CS 11 Data Structures and Algorithms

Lesson 9: Arrays

Section 9.1: Introduction to Arrays

Sometimes when we are writing a program we find that we want to store many different values at once. So far, we have not had to do this, even when we were dealing with a list of numbers unlimited in length. For example, we found the largest and smallest items in a list of numbers and the sum and average of a list of numbers without ever storing more than 1 or 2 of the numbers. Suppose now that we want to write a program that reads in a list of numbers that the user types in and then writes them back on the screen in reverse order. In order to do this, we need to store all of the numbers in the list. Is this possible given what we have learned so far in C++? If we know ahead of time how many numbers there will be, we can write this program by declaring that many variables. For example, if we know that there will always be six numbers typed in, our program might look like this:

```
int main()
{
    int num0;
    int num1;
    int num2;
    int num3;
    int num4;
    int num5;

    cout << "Enter a number: ";
    cin >> num0;
    cout << "Enter a number: ";
    cin >> num1;
    cout << "Enter a number: ";
    cin >> num2;
    cout << "Enter a number: ";
    cin >> num3;
    cout << "Enter a number: ";
    cin >> num4;
    cout << "Enter a number: ";
    cin >> num5;
    cout << "The numbers in reverse order are: ";
         << num5 << " " << num4 << " " << num3 << " "
         << num2 << " " << num1 << " " << num0 << endl;
}
```

```
Enter a number: 8
Enter a number: 13
Enter a number: 5
Enter a number: 12
Enter a number: 21
Enter a number: 19
The numbers in reverse order are 19 21 12 5 13 8
```

This program is bad in at least 3 ways, which get worse as we go:

1. The program seems kind of awkward because there are so many things in the program that we have to repeat over and over again -- like, for example, the line

   ```
   cout << "Enter a number: ";
   ```

   It would be better if we could use a loop to keep from having to repeat lines like this one six times in our program.

2. The program would begin to get ridiculous if we had to read many numbers. Imagine, for example, what the program would look like if we had to read and reverse 1000 numbers!

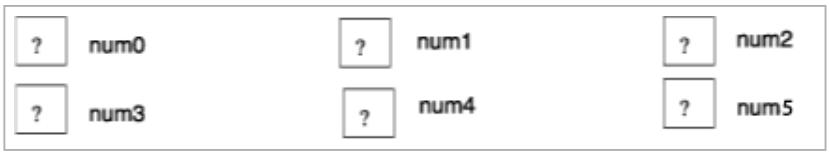
3. Worst of all, what if we didn't know in advance how many numbers there would be? It would be (nearly) impossible to write this program.

In order to have a well written program that would work with any number of numbers, we would need to use two loops: a loop to read in the numbers until a negative number is entered, and then a second loop to write the numbers out again in reverse order. Here is the pseudocode for this algorithm:

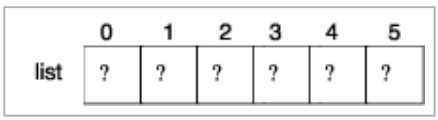
```
the loop to read the numbers in:

    count = 0;
    Read a number;
    While the number >= 0 {

       Store the number in the appropriate variable.  For example, if count is 0,
       store the number in number0.  If count is 1, store the number in number1.
       If count is 2, store the number in number2.  And so on.

       Add 1 to count;
       Read the next number;

    }


the loop to write the numbers out again in reverse order:

    numItems = count;
    for (count = numItems - 1; count >= 0; count--) {

        print out the appropriate variable.  For example, if
        count is 0, print out number0.  If count is 1, print
        out number1.  If count is 2, print out number2.  And
        so on.
    }
```

Section 9.2: Reverse Input Example

In order to turn the pseudocode from section 1 into C++ we have to introduce the concept of an **array**. At the most basic level, an array is simply a way of sticking a bunch of individual variables together into one big combined variable. In the program we wrote at the beginning of this section we declared 6 individual variables. We picture this situation like this:



If we were to use an array, it would mean sticking those 6 individual variables together. Then we would picture the situation like this:



We tell C++ to create a combined variable like this much the same way we tell it to create individual ones: using a declaration statement. Here's the declaration statement we would use to create an array that represents this list of six integers:

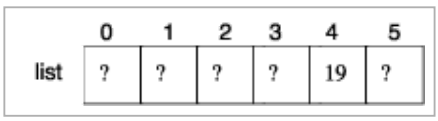
```
int list[6];
```

This declaration statement tells C++ to create an array variable called list. The int in front tells C++ that each item in the array will be of type int. The 6 in the square brackets tells C++ to create an array with 6 items. C++ automatically numbers the items from 0 to 5. What goes inside the square brackets **must be either a literal constant or a named constant**, and it must be an integer. You can use an integer value as I have done in the example, or you can use the name of a constant that you have declared previously (the latter will more typically be the case). You cannot use a variable or other arbitrary expression.

