COMSC-110 Chapter 13

Keeping a list: Array-Based Lists

Valerie Colber, MBA, PMP, SCPM

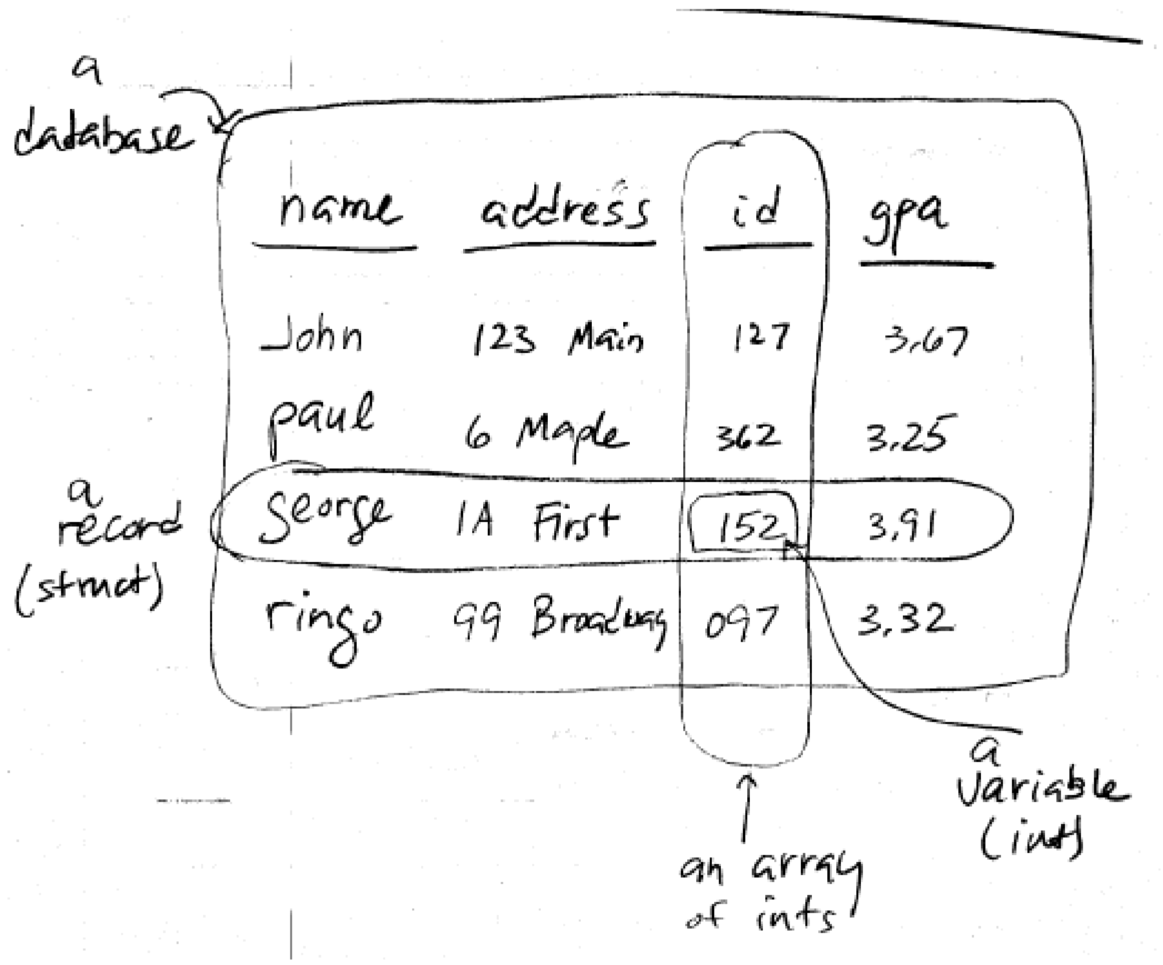
Chapter 13

- This chapter is an opportunity to introduce concepts that you will run into if you go further in your study of computer science, and to provide an application of the programming features we studied in the preceding chapters.
- This chapter uses arrays as the data structure but conceptually in a different way to introduce the concept of a list.

