SUMMER 2016 CS61C DISCUSSION PREPARATION NOTES

ALEX JING

Contents

1. C Intro clarification

1.1. chars, strings, and list of strings.

- char is a C primitive(most basic unit), it is declared by single qoute. Notice there is a difference between 'a' and "a".(first is a char, second is a string.)
- string in C is just an null-terminated array of char. Null-terminated means at the end of every string there is a implicit null character '\n'. You can declare a string as **char a[]** or **char a***. While the former may make more sense if you know java, the latter is actually used more frequently. It also makes sense because a string is just a char array, you can simply declare a pointer to the first char in the array and the compiler will figure out the end by the null character at the end (which can be tricky for code-writing in certain cases).
- To generalize, you can always declare an array with an pointer. This is both hacky and dangerous, so be sure to know what you are doing.
- if you know the length of your char (or maximum length in most cases), you could also do **char a[LENGTH]**, which declares a string of the given length **on the stack**
- Lastly, if you see things like **char **my_string_list**, i.e. double pointers, in <u>most</u> cases in this class, it means an array of arrays, and since it is a char pointer, it <u>usually</u> means an array of string.

1.2. pass by value and pass by reference.

C functions always pass by value, which means when a function foo calls another function bar, the arguments given by foo is copied to the stack of bar. This explain why pointers are so useful, because we can dereference the pointer, go to the memory address and retrieve the data. This way we can achieve communications between functions.

1.3. typedef and struct declaration.

- In class we have the following way to declare a struct and its usage:

  ```
  typedef struct{
          int a;
          int* pointer;
  } myStruct

  myStruct myS;
  ```

  Essentially what typedef is doing here is to declare a struct as a type named myStruct, such that the compiler would know **myStruct** as any other built in C types such as int, char etc.

- This is really a special use of typedef, you can really use typedef to give a customized name to anything if necessary. For example, if for some reason you want to call a 32-bit unsigned integer as **myNum**, you could do **typedef uint32_t myNum**, and you could declare any 32-bit unsigned integer as **myNum**. This is very common in more advanced C programming because it improves your code readability by a lot.

- You can also declare a struct like this (you will probably see this more often in actual C codes):
  ```
  struct myStruct{
          int a;
          int* pointer;
  }

  struct myStruct myS;
  ```
  Here, I am only declaring a struct, not a type (in contrast to typedef in previous case). As a result, notice that you will have to call it **struct myStruct** when declaring a variable.

- to summarize the nuance difference between these two types, typedef defines a type alias that is a struct; struct declaration defines a struct that compiler only knows if you state specifically that it is a struct.

- you can also combine both as the following, more info at Here:

  ```
  typedef struct myStruct{
          int a;
          int* pointer;
  }myStruct
  ```

1.4. **#ADVANCED rvalue and lvalue.** (for the curious minds)

- Values in C are divided into two types, lvalue and rvalue.
- lvalues are values that have an identifiable addresses, rvalues are everything else (**r**est of the values)
- Since we know that memory can be visualized as indexed cells, lvalues are values that could live in one of these cells.
- On the other hand, rvalues are values that do not live in memory at all. As a result, you could not get their addresses.
  e.g. if you do **int *integer_pointer = &1**, this will throw an error because 1 is a rvalue here.
  Similarly, if you do **int *pointer_to_address = &malloc(100)**, this will also give you an error because malloc returns an address, which itself does not have an address.

2. Uncommented Code

(1) pointer arithmetic/recursion
(2) ! is boolean not, ˜ is one's complement. Use ! on a false value(a.k.a integer 0 and null pointer 0x0) will give you 1. Use ! on everything else will give you 0.
(3) ∧ is binary XOR. Remember C function adopts pass by value.

3. Pointers

(1) dereference pointer and assignment
(2) difference between *p++, ++*p, *++p.

First, on *p++:
There are two ideas that are potentially confusing:
• order of execution, which is the chronological order on how expressions are evaluated
• order of precedence, which is how 'tightly' an operator binds to its operand.

On order of execution, while implementation dependent, the most intuitive way to understand it is to read it from left to right or from prefix to postfix. Specifically, *p++ is carried out as:

1. we first dereference p and return the result stored in p. (prefix)
2. we increment the pointer p, not the value stored in p. (postfix)

To translate the one-liner to more readable code, it goes as the following:

```
return *p;
p = p+1;
```

You can read more Here.

On order of precedence, postfix generally have higher precedence than prefix. This means postfix ++ binds more tightly to p than *. As a result, ++ acts on p, the pointer, rather than *p, the value stored in the pointer. However, tighter binding does not necessarily mean earlier execution (it is in fact implementation dependent.).
The baseline is, after this expression, the original value stored in p will be return, the pointer p will be advanced to the next cell in memory. Second,

++*p and *++p are more straightforward.

Since there is not postfix, we can just defer the order of precedence by the rule of associativity, which is from right to left (from closet to farthest.).
Specifically, ++*p will dereference p first the then ++ will act on the dereferenced value.
*++p will advance the pointer first then dereference the value the new pointer points to.

(3) hacky strlen (so overly used tbh....)
But mechanistically, str will be dereferenced first then incremented. So when str returns the null terminator of the string, the null character will be returned and str will be pointing to the one character past the null terminator (which is allowed by C99 standards.). The returned null terminator will evaluate to false, hence break out of the loop.

4. Debugging

(1) argc, argv and array pointer conversion.
When you pass an array declared as **char arr**[] to a function, it will automatically be converted to a pointer (pointing to the first element in the array.). Therefore, you must have a way to tell the next function how long the array is, otherwise horrible things will happen as you step out of the bound. Here, v stands for value, c stands for count. Also, **sizeof(summands)** will give you the size of a pointer, not the entire array.

(2) pointer arithmetic go wild. Think about what if string has the maximum allowed address and you are trying to increment it. (KERNEL PANIC!!!!)

(3) hacky copy (again, never ever write code like this....people reading this will have a thousand reasons to kill you)
Mechanistically, this is similar to the hacky strlen. One thing to note is that in C99 6.5.16, **An assignment expression has the value of the left operand after the assignment**.
This means that once the null terminator in src is copied over, it will be assigned to the corresponding position in dst, after which the value at dst will be returned (which is the null character we just assigned it), the null terminator breaks the loop.