Arrays

Arrays are useful, because they allow us to store many values of the *same type* under the same name. We can think about arrays like this:

[ ] [ ] [ ] [ ] [ ] [ ]

Each of the bracket pairs is a slot or *element* in the array, and you can store information in each one of them. It is like having a group of variables side by side.

Another approach is to think of an array as a simple variable with indices or subscripts added (a_0, a_1, a_2, a_3, a_4, a_5, …). The brackets [ ] are used to contain the array subscripts.

$$a = [\ a_0\ ]\ [\ a_1\ ]\ [\ a_2\ ]\ [\ a_3\ ]\ [\ a_4\ ]\ [\ a_5\ ]$$

<u>Example:</u>
Suppose we want to record three temperature measurements. One approach could be to declare three different variables:

int temp0, temp1, temp2;

Suppose instead we need to record 500 temperatures. Using unique variables is probably not a good idea. We need to use arrays. To use temp[0], temp[1], temp[2], … , temp[499] in a program, we could simply declare

int temp[500];

Declaring and Initializing Arrays

Declaring an array involves two things: allocating space in memory (mandatory when the array is first defined), and initializing values of the array element(s) (optional).

Declaration syntax

type array_name [*size*]; // this array is called "array_name" and has "size" number of elements.
// "size" MUST be a <u>constant integer expression</u> (NOT a variable)
// each element in the array is of the SAME data type "type"
// to reference or *index* an element within the array, we can use:
// array_name [0], array_name [1], … , array_name [size –1]

Examples:

```
int    a1 [100];          // this declares an array of 100 integers
char a2 [90+10];          // an array of 100 elements – each of which are of type char
float a3 [20];            // an array of 20 numbers of type float
float a4 [5*4];           // also an array of 20 numbers of type float
```

To access a specific element of an array, you must write the array name followed by an index number in brackets [ ] . **The indices begin with 0 and end with (number of elements – 1). 0 – 99** in a 100-element array. In example above, elements of array a3 are referenced by a3[0], a3[1], …, a3[19].

- Array size must be defined in declarations using a constant integer expression (no variables!)
- Indices of a size n array start at 0 and go to n-1. Indices must be integers.
- Array elements must be referenced with [ ], not just the array name

Like variables, arrays need to be initialized or values input into the elements before use…

Initializing Arrays in Declaration Statements

An array may be initialized in its definition/declaration.

```
float a[5] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0};
// This initializes a[0] to 0.0, a[1] to 1.0, and so on…

int a[100] = {0};  // initializes all elements of a to 0
int a[500] = {9};  // initializes a[0] = 9, and all other elements (a[1] to a[499]) are set to 0
int a[250];        // contents of a are UNKNOWN garbage. can't assume anything about contents
```

If an array is initialized in its definition, its size may be omitted:
char vowels [ ] = {'A', 'E', 'I', 'O', 'U'};

```
int    a [ ] = {2, 4, 5, -9};        // and
int    a [4] = {2, 4, 5, -9};        // are EQUIVALENT declarations
```

This kind of "mass" initialization can ONLY be done in the declaration statement. Within the actual program, you must explicitly assign each cell one by one (manually, via a formula, via for loops). For example: a[0] = 1; a[1] = 15; a[2] = 0; a[3] = 4; and so on…

After arrays have been initialized, cell indices can be represented by variables of **int** type.

Bounds Checking and Access Violations

C++ does not do bounds checking on arrays. That is, the system does *not* check whether an index expression falls between 0 and (*size* – 1), where *size* is the array's size; but errors will occur (overflow, underflow).

```
int a[3];       // legal indices are 0, 1, and 2
a[0] = 9;       // ok
a[1] = 8;       // ok
a[2] = 7;       // ok
a[3] = 6;       // ERROR – ARRAY OVERFLOW
a[-2] = 1;      // ERROR – ARRAY UNDERFLOW
```

Array overflow and underflow typically result in access violations, often preventing execution of your program. It is your responsibility to ensure that your indices are legal!

Arrays and Functions

An array can be passed as an argument to a function. Any array parameters in the function prototype must include square brackets to indicate that that parameter is indeed an array; however, the number of cells is *not* enclosed in the square brackets (for a 1-dimensional array or *vector*).