Storing values in a program

- Literal values
- Variables: store one value
- Arrays: store multiple values of the *same data type*
- Structs: database records
- Array-Based Lists: arrays in which not all of the elements are used, but are available for use instead.

Variables, arrays, records, and data bases



name	address	id	gpa
John	123 Main	127	3.67
paul	6 Maple	362	3.25
george	1A First	152	3.91
ringo	99 Broadway	097	3.32

List

- A *list* is a collection of variables -- usually struct variables, but they can be integers or floating points or text, too.
- The language allows the data types for these to be different from one another, even in the same collection!
- The concept is that of a "bag", which can contain lots of different items.
- *But for purposes of our study, we will consider collections of like data types only.*

Lists

- A list can be empty -- that is, it can have zero values in it.
- Lists cans grow in size by adding values, and shrink by taking them away.
- With a list we could find lowest test score and delete it, or insert a score for a late assignment.
- By contrast, the arrays we studied were of *fixed*, nonzero size, and all elements had a value.
- So there are similarities and differences between arrays and lists.

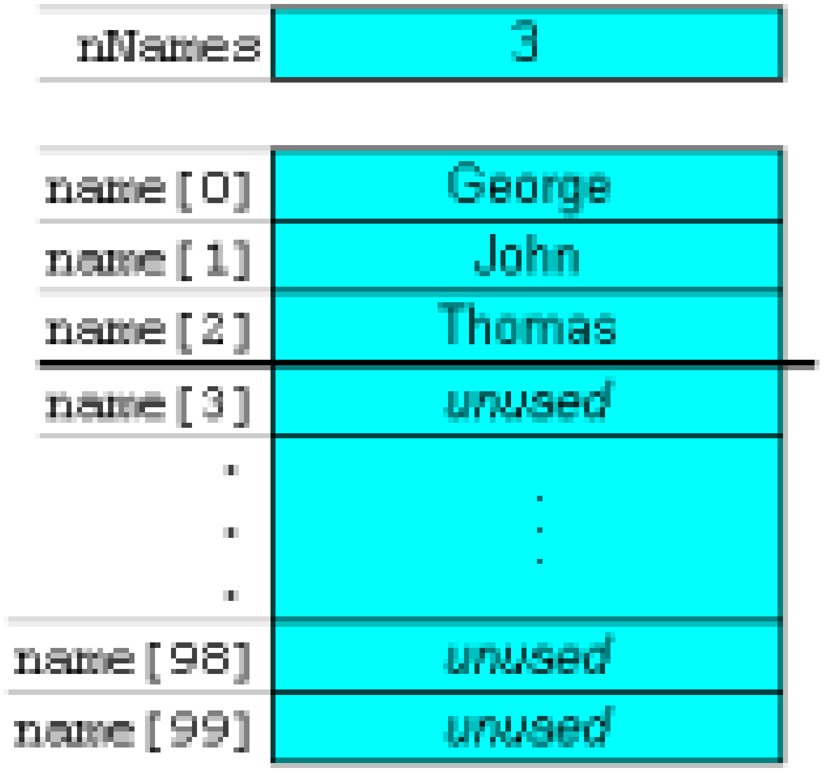
Array-Based Lists

- Arrays are sized when they are created, and we normally consider all of their elements to have valid values.
- An *array-based list* takes the idea of an array, but considers that the elements may not all be in use.
- The elements are place holders for values that may or may not ever be used.

Array-Based Lists

- The array is divided into two parts -- the front part, which has elements with valid values, and the back part, which has elements in reserve for possible future use.
- To mark the dividing point, an *integer* (not a constant like in arrays) is used to count the number of elements that have valid values.

