Arrays

Thomas Merkh
tmerkh55@gmail.com

Section 2, Week 9

May 26, 2020
1 Section

2 Next Time
Last time we introduced vectors. As a refresher:

- Vectors are containers which hold data of a specified type (data structures)
- There are many pre-built methods for determining the size of the vector, modifying its values, and iterating over it
- Vectors are in some sense a nicer version of arrays, which is today’s topic
An array is a series of elements of the same type placed in contiguous memory locations. Essentially, it is a vector that cannot change sizes, and has different methods associated with it.

- Just like vectors, since data is stored contiguously, we can access all of the values by incrementing a single index.
- As always, arrays must be declared before using it, and this usually looks like
  type name [\# of elements];
- One may also omit the number of elements and let the compiler automatically determine the size (which cannot change after initialization). For example,
  int name [] = {1,2,3,4};
Let’s consider several valid C++ statements:

```cpp
int foo [5];
int foo [5] = { 2,3,4,7,8 };
int bar [5] = { 10, 20, 30 };
int foo [] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```
Let’s consider several valid C++ statements:

```cpp
int foo [5];
int foo [5] = { 2,3,4,7,8 };  
int bar [5] = { 10, 20, 30 }; 
int foo [] = { 10, 20, 30 }; 
int foo[] { 10, 20, 30 };
```

In order, these perform:

- Initializing an integer array of size 5
- Initializing an integer array of size 5 with certain values
- The same as previously, except only the first 3 values will be non-default values
- Initializing an integer array of size 3
- Exactly equivalent to the previous one, with different notation
Before moving on to multi-dimensional arrays, note that we access elements of the array the same as vectors. For example,

\[ x = \text{foo}[2]; //access the 3rd value. \]
Multidimensional Arrays

Before moving on to multi-dimensional arrays, note that we access elements of the array the same as vectors. For example,

\[ x = \text{foo}[2]; \] //access the 3rd value.

Arrays need not be one dimensional. We can create arrays of arbitrary dimension by including extra \([]\)'s. For example, consider forming a 5 by 5 array:

\[ \text{int bythearray}[5][5]; \]

- The secret is: the computer actually stores this continuously as a block of 25 integers, no different than \[ \text{int alt}[25];\]
- These serve as an abstraction for programmers which can simplify the placement of the elements
- The compiler “remembers” the depth of each dimension, but this is reflected in the actual computer memory
To accept an array as a parameter for a function, it's recommended using empty brackets to omit the actual size of the array. This improves the generalization ability of the function written. This looks like

```c
void procedure (int arg[])
```

Then passing an array into `procedure` looks like

```c
int mee [10];
procedure(mee);
```

(success - no errors here)
Some people didn't care for remembering the differences in how arrays and vectors are declared. They formed the array library\(^1\). Consider:

```cpp
#include <iostream>
#include <array>
using namespace std;

int main()
{
    array<int,3> mee {10,20,30};

    for (int i = 0; i < mee.size(); ++i)
    {
        mee[i]++;
    }

    for (int elem : mee)
    {
        cout << elem << "\n";
    }
}
```

For some, this is easier to remember than the bracket notation since brackets are also used to access data containers.

\(^1\)I don’t actual know who made this library or why, but its standard.
Drawbacks and What Will Be Annoying

Several things are worth noting about arrays and vectors:

- Arrays exist in both the C and C++ language
- Vectors *only* exist in C++
- Arrays do not have any guard against going “out of bounds”
- Errors will not be thrown, but unexpected results will occur and be difficult to locate
Example of Not Bounds Checking

What do you think will print to the console?

```cpp
#include <iostream>
using namespace std;

int main()
{
    int arr[2];
    cout << arr[3] << ' ' ' ';
    cout << arr[-2] << ' ' ' ';
    return 0;
}
```

If you said random numbers, like 2008101287 4195777, then you'd be correct. Unintended consequences, so be careful with bounds checking!

Thomas Merkh tmerkh55@gmail.com
May 26, 2020 11 / 15
What do you think will print to the console?

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[2];
    cout << arr[3] << ' ' ' ';
    cout << arr[-2] << ' ' ' ';
    return 0;
}
```

If you said random numbers, like 2008101287 4195777, then you’d be correct. Unintended consequences, so be careful with bounds checking!
Arrays, References, and Pointers

Consider the following code:

```c++
int A[] = { 1,2,3,4,5 };
cout << A << endl;
cout << &A << endl;
cout << A[0] << endl;

int *p = A;
cout << *p << endl;
cout << p + 1 << endl;
cout << *(p + 1) << endl;
```

The output of this code is:

```
0133FA98
0133FA98
1
1
0133FA9C
2
```
Consider the following code:

```cpp
int A[] = { 1, 2, 3, 4, 5 };
cout << A << endl;
cout << &A << endl;
cout << A[0] << endl;

int *p = A;
cout << *p << endl;
cout << p + 1 << endl;
cout << *(p + 1) << endl;
```

The output of this code is:

```
0133FA98
0133FA98
1
1
0133FA9C
2
```
What is going on here?

- The name of an array doesn’t exist separately in memory anywhere.
- The name of an array is basically a reference to the memory address of the array’s first element.
- Printing $A$ and $&A$ will output the same address.
- A *pointer*, or an object which *points* to a location in memory, can be set to point to $A$.
- Dereferencing the pointer, i.e. $*p$, will fetch the value of $A[0]$.
- Printing $p + 1$ will output the address of $A[2]$ to the console.
Section

Next Time
Next time we will cover our last topic, streams.

Questions?