Chapter 7: Arrays

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Arrays

- Arrays enabling storing multiple values under a single variable
- Lists are intuitive to humans, and arrays let us mimic a list of items
  - One note: in C++, all lists must have the same type.
- If multiple values are stored in a single variable, we need a way to access each value
- We access values stored in an array using **indices**, called *subscripts*
- Values inside of an array are *homogeneous*, meaning they all have the same type
  - Can't mix `int`s with `float`s or vice-versa
- Later we will introduce the idea of a *pointer*, which extend the use of arrays

Declaration and Initialization

- C++ arrays are declared in the following form

```
type name[number of elements];
```

- `type` specifies the type of every element in the array (since arrays are homogeneous, we only specify one type)
- `name` is the identifier/variable name we will use to refer to the array
- `number of elements` is the number of `type` elements that the array can store
- To declare an array of 6 integers called `numbers` we would use:

```
int numbers[6];
```

- To declare an array of 6 characters called `letters` we would use:

```
char letters[6];
```

- We can initialize the array when we declare it using curly braces and initialization values using an initializer list:

```
int point[6] = {0,3,1,6,7,2};
```

- Or we can only initialize the first few elements (this initializes the first 3):

```
int parital[6] = {1,2};
```

- We can also omit the size of the array and use the size of the initializer as the size of the array (this will have space for 6 integers):

```
int point[] = {0,3,1,6,7,2};
```

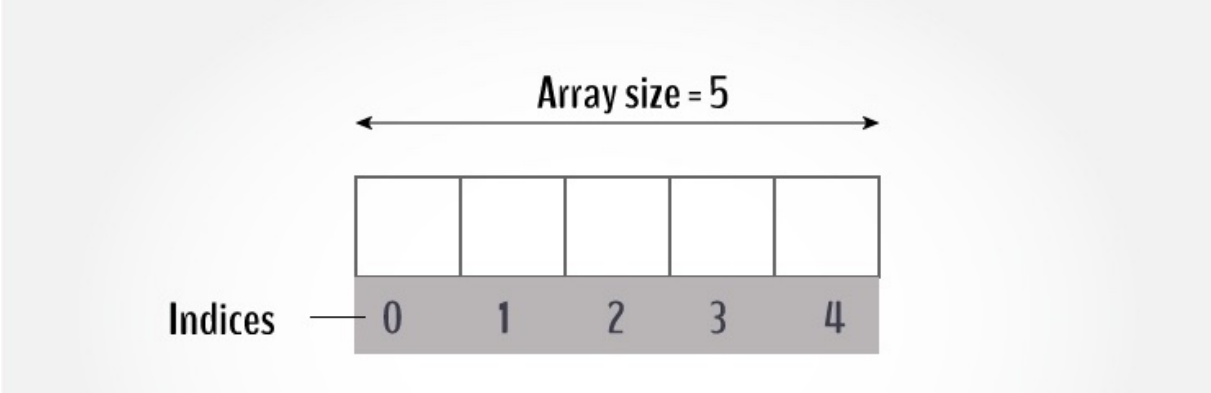

Figure 1: C++ Array

Array Access

- Now we know how to declare and initialize and array, but how do we access elements in an array?
- We'll use an *index* or *subscript* to specify which element of the array we want to access
- Arrays are 0-indexed in C, meaning the index of the first element in the array is 0, the second element in the array is 1, the third is 2, and so forth.
  - Important note: the last **valid** index in an array is the size-1. For instance, in an array of length 6 (an array that can store 6 elements), the last valid index is 5. 0-5 is 6 numbers.
- Example:

```
int point[6] = {0,3,1,6,7,2};
int thirdEle = point[2]; // arrays are 0-indexed in C, so thirdEle will
    have the value of 1
```

- What happens if you access an array with an index is out of the bounds of the array (i.e. use 6 as an index to the `point` array?
  - It depends. Sometimes the compiler can catch the error, but it's not guaranteed to.