Array-Based Lists

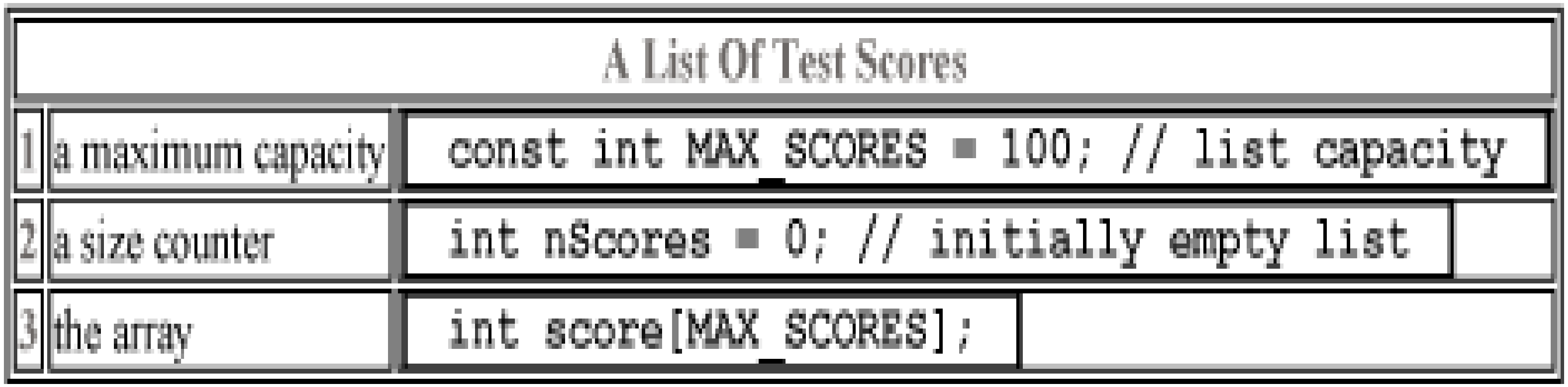


Conceptual representation

Array-Based Lists must have:

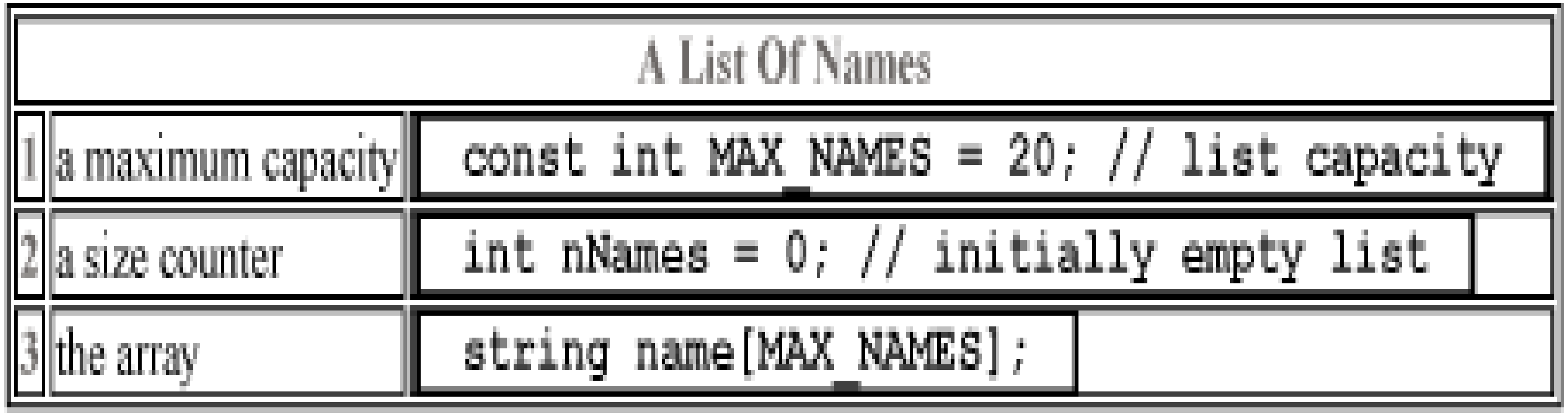
- A maximum capacity: the maximum number of values that the list can possibly contain
- A size counter: to keep track of the number of values in the list
- The array: the data structure where the values are stored during the execution of the program.

