ECE 264: Advanced C Programming

Lecture Notes 6/13/19

1 Topics

1. Short demo of using Putty to connect to ecegrid machine
2. Wrapup of strings
3. Arrays, Typedef, Structures, Addresses (brief intro)

2 Putty and ThinLinc

You can login to a remote machine from your laptop/desktop and run programs on the remote machine. One option provides you just a terminal (Putty) so that you can execute your commands on the remote machine, while the other provides a richer experience by allowing you to work with a graphical user interface(GUI) based system (ThinLinc).

https://www.chiark.greenend.org.uk/ sgtatham/putty/

https://en.wikipedia.org/wiki/ThinLinc

If you need just a terminal (cannot see a Desktop screen of the remote machine upon login) use Putty. If you like cut-copy-paste operations (on anything on your laptop) using a mouseuse then use ThinLinc.

If you are using a MAC or Linux based system, type on your terminal (on MAC and Linux the terminal is already installed for you).

`ssh <username>@<remote machine FQDN/IP-address>`

remote machine FQDN would look like wabash.purdue.edu.

If you are using Windows, you would use Putty and type in the details in the window provided upon clicking Putty executable.

3 Wrapup of Strings

We wrapped-up our discussion of strings by first comparing and contrasting the properties of strings initialized in different ways:

`char s[]="ECE"` and `char* s="ECE"`

We also saw how strings are printed and how to obtain the length of a string.

4 Arrays, Typedef, Structures, Addresses

Arrays while strings in C are arrays of `char` types, we can have arrays of any type in C. There are multiple ways we can create arrays:

1. using initializer lists to declare and initialize an array — `int arr[]=1,2,3;`. Note that array size is not required here.

2. declaring an array using an array size — `int arr[3];`

Array elements are accessed using indices with array names. E.g. arr[0] accesses the first element within the array, arr[1] accesses the second element and so on.

Literals just as we saw string literals in previous class, we can have literals or constants of other types. 'E', 1234, 128.2 are examples of literals. 078 is not a valid literal in C. This is because any number preceeded by a zero is assumed to be an Octal number (base-8). In Octal number there is no digit 8. 0xFee is a valid literal. This is because in hexadecimal notation, we write a number starting with a 0 following by an 'x' followed by any sequence of digits 0 to 9 and a to f (including capitals).

Typedef sometimes it is convenient to rename existing types in C. the `typedef` command renames an existing type to a new type. E.g.

```
typedef unsigned char BYTE;
```

The renamed type can now be used as:

```
BYTE b1, b2; //declares two unsigned char types b1, b2
```

While `typedef` allows you to rename types and give new names to existing types, you can actually create new types in C. *Structures* are a way to create new types.

Structures C does not have many built in datatypes: `int`, `char`, `short`, `long`, `float`, `double`, and a few others, as well as arrays of each and pointers to each. But what if you want to talk about more complex pieces of data?

What if we want to represent a point on a graph? We cannot represent that point with just a single value, like a `float`. We need *two* values: an x coordinate and a y coordinate: `float point_x; float point_y;`

But what *is* a point? It's not a `float`. It's a `float` representing its x coordinate and a `float` representing its y coordinate. Can we define data types that let us say that a variable is <thing one> and <thing two> and <thing three>?

C structures let us do this. We can define a new type that lets us say that if a variable is a point, it is two floats!

```
typedef struct { float x; float y; } Point;
```

And now when we declare a new variable, we can say that it *is a Point*:

```
Point p1; Point p2;
```

To access the components of a structure, we use '.' (dot) operator:

```
p1.x = 2.5; p1.y = 3.7; p2.x = p1.x - 3; p2.y = p1.x * 2;
```

How to initialize structures? `typedef struct { float x; float y; } Point;` //Simple style: `Point p; p.x = 1.5; p.y = 2.5;`

Bad because it separates the declaration from the initialization

//All-at-once style: `Point p = {1.5, 2.5}`

Bad because the order of fields isn't obvious, and you may mess it up

//Best style:

```
Point p = {.x = 1.5, .y = 2.5}
```

Makes clear which fields of the struct are initialized to what.

Nesting structures? structures are data types, too, which means you can make a structure a field of another