Each individual item in an array is called an **element** of the array. When we want to use a particular individual variable that is part of this array, we put the number of the individual variable (called the **index**) in brackets after the name of the array. For example, the statement

```
list[4] = 19;
```

tells C++ to put the value 19 into the element whose index is 4 of the array variable `list`. After executing this statement, our situation would look like this:



In this context (unlike the case of the declaration statement) we can put any integer expression inside the brackets. So, for example, if we say

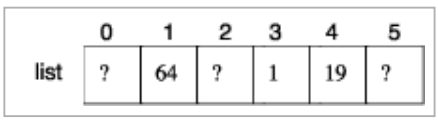
```
numbers[2 * count] = 19;
```

we mean "put the value 19 into the element of the array that has an index equal to `2 * count`".

This ability to use any expression inside the square brackets can lead to some interesting code. For example, consider what happens if we execute the following code:

```
int array[6];
array[4] = 19;
array[array[4]/5] = 1;
array[array[3]] = 64;
```

Here is a picture of the resulting array. You should try to figure out what the code will produce first, and then look at the picture. Make sure that you understand how the code produces the array in the picture!!



Let's now use arrays to translate our pseudocode that we developed at the end of the last section into C++ code.

```
#include <iostream>
using namespace std;

const int ARRAY_SIZE = 1000;

int main()
{
    int count;
    int number;
    int numItems;
    int list[ARRAY_SIZE];

    count = 0;

    cout << "Enter a number (negative number to quit): ";
    cin >> number;
    while (number >= 0){
        list[count] = number;
        count++;
        cout << "Enter a number (negative number to quit): ";
        cin >> number;
    }

    numItems = count;
    cout << "The numbers in reverse order: ";
    for (count = numItems - 1; count >= 0; count--){
        cout << list[count] << " ";
    }
}
```

```
Enter a number (negative number to quit): 7
Enter a number (negative number to quit): 14
Enter a number (negative number to quit): 9
Enter a number (negative number to quit): 30
Enter a number (negative number to quit): 8
Enter a number (negative number to quit): 24
```

```
Enter a number (negative number to quit): 17
Enter a number (negative number to quit): -1
The numbers in reverse order: 17 24 8 30 9 14 7
```

Notice in this example that even though we declared our array to be quite large, we only used a very small portion of it. We kept track of how many elements of our array we were using by having a variable called `numItems`. We can ignore all of the other elements in our array. For example, if `numItems` is equal to 8, that means that we can ignore elements 8 through 999 in our array. This is a very common technique.

Section 9.3: Arrays as Arguments and Parameters

Let's write a program that reads in a list of numbers and then prints them back out again, only it leaves out the smallest number. Sample screen output:

```
Enter a number (negative number to quit): 7
Enter a number (negative number to quit): 14
Enter a number (negative number to quit): 9
Enter a number (negative number to quit): 30
Enter a number (negative number to quit): 8
Enter a number (negative number to quit): 2
Enter a number (negative number to quit): 17
Enter a number (negative number to quit): -1
The numbers you entered (excluding the smallest):
7 14 9 30 8 17
```

As usual, we'll use stepwise refinement to write this program, and so we start by designing our main function. We'll want something like this:

```
int main()
{
    int list[ARRAY_SIZE];
    int smallest;
    int numItems;

    readNumbers(??);
    smallest = findSmallest(??);
    cout << "The numbers you entered (omitting the smallest): "
         << endl;
    printListExcept(??);
}
```

Before we can finish this program up, there are three important pieces of information you need to know about how to use arrays as arguments and parameters. They are (1) what to do when you **call** the function, (2) what to do when you **define** the function, and (3) the **special rule** for pass-by-reference with arrays.

1. When you **call** the function, you simply put the name of the array in the parentheses, just like you would with, say, an int variable. So, when we fill in the ??'s from the main function we just designed, we get something like this:

   ```
   int main()
   {
       int list[ARRAY_SIZE];
       int smallest;
       int numItems;

       readNumbers(list, numItems);
       smallest = findSmallest(list, numItems);
       cout << "The numbers you entered (omitting the smallest): "
            << endl;
       printListExcept(list, numItems, smallest);
   }
   ```

   Notice that you do not put square brackets next to the name of the array!

