CSCI 1300

Intro to Computing

Gabe Johnson

Lecture 34 Apr 10, 2013

Pointers Part 2

Lecture Goals

1. HW Issues
2. Recap Pointer Syntax
3. Recap Pointers in Memory
4. C++ Arrays
5. Heap vs. Stack
6. Structs

Upcoming Homework Assignment

HW #8 **Due: Friday, Apr 12**

Pointers, Arrays, Structs

This HW is mostly about pointers. It is tricky. The trickiest one yet. You can get 5 points extra credit, since it is scored 15 out of 10.

Start early.

Make sure you get it to compile and run *before* you start messing around. If you don't you are likely to have so many problems with your file that you won't be able to fix things.

HW Issues

1. You need all the files from GitHub. They are:

Makefile
UTFramework.cpp
UTFramework.h
pointers.cpp ←
pointers.h
pointers_driver.cpp

2. The pointers.cpp file is the only one you will *need* to edit, and the only one you should turn in. Feel free to edit the driver on your side to debug, though.

HW Issues

3. To build the HW you cd to the directory where all those files are, and type ‘make’.

If you get the following error:

```
cc1plus: error: unrecognized command line option "-std=c++11"
```

... it means the compiler doesn’t know about the new C++ standard. Not a problem. Next slide to fix...

HW Issues

```
cc1plus: error: unrecognized command line option "-std=c++11"
```

To fix this, edit the Makefile and go to the line that says -std=c++11 and remove *just that part of the line*. In other words, change this:

CXXFLAGS += -g -Wall -Wextra -std=c++11

To this:

CXXFLAGS += -g -Wall -Wextra

Pointer Syntax

```
values
pointers
addresses
```

Pointer Syntax

```
int x=42;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

Pointer Syntax

```
int x=42;
int* x_ptr;
```

`x` is an integer that has the *value* 42. It is declared with the data type `int`, and stores the value 42.

`x_ptr` is a *pointer to an int*. It is declared with the data type `int*`, but its value is unspecified.

Pointer Syntax

```
x_ptr = &x;
```

We know the value of an int is some number. The value of `x` is 42.

The *value* of a *pointer variable* is an *address.*

In the above code, we take the *address of* ***x***, and store it in `x_ptr`.

Pointer Syntax

`x` ← This is our int x. Value is 42.

`&x` ← Memory address of x.

`x_ptr` ← int pointer. Value is the address of int x.

`*x_ptr` ← Dereference an int pointer. Value is 42.

Pointers in Memory

```
int x = 42; // given memory cell 23
int* x_ptr = &x; // given memory cell 30
int* other = &x; // cell 18
```

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										

var	type	addr
x	int	23
x_ptr	int*	30
other	int*	18

Pointers in Memory

Be aware that this is a simplification. Variables can take up several contiguous cells in memory; we know how many they take up based on their data type.

	0	1	2	3	4	5	6	7	8	9
0										
10									23	
20				42						
30	23									
40										

Also, this is a tiny memory. A real computer has **billions** of bytes of memory that we can store things in.

C++ Arrays

```
int some_data[123];
```

An *Array* in C++ has (sort of) the same purpose as Java's arrays and Python's lists. In C++ it is a contiguous stretch of memory, which is divided into elements that contain some value.

In this case, some_data is an array of integers, and there are 123 elements reserved.

C++ Arrays

```
... more above
52: 0
53: 0
54: 0
55: 0
56: -1608234075
57: 1175298596
58: 50716776
59: 1
60: 1
61: 0
... more below
```

If we were to print out each element of some_data, we might see something like this.

Declaring an array (or anything else in C++ for that matter) does *not* initialize that memory. If we read from the variable without setting a value, we might get *garbage*.

Alternate initializers

```
int stack_arr[] = { 5, 6, 20, 3 };

int* dyna_arr = new int[6];
```

Here are two other ways of declaring and initializing arrays. The first gives the values of the array, and the length is derived by how many values are there.

The second is tricky and is discussed later.

C++ Arrays

Reading and writing to arrays should be familiar syntax. Here we initialize each cell to 0, 1, 2, 3, and so on.

```
int ret[num];
for (int i=0; i < num; i++) {
  ret[i] = i;
}
```

C++ Arrays

Reading from arrays uses the same syntax with the square brackets. Note that uninitialized arrays will have garbage in them when we read.

