Introduction to Software Development – CS 6010

Lecture 10 – Pointers, C Arrays, and Command Line Args

Master of Software Development (MSD) Program

Varun Shankar

Fall 2022

Miscellaneous

- Monday is a Holiday!

Lecture 10 – Pointers, C Arrays, And Command Line Arguments

- Topics
  - Arrays
  - Pointers
  - Command Line Arguments
  - Code Review – Rainfall Analysis

Built in Arrays (C Arrays)

- To create a built-in array, we usually need to know its size at compile time (there's an exception).
- `type arrayName[ size ];`
- int numbers[10]; // Create space for 10 integers
- numbers[0] = 99; // Assign the 1st (0th?) position of the array
- cout << numbers[3]; // Display the 4th number in the array
- We index into the array just like with a vector or a string using [ ]. But none of the helper functions (methods) are available (e.g.: .size())
- We cannot change the size of the array after it is created
- Note, if your data is a fixed size, using a std::array<type, size> variable is better as it works just like a fixed-size vector.
  - And has methods…

Weirdness With Functions

- What might you expect this code to do:

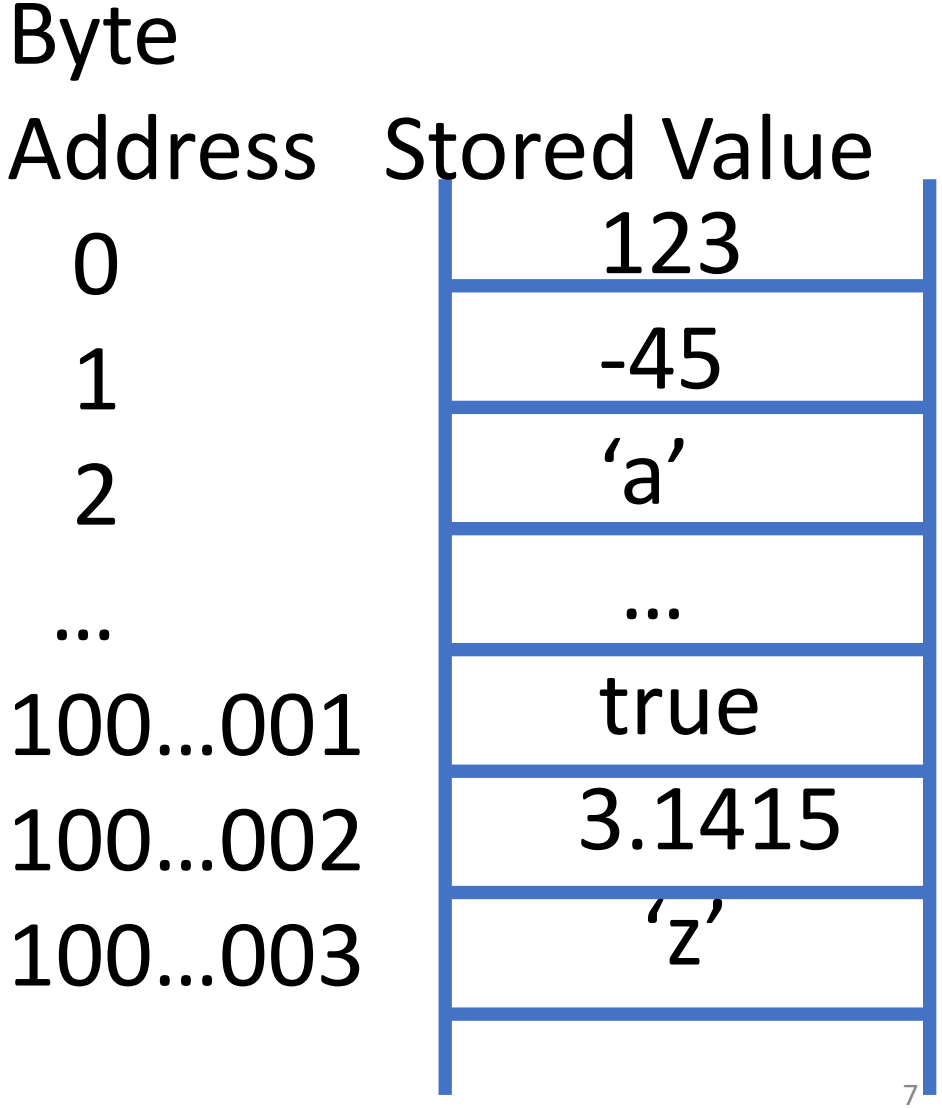
```
void myFunction( int array[3] ) {          // Looks like pass by value (copy).
        array[0] = 0;
        array[1] = 1;
        array[2] = 2;
}
int main() {
        int a[3] = { 5, 5, 5 };
        myFunction( a );                   // Prints 1 2 3??? Why is this?
        cout << a[0] << a[1] << a[2];      // Because arrays are pointers…
}
```