   Actually, there is a more concise way to write this main function that you should get used to using. Rather than having that extra local variable named `smallest`, let's just put the call to `findSmallest`

right where we want the value to go, in the place of `smallest` in the call to `printNumbers`, as illustrated here:

```
int main()
{
    int list[ARRAY_SIZE];
    int numItems;

    readNumbers(list, numItems);
    cout << "The numbers you entered (omitting the smallest): "
         << endl;
    printListExcept(list, numItems, findSmallest(list, numItems));
}
```

This is the preferred style. You will be penalized on your programs if you don't utilize this technique.

2. When you **define** the function, you have to declare the variable all over again, including the type and the brackets. Rather than putting the size of the array inside the brackets, you can just leave the brackets empty. Having the brackets next to the word `list` in the following example tells C++ that the parameter is an array, not just an integer. As an example, here is a first draft of our function definition for the findSmallest function.

```
int findSmallest(int list[], int numItems)
{
    <body of function>
}
```

3. There is a **special rule** for pass-by-reference with arrays. When you are passing an array, C++ always assumes that you are using pass-by-reference. Therefore, you should never ever use & when passing an array as an argument. If you do you will get an error message. For example, when we define the function readNumbers, we intend for the changes to `list` that take place inside that function to also affect the variable list in the main function. However, because of the "never use & with arrays" rule, we don't use &. Our function will look like this:

```
void readNumbers(int list[], int &numItems)
{
    <body of function>
}
```

This brings up another detail that we need to discuss here. We do not want the findSmallest function to have access to the variable `list` declared in main. Usually we would accomplish this by making `list` a pass-by-value parameter, but that won't work in this case because arrays are always pass-by-reference automatically. So we use the `const` keyword, just like we did in lesson 9.9. In our `findSmallest` function that will look like this:

```
int findSmallest(const int list[], int numItems)
{
    <body of function>
}
```

The program is given in the next section of this lesson.

Section 9.4: Skip Smallest Example

Before getting to the code for the skip-smallest program, let's talk about our code that fills the array with numbers. (This code will appear in the function we named `readNumbers`.) We did the same thing in our reverse-input program from section 2 of this lesson, but now we are going to refine the code a bit. Here is the code as it appeared in section 2:

```
int count;
int number;
int list[ARRAY_SIZE];

count = 0;

cout << "Enter a number (negative number to quit): ";
cin >> number;
while (number != -1){
    list[count] = number;
    count++;
    cout << "Enter a number (negative number to quit): ";
```

```
    cin >> number;
}

numItems = count;
```

The problem with this code is that it does not take into consideration the case where the user types in more numbers than there is room for in the array. This is a very serious problem, because C++ does not check to see if the array index you use is actually a valid index into the array. For example, if ARRAY_SIZE is 10 (so the indices of the array are 0 through 9), and count is equal to 10, then the statement `list[count] = number;` will cause C++ to put `number` into an array element that does not exist. This is called an array index out of bounds error. The outcome of this action is unpredictable but always bad. You might get lucky and list[10] might just be available and your program will work; however, it might not work the next time you run it. More likely, in executing the statement C++ will write over some other data that has been stored in that memory location, causing your program to work incorrectly or, usually, to simply crash. Or the expression list[10] might refer to a memory location that is invalid. Often array index out of bounds errors are extremely difficult to debug because you run your program and you simply get an error message that says (essentially) "something bad happened while your program was running". (To be more specific, in CodeWarrior your program will end execution immediately and you will get a dialog box that says "unhandled exception".) Not very helpful.

A first attempt to solve this problem might be to add a condition to the while-loop header that prevents execution from entering the loop when the array is full, like this:

```
int count;
int number;
int list[ARRAY_SIZE];

count = 0;

cout << "Enter a number (negative number to quit): ";
cin >> number;
while (number != -1 && count < ARRAY_SIZE){
    list[count] = number;
    count++;
    cout << "Enter a number (negative number to quit): ";
    cin >> number;
}

numItems = count;
```

This is a vast improvement over our original code, since this will keep the program from crashing. To see the remaining problem you'll have to trace through the code very carefully. Let's again say ARRAY_SIZE is 10. When count is 9, the loop will be entered (since 9 is less than ARRAY_SIZE). `number` will be assigned to list[count], and count will be increased to 10. Things should stop there, since the array is full, and if the user enters more numbers there will be no room for them anyway. But instead of stopping, our code gets another number from the user, and then stops. The user is likely to think that the number that was entered has been inserted into the array, and will not know why the program stopped asking for more numbers.

