Arrays

Motivation

Suppose that we want a program that can read in a list of numbers and sort that list, or find the largest value in that list. To be concrete about it, suppose we have 15 numbers to read in from a file and sort into ascending order. We could declare 15 variables to store the numbers, but then how could we use a loop to compare the variables to each other? The answer is that we cannot.

The problem is that we would like to have variables with subscripts or something like them, so that we could write something like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
    if ( number_i > max )
        max = number_i ;
}
```

where somehow the variable referred to by `number_i` would change as `i` changed.

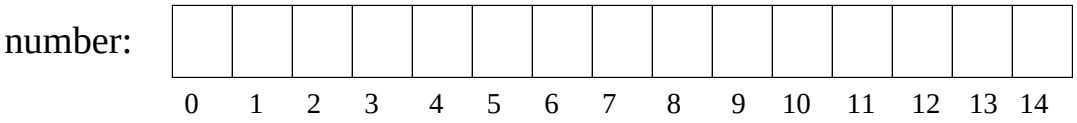
You have seen something like this already with C++ strings: if we declare

```
string str;
```

then we can write a loop like

```
for ( int i = 0; i < str.size(); i++ )
    cout << str[i] << "-" ;
```

in which each individual character in `str` is accessed using the subscript operator `[]`. The characters in a string form what we call an ***array***. An array is conceptually a linear collection of elements, indexed by subscripts, all of the same type. If we could create an array named `number` with 15 elements, it would look like this:



Each element could be accessed using the subscript operator, as in `number[1]` or `number[7]`, and we could write a loop like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
    if ( number_i > max )
        max = number[i] ;
}
```

This would make it possible to manipulate large collections of ***homogeneous*** data, meaning data of the same type, with a single subscripted variable. Such is possible in C and C++ and all modern programming languages. ***Arrays are of fundamental importance to algorithms and computer science.***

Declaring a (one-Dimensional) Array

Syntax:

```
elementtype arrayname [ size_expression ]
```

where

- *elementtype* is any type that already exists
- *arrayname* is the name of the array
- *size_expression* is the number of elements in the array

This declares an array named `arrayname` whose elements will be of type *elementtype*, and that has *size_expression* many elements.

Examples

```
char    fname[24];            // an array named fname with 24 chars
int     grade[35];            // an array named grade with 35 ints
int     pixel[1024*768];      // an array named pixel with 1024*768 ints

const int MAX_STUDENTS = 100;
double average[MAX_STUDENTS]; // an array named average with 100 doubles

string fruit[5];              // an array of 5 C++ strings
```

The element type of an array is often called its ***base type***. The first example is an array with base type `char`, for example. One can say that `fname` is "an array of `char`."

Things to remember about arrays:

- The starting index of an array is 0, not 1.
- The last index is one less than the size of the array.
- If the array has `size` elements, the range is 0..`size`-1.
- Arrays contain data of a single type.
- An array is a sequence of consecutive elements in memory and the start of the array is the address of its first element.
- Because the starting address of the array in memory is the address of its first element, and all elements are the same size and type, the compiler can calculate the locations of the remaining elements. If B is the starting address of the array `array`, and each element is 4 bytes long, the elements are at addresses B, $B+4$, $B+8$, $B+12$, and so on, and in general, element `array[k]` is at address $B+12k$.
- Although C and C++ allow the size expression to be variable, you should not use a variable, for reasons having to do with concepts of dynamic allocation and the lifetime of variables.
- Use constants such as macro definitions or `const int`s as the sizes of arrays.

Initializing Declarations

Arrays can be initialized at declaration time in two different ways.

```
elementtype arrayname[size expr] = { list with <= sizeexpr vals };
elementtype arrayname[ ]         = { list with any number of values };
```

Examples

```
#define MAXSCORES 200
#define NUMFRUIT    5
const int SIZE = 100;

double  numbers[SIZE];    // not initialized
string  fruit[NUMFRUIT]   = {"apple","pear","peach","lemon","mango"};
int     power[ ]          = {0,1,2,4,9,16,25,36,49,64,81,100};
int     counts[SIZE]      = {0};
int     score[MAXSCORES]  = {1,1,1};
```

The first declaration declares but does not initialize the array named `numbers`. The next declares and initializes an array named `fruit` with five strings:

apple	pear	peach	lemon	mango
0	1	2	3	4

The third initializes an array named `power`, whose size is determined by the number of values in the brace-delimited list. When the array size is given in square brackets but the number of values in the list is less than the size, the remainder of the array is initialized to 0. In the fourth example, all elements will be set to 0, and in the last, the first three are set to 1 and the rest, to 0.