Pointers

- Pointers are sort of like references in that they give a new name to an existing value.
- A pointer is a data type that stores an “arrow” that “points to” another value. In reality it is actually an index into the giant RAM array.
- Pointer syntax can be a bit weird, and the “punctuation” has different meanings depending on context!

Pointers –> Index into Memory

- Memory is just a giant array of bytes.
- A pointer *points* to (addresses) a byte.
- Think of it as an arrow that points to a where a variable lives.
- int * pInt = 0; // Assign the *address* 0
- cout << *pInt; // 123 – What if this is an array of data?
- pInt = 1; // pInt is now pointing at address 1
- cout << *pInt; // -45
- pInt = 100000001; // Real addrs look more like this.
- cout << *pInt; // true
- Note: Assigning arbitrary addresses to your pointer (like above) is a really bad idea.



Returning Invalid Memory

```
int * getNumbers() {
    int nums[3] = { 1, 2, 3 };
    return nums;
}

int main() {
        int * x =
getNumbers();
        someOtherFunction()
;
}
```

- What *type* of data does the getNumbers() function return?
  - Pointer to an Integer
- What does "return nums" actually return?
  - Returns a pointer to the address where *nums* is.
- Where is *nums* allocated? (Where does it exist?)
  - On the call stack for getNumbers()
- What value did *x* receive?
  - The address of *nums*
- When getNumbers() returned, what happened to its callstack?
  - It went away…
- So what is *x* pointing to now?
  - Invalid memory

Quick Reminder for & and *

- & means at least 3 things:
  - reference
  - Boolean And (Actually a single & is bitwise and)
  - Give me the address of a variable.
- * has at least 3 meanings:
  - multiplication
  - create a pointer
  - dereference a pointer (follow the arrow)

Pointer Syntax

- C++ uses the * to designate a pointer to a type of data (during declaration), and as a way to *dereference* the pointer when using it.
- int * pointerToInt; // pointerToInt is not an integer, it is an “arrow” pointing to an integer
- int x = 5; // Created a normal integer variable
- pointerToInt = &x; // & means *address of*. pointerToInt now points to the variable x (or pedantically, to the address where x is stored.
- int j = *pointerToInt; // When *using* a pointer, the * means *dereference* it. In other words, use the value it is pointing at (or perhaps “follow the arrow”). In this case, j now has the value 5.
- int k = 7;
- *pointerToInt = k; // We can use *dereferencing* on the left-hand side of the equation too. In this case, the value in the variable *k* is put into the variable *x* (because pointerToInt points to *x*.)

What's the Point(er)

- Originally used to do similar things to what we are doing with references
  - It is best to use references as they are somewhat safer than pointers.
  - Pointers allow for manipulation of memory in any way you wish. This can create code that is hard to follow and bug prone.
  - Many languages don't even provide a pointer datatype.
- While a reference is fixed with respect to what it refers (points) to, a pointer can be changed to refer to different pieces of data over time.
- Pointers are useful for working with built-in arrays.
- A pointer's value can be set to nullptr (This used to be NULL, but NULL is deprecated – don't use it). A value of *nullptr* means a pointer is not pointing at anything.
- These abilities are often necessary for low level code. (e.g.: std::vector uses pointers internally)

Pointers and Strings

- Before C++ had std::string, text was presented as character arrays (which technically is a character pointer.
  - char * myString; // Pointer to a char (actually pointer to 1 or more characters)
  - char myString[ ]; // Char array (actually a pointer to 1 or more characters)

- A “pointer to a (single) character” can also be used as a pointer to the first character of many.

- Don’t use char * strings unless you have a very good reason for it.

Pointers and Arrays

- In C, arrays were passed to functions using pointers
- void findSum(vector<int> & numbers); // Was written as…
- void findSum(int * numbers, int size); // It was necessary to keep track of the size of the array in a separate variable.
  - Remember, in this case *numbers* points at the first integer in the array…
  - …But will allow us to work our way down through the array.
- Note, we can use the same syntax to access the data: [ ]

C Style Swap

- Before references, we had pointers.
- How to write swap() with pointers? Let's take a look.

```
void swap( int * a, int * b ){
        int temp = *a;
        *a = *b;
        *b = temp;
}
int main() {
        int x = 1; int y = 2;
        swap( &x, &y );
}
```

Pointer Arithmetic

- Given:

```
int i[ 10 ];

int * pInt = & i; // Note pInt is short for pointer to int
```

- What happens if I write:

```
pInt = pInt + 1;
```

- This does not modify the value in *i*, it modifies the “arrow”
- Basically, the compiler assumes that the pointer is pointing to an element in an array. Adding one to the pointer actually makes it point to the next element in the array.
  - As a side note, this is why we count from 0. Array syntax is shorthand for the following:
  - array[0] == *(array + 0)
  - array[1] == *(array + 1)
  - array[2] == *(array + 2)
- Most of the time you shouldn’t write code to do this… but it is good to know that this exists.

Print Array

```
// What does the printArray function declaration look like?
void printArray( int arr[], int size ){
        for( int i = 0; i < size; i++ ) {
                // What does *arr mean? [Above in the declaration?   Where when it is used?]
                        // Above: Specifies arr is a pointer to an int.
                        // When used in code, it means dereference (i.e.: follow the pointer – get
the value
                                                // the pointer is pointing at.)
                // What does arr + 1 mean?
                        // Refer to the next location in memory
                // What about *(arr + 1)
                        // Go to the 2nd position of the array arr and get its value.
                cout << *(arr + i); // This is pointer Arithmetic
                // There is a shorthand for *(arr + i) – what is it?
                // arr[ I ]
        }
}
```

```
int main() {
    int array[3] = { 0, 2, 4 };
    // Must provide the size of the array
    printArray( array, 3 );
}
```

Command Line Arguments

- int main(int argc, char** argv) // pointer to a pointer (????)
- int main(int argc, char* argv[]) // Array of strings
- argc == argument count
- argv == argument vector
- string firstArg = argv[0];
- Where do these commands come from?
- Where have we seen args passed to a program?
  - Running programs on the command line.

Some Reasons This is Important

- Remember seeing:

int main(int argc, char**argv) // What are these parameters?

- They are used to provide the command line arguments to the program.
- argv is a pointer to the first *c-string* in an array
- argc is the number of strings in the array
- If main was re-written in modern C++, it would look like:

int main(vector<string> args)

- We can treat argv like an array of strings – ie, use [ ] to index into it.
  - But remember, we can't use *.size()* or other methods. (We use *argc* instead.)

More Pointers To Come… But Later

- For now we need to know the basic idea behind pointers
- Later we will dig deeper into how memory is actually organized and we will be forced to use pointers.

Today’s Assignment(s)

- Code Reviews – Deck of cards and Poker
- Lab – Pointers and Arrays
- Homework – Book Analyzer