  - If your program executes, it will be in *undefined behavior* (UB), which means the rest of your program's output is rendered meaningless and unpredicable, even if it outputs the correct thing
    * Undefined behavior is a large and somewhat esoteric definition, but the point is that C++ makes zero guarantee about what will happen after you've triggered undefined behavior.
- Examples:

```
char y;
int z = 9;
char point[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not
   always raised
y = point[15];
y = point[-4];
y = point[z];
```

- Your program may continue running normally after these cases, but you have entered UB. This must be avoided at all costs!
- But there's got to be a better way to make sure we stay within the bounds…
  - Well not for every case, but for any type of loop, we can use `sizeof()` to as the limit on the number of iterations the loop executes
  - Here's an example:

```
int i;
int arr[] = {3, 6, 9, 12, 15};

cout << "sizeof(arr): " << sizeof(arr) << endl;
cout << "sizeof(int): " << sizeof(int) << endl;

int arr_len = sizeof(arr) / sizeof(int);

cout << "array is length " << arr_len << endl;

for (i = 0; i < arr_len; ++i)
{
  cout << "arr[" << i << "]: " << arr[i] << endl;
}
```

- This is a great way to ensure you stay within the bounds of the array!

Array size

- Note that for native C++ arrays, the array size is fixed after you declare the size.
  - It is not possible to make an array of length 6 and extend it to size 10, or shrink it to size 3 (or any size change)
  - We typically get around this by allocating more memory than we need, and use a variable to keep track of how much of the array is actually used.
- Array sizes cannot be changed, but later we learn about the `vector` class that allows resizing.
  - The `vector` class is part of the Standard Library, so it is not a functionality of the C++ language itself.

Passing arrays to functions

- To pass an array to a function, we'll pass the name of the variable of the array.
- However, in the function signature, we must tell the compiler we are passing an array:

```cpp
#include <iostream>

// [] after the variable name indicates the variable is an array
float average(float age[]);

int main()
{
    float avg
    float age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
    int arr_len = sizeof(arr) / sizeof(float);

    avg = average(age, age_len); /* Only name of array is passed as
        argument. */

    cout << "Average age=" << avg << endl;
    return 0;
}

// [] after the variable name indicates the variable is an array
float average(float age[], size_t age_len)
{
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < age_len; ++i) {
        sum += age[i];
    }
```

```
26      avg = (sum / 6);
27      return avg;
28  }
```

Returning arrays from functions

- We'll have to introduce a symbol we will talk in greater detail about when we discuss pointers and passing-by-reference. We need to cover this for the homework assignment, but the concept will be covered later.
- We'll use the pointer type-qualifer `*` as a part of the return type to indicate we wish to return an array.
- Inside of the function, we'll return the symbol of the array **without accessing an element using an index**
- Example:

```
1  //NOTICE: the asterisk (star) next to int indicates we are returning an
       array
2  int* add_to_zeroth_element(int arr[], size_t arr_len, int value){
3    // this is just a dummy array operation, in practice you'll do
         wonderful and amazing things here
4    arr[0] += value;
5    // NOTICE: return the array, we don't use [] here, just the name of
         the array.
6    return arr;
7  }
8
9  int main(){
10   int arr[] = {1,2,3};
11   // notice the type here has to match the return type of the function.
         Exactly what's going on here will be covered with pointers.
12   int* result = add_to_zeroth_element(arr, 3, 5);
13 }
```

- Note that we aren't required to return the array. Since the array is effectively passed-by-reference, any changes we make to `arr` in `add_to_zeroth_element()` will persist in the `arr` in `main()`

Example: `fill_array_with_input.cpp`

```
1  // Array1.cpp : Defines the entry point for the console application.
2  //
```

```
3
4  #include <iostream>
5  #include <cstdlib>
6  #include <fstream>
7  #include <cctype>
8  #include <string>
9
10 using namespace std;
11
12 void fillarraywithinput( int array[], const int& size );
13
14 int main(int argc, char* argv[])
15 {
16     int size = 1024;
17         int dataarray[ 1024 ];
18     int i;
19
20     // read from cin
21     fillarraywithinput( dataarray, size );
22
23     // sort
24     for (i = 0; i < size; i++)
25         for (int j = 0; j < size; j++)
26             if (dataarray[i] < dataarray[j]) {
27                 int temp = dataarray[ i ];
28                 dataarray[ i ] = dataarray[ j ];
29                 dataarray[ j ] = temp;
30             }
31
32     // print out
33     for (i = 0; i < size; i++) {
34         cout << dataarray[i] << " ";
35     }
36     cout << endl;
37
38     return 0;
39 }
40
41 void fillarraywithinput( int array[], const int& size ) {
42     string data;
43     int k = 0;
44     int startcntr = 0, endcntr = 0;
45     cout << "Enter one line of data to sort" << endl;
```

```
46      getline( cin, data );
47      data += " ";  // in case string does not end with whitespace
48      for (endcntr = 0; endcntr < data.length(); ++endcntr) {
49          if (isspace(data.at(endcntr)) && startcntr <= endcntr) {
50              string bit = data.substr( startcntr, endcntr - startcntr +
                    1 );
51              if (isspace( bit.at(0) )) {
52                  startcntr = endcntr + 1;
53                  continue;
54              }
55              int value = atoi( bit.c_str() );
56              array[ k++ ] = value;
57              startcntr = endcntr + 1;
58          }
59      }
60      size = k;
61  }
```