Rules

- If the array size is given in brackets, then the initialized list must have at most that many values in it. If it has fewer, the rest of the array is initialized to 0's.
- If the size is not given in brackets, then the size is equal to the number of elements in the initializer list.
- If there is no initializer list, none of the array elements are initialized. They are not set to 0.

Advice

- Always named constants for the sizes of arrays. It makes it easier to write, maintain, and read the code.

Accessing Array Elements

An element of an array is accessed by using the ***subscript operator***. The syntax is:

```
arrayname [ integer-valued-expression-within-range ]
```

Examples

```
cout << fruit[0] ;                      // prints apple
cout << powers[1] << " " << powers[2] ; // prints 1 2
fruit[3] = "apricot";                   // replaces "peach" by "apricot in fruit[3]
counts[SIZE-10] = counts[SIZE-11] + 2;  // sets counts[90] to counts[89] + 2 = 2
cout << score[power[4]];                // power[4] = 9, so this outputs scores[9]
```

Notice in the last two examples that the index expression can be arbitrary integer-valued expressions, even the value of an array of integers.

Loops and Arrays

Without loops it is very hard to use arrays. Conversely, with them they are easy to use. In general, a loop can be used to initialize an array, to modify all elements, to access all elements, or to search for elements within the array. An example follows.

Example 1. Finding a minimum element.

This example shows how an array can be initialized with values from an input stream using a for-loop. The for-loop guarantees that the array is not "overfilled" because it runs only as many times as the array has elements. The example uses the preceding declarations.

```
// read values into array from standard input
for ( i = 0; i < MAXSCORES; i++ )
    cin >> score[i];

// Let minscore be the first element as a starting guess
int minscore = score[0];

// Compare remaining array alements to current maxscore
for ( i = 1; i < MAXSCORES; i++ ) {
    if ( minscore > score[i] )  // if current element < minscore
        minscore = score[i];    // make it new minscore
    // At this point, maxscore >= score[j], for all j = 0,1,2,...i
}

cout << The minimum score is  << minscore << endl;
```

In this example, the score array is filled from values entered on the standard input stream, `cin`. After the entire array has been filled, a variable named `minscore` is set to the value of `score[0]`. Then a second loop examines each element of score from `score[1]`, to `score[2]`, and so on up to `score[MAXSCORE-1]`. If an element is smaller than `minscore`, its value is copied into `minscore`. This implies that in each iteration of the second loop, `minscore` is the smallest of the elements seen so far. This is how one searches for the minimum value in an unsorted array.

Danger

If your program tries to access an element outside of the range of the array, bad things can happen. Typically the program will be aborted by the opeating system with an error message. As an example:

```
int score[10];
for ( int i = 0; i < 20; i++ )
    cin >> score[i];
```

When `i` eventually has the value 10, `score[i]` is a memory address outside of the `score` array (since it is 0-based and the last index is 9) and the program will be aborted.

Partially Filled Arrays

Although an array might be declared to have so many elements, that does not always mean that all of the cells in the array actually have meangingful data. Sometimes a program does not know how much data it will read in from a file or external source, so it declares an array much larger than it needs. Then after it reads the data, only part of the array has been filled. If a loop that is processing the array tries to process all array elements, errors will result.

This implies that when an array is filled and the number of data items is not known in advance, then the code needs to count how many values are written into the array, and stop if it exceeds the array size, or if it runs out of data first. The while loop has two conditions to check in this case:

```
int list[100];
int temp;
length = 0;

cin >> temp;  // try to read to force eof() to become true if the file is empty
while ( length < 100 && !cin.eof() ) {
    list[length] = temp;  // temp has a value, so copy it into the list
    length++;             // increment because we copied a value into list
    cin >> temp;          // read a new value; might make eof() true
}
```

The variable `length` keeps a count of what has been put into the array and when the loop ends, it is guaranteed to be at most 100. Either the loop ends because `length` became 100 or because the data stream ended. In either case, the array has been filled with `length` many values. Therefore, all future processing of this array must iterate only from 0 to `length` -1. The following code would find the minimum element of `list`:

```
for ( i = 1; i < length; i++ ) {
    if ( min > list[i] )
        min = list[i];
 }
```

Arrays in Functions

Suppose that you want to write a function that can find the minimum element of any array it is given. Can we write such a function? Let us rephrase a simpler question for now. Can we write a function that can find the minimum integer in an array of integers? We can safely answer yes to this question, whether it is C or C++. The heart of the matter is how we can pass an entire array into a function.