To remedy this, in the final version of the program below we will place an if statement after count is incremented so that the program does not get that last number from the user. In addition, we will print a message explaining why the program stopped asking for numbers.

```
#include <iostream>
using namespace std;

const int ARRAY_SIZE = 5;

int smallest(const int list[], int numItems);
void printListExcept(const int list[],
                           int numItems,
                           int numberToSkip);
void readNumbers(int list[], int &numItems);


int main()
{
    int list[ARRAY_SIZE];
    int numItems;

    readNumbers(list, numItems);
    cout << "The numbers you entered (omitting the smallest):"
```

```
             << endl;
        printListExcept(list, numItems, smallest(list, numItems));
    }

    void readNumbers(int list[], int &numItems)
    {
        int number, count;

        cout << "Enter a number (-1 to quit): ";
        cin >> number;

        count = 0;
        while ((number != -1) && (count < ARRAY_SIZE)){
            list[count] = number;
            count++;
            if (count < ARRAY_SIZE){
                cout << "Enter a number (-1 to quit): ";
                cin >> number;
            } else {
                cout << "the array is now full." << endl;
            }
        }
        numItems = count;
    }


    // Print out each item in the array, being sure not to
    // print the numberToSkip.  We do this by looking at
    // each item in the array.  If it is not equal to the
    // numberToSkip, we print it.

    void printListExcept(const int list[],
                               int numItems,
                               int numberToSkip)
    {
        for (int count = 0; count < numItems; count++){
            if (list[count] != numberToSkip){
                cout << list[count] << " ";
            }
        }
    }


    // Go through the array looking for the smallest number.
    // We start by saying that the smallest number is the first
    // number in the array (list[0]).   Then we look at each
    // remaining item.  If we see one that is less than our
    // smallest number, we say that that number is now our
    // smallest number.


    int smallest(const int list[], int numItems)
    {
        int small = list[0];

        for (int count = 1; count < numItems; count++){
            if (list[count] < small){
                small = list[count];
            }
        }
        return small;
    }
```

Section 9.5: Reverse List Example

Before getting into our next example, a quick word about writing and testing functions. You may be used to doing programming problems which require you to write an entire program. The questions usually start with "write a program that...." However, you will often be asked simply to write a function that does a particular task. For example, "write a function that ...." The problem with this is that we can't just type a function into the computer and see whether it works. When you are presented with this type of problem, you not only have to write the function, but you need to write a program to put the function into to see whether it works. A program written for this purpose is called a driver. This is the technique we will be illustrating in this section.

Problem: Write a function that has two parameters: an array of integers `list` that represents a list of integers, and an integer `numItems` that tells us how many items are in the list. The function should reverse the positions of the items in the list within the array.

Solution: Before we actually start writing C++ statements to solve this problem, we need to think about what our strategy will be. In general, we should always be able to express our solution to a problem in English as a sequence of steps before we start writing the code.

Our strategy for this program will be the following. We will have two counters, one which starts at the beginning of the list and counts forwards, and one which starts at the end of the list and counts backwards. Our pseudocode might look like this:

```
repeat
    swap list[forwardCount] and list[backwardCount]
    increment forwardCount
    decrement backwardCount
until the entire list is reversed.
```

How will we know when the entire list is reversed? There are several ways to accomplish this. One way would be to use the fact that forwardCount is getting bigger and backwardCount is getting smaller. When backwardCount becomes less-than-or-equal-to forwardCount, the list is reversed. Another way would be to compute before the loop starts how many iterations of the loop are required. If there are 6 items in our list, we need to swap three pairs of numbers. If there are 7 items, we also need to swap three pairs of numbers. If there are 8 items, we need to swap four pairs of numbers. So, we need to repeat our statements numItems/2 times. Here is our function. We will call a function called "swap" which we will write later. (There is also a built in function named "swap" which we could use; but we will write the "swap" function here for the sake of illustration.)

```
void reverseList(int list[], int listSize)
{
    int forwardCount;
    int backwardCount;

    backwardCount = listSize-1;
    for (forwardCount=0; forwardCount < listSize/2; forwardCount++){
        swap(list[forwardCount],list[backwardCount]);
        backwardCount--;
    }
}
```