Multi-dimensional arrays

- Muti-dimensional arrays are arrays-of-arrays.
- The most basic multi-dimensional is a 2-dimensional array, which creates a rectangular array. Each row has the same number of columns.
- To get an int array with 3 rows and 5 columns, we write:

```
1  int arr[3][5];
```

- To access/modify a value in the array, we need two subscripts: one for the row we wish to access, and a second for the column we wish to access:

```
1  arr[1][3] = 5; // sets the element in the second row and forth column
       to 5
```

- We can also initialize a multi-dimensional array in a similar fashion as a single-dimension array using an initializer list:

```
1  int two_d[2][3] = {{ 5, 2, 1 },
2                     { 6, 7, 8 }};
```

- The amount of columns must be explicitly specified, but the compiler will sort out how many rows are needed based on the initializer list. We could have written

```
int two_d[][3] = {{ 5, 2, 1 },
                  { 6, 7, 8 }};
```

Passing multi-dimensional arrays to functions

- Exactly the same as passing single-dimension, except we must specify the number of columns
  - Can also specify both rows and columns if you only want a

```
#include <iostream>

void print_arr(int num[][2]);

int main()
{
  const int nr=2, nc=2;
  int num[nr][nc], i, j;
  for (i = 0; i < nr; i++)
  {
    for (j = 0; j < nc; j++)
    {
      cout << "element - [" << i << "][" << j << "]: ";
      cin >> num[i][j];
    }
  }
  // passing multi-dimensional array to function
  print_arr(num, nr);

  return 0;
}

void print_arr(int num[][2], size_t num_len)
{
  int i, j;
  for (i = 0; i < num_len; ++i)
  {
    for (j = 0; j < 2; ++j)
    {
      cout << num[i][j] << "  ";
    }
    cout << endl;
  }
```

```
34 }
```

Returning multi-dimensional arrays from functions

- This is a bit trickier and we will cover this when we cover pointers

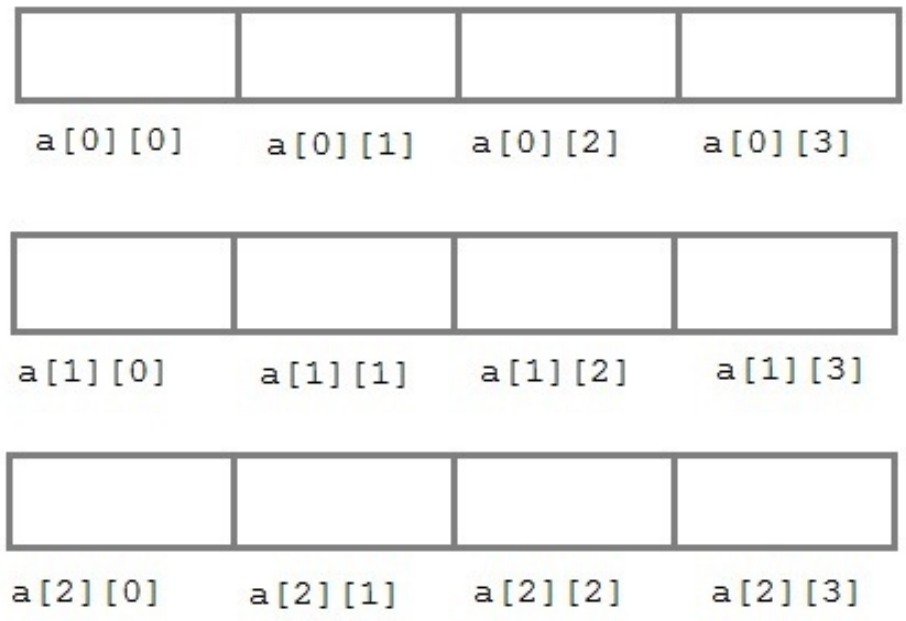

Figure 2: Multi-dimensional arrays

Exercises

1. Write a program in C++ to store 10 elements inputted by the user and write a function to print the contents of the array.

```cpp
#include <iostream>

using namespace std;

int main()
{
  int arr[10];
  int i;
```

```
  cout << "\n\nRead and Print elements of an array:\n";
  cout << "----------------------------------------\n";

  cout << "Input 10 elements in the array :\n";
  for(i=0; i<10; i++)
  {
    cout << "element - " << i << " : ";
    cin >> arr[i];
  }

  cout << "\nElements in array are: ";
  for(i=0; i<10; i++)
  {
    cout << arr[i] << "  ";
  }
  cout << endl;
}
```

2. Write a program in C++ to prompt for the number of elements the user wishes to input (n < 100) and then prompt for the user to input each element. Then print all unique elements in an array.

```
#include <iostream>

using namespace std;

int main()
{
  int arr1[100], n, count_ele = 0;
  int i, j, k;

  cout << "Input the number of elements to be stored in the array (must
      be less than 100):";
  cin >> n;

  cout << "Input " << n <<" elements in the array:\n";
  for (i = 0; i < n; i++)
  {
    cout << "element - " << i << " : ";
    cin >> arr1[i];
  }

  /*Checking duplicate elements in the array */
  cout << "\nThe unique elements found in the array are: \n";
```

```
  for (i = 0; i < n; i++)
  {
    count_ele = 0;

    /*Check duplicate before the current position and
     increase counter by 1 if found.*/
    for (j = i - 1; j >= 0; j--)
    {
      /*Increment the counter when the search value is duplicate.*/
      if (arr1[i] == arr1[j])
      {
        count_ele++;
      }
    }
    /*Check duplicate after the current position and increase counter
       by 1 if found.*/
    for (k = i + 1; k < n; k++)
    {
      /*Increment the counter when the search value is duplicate.*/
      if (arr1[i] == arr1[k])
      {
        count_ele++;
      }
    }
    /*Print the value of the current position of the array as unique
       value
     when counter remain contains its initial value (zero).*/
    if (count_ele == 0)
    {
      cout << arr1[i] << " ";
    }
  }
  cout << "\n\n";
}
```

3. Write a program in C++ to store a 2x2 2-dimensional array. Elements are inputted by the user. Print the matrix and find the sum of rows an columns of the matrix.

```
#include <iostream>
#include <iomanip>

using namespace std;

```

```
6  int main()
7  {
8    const int n = 2;
9    int arr1[n][n], rsum[n], csum[n];
10
11   cout << "Input elements in the 2x2 matrix:\n";
12   for (int i = 0; i < n; i++)
13   {
14     for (int j = 0; j < n; j++)
15     {
16       cout << "element - [" << i << "][" << j << "]: ";
17       cin >> arr1[i][j];
18     }
19   }
20   cout << "The matrix is:\n";
21   for (int i = 0; i < n; i++)
22   {
23     for (int j = 0; j < n; j++)
24       cout << std::setfill('0') << std::setw(4) << arr1[i][j] << " ";
25     cout << endl;
26   }
27
28   /* Sum of rows */
29   for (int i = 0; i < n; i++)
30   {
31     rsum[i] = 0;
32     for (int j = 0; j < n; j++)
33       rsum[i] = rsum[i] + arr1[i][j];
34   }
35
36   /* Sum of Column */
37   for (int i = 0; i < n; i++)
38   {
39     csum[i] = 0;
40     for (int j = 0; j < n; j++)
41       csum[i] = csum[i] + arr1[j][i];
42   }
43
44   cout << "The sum of the rows the matrix is:\n";
45   for (int i = 0; i < n; i++)
46   {
47     cout << std::setfill('0') << std::setw(4) << rsum[i] << " " << endl
         ;
```

```
48    }
49    cout << endl << "The sum of the cols the matrix is: " << endl;
50    for (int j = 0; j < n; j++)
51    {
52      cout << std::setfill('0') << std::setw(4) << csum[j] << " " << endl
          ;
53    }
54    cout << endl << endl;
55  }
```