How can you pass an entire array into a function?

An array parameter in a function is written as a name followed by empty square brackets:

```
result_type function_name ( ..., elementtype arrayname [ ] ,... ) ;
```

For example,

```
int max( int array[] );
void foo( int a[], int b[], double c[]);
```

The size of the array is not put between the square brackets. When you declare a function with an array parameter, you just put the type and a name for the parameter followed by a pair of empty square brackets. This tells the compiler that the corresponding argument will be an array of that type.

How does the function know how many elements are in the array?

It doesn't. Arrays do not "know" their size. It is not stored anywhere. The function acts on any array of any size whose type matches the base type.

So how can you write a function that works with an array parameter?

You must always pass the size of the array as an extra parameter. For example, if we want to write a function that finds the minimum in an integer array, the function prototype would be

```
int min( int array[], int size );
```

We would call the function by passing just the array name to it. not the name with square brackets after it. For example

```
int score[MAXSCORES];
for ( i = 0; i < MAXSCORES; i++ )
    cin >> score[i];
cout << "The minimum score is " << min(score, MAXSCORES);
```

Notice that the `score` array is passed to the `min()` function just by writing its name. The `min()` function definition would be

```
int min( int array[], int size )
{
    int minvalue = array[0];

    // Compare remaining array alements to current minvalue
    for ( int i = 1; i < size; i++ ) {
        if ( minvalue > array[i] )
            minvalue = array[i];
        //  minvalue <= array[j], for all j <= i
    }
    return minvalue;
}
```

When the function is called as in `cout << max(score, MAXSCORES)`, the `score` array is passed to the array parameter `array`, and within the function `array[i]` is in fact a ***reference*** to `score[i]`, ***not a copy of it***, but a reference to it, i.e., another name for it. This means that changes made to `array[i]` within the function are actually made to `score[i]` outside of the function. The following example will make this clearer.

```
void initSquares( int array[], int size )
{
    for ( int i = 0; i < size; i++ ) {
        array[i] = i*i;
    }
}
```

The function `initSquares()` puts into each element `array[i]` the value `i*i`. The remarkable thing about array parameters is that each element of the array parameter acts like a reference to the corresponding element of its argument, so the output of this code snippet

```
int squares[20];

initSquares( squares, 20);
for ( int i = 1; i < 20; i++ )
    cout << squares[i] << endl;
```

will be the squares 0, 1, 4, 9, ... 400, because the changes made to the array parameter `array[]` are actually changes made to the calling program's `squares` array. To emphasize this

There are three types of parameter passing:

- call by value parameters
- call by reference parameters
- array parameters

Array parameters are like call by reference parameters – changes made to the array are changes made to the corresponding array argument.

Searching and Sorting

Two of the most common types of processing of array data are searching for specific values and sorting the elements.

Searching

The problem of searching for a particular value in an array or in a list of anything is called the ***search problem***. The value being sought is called the ***search key***, or just the ***key*** when the meaning is clear.

Suppose, for example, that want to know whether anyone had a score of 75, assuming that we already read a list of scores into our array `scores`. We can search through our array of scores for the number 75 as follows:

```
cout << "Enter the number to look for: ";
cin >> searchkey;

bool found = false;  // flag to control loop
int  i     = 0;      // index to go through array

while ( !found &&  (i < length) ) {
    if ( searchkey ==  score[i] )  // found it!
        found = true;
    else
        i++;   // didn't find it yet, so advance to next element
}

// If found == false then we did not find the item
if ( found )
    cout << searchkey <<  " is in index " << i << endl;
```

There are some very important concepts here.

- We do not want to use a for-loop because if we find what we are looking for, we want to stop searching. So we use a while-loop that is re-entered if we have not found what we are looking for *and* our index variable is still within the range of the array. Therefore we set `found` to `false` initially and set it to `true` if we find a matching value in the array.
- If we do find an element in the array whose value equals `searchkey`, then we set `found` to `true`. This causes the loop to be exited. Because `i` is declared outside of the loop body, it still has a valid value after the loop exits. It was not incremented since `found` became `true`, so it is the index at which `score[i] == searchkey`.
- If we search through the entire array and do not find `searchkey`, then `found` is `false` when the loop exits, because the condition `(i < length)` became false. By checking the value of `found` after the loop, we can tell whether it was found or not.

