Lecture 11 Notes

Derek Harter

2015-04-07

1 First Session (11 - 11:40)

1.1 Refresher on memory concepts

- Memory consists of a sequence of numbered addresses (each holding 1 byte of information)
- Addresses can be accessed randomly (Random Access Memory). It takes the same amount of time to access any particular address in primary memory (O(1), or constant time).
- When you declare a variable in C, you allocate memory at some particular address in memory to hold the value you assign to the variable.
- The type of the variable determines the size, or number of bytes of memory, needed to hold the variable.
- The type also determines the interpretation of the bit pattern in memory of those bytes.

1.2 Pointers Introduction

- A declared variable actually is associated with a memory address. In C compiler, a look up table translates every variable name reference to the associated address when creating the machine code.
- Pointer variables contain **memory addresses** as their values.
- **Pointer variables contain MEMORY ADDRESSES as their values.**
- We use pointers indirectly. **indirect referencing**

- We dereference a pointer, in order to access the value it is pointing too.
- Confusing for some people, a pointer variable holds a memory address, but we still associate a type with a pointer variable.
  - Because, when we dereference a pointer variable, we need to know how to interpret the bit pattern in memory that it is pointing too.

1.3 Pointer Variables

- Pointers, like any other variable, must be declared before they can be used.

```
int count;
int* countPtr;

double x;
double* xPtr;
```

- Do not declare multiple pointer variables using single line declaration. Always declare pointers on their own line.
- It is good practice to use Ptr on name of variable, to indicate they are pointer variables.
- Please follow our class convention, and put the * next to type, to indicate it is a pointer to the indicated type.
- Pointers must be initialized before they can be used (like any other variable).
  - It is a common, and **very bad not to ever be done** error to try and dereference an uninitialized pointer. Bad things will happen, don't do it.
- Pointers can be initialized to 0 or NULL, but **never dereference a null pointer**.

2 Second Session (11:45 - 12:30)

2.1 Pointer Operators: Address of Operator

- How do we initialize a pointer?

- Well obviously, pointers hold memory addresses, so we have to initialize with a memory address.
- How do we found out memory address of interest?
- use `&` operator. Read as the **address of** operator.

```
int y = 5;
int* yPtr;

yPtr = &y; // assign address of y to yPtr
```

- As we just did to initialize y to an integer value when declaraing, we can also initialize a pointer variable in declaration, thus:

```
int y = 5;
int* yPtr = &y;
```

2.2 Pointer Operators: dereference operator

- Ok we can declare a pointer, and point it at something. What use is that?
- When a pointer is validly referencing memory, we can derefrence it and use the dereferenced location anywhere we could use a simple variable of the given type.
- Derefernce to read value out and display it. Operator `*` is the **dereference** operator.

```
cout << *yPtr << endl;
```

- Can dereference to read value out, and use it in any arbitray expression

```
int x = 3;
x = *yPtr + 5;
```

- Dereference to write value to reference memory location

```
*yPtr = 9; // assign value 9 to location referenced in yPtr (e.g. to variable y)
cout << *yPtr << endl;
cout << y << endl;
```

- Can also assign value using IOstream input

```
cin >> *yPtr;
```

2.3 Memory Addresses, again

- Lets look at the actual values in one of our pointer variables

```
int a;     // a is an integer
int* aPtr; // aPtr will point to the a integer

a = 7;
*aPtr = &a;

cout << "The address of a is " << aPtr << endl;
```

- Also, remember that `&` and `*` (address of and dereference operators) are inverses of one another.

2.4 sizeof operator

- We have seen examples of this. Can be used to determine number of bytes a variable is using.
- Can be used for arrays.
- In C and typed languages, the type of a variable is tied to its size, each particular type needs some number of bytes of memory to represent values of that type.
- The sizeof a pointer variable will depend on the size used to represent and hold memory addresses on the machine you are running. But typically it is 4 bytes (32 bits) on 32-bit architectures or 8 bytes (64 bits) on 64-bit architecture.

3 Third Session (12:40 - 1:40)

3.1 Pass by Reference

- Pass by reference used not to be built in to C language.
- But you can pass by reference by passing pointer parameters.
- They are equivalent, function parameters declared as pass by reference are set to pass in a pointer (a memory reference). Only difference is that a reference variable in a function is dereferenced for you behind the scenes.

3.2 Multiple Indirection

- Pointer dereferences can be chained together, to create multiple levels of reference/dereference.
- This can be useful in many cases, e.g. an array of pointers to structures, that have pointers to link to other structures. We will see more of this when we look at data structures using pointers to create links between items.
- As an example, lets use 2 levels of reference.

```
int x;
int* xPtr = &x;
int** xPtrPtr = &xPtr;

**xPtrPtr = 42;
cout << x;
```

3.3 Pointers to Structs and Struct Members

- Structs have a special operator used to reference members of a struct pointed to by a pointer reference.
- This was so commonly done, the operator `->` was developed. Read as **pointer to member**.

```
struct Trial
{
   string name;
   string gender;
   float reactionTime; // ms
   int numberOfPresses;
}; // don't forget the semicolon

Trial t;
Trial* tPtr;

tPtr = &t;
t->name = "Jane Student";
t->gender = "Female";
t->reactionTime = 42;
```