Arrays

An array is a sequence of consecutive memory locations, of the same type, any one of which can be referenced via a common pointer to the first entry in the sequence using an index. This is an implementation description and not the description of an abstract data type.

Declaring an Array in C++

Most common way:

```cpp
type  arrayname[constant integer expression];
```

*type* is the common data type of all entries. *arrayname* is a user-defined identifier. [*] contains the size or length of the array which is a constant integer expression.

Examples

- `int list[5];`
- `float x[100];`
- `double y_coord[100];`
- `char ch[1000];`

These only declare the array; i.e., reserve the memory. None of the individual memory locations has been accessed, and no values have been assigned to array entries.
Accessing Elements of the Array

Consider the example:

```java
int list[5];
list[0]=2;  // Put 2 in first location.
list[1]=8;  // Put 8 in second location.
list[3]=100; // Put 100 in fourth location.
list[4]=300; // Put 300 in fifth and last location.
```

Range of Indices

In the previous examples, the quantity inside the `[ ]` is called the index. What do you notice about the index and the location of the array referenced?

Position vs. Index

When referencing an array, the index of the array position is always one less than the sequence number of a location in the array. In other words, 0 references the first location, 1 the second, and so forth.

So, what can the largest index of an array position be?
Would the following be allowed with the array declaration given (int list[5];)?

list[5] = 400;

Syntactically: Yes
Run Time: Maybe yes, Maybe no.
Logically: A potential disaster.

list[5] = 400; (cont)

This merely places 400 into the “sixth” sequential memory position from the beginning of the array. It could result in an illegal memory reference, if you are lucky; otherwise, it overwrites a memory location that could have been referenced by another variable, creating a logical error.

Indices

In summary, if an array is declared by

type arrayname[N];

where $N$ is a constant integer value, then the indices for this array range from 0 to N-1 inclusive.

If an index is out of bounds and doesn't cause a runtime error, it often creates a logical error that is difficult to debug. So, be careful with indices of arrays.
double y_coord[100];

This is a larger array, and we likely would not want to fill it up by typing 100 assignment statements. For each i in the correct range, suppose we want y_coord[i] to have the double value of i.

for(i=0;i<100;i++)
   y_coord[i]=static_cast<double>(i);

Would we need the cast operator in this case?

Alternate Declaration Methods

int list1[5] = {1,2,3,4,5};
int list2[5] = {0};
int list3[5] = {1,2,3};
int list4[ ] = {6,2,7,9,10};
int list5[ ] = {1,2,3,4,5,6,7};

What happens with each of these declarations?

A Syntax Error

int list6[5] = {1,2,3,4,5,6,7};

This is a syntax error, because the right hand side of = has more elements to insert than the size of the array declared on the left hand side.
Valid Indices

Consider
list[expression] = computed value;
What can the index expression be?
It can be any integer expression whose computed value falls within the legal range of the array indices.

What is the Array Name?

If you declare an array, the name of that array is an array type, which can be thought of as a constant pointer whose value is the address of the first location in the array.
int list[5];
Think of list as a pointer. Its value is the address of the first of the five integers in the array sequence. It is constant in the sense that it cannot be changed; i.e., you can’t assign another address or value to list.

Accessing the Array Elements

When you then write list[i] you are “de-referencing” the array and this means “the value of the (i+1)st location in the array”. The quantity i is the index and indicates how many locations to “skip over” to reach the specified location in the array. It merely adds i times the array type size to the address stored in list to find the starting byte of the specified array element.
Selection Sort

```c
int MxInd;  //MxInd, index to
//current max val
int i, j;
for(i=N-1; i>0; i--){
    MxInd = 0;  //initialize to first item
    for(j=1; j<i; j++)
        if(A[j] > A[MxInd])
            MxInd = j;  //MxInd has
            //current max val
    swap(A[i],A[MxInd]);
}
```

Cost of this Algorithm

Consider the statement if(A[j] > A[MxInd]) that appears in the algorithm. How many times is it executed; i.e., how many element comparisons are done? For each value of i, j runs from 1 to i; therefore, for each i, the if statement is done i times. Then on the outer loop, i runs from N-1 down to 1. 

\[ N-1 + N-2 + N-3 + \ldots + 3 + 2 + 1. \]

In other words we need to sum up the first N-1 positive integers.

Sum of first N-1 positive integers.

\[ 1 + 2 + 3 + \ldots + (N-1) = \frac{(N-1)N}{2} \]
When multiplied out, the previous expression becomes:

\[(N^2 - N)/2 = N^2/2 - N/2\]

What is the dominating term in this last expression? Based on that, what happens to the total work when \(N\) is doubled?

---

**Passing an Array to a Function**

Suppose that instead of finding the \(Mx\) within the body of the loop we desired to do that with a function. We would like to have:

\[MxInd = \text{IndexOfMax}(A, i);\]

Note we are passing the array name and the last index in the array we want to consider. The function will find the index of the maximum value in \(A[0..i]\).

---

**The Function Definition**

```c
int IndexOfMax(int X[], int last)
{
    int i;
    int Mxi = 0;
    for(i=0; i<=last; i++)
        if(X[i]>X[Mxi])
            Mxi = i;
    return Mxi;
}
```
Consider the Header

```c
int IndexOfMax(int X[], int last)

Note the return type; the parameter corresponding to the array type; and the parameter last which provides the function with information regarding how far to go in addressing the array elements.

Note also that in X[] the array size is not specified. No effect would take place by putting a number inside [], rather that information is passed through the variable last.
```

Revised Code with Function

```c
int MxInd;  //MxInd, index to //current max val
int i;
for(i=N-1; i>0; i--){
    MxInd=IndexOfMax(A,i);
    swap(A[i],A[MxInd]);
}
```

Do it all with functions:

Suppose that in `main()`, we don't want any code details, rather just the statement:

```c
SelectSort(A,N-1);  //sort  //A[0..N-1]
```
Definition for SelectSort

```c
void SelectSort(int Y[], int last)
{
    // Sorts Y[0..last]
    // This Function calls IndexOfMax
    int MxInd, i;
    for(i=last; i>0; i--){
        MxInd = IndexOfMax(Y, i);
        swap(Y[i], Y[MxInd]);
    }
    return;
}
```

Two More Sorting Routines

- Bubble Sort
- Insertion Sort

The method behind these will be discussed, and discussion will be followed by an activity to write the C++ code for it.