This is the first example of a non-trivial algorithm. We can state its requirement more generally as,

> "Given an array A of N values, and a searchkey X, find the position in A in which X occurs or report that it is not there."

The preceding solution searched through the array one element at a time, starting with the first and ending with the last, until either it found it or it examined every element.

This is the best we can do when searching through a list of values that is not in sorted order. We have to look at every value in the array until we find the key. If there are N items in the array, we need to check compare our search key to N values in the absolute worst case that either it is not there or it is in the last position.

If however, the array is sorted, then it is possible to search in a much faster way. In general it is better to sort the values in the array for faster searching later. Presumably we will search the data very often, so if sorting it costs us a bit of time, it is worth the savings we get from faster searching later.

Sorting

There are many different ways to sort an array, some much faster than others. Some are easy to understand and some are harder to understand. Some require additional memory to sort, others can be done "in place", meaning by rearranging the elements of the array without using extra memory. The sorting problem can be stated as follows,

> "Given an array A of size N, rearrange the elements of A so that A[0] <= A[1] <= A[2] <= ... <= A[N-1]."

Notice that the elements are rearranged. In other words, sorting is a permutation, a rearranegment of an array.

We will start with a very intuitive sorting algorithm that is called selection sort. You will see why it is called selection sort very soon. The idea is simple:

> Find the minimum element of the array from 0 to N-1. Suppose it is in index K.
>
> Swap A[0] and A[K]. Now A[0] has the smallest element.
>
> Find the minimum element of the array from 1 to N-1. Suppose it is in index K.
>
> Swap A[1] and A[K]. Now A[1] has the second smallest element.
>
> Find the minimum element of the array from 2 to N-1. Suppose it is in index K.
>
> Swap A[2] and A[K]. Now A[2] has the third smallest element.
>
> Repeat this procedure until there are two elements remaining: A[N-2] and A[N-1}. Find the smaller of the two and swap it into A[N-2].

When the above procedure has finished, the array is in sorted order, smallest to largest. The procedure can be stated more succinctly and abstractly as follows:

```
let bottom = 0, 1, 2, ... N-2 {
    let k = index_of_minimum(A, bottom, N-1)
    swap(A[k], A[bottom)
}
```

This pseudocode translates directly into the following C++ function. Because you can sort any types of values that can be compared using ordinary comparison operators, rather than writing it with a fixed type, we use the type `T`, and a `typedef` that can be changed easily that here makes `T` another name for the `string` type.

```
typedef string T;

// Returns the index of the smallest element in A between
// A[bottom] and A[maxIndex]
int indexofMin(const T A[], int bottom, int maxIndex);

// Swaps the values stored in x and y
void swap( T & x, T & y )


// Precondition:  A[0..N-1] contains data of type T, to be sorted
// Postcondition: A[0..N-1] is sorted in ascending order.
void selectionSort ( T A[ ], int N)
{
    int smallest; // index of smallest item in unsorted part of A
    int first;    // index of first item in unsorted part of A

    first = 0;  // start with last at highest index in A
    while (  first < N-1) {
        smallest = indexofMin(A, first , N-1);
        swap( A[smallest], A[first]);
        first++;
    }
}

int indexofMin(const T A[], int bottom, int maxIndex)
{
    int m = bottom;
    for (int k = bottom+1; k <= maxIndex; k++)
        if ( A[k] < A[m])
            m = k;
    return m;
}

void swap( T & x, T & y );
{
    T temp = x;
    x = y;
    y = temp;
}
```

Notes

- This algorithm can be written symmetrically using maximums instead of minimums. That is a good exercise to try.
- If the call to `swap()` were replaced by the three instructions that do the swap, this would compile with a C compiler.
- This always takes the exact same amount of time, even if the array is already sorted. The call to `indexofMin()` always iterates its loop (`N-first`) times and the swap is always performed, even if the two indices are the same. It will swap a value with itself in this case. It is an example of an ***unnatural sort***. A sorting algorithm is ***natural*** if arrays that are already in order require less swapping than arrays that are very out of order.

Once an array has been sorted, searching it is much faster. Later we will see a method of searching called binary search, in which the portion of the array being searched is cut roughly in half in each iteration.

Multidimensional Arrays

The arrays we have studied so far have a single dimension – they represent data items that can be arranged in a sequence, and therefore they are also called ***linear arrays***. Arrays can have more than one dimension in C and C++.

Declaring a Multidimensional Array

You declare an array with multiple dimensions using a syntax that is similar to one-dimensional arrays.