We are done with our function now, but of course we can't just type this function into the computer and expect it to do anything. We need to write a driver. Here is an example of a driver that could be used to test our reverseList function:

```
#include <iostream>
using namespace std;

void readNumbers(int list[],int &numItems);
void reverseList(int list[], int numItems);
void printNumbers(const int list[],  int numItems);

const int ARRAY_SIZE = 100;

int main()
{
    int numItems;
    int list[ARRAY_SIZE];

    readNumbers(list,numItems);
    reverseList(list,numItems);
    printNumbers(list,numItems);
}


void readNumbers(int list[], int &numItems)
{
    int number, count;

    cout << "Enter a number (-1 to quit): ";
    cin >> number;

    count = 0;
    while ((number != -1) && (count < ARRAY_SIZE)){
        list[count] = number;
        count++;
        if (count < ARRAY_SIZE){
            cout << "Enter a number (-1 to quit): ";
            cin >> number;
        } else {
            cout << "the array is now full." << endl;
        }
    }
    numItems = count;
}


void reverseList(int list[], int numItems)
```

```
{
    int forwardCount;
    int backwardCount;

    backwardCount = numItems-1;
    for (forwardCount=0; forwardCount < numItems/2; forwardCount++){
        swap(list[forwardCount],list[backwardCount]);
        backwardCount--;
    }
}



void printNumbers(const int list[],  int numItems)
{
    int count;

    for (count = 0; count < numItems; count++){
        cout << list[count];
    }
}
```

Notice that if you have an exercise where you are asked to write a function and then test it in a driver, you have quite a lot of freedom to test it in whatever way you want. The program we have written above allows you to enter different numbers each time you run the program. This is good because you can run it several times, entering different numbers each time, to make sure that your program works no matter what numbers you type in. For example, to test our function and make sure that it works correctly, you should be sure to run the program once with an even number of items and once with an odd number of items. You should also try some boundary cases. What happens if we enter just one number? Does it work? What if we enter zero numbers? Does it work? When writing a driver, make sure that it is flexible enough to allow you to test your function thoroughly.

Section 9.6: Selection Sort Example

The problem of sorting an array so that the items in the array are in ascending or descending order is the subject of an entire area of Computer Science. As an introduction, we will write a function that uses a sorting algorithm named "selection sort". The idea behind the selection sort is that you find the smallest item in the list and swap it with the first item in the list. Then you repeatedly find the smallest item in the list (not counting the items that have already been placed in their correct position) and swap that item with the next item in the list that has not been placed yet. You repeat this process until the list is sorted. For example, if the original list is

```
5, 7, 3, 2, 9, 0, 14, 8, 10, -2
```

you would begin by determining that the smallest item in the list is -2 and swapping that item with the first item, 5:

```
-2, 7, 3, 2, 9, 0, 14, 8, 10, 5
```

Next you ignore the -2 and determine that the smallest item in the list is 0 and swap that item with the first still-unsorted item, which is 7:

```
-2, 0, 3, 2, 9, 7, 14, 8, 10, 5
```

Next you ignore the -2 and the 0 and determine that the smallest item in the list is 2 and swap that item with the first still-unsorted item, which is 3:

```
-2, 0, 2, 3, 9, 7, 14, 8, 10, 5
```

and so on until the list is sorted.

In the code below, the most difficult function is the Sort function. In that function you will find a for loop going from 0 to numItems - 1, since this is how many times we will have to repeat the process described above in order to ensure that the list is sorted. Each time through this for loop we want to swap the smallest item in the list (indicated by the function call to "indexOfSmallest") with the next item in the list (indicated by the variable "count").

```
#include <iostream>
using namespace std;

const int ARRAY_SIZE = 1000;
```

```
void printList(const int list[],
                    int numItems);
int indexOfSmallest(const int list[], int startingIndex, int numItems);
void sort(int list[], int numItems);
void readNumbers(int list[], int &numItems);

int main()
{
    int list[ARRAY_SIZE];
    int numItems;


    readNumbers(list, numItems);
    sort(list, numItems);
    cout << "The sorted list: "
         << endl;
    printList(list, numItems);
}



void readNumbers(int list[], int &numItems)
{
    int number;
    int count = 0; // the number of numbers that have been entered

    cout << "Enter a number (negative number to quit): ";
    cin >> number;
    while (number >= 0 && count < ARRAY_SIZE){
        list[count] = number;
        count++;
        if (count < ARRAY_SIZE){
            cout << "Enter a number (negative number to quit): ";
            cin >> number;
        } else {
            cout << "The array is now full." << endl;
        }
    }
    numItems = count;
}


void sort(int list[], int numItems)
{
    for (int count = 0; count < numItems - 1; count++){
        swap(list[indexOfSmallest(list, count, numItems)],
             list[count]);
    }

}


int indexOfSmallest(const int list[], int startingIndex, int numItems)
{
    int targetIndex = startingIndex;

    for (int count = startingIndex + 1; count < numItems; count++){
        if (list[count] < list[targetIndex]){
            targetIndex = count;
        }
    }

    return targetIndex;
}

void printList(const int list[],
                    int numItems)

{
    for (int count = 0; count < numItems; count++){
        cout << list[count] << " ";
    }
}
```