Example list of test scores



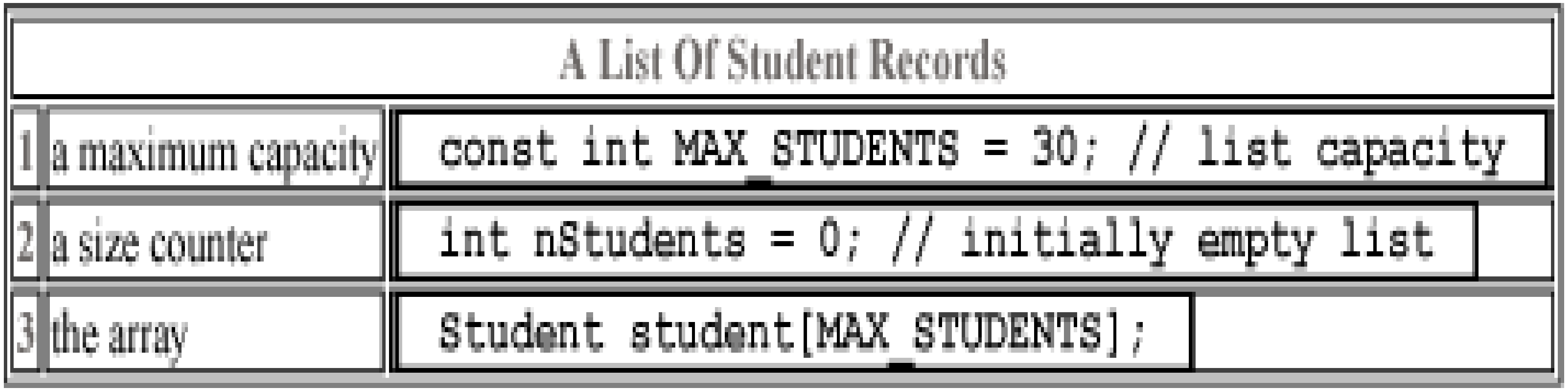
A List Of Test Scores		
1	a maximum capacity	`const int MAX_SCORES = 100; // list capacity`
2	a size counter	`int nScores = 0; // initially empty list`
3	the array	`int score[MAX_SCORES];`

Example list of names



A List Of Names		
1	a maximum capacity	`const int MAX_NAMES = 20; // list capacity`
2	a size counter	`int nNames = 0; // initially empty list`
3	the array	`string name[MAX_NAMES];`

Example list of students



A List Of Student Records		
1	a maximum capacity	`const int MAX_STUDENTS = 30; // list capacity`
2	a size counter	`int nStudents = 0; // initially empty list`
3	the array	`Student student[MAX_STUDENTS];`

Naming convention used in examples

- the array names are not plural, so that **student[0]** is the first Student.
- The name of the counter is the same as that of the array, but plural, with a leading "n", and the first letter uppercased.
- The capacity is named "MAX_" plus the uppercase, plural name of the array.
- There is nothing special or universal about these naming conventions, but it is consistent.

Adding values to a list

- By default, new values are added at the end of the list.

```
int aScore;
... // set aScore's value
if (nScores < MAX_SCORES) score[nScores++] = aScore;
```

Adding values to a list

```
string aName;
... // set aName's value
if (nNames < MAX_NAMES) name[nNames++] = aName;
```

Adding values to a list

```
Student aStudent;
... // set aStudent's field values
if (nStudents < MAX_STUDENTS) student[nStudents++] = aStudent;
```

Adding values to a list

- Inside the square brackets of each assignment statement is the *post-increment operator* on the array index.
- What this does is to add one to the counter *after* the assignment takes place.
- The statement

score[nScores++] = aScore;

is the same as these two statements:

score[nScores] = aScore;

nScores++;

but it is more concise.

Scores.txt <input file>

```
66
98
87
71
56
82
84
90
98
84
67
63
80
81
99
```

Example

listFile.cpp

listFile.cpp

- Note that the first loop reads values from the file, until the end of the file is reached.
- The "if" statement inside the loop protects against overfilling the array that supports the list.
- There is no error message in this example, in case the list capacity is exceeded -- new values are simply ignored.
- But it would not be difficult to add code to either print a message or set a boolean flag!

Searching and sorting a List

- With array-based lists, the code for searching and sorting is almost the same as it was for the arrays.
- The difference is that the *size counter* is used for the end of count-controlled traversal loops instead of the array capacity.