Syntax:

```
elementtype arrayname [ size1 ] [ size2 ] ... [ sizeD ];
```

where

- *elementtype* is any type that already exists
- *arrayname* is the name of the array
- *size1* is the number of elements in the first dimension, size2, the number in the second dimension, and so on, and D is the number of dimensions in the array.

Basically you tack on a pair of square brackets with a number in between for every dimension that you want the array to have.

Examples

```
char    tictactoe [3][3];  // a 3 by 3 grid of chars
string  chessboard[8][8];  // an 8 by 8 grid of strings
int     pixel[1024][768];  // a 1024 by 768 grid of ints
double  cube  [5][5][5];   // a 5 by 5 by 5 cubic grid of doubles
```

In two dimensions, we think of the first dimension as the row, the next as the column. There is no rule for this and there is nothing in the language that associates the dimensions with rows or columns. It depends how you program it. In a three dimensional array, the first subscript would be like the row, the second like the column, and the third, like the plane in which this row and column are located, i.e., the height. People usually refer to the first dimension as the row, the second as the column, and the third, if it exists, as the plane.

Accessing Elements of Multidimensional Arrays

Higher dimensional arrays are accessed just like one dimensional arrays. The subscripts match their dimension:

```
tictactoe[0][2] = 'x';          // puts an 'x' in first row, last column.
chessboard[7][7] = "redqueen"; // puts the string "redqueen" into the lower
                                // right hand corner of the chessboard.
```

The following example shows how we could "color" the elements of a checkerboard. It demonstrates accessing two-dimensional array accesses.

```
const char RED = 'r';
const char BLACK = 'b';
char   checkerboard[ROWS][COLS];
int i, j;

for ( i = 0; i < 8; i++ )
    for ( j = 0; j < 8; j++ )
        if ( ( i + j ) % 2 == 0 )
            checkerboard[i][j] =  RED;
        else
            checkerboard[i][j] =  BLACK;

for ( i = 0; i < 8; i++ ) {
    for ( j = 0; j < 8; j++ )
        cout << checkerboard[i][j] << ' ';
    cout << "\n";
}
```

Processing Higher Dimensional Arrays

The most commonly used higher dimensional arrays are two-dimensional. They can be used to represent a wide assortment of objects, such as graphical images, mathematical matrices, screen representations, two-dimensional surfaces in general, mazes and other 2D cellular arrangements (chess boards, game boards). Just as one dimensional arrays go hand in hand with for-loops, two-dimensional arrays are naturally paired with nested for-loops.

The following example shows a code snippet that creates a random grid of asterisks. Notice that the middle loop is not a nested loop. Sometimes two-dimensional data can be processed within a single loop.

```
const int   COLS = 100;
const int   ROWS = 80;
const int   MAXSTARS = COLS*ROWS/10;
char   grid[ROWS][COLS];
int i, j, count = 0;

srand(time(0));
```

```
// Initialize the grid to all blanks
for ( i = 0; i < ROWS; i++ )
    for ( j = 0; j < COLS; j++ )
        grid[i][j] = ' ';

while ( count < MAXSTARS ) {
    i = random(ROWS);   // generate a random row index
    j = random(COLS);   // generate a random column index
    grid[i][j] = '*';   // put an asterisk there
    count++;
}

// Print out the grid
for ( i = 0; i < ROWS; i++ ) {
    for ( j = 0; j < COLS; j++ )
        cout << grid[i][j];
    cout << "\n";
}
```

Multidimensional Array Parameters[1]

To understand how to use multidimensional array parameters, you need to remember how one-dimensional array parameters are used. When you write a function declaration such as

```
void process( double data[], int size);
```

you are telling the compiler that the function named `process` will be given an array of basetype `double` as the first argument and an integer as the second argument. The array parameter `data[]` is just the starting address of the array that is passed to it. The compiler knows that it is of type `double`, so it knows where the remaining array elements are loacted in memory, because it knows how many bytes are in the double type. In other words, the array parameter tells the compiler where the start of the array is and how many bytes are in each entry, and this is all that the compiler needs.

This same logic applies to multidimensional arrays. The compiler needs to know the size of each element and the starting address. This means that the array parameter must be written with the size of every dimension except the leftmost one, as in

```
void processdata( int data[][10][20], int size);
```

which declares that processdata is a function with a 3D array parameter whose second dimension has 10 elements, and third has 20 elements. The compiler thus knows that `data[1]` starts 200 integers after `data[0]` and `data[2]` starts 200 integers after `data[1]`, and so on.

[1] This material will not be covered in class.