Arrays

- An array is a collection of a fixed number of variables, all of the same data type.
- individual variables in an array are called elements.
- Each element is in contiguous (adjacent) memory space.
- Elements in an array are accessed with an index number. e.g. `arr[0]`
- The first element in an array has an index of 0.
- A common pogroming error is not to account for the zero-based index in arrays, resulting in being “off-by-one.”

One-Dimensional Arrays

- A one dimensional array is an array in which all elements are in list form.
- To initialize an `int` array as having 10 values. `int arr[10];`
- All variables in an array share the same data type.
- A `for` loop initializing each value in the array.

```
int arr[10];
for (int i = 0; i < 10; ++i)
{
    arr[i] = i;
}
```

- Use square brackets to access an element in the array.

```
cout << arr[4] << endl;
```

- Remember the first element in the array is the *zeroth* element.

```
cout << arr[0] << endl;
```

- So if you initialize an array with 10 elements the last element will be at index 9 not 10.

```
cout << arr[9] << endl;
```

- Unfortunately C++ arrays *do not know their own size* and they *do not check if the index value is within the range*.
- Arrays in C++ are really just a pointer, and it have no range or size information for the array, this is also discussed later in the section on Pointers.
- If you try to access elements outside the array the compiler will give you whatever memory happens to be beyond the array. Potentially this can be very dangerous! C++ does not give an "index out of bounds" error like most other languages.
- Try `cout << arr[12] << endl;` on this array. In my case I get `1606416472` which is just happens to be what’s in memory in my machine at that location at the moment.
- You can make arrays of any data type. `float fAverages[30]; double dResults[3];`
- If you don’t initialize elements in the the array the compiler will give you whatever happens to be in memory at the free space it allocated.
- A simpler way to initialize an array. `int arr[5] = {0, 1, 2, 3, 4};`
- This also works. `int arr[] = {0, 1, 2, 3, 4};` In which case the size of the array is initialized to the number of items in the `{}` initializer.
- `int arr[10] = {};` Is shorthand to initialize all elements in the array to `0`.
- The following `for` loop might be used to have the use input data into the array,

```
for (int i = 0; i < 5; ++i) {
    cout << "Enter int element at index " << i << ": ";
    cin >> arr[i];
}
```

Passing the Array Size

- As C++ does not know the size of the array, you must pass the array size as a separate parameter when passing an array to a function.
- If you compute an array's size *in the function that it is declared* then, yes, you get its size,

```
int arr[5] = {0, 1, 2, 3, 4};
cout << sizeof(arr) << endl;
```

- Here we get 4 bytes x 5 elements = 20 bytes in size.
- But when you're *in a function the array has been passed to*, you can't tell the size of the array. All you have is a pointer, and that's it. No indication of how big that array is. Now if you try to compute the array's size using `sizeof()`, you just get the size of the datatype of the array, or similar, depending on the compiler.

```
// Compiler translates arr's type to int*
void foo(int arr[], int size)
{
    // Prints 4 (the size of int) or 8, depending on compiler
    cout << sizeof(arr) << endl;
}
```

- Notice now we get 4, the size of the int datatype, or some compilers will give 8.
- When passing an array to a function the compiler translates the array type (`arr[]`) to a pointer pointing to the first element in the array, with a data type same as that of the array (in this case, an `int*`). See the section on Pointers for more information on this.
- So, if you declare a *local* array, you can get the size of the array. However, once you pass that array, all that's passed is the address. There's no information about the array size anymore.
- Notice that arrays are not really passed as whole arrays. In C++ *arrays are passed by reference only*.
- Note: if you specify size of a one-dimensional array when passing it as a parameter that size is ignored by the compiler. Typically one-dimensional array size is omitted as a parameter.