A counting search loop for a List

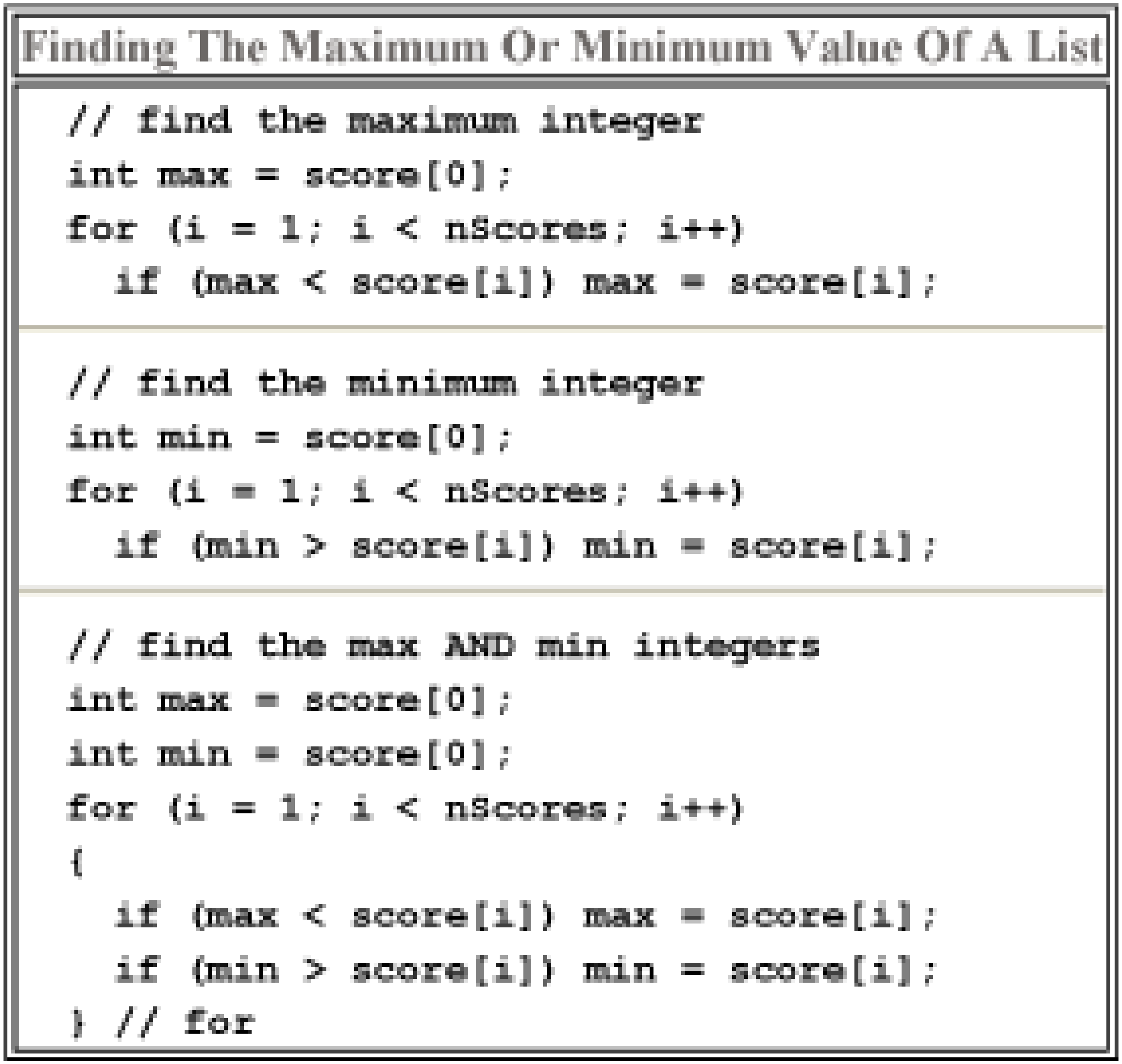
```
int nGreater = 0;
 for (i = 0; i < nScores; i++)
   if (score[i] > average) nGreater++;
```

Two Boolean search loops for a List

```
bool hasPerfectScore = false;
 for (i = 0; i < nScores; i++)
 {
   if (score[i] == 100)
   {
     hasPerfectScore = true;
     break;
   } // if
 } // for
```

