Dynamically Allocated Arrays

CS 211

Adam Sweeney

November 3, 2017

Wichita State University, EECS

Introduction

- We have learned about the differences between the stack and the heap
- We saw how to allocate dynamic variables from the heap
- But a dynamic `int` doesn't seem all that useful
- We'll discuss a much better utilization of heap memory today

Agenda

- Section 9.2
- Arrays. Pointers in disguise
- Declaring and using dynamic arrays
- Pointer Arithmetic

Arrays. Pointers in Disguise

Arrays. Pointers in Disguise

- Recall that arrays are stored in memory by grabbing a large chunk of continuous memory that can hold all of its elements
- It was briefly touched on that the successive elements are accessed by doing some memory manipulation operations, based on the knowledge that we know how many bytes the data type occupies
- Also recall that when an array is passed to a function, we said that it was "like a reference," but not that it was specifically passed by reference
- All of this should add up to indicate that array variables behave a little oddly
- It all makes sense if we apply our knowledge of pointers
- An array variable is actually a pointer, which means that its value is just a memory address

What Does an Array Point to?

- The address held by an array variable is the address of the first element of the array
- Because the concept of arrays is more important than the fact that it's a pointer, it is abstracted away for us
- But the knowledge that arrays are pointers is important for understanding how to request an array from the heap
- `https://repl.it/N5n2/latest/426347`

A Distinction

- While it is true that an array variable is a pointer, there is one thing that separates it from a "real" pointer
- Once given an address, an array variable cannot be reassigned to point to something else
- A regular pointer can be reassigned to point to different objects (so long as they are of the correct data type)
- An array variable is a pointer, but it is a special pointer

Declaring and Using Dynamic Arrays

Declaring and Using Dynamic Arrays

```
int size;
cout << "How many elements? ";
int arr[size];
```

- You might be surprised to learn that this code works
- After all, we have gone on and on about how an array's size needs to be known at compile time
- What is happening here is that C++ has inherited what is called a Variable Length Array (VLA) from C99 (A C language specification, ratified in 1999)
- These VLA's allow for arrays to be declared at run-time, on the stack
- While on the surface it seems like a win-win scenario, there is quite a lot of potential for error

VLA

- They are limited in use-case, they have become optional in C2011, and they can make it very easy to overflow the stack
- We have spoken about the heap being limited, but so is the stack
  - Moreso than the heap, and if we think about it, that makes sense
  - The stack is supposed to a highly organized memory space with no wiggle room
- So, we will avoid using VLA's and instead dynamically allocate our arrays

Review: Why Do We Care About This Again?

- When we need an array, it often difficult or impossible to know how large our array needs to be
- Many times, the size of the array will vary from execution to execution
- When statically declared, we are making a guess at the size that we hope will cover most of our needs
- I am calling this a Goldilocks problem
- More often than not, we will declare arrays that are too big, therefore wasting memory but giving us some breathing room
- However, it is still possible that our declared size is too small, thereby limiting our program's functionality
- Wouldn't it be nice if we could get an array that is sized just right every time?

Declaring a Dynamic Array

- We know how to dynamically allocate a single variable, using `new`
- An array is not so different
- The key is that we still have to specify how many elements we need (or rather, how much total memory we need)

```
int size;
cout << "How many readings will be recorded? ";
cin >> size;

double *readings = new double[size];
```

- https://repl.it/N5ns/latest/426347

An Explanation

```
int size;
cout << "How many readings will be recorded? ";
cin >> size;

double *readings = new double[size];
```

- We should be familiar with most of what we see above
- Recall that when the heap provides the necessary dynamic memory, it is not named
- We are only given the starting address, and we can only access this memory through the pointer
- We are used to seeing an array declared with the square brackets next to the name
- Again, heap memory is never named, so we place the square brackets next to the data type

Using a Dynamically Allocated Array

- The short and long of it is that once it is declared, you can use a dynamically allocated array exactly as you would use a statically allocated array
- There is no need to dereference the pointer, or anything like that

```
double *readings = new double[size];

for (int i = 0; i < size; i++)
    readings[i] = getReading();
```

- In the example above, the only reason we know the array has been dynamically allocated is because of the declaration
- The usage is identical to what we've been doing all along

If we use `new`...

- Just like heap allocated memory of single variables, heap allocated arrays are using borrowed memory
- We need to make sure to return it when we are done
- The syntax is only a little bit different

```
#include <string>

string *arr = new string[size];
// Do stuff with array
delete [] arr;
arr = NULL;
```

- The syntax is nearly identical, with the exception of the [ ]
- Take note of their location in the command

But Why, Though?

- Remember that arrays are pointers in disguise
- However, the concept of an array is more important than the fact that they are pointers
- So, the pointer nature of array variables was abstracted away to allow the more important concept to take center stage
- But now that we've seen that we can use an array exactly the same whether it was statically or dynamically declared, the hope is that we are now a bit curious about what's happening behind the scenes

Pointer Arithmetic

Pointer Arithmetic

- What's going on behind the scenes is known as pointer arithmetic
- It is akin to regular arithmetic, if we remember what pointers are
- If we are dealing with regular numbers, `56 + 1 = 57`
- But what about this?

```
double x, *dptr;
dptr = &x;     // Let's say that the address of x = 56

dptr++;        // What is the new value of dptr?
```

- HINT: It's not 57

Pointer Arithmetic (cont.)

```
double x, *dptr;
dptr = &x;     // Let's say that the address of x = 56

dptr++;        // What is the new value of dptr?
```

- First recall what a pointer is
- It is the address of the first byte of the data type it is pointing to
- Most data types require more than a single byte
- In the case of a `double`, 8 bytes are required
- If incrementing the value of a pointer to a double only incremented the byte value by one, you would lose the first byte of real data and gain a single byte of garbage data

Pointer Arithmetic (cont.)

```
double x, *dptr;
dptr = &x;     // Let's say that the address of x = 56

dptr++;        // What is the new value of dptr?
```

- What happens instead is that the pointer is incremented to the next possible memory address that could hold another double
- In the case of the example above, `dptr` would be equal to 64, as that is first available address that could hold another double
- Again, a `double` is 8 bytes. So, starting with address 56, 8 bytes are required to store the `double`
- The pointer cannot point to bytes 57 - 63 because it knows that the `double` it is pointing to occupies that space

Pointer Arithmetic (cont.)

- To reiterate, when a pointer is incremented, the memory address does not simply go up by one byte
- The pointer will point to what is the earliest available address, given the data type it is pointing to, of where the next variable can be stored
- `https://repl.it/N5zb/latest/426347`