```
for (int i=0; i < n; i++) {
  cout << i << ": "
       << ret[i] << endl;
}
```

C++ Arrays as pointers

Earlier I mentioned this one:

```
int* dyna_arr = new int[6];
```

... and pointed out that it was tricky. The *value* of an array (remember how variables *all contain values*?) is actually the address of some spot in memory.

The statement ‘new int[6]’ creates an array and returns a pointer to that array.

C++ Arrays as pointers

```
int numbers[] = { 5, 6, 20, 3 };
print_array(numbers, 4); // prints 5, 6, 20, 3
int* weird = numbers; // note, NOT &numbers
print_array(weird, 4); // prints 5, 6, 20, 3
```

Since the *value* of an array variable is the address of its first element, we can read that array without the square brackets to get the address where the array data lives.

Heap vs. Stack

When we ask for memory to store variables, the operating system will allocate some spots in memory for us to use. There's two kinds of memory you need to be aware of: *stack* and *heap* memory.

I'll start with *stack* memory since it is related to *scope*, another idea we've talked about before.

Heap vs. Stack

When we declare a variable inside some block of code, that variable can be used inside that block. This is that variable's *scope*. But it will be unavailable once we leave that scope.

Scope

```
int get_sum_times_two(int a, int b) {
  int sum = a + b;
  int twice = sum * 2;
  return twice;
}
```

```
int main() {
  int ten = get_sum_times_two(2, 3);
  cout << "sum: " << sum << endl;
}
```

foo.cpp:19:22: error: use of undeclared identifier 'sum'
 cout << "sum: " << sum << endl;

Scope

```
int get_sum_times_two(int a, int b) {
  int sum = a + b;
  int twice = sum * 2;
  return twice;
}
```

Inside this function we declare two variables: **`sum`** and **`twice`**. Once the **`get_sum_times_two`** function completes, the computer no longer needs them. So the memory that we were using to store them is recycled.

Scope

```
int get_sum_times_two(int a, int b) {
   int sum = a + b;
   int twice = sum * 2;
   return twice;
}
```

The variables we declare here are *stack variables*. These are normal variables that we expect to use only temporarily. The operating system is doing us a service by cleaning up our mess. *But what if we wanted them to stay in memory afterwards?*

Heap vs. Stack

The other kind of variable is one that lives in the *heap* section of memory. These are persistent—the operating system will not recycle that memory until we explicitly tell it to.

(And, if you forget to do this, you end up with memory leaks and you gradually run out of free memory.)

Heap vs. Stack

To create a heap variable in C++, use the ‘new’ keyword. This is not in plain C (though C has its own way of doing this, and we won’t go into it).

```
int* dyna_arr = new int[6];
```

We use ‘new’ to allocate memory. This process always returns a *pointer* to the memory it allocated. Here, `dyna_arr` is a pointer to the block of memory holding our array of 6 integers.

Heap vs. Stack

```
int* dyna_arr = new int[6];
```

When the dyna_arr variable is no longer in scope:

- The variable dyna_arr is no longer valid, *but:*
- The memory it points to is still safe!

Heap vs. Stack

Check out the file:

cs1300/code/cpp/
stack_vs_heap_array_pointers.cpp

This demonstrates how you can really screw things up by using a stack variable when you're going to need to refer to its memory later on with a pointer.

Heap vs. Stack

```
int* make_stack_array(int num) {
  int ret[num]; // stack variable ‘ret’
  for (int i=0; i < num; i++) {
    ret[i] = i;
  }
  return ret; // ret’s memory will be recycled
}

int* make_dynamic_array(int num) {
  int* ret = new int[num]; // heap variable ‘ret’
  for (int i=0; i < num; i++) {
    ret[i] = i;
  }
  return ret; // ret’s memory will be left alone
}
```

Heap vs. Stack

```
int main() {
  int* stack_arr = make_stack_array(10);
  cout << "make_stack_array(10):" << endl;
  print_array(stack_arr, 10);

  int* dynamic_arr = make_dynamic_array(10);
  cout << "make_dynamic_array(10):" << endl;
  print_array(dynamic_arr, 10);
}
```

stack_arr contains garbage at this point.
dynamic_arr contains the data we want.

Structs

```
struct things{
   int x;
   int y;
   int add_stuff() {
      return x + y;
   }
};
```

Be sure to put the semicolon here!

One more thing you need to know about before you can do the HW: *structs*.

They are sort of like classes in Java and in Python. They can have member variables and functions.

Structs

```
struct things{
  int x;
  int y;
  int add_stuff() {
    return x + y;
  }
};
int main() {
  things* t = new things;
  t->x = 30;
  t->y= 8;
  int z = t->add_stuff();
  cout << t->x << " + " << t->y
       << " = " << z << endl;
}
```

To create an instance of the things structure, you can either make one on the heap (which is done in this example), or on the stack.

Structs

```
struct things{
  int x;
  int y;
  int add_stuff() {
    return x + y;
  }
};
int main() {
  things* t = new things;
  t->x = 30;
  t->y= 8;
  int z = t->add_stuff();
  cout << t->x << " + " << t->y
       << " = " << z << endl;
}
```

Notice we have to use the arrow operator to access members. This is true for all pointers to structs or objects. Rule of thumb is: if it is a pointer, we have to use the arrow to access things.

Structs

```
struct things{
  int x;
  int y;
  int add_stuff() {
    return x + y;
  }
};
int main() {
  things t;
  t.x = 17;
  t.y = 3;
  int z = t.add_stuff();
  cout << t.x << " + " << t.y
       << " = " << z << endl;
}
```

Here we make a things instance on the stack. The syntax is a little cleaner, but now the memory associated with it will go away. This is often something we want and depend on.