...or...

```
bool hasPerfectScore = false;
for (i = 0; i < nScores; i++)
  if (score[i] == 100) hasPerfectScore = true;
```



Finding The Maximum Or Minimum Value Of A List

```
// find the maximum integer
int max = score[0];
for (i = 1; i < nScores; i++)
   if (max < score[i]) max = score[i];
```

```
// find the minimum integer
int min = score[0];
for (i = 1; i < nScores; i++)
   if (min > score[i]) min = score[i];
```

```
// find the max AND min integers
int max = score[0];
int min = score[0];
for (i = 1; i < nScores; i++)
{
   if (max < score[i]) max = score[i];
   if (min > score[i]) min = score[i];
} // for
```

Example

sortFileList.cpp

Functions and Array-Based Lists

- When sharing an array-based list with a function, remember that there are now two values that identify the list -- the array and the size counter.
- So any function that receives the array in its parameter list also needs the size counter.
- Note that the size counter is passed to the function *by value*, meaning that the version in the function code block is a copy.
- So if you change it there, it will not affect the *real* size of the list.

An Averaging Function

```
double getAverage(int* score, int n)
{
  double average = 0;
  int i = 0;
  for (i = 0; i < n; i++)
    average += score[i];
  average = average / n;
  return average;
} // getAverage

int main()
{
  ...
  cout << "Average = " << getAverage(score, nScores) << endl;
  ...
  return 0;
} // main
```

Inserting and deleting values

- While it is certainly possible to write code for inserting values at positions in the list other than at the end, and to remove values, array-based lists do not really lend themselves very well to these operations.
- The problem is that such operations require mass movement of elements to create/close "gaps" in an array.
- There are better ways to represent lists, if you need features such as these.

Inserting and deleting values

- But it is actually not hard to remove the *last* value in an array-based list -- the index for its value is the size counter minus one. For example,

  score[nScores - 1]

- Then you would have to subtract one from the size counter. For example,

  --nScores

```
if (nScores > 0)
{
  int aScore = score[--nScores];
  ...
} // if
---------------------------------------------------
if (nNames > 0)
{
  string aName = name[--nNames];
  ...
} // if
---------------------------------------------------
if (nStudents > 0)
{
  Student aStudent = student[--nStudents];
  ...
} // if
```

In each of these examples, a value is removed from the list (even though it actually remains in the array!), and is retrieved with a new, temporary alias (like "aScore") for further processing in the remainder of the **if** code block (represented by "...").

Other List models

- Besides array-based lists, there are "linked lists" which do not use arrays at all, and there are library-based collections which do most of the coding for you and provide a wide range of features.
- Next we will study linked lists, and collections.

Array-Based List using struct programmer defined data type Example

studentList.cpp

students.txt

studentList.cpp

- Student records are read from a text file, stored in a list, sorted, and printed.
- Note that the input text file has **----------** separating each "student" record -- this is to make the file easier for humans to read.
- You see in "main" that there is code to skip the separator.
- The sort code is based on student names, low to high alphabetically, and case-dependent (so that all uppercase letters come before all lowercase letters).
- To sort on any other field, you would rewrite the "if" statements that precede the swap code.

alternate sort comparisons for the other fields in struct **Student**

```
if (student[i].id > student[j].id) // then swap
{
  Student temp = student[i];
  student[i] = student[j];
  student[j] = temp;
}
-------------------------------------------------------------------
if (student[i].gpa > student[j].gpa) // then swap
{
  Student temp = student[i];
  student[i] = student[j];
  student[j] = temp;
}
```

Alternate sort comparisons for other fields in struct Student

- The print function has 2 values in its parameter list the array *and* the size counter.
- The fields are aligned in columns.
- The name is left-justified in 30 columns by this:

  << left << setw(30) << student[i].name
- The ID is right-justified in 7 columns by this:

  << right << setw(7) << student[i].id
- So that the spaces after the ID are filled with leading zeros, we have **cout.fill('0');**, and then **cout.fill(' ');** resets it back to spaces (which the name needs).
- With all of the details moved to functions, "main" is pretty easy to follow.

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns