElems; j++) //look for it
if( value == v[j] )
break;
if(j==nElems) //can't find it
return false;
else //found it
{
for(int k=j; k<nElems; k++) //move higher ones down
```

```
v[k] = v[k+1];
nElems--; //decrement size
return true;
}
} //end delete()
//-------------------------------------------------------------
void display() //displays array contents
{
for(int j=0; j<nElems; j++) //for each element,
cout << v[j] << " "; //display it
cout << endl;
}
//-------------------------------------------------------------
}; //end class HighArray
////////////////////////////////////////////////////////////////
int main()
{
int maxSize = 100; //array size
HighArray arr(maxSize); //vector
arr.insert(77); //insert 10 items
arr.insert(99);
arr.insert(44);
arr.insert(55);
arr.insert(22);
arr.insert(88);
arr.insert(11);
arr.insert(0);
arr.insert(66);
arr.insert(33);
arr.display(); //display items
int searchKey = 35; //search for item
if( arr.find(searchKey) )
cout << "Found " << searchKey << endl;
else
cout << "Can't find " << searchKey << endl;
cout << "Deleting 0, 55, and 99" << endl;
arr.remove(0); //delete 3 items
arr.remove(55);
arr.remove(99);
arr.display(); //display items again
```

```
return 0;
} //end main()
```

The HighArray class is now wrapped around the array (actually a vector). In main(), we create an object of this class and carry out almost the same operations as in the lowArray.cpp program: we insert 10 items, search for an item—one that isn't there—and display the array contents. Because it's so easy, we delete three items (0, 55, and 99) instead of one, and finally display the contents again. Here's the output:

```
77 99 44 55 22 88 11 0 66 33
Can't find 35
Deleting 0, 55, and 99
77 44 22 88 11 66 33
```

Notice how short and simple main() is. The details that had to be handled by main() in lowArray.cpp are now handled by HighArray class member functions.

1. In the HighArray class, the find() member function looks through the array for the item whose key value was passed to it as an argument. It returns true or false, depending on whether it finds the item or not.
2. The insert() member function places a new data item in the next available space in the array. A data member called nElems keeps track of the number of array cells that are actually filled with data items. The main() function no longer needs to worry about how many items are in the array.
3. The delete() member function searches for the element whose key value was passed to it as an argument, and when it finds that element, it shifts all the elements in higher index cells down one cell, thus writing over the deleted value; it then decrements nElems. We've also included a display() member function, which displays all the values stored in the array.

Abstraction

The process of separating the *how* from the *what*—how an operation is performed inside a class, as opposed to what's visible to the class user—is called *abstraction*. Abstraction is an important aspect of software engineering. By abstracting class functionality we make it easier to design a program because we don't need to think about implementation details at too early a stage in the design process.

Summary

In this hour, you've learned the following:

- Unordered arrays offer fast insertion but slow searching and deletion.
- Wrapping an array in a class protects the array from being inadvertently altered.
- A class interface comprises the member functions (and occasionally data members) that the class user can access.
- A class interface can be designed to make things simpler for the class user (although possibly harder for the class designer).

Quiz

1. On average, how many items must be moved to insert a new item into an unsorted array with N items?
2. On average, how many items must be moved to delete an item from an unsorted array with N items?
3. On average, how many items must be examined to find a particular item in an unsorted array with N items?

4. What is a class interface?
5. Why is it important to make things easier for the class user than for the class designer?
6. What are the advantages of wrapping an array in a class?
7. What's an example of an operation that's easier to perform on an array that's in a class than on a simple array?
8. What is abstraction?

Abstract Data Types

Stacks

Queues and Priority Queues

Linked Lists

Abstract Data Types

Specialized Lists

Stacks

So far we've been looking at one kind of data structure: the array (often implemented as a vector). In this chapter we'll examine a different kind of structure: a stack. You'll learn

- How stacks differ philosophically from arrays
- How a stack operates
- How to create stacks in C++
- Some applications for stacks

Uses for Stacks and Queues: Programmer's Tools

The array—the data storage structure we've been examining thus far—as well as many other structures we'll encounter later in this book (linked lists, trees, and so on), are appropriate for the kind of data you might find in a database application. They're typically used for personnel records, inventories, financial data, and so on—data that corresponds to real-world objects or activities. These structures facilitate access to data: They make it easy to insert, delete, and search for particular items. The structures and algorithms we'll examine in this hour and the next, on the other hand, are more often used as programmer's tools. They're primarily conceptual aids rather than full-fledged data storage devices. Their lifetime is typically shorter than that of the database-type structures. They are created and used to carry out a particular task during the operation of a function within a program; when the task is completed, they're discarded.

Stacks and Queues: Restricted Access to Data

In an array, any item can be accessed, either immediately—if its index number is known—or by searching through a sequence of cells until it's found. In stacks and queues, however, access is restricted: Only one item can be read or removed at a given time. The interface of these structures is designed to enforce this restricted access. Access to other items is (in theory) not allowed.

Stacks and Queues: More Abstract

Stacks and queues are more abstract entities than arrays and many other data storage structures. They're defined primarily by their interface—the permissible operations that can be carried out on them. The underlying mechanism used to implement them is typically not visible to their users.
For example, the underlying mechanism for a stack can be an array, as we will show in this chapter, or it can be a linked list.

To better understand these ideas, let's look at how stacks work.

Understanding Stacks
A stack allows access to only one data item: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on. This is a useful capability in many programming situations. In this section, we'll see how a stack can be used to check whether parentheses, braces, and brackets are balanced in a computer program source file. Stacks also play a vital role in parsing (analyzing) arithmetic expressions such as 3*(4+5).
A stack is also a handy aid when you're programming algorithms for certain complex data structures. In Hour 16, "Traversing Binary Trees," for example, we'll see a stack is used in the code that traverses the nodes of a tree.
Most microprocessors (like the one in your computer) use a stack-based architecture. When a member function is called, its return address and arguments are pushed onto a stack, and when the function returns they're popped off. The stack operations are built into the microprocessor.
Some older pocket calculators used a stack-based paradigm. Instead of entering arithmetic expressions using parentheses, you pushed intermediate results onto a stack.

Placing a data item on the top of the stack is called *pushing* it. Removing it from the top of the stack is called *popping* it. These are the primary stack operations. A stack is said to be a Last-In-First-Out (LIFO) storage mechanism because the last item inserted is the first one to be removed.

Implementing a Stack in C++
Let's examine a program, Stack.cpp, that implements a stack using a class called StackX. Listing 6.1 contains this class and a short main() routine to exercise it.
LISTING 6.1 THE Stack.cpp PROGRAM

```
//Stack.cpp
//demonstrates stacks
#include <iostream>
#include <vector>
using namespace std;
////////////////////////////////////////////////////////////////
class StackX
{
private:
int maxSize; //size of stack vector
vector<double> stackVect; //stack vector
```

```
int top; //top of stack
public:
//--------------------------------------------------------------
StackX(int s) : maxSize(s), top(-1) //constructor
{
stackVect.reserve(maxSize); //size the vector
}
//--------------------------------------------------------------
void push(double j) //put item on top
{
stackVect[++top] = j; //increment top,
} //insert item
//--------------------------------------------------------------
double pop() //take item from top
{
return stackVect[top--]; //access item,
} //decrement top
//--------------------------------------------------------------
double peek() //peek at top of stack
{
return stackVect[top];
}
//--------------------------------------------------------------
bool isEmpty() //true if stack is empty
{
return (top == -1);
}
//--------------------------------------------------------------
bool isFull() //true if stack is full

{
return (top == maxSize-1);
}
//--------------------------------------------------------------
}; //end class StackX
////////////////////////////////////////////////////////////////
int main()
{
StackX theStack(10); //make new stack, size 10
theStack.push(20); //push items onto stack
```

```
theStack.push(40);
theStack.push(60);
theStack.push(80);
while( !theStack.isEmpty() ) //until it's empty,
{ //delete item from stack
double value = theStack.pop();
cout << value << " "; //display it
} //end while
cout << endl;
return 0;
} //end main()
```

The main() function creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it's empty. Here's the output:

80 60 40 20

Notice how the order of the data is reversed. Because the last item pushed is the first one popped, the 80 appears first in the output.

Error Handling

There are different philosophies about how to handle stack errors. What happens if you try to push an item onto a stack that's already full? Or pop an item from a stack that's empty?

In Stack.cpp we've left the responsibility for handling such errors up to the class user.

The user should always check to be sure the stack is not full before pushing a new item:

```
if( !theStack.isFull() )
theStack.push(item);
else
cout << "Can't insert, stack is full";
```

In the interest of simplicity, we've left this code out of the main() routine (and anyway, in this simple program, we know the stack isn't full because it has just been initialized).

We do include the check for an empty stack when main() calls pop(). Many stack classes check for these errors internally, in the push() and pop() member functions. This is the preferred approach. In C++, a good solution for a stack class that discovers such errors is to throw an exception, which can then be caught and processed by the class user.

Now that we've seen how to program a stack in general, let's look at some programs that solve real problems by using a stack.

Stack Example 1: Reversing a Word

For our first example of using a stack, we'll examine a very simple task: reversing a word. When you run the program, it asks you to type in a word. When you press Enter, it displays the word with the letters in reverse order.

A stack is used to reverse the letters. First the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped off the stack and displayed.
Because of its last-in-first-out characteristic, the stack reverses the order of the characters. Listing 6.2 shows the code for the reverse.cpp program.

THE reverse.cpp PROGRAM

```
//reverse.cpp
//stack used to reverse a word
#include <iostream>
#include <vector>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class StackX
{
private:
int maxSize;
vector<char> stackVect; //vector holds stack
int top;
public:
//--------------------------------------------------------------
StackX(int max) : maxSize(max), top(-1) //constructor
{
stackVect.resize(maxSize); //size the vector
}
//--------------------------------------------------------------
void push(char j) //put item on top of stack
{ stackVect[++top] = j; }
//--------------------------------------------------------------
char pop() //take item from top of stack
{ return stackVect[top--]; }
//--------------------------------------------------------------
char peek() //peek at top of stack
{ return stackVect[top]; }
//--------------------------------------------------------------
bool isEmpty() //true if stack is empty
{ return (top == -1); }
//--------------------------------------------------------------
}; //end class StackX
////////////////////////////////////////////////////////////////
```

```
class Reverser
{
private:
string input; //input string
string output; //output string
public:
//--------------------------------------------------------------
Reverser(string in) : input(in) //constructor
{ }
//--------------------------------------------------------------
string doRev() //reverse the word
{
int stackSize = input.length(); //get max stack size
StackX theStack(stackSize); //make stack
for(int j=0; j<input.length(); j++)
{
char ch = input[j]; //get a char from input
theStack.push(ch); //push it
}
output = "";
while( !theStack.isEmpty() )
{
char ch = theStack.pop(); //pop a char,
output = output + ch; //append to output
}
return output;
} //end doRev()
//--------------------------------------------------------------
}; //end class Reverser
////////////////////////////////////////////////////////////////
int main()
{
string input, output;
while(true)
{
cout << "Enter a word: ";
cin >> input; //read a word from kbd
if( input.length() < 2 ) //quit if one character
break;
//make a Reverser
```

```
Reverser theReverser(input);
output = theReverser.doRev(); //use it
cout << "Reversed: " << output << endl;
} //end while
return 0;
} //end main()
```

We've created a class Reverser to handle the reversing of the input string. Its key component is the member function doRev(), which carries out the reversal, using a stack. The stack is created within doRev(), which sizes it according to the length of the input string. In main() we get a string from the user, create a Reverser object with this string as an argument to the constructor, call this object's doRev() member function, and display the return value, which is the reversed string. Here's some sample interaction with the program:

```
Enter a word: part
Reversed: trap
Enter a word:
```

Stack Example 2: Delimiter Matching

One common use for stacks is to parse certain kinds of text strings. Typically the strings are lines of code in a computer language, and the programs parsing them are compilers. To give the flavor of what's involved; we'll show a program that checks the delimiters in a line of text typed by the user. This text doesn't need to be a line of real C++ code (although it could be), but it should use delimiters the same way C++ does. The delimiters are the braces { and }, brackets [and], and parentheses (and). Each opening or left delimiter should be matched by a closing or right delimiter; that is, every '{' should be followed by a matching '}', and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. Here are some examples:

```
c[d] // correct
a{b[c]d}e // correct
a{b(c]d}e // not correct; ] doesn't match (
a[b{c}d]e} // not correct; nothing matches final }
a{b(c) // not correct; Nothing matches opening {
```

Opening Delimiters on the Stack

The program works by reading characters from the string one at a time and placing opening delimiters, when it finds them, on a stack. When the program reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter. If the delimiters are not the same type (there's an opening brace but a closing parenthesis, for example), an error has occurred. Also, if there is no opening delimiter on the stack to match a closing one, or if a delimiter remains on the stack when the parse has ended, an error has occurred.

Let's see what happens on the stack for a typical correct string: a{b(c[d]e)f}

Table 6.1 shows how the stack looks as each character is read from this string. The entries in the second column show the stack contents, reading from the bottom of the stack on the left to the top on the right. As it's read, each opening

delimiter is placed on the stack. Each closing delimiter read from the input is matched with the opening delimiter popped from the top of the stack. If they form a pair, all is well. Non delimiter characters are not inserted on the stack; they're ignored.

C++ Code for brackets.cpp

The code for the parsing program, brackets.cpp, is shown in Listing 6.3. We've placed check(), the member function that does the parsing, in a class called BracketChecker.

THE brackets.cpp PROGRAM

```
//brackets.cpp
//stacks used to check matching brackets
#include <iostream>
#include <string>
#include <vector>
using namespace std;
////////////////////////////////////////////////////////////////
class StackX
{
private:
int maxSize; //size of vector
vector<char> stackVect; //vector for stack
int top; //top of stack
public:
//--------------------------------------------------------------
StackX(int s) : maxSize(s), top(-1) //constructor
{ stackVect.resize(maxSize); }
//--------------------------------------------------------------
void push(char j) //put item on top of stack
{ stackVect[++top] = j; }
//--------------------------------------------------------------
char pop() //take item from top of stack
{ return stackVect[top--]; }
//--------------------------------------------------------------
char peek() //peek at top of stack
{ return stackVect[top]; }
//--------------------------------------------------------------
bool isEmpty() //true if stack is empty
{ return (top == -1); }
//--------------------------------------------------------------
}; //end class StackX
```

```
////////////////////////////////////////////////////////////////
class BracketChecker
{
private:
string input; //inputstring
public:
//------------------------------------------------------------
BracketChecker(string in) : input(in) //constructor
{ }
//------------------------------------------------------------
void check()
{
int stackSize = input.length(); //get max stack size
StackX theStack(stackSize); //make stack
bool isError = false; //error flag
for(int j=0; j<input.length(); j++) //get chars in turn
{
char ch = input[j]; //get char
switch(ch)
{
case '{': //opening symbols
case '[':
case '(':
theStack.push(ch); //push them
break;
case '}': //closing symbols
case ']':
case ')':
if( !theStack.isEmpty() ) //if stack not empty,
{
char chx = theStack.pop(); //pop and check
if( (ch=='}' && chx!='{') ||
(ch==']' && chx!='[') ||
(ch==')' && chx!='(') )
{
isError = true;
cout << "Mismatched delimeter: "
<< ch << " at " << j << endl;
}
}
```

```
else //prematurely empty
{
isError = true;
cout << “Misplaced delimiter: “
<< ch << “ at “ << j << endl;
}
break;
default: //no action on other characters
break;
} //end switch
} //end for
//at this point, all characters have been processed
if( !theStack.isEmpty() )
cout << “Missing right delimiter” << endl;
else if( !isError )
cout << “OK” << endl;
} //end check()
//--------------------------------------------------------------
}; //end class BracketChecker
////////////////////////////////////////////////////////////////
int main()
{
string input;
while(true)
{
cout << “Enter string containing delimiters “
<< “(no whitespace): “;
cin >> input; //read a string from kbd
if( input.length() == 1 ) //quit if ‘q’, etc.
break;
//make a BracketChecker
BracketChecker theChecker(input);
theChecker.check(); //check brackets
} //end while
return 0;
} //end main()
```

Efficiency of Stacks

Items can be both pushed and popped from a stack in constant O(1) time. That is, the time is not dependent on how many items are in the stack, and is therefore very quick. No comparisons or moves are necessary. Of course access is, by design, restricted to a single item.

Summary

In this hour, you've learned the following:

- A stack allows access to the last item inserted, at the top of the stack.
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item from the top.
- A stack is often helpful in parsing a string of characters, among other applications.
- A stack can be implemented with an array or with another mechanism, such as a linked list.