Printing and Searching an Array

- Here is a function to print an array in C++

```
void printArray(int arr[], int size)
{
      for ( int i = 0; i < size; ++i )
      {
            cout << arr[i] << ' ';
      }
      cout << endl;
}
```

- As you can see here, you *have* to pass in the size of the array to tell it how many elements to print. When passing an array to a function you're just passing a starting address, so for this to work a size must also be passed into the function.
- Try initializing and printing various arrays.

```
int arr1[6] = { 0, 1, 2, 3, 4, 5 };
int arr2[] = { 9, 8, 7, 6, 0, 1, 2, 3, 4, 5 };
int arr3[12] = {};
```

```
printArray(arr1, 5);
printArray(arr2, 10);
printArray(arr3, 12);
```

- Try modifying the printArray() function to print nicely using curly brackets and commas.

```
void printArrayNice(int arr[], int size)
{
	cout << "{";
	for ( int i = 0; i < size; i++ )
	{
		if (i > 0)
		{
			cout << ", ";
		}
		cout << arr[i];
	}
	cout << "}" << endl;
}
```

- Try making a function that searches for a value and returns the index of the value being searched for.

```
int searchArray(int arr[], int size, int value)
{
	for (int i = 0; i < size; ++i)
	{
		if (arr[i] == value)
		{
			return i;
		}
	}

	return -1;
}

int main()
{
	// array to search
	int arr[10] = { 3, 6, 2, 8, 11, -3, -7, 16, 21, 42 };

	int value;
	cout << "Value to search for: ";
	cin >> value;
	cout << endl;

	int index = searchArray(arr, 10, value);
	cout << “index = " << index << endl;
}
```

Functions cannot return whole Arrays

- C++ does not allow a function to return an Array.
- If you have a function `foo(int arr[])` you can't return `arr[]`.
- But the value of an array's name is implicitly convertible to the address of the array's first element. `arr` is implicitly convertible to the pointer `&arr[0]`, which, in this case, an integer pointer `int*`. See the section on Pointers to better understand this.
- So, you can return a pointer that points to the beginning of the array, e.g.,

```
// Compiler translates arr's type to int*
int* foo(int arr[], int size)
{
    // Prints 4 (the size of int) or 8, depending on compiler
    cout << sizeof(arr) << endl;
    return arr;
}
```

- In C++ *arrays are passed by reference only*.

Copying Arrays

- As an array is really just a reference, you can't copy whole arrays just by assigning the reference. e.g.,

```
int yourArr[5] = {0, 1, 2, 3, 4};
int myArr[5] = {3, 4, 5, 6, 7};
yourArr = myArr;    // illegal
```

- Outright assigning one array to another is *illegal* in C++.
- If you want to copy the array you need to make what is known as a "deep copy" of the array, iterating over every element in the source array and copying it to the target array, e.g.,

```
int main() {
    const int SIZE = 5;
    int yourArr[SIZE] = {0, 1, 2, 3, 4};
    int myArr[SIZE] = {3, 4, 5, 6, 7};
    printArray(yourArr, SIZE);

    for (int i = 0; i < SIZE; ++i) {
        yourArr[i] = myArr[i];
    }
    printArray(yourArr, SIZE);
    return 0;
}
```

- This is called a "deep copy" as you go deep and copying every element in the array to the other array.
- A "shallow copy" would be if you just copied the reference to the array, e.g.,

```
int* shallowCopy = myArr;
```

- A shallow copy isn't really a copy at all but is simply a copy of the reference to the array.

Comparing Arrays

- Similarly to copying arrays, you can't judge if two arrays have the same content by just looking at the reference. To compare two arrays you need to iterate through all elements in the array.
- Here is a function that compares two integer arrays.

```
bool compareArrays(int arr1[], int arr2[], int size)
{
    for (int i = 0; i < size; ++i)
    {
        if (arr1[i] != arr2[i])
        {
            return false;
        }
    }
    return true;
}

int main()
{
    const int SIZE = 5;
    int yourArr[SIZE] = {0, 1, 2, 3, 4};
    int myArr[SIZE] = {3, 4, 5, 6, 7};

    cout << compareArrays(yourArr, myArr, SIZE) << endl;
    return 0;
}
```

Array Limitations Summary

- Arrays *cannot be compared* using relational and equality operators—you must use a loop to compare two arrays.
- Arrays *cannot be assigned* to one another—an array name is effectively a pointer that is `const`.
- Arrays *don't know their own size*—a function that processes an array typically receives *both* the array's name and its size as arguments.
- Arrays *don't have automatic bounds checking*—you must ensure that array-access is within the array's bounds.

Two-Dimensional Arrays

- A two dimensional array is a collection of a fixed number of components arranged in rows and columns (two dimensions), with all components of the same type.
- Two-dimensional arrays are created similarly to one-dimensional arrays except are declared with two separate element numbers, e.g.

```
int arr2d[3][3];
```

- Nested `for` loops are useful for initializing 2d arrays.

```
int arr2d[3][3];
int counter = 0;

// initialize 2d array
for (int y = 0; y < 3; ++y)
{
      for (int x = 0; x < 3; ++x)
      {
            arr2d[x][y] = counter++;
      }
}
```

- Use double square brackets to access an element in the array.

```
cout << arr2d[1][1] << endl;
```

- A simpler way to initialize a 2d array.

```
int arr2d[][3] = { {0, 3, 6}, {1, 4, 7}, {2, 5, 8} };
```

- When you have a 1-D array as a parameter to a function, the size of the array is usually omitted. But C++ stores 2-D arrays in row oder form. To compute the address of an element correctly, the compiler needs to know where one row ends and the next begins.
- As such, when you have a two-dimensional array as a parameter to a function, there's no need to specify the number of rows, but you *must* specify the number of columns.
- Here is a function to print a 2d array in C++

```
void printArray2d(int arr2d[][3], int size)
{
      // print 2d array
      for (int y = 0; y < size; ++y)
      {
            for (int x = 0; x < size; ++x)
            {
                  cout << arr2d[x][y] << ' ';
            }
            cout << endl;
      }
}
```

Character Arrays (AKA "C Style" Strings) [C]

- A 1-D array of type `char` is effectively a string, and is treated as such in regular C.
- A `NULL` character, a `NULL` is the same as `'\0'`, and is used to terminate the sequence.

```
char name[6] = { 'B', 'r', 'u', 'c', 'e', '\0'};
cout << name << endl;
```

- Be sure to assign enough room for the NULL terminator.
- This may also be assigned like so,

```
char name[] = "Bruce";
```

- Try making a version of searchArray() that reads `char` arrays instead of `int`.

```
int searchChars(char name[], int size, char value)
{
      for (int i = 0; i < size; ++i)
      {
            if (name[i] == value)
            {
                  return i;
            }
      }

      return -1;
}
```

- The `NULL` terminator may be used to detect the end of the sequence. Try modifying the `searchChars()` function so it checks for the NULL rather than needing the `int size` variable passed in.

```
int searchCharsBetter(char name[], char value)
{
      int i = 0;
      char ch;

      do
      {
            ch = name[i];
            if (ch == value)
            {
                  return i;
            }
            ++i;
      }
      while (ch != '\0');

      return -1;
}
```

- The above is a good example of a `do while` loop.

Array and Vector Class Templates Mention

- C++11 introduced `Array` objects for dealing with fixed-size collections of of data items of the same type, and `Vector` objects for dealing with with collections that may grow and shrink dynamically at execution time.
- Both `Array` and `Vector` are C++ standard library class templates.
- To use `Array` and `Vector` you must include the `<array>` and `<vector>` headers, respectively.
- The `Array` and `Vector` class templates are very helpful when dealing with large collections in C++.
- For more information on `Array` and `Vector` class templates see http://www.cplusplus.com/reference/array/array/ and http://www.cplusplus.com/reference/vector/vector/