Example.

function header or prototype:

```
    int sum ( int  a[ ], int  b)      // parameters of function "sum" are array "a" and integer "b"
```

function call:

```
    sum ( array_name, value);  // to pass an array, we simply enter its name as the argument
```

A function with an array parameter operates on the ACTUAL cells of the array – not on a copy of the cells, as we may be used to from our previous work with functions. In other words, any assignment statements or other commands within the function that change the array parameter(s) *will consequently change the corresponding array argument that was sent to it*.

```
// Sample function where a passed array argument will be changed after leaving the function
// Assume first n elements of arrays "a1" and "a2" are defined
// the first n elements of input arrays "a1" and "a2" are being added and stored in array "sum"
void add_arrays (double a1[], double a2[], double sum[], int n)
{       int i;
        for (i = 0; i < n; i++)
                sum[i] = a1[i] + a2[i]
}
```

IN-CLASS PRACTICE … Write a Function

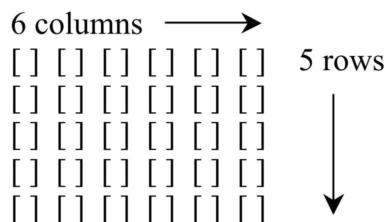
- Write a function that squares every element of an array, sums up the squares and returns this sum of the squares (SS)
- Input 4-10 numbers from the keyboard, into an array
- Send the array to the function, plus the size of the array, square each element, sum them. Return SS.

Multidimensional Arrays

An array with one pair of brackets [] is 1-dimensional. An array with two pairs of brackets [] [] is 2-dimensional; and so on. Arrays with more than 1 dimension are called multidimensional arrays. You could make a 3-dimensional or even a 100-dimensional array, though you probably will not need to; i.e., there are no restrictions in C++ to how many dimensions you can have.

A 1-dimensional array is called a vector.
A 2-dimensional array is called a matrix.

You can visualize a 2-dimensional (2-D) array as a table consisting of rows and columns. For example, here is a 5 x 6 array (read this as "5 by 6" array), where there are 5 rows and 6 columns:



If we wanted to use such an array in our program, we could define the array "b" as follows:

int b [5] [6];

To access elements in the 2-D array b, you would now two sets of indices: one for the row you choose, and one to specify which column within that row the desired element is. **Again, each set of indices begin with 0.** Specifically, the array elements of b are indexed as follows:

b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]	b[0][5]
b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]	b[1][5]
b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]	b[2][5]
b[3][0]	b[3][1]	b[3][2]	b[3][3]	b[3][4]	b[3][5]
b[4][0]	b[4][1]	b[4][2]	b[4][3]	b[4][4]	b[4][5]

Declaring and Initializing 2-Dimensional Arrays

int array1 [k] [n]; // defines array1 to be a (k x n) matrix of integers, i.e., k rows and n columns

Question: How many cells are in this array?

Multidimensional arrays can be initialized within the declaration statement as well. Initialization is similar to vectors, except now you need the elements of each row separated by braces:

Example.

```
int nums [2] [3] = { {1, 3, 5},      // 1st row of 2 x 3 matrix
                     {7, 9, 11} }; // 2nd row of 2 x 3 matrix
```

nums =

1	3	5
7	9	11

You can also set the entire matrix equal to 0 with the statement, just like the 1-dimensional case:

```
int nums [2] [3] = {0};
```

Then nums =

0	0	0
0	0	0

Again, this kind of "mass" initialization can ONLY be done in the declaration statement. Within the actual program, you must explicitly assign each cell one by one (manually, via a formula, via nested for loops).

This kind of initializing is sufficient for testing your code for small matrices. Very tedious for large arrays or for arrays with many dimensions.

Processing 2-Dimensional Arrays

Use nested for loops…

```
for (i=0; i <= NROWS - 1; i++)
{
        for (j=0; j<=NCOLS - 1; j++)
                {
                        cin >> sound [i][j];        // put user inputs into i x j matrix sound
                }
}
```

Example.
Write a function to check whether a 3x3 box of characters is completely filled:

char box [3] [3] ;

An example of the contents of box could be:

X	X	X
X	X	
X		X

```
int filled (char box [3][3])    // function "filled" takes 3x3 array "box" and returns 1 if it's filled
                                // and returns 0 it any of the cells are empty
{ int  row, col,                // row and column subscripts
       answer;                  // answer =1 if box is filled, otherwise = 0

  answer = 1;                   // we assume box is filled, until a blank is found
        for (row = 0; r < 3; r++)
              for (col = 0; c < 3; c++)
                    if (box[row][col] == ' ')        // set answer = 0 if we find a blank
                          answer = 0;
   return (answer);
}
```

Passing Multidimensional Arrays to Functions

Same as 1-dimensional case, except that to declare a parameter for a multidimensional case, we MUST SPECIFY THE NUMBER OF CELLS AS A CONSTANT IN ALL DIMENSIONS BEYOND THE FIRST.

Example.

function header or prototype:

```
void print_table ( int  jobs[ ] [4])      // this function prints each entry in an n x 4 table using the
                                          // 2-dimensional array parameter jobs.
```

function call:

```
int job_table [100] [4];                 // define some 100 x 4 "job_table" array
print_table ( job_table);                // pass the 2-D array "job_table" to the function print_table
```

IN-CLASS PRACTICE

Write the C++ code to have the user type in data to a 3 by 3 matrix, and print the data out. Use doubles.