CS 521: Systems Programming

Pointers

Lecture 6

Today's Schedule

- Pointers
- Argument Passing Conventions

Today's Schedule

- **Pointers**
- Argument Passing Conventions

Passing by Value

- In C, function arguments are passed by **value**
  - **NOT** pass by reference
- This means that changes to the argument *inside* the function are not reflected *outside* the function
  - When you call a function, like: `location(2, 4);`
  - Copies will be made of `2` and `4` and passed to `location()`
- Sometimes we actually do want to change the value of a variable when it's passed into a function, though…

Passing by Reference [1/2]

Here's what a ***swap*** function should produce, but it doesn't seem possible if `a` and `b` are just copies:

```
int a = 3;
int b = 8;
printf("%d, %d\n", a, b);
swap(a, b);
printf("%d, %d\n", a, b);
```

Output:

```
3, 8
8, 3
```

Passing by Reference [2/2]

- If you want to make outside changes to a variable passed to a function, then you must use **pointers**
- Pointers are a special type of integer that hold a memory address
  - They are still passed by value; the value is the memory address
  - However, we can use the memory address to access a variable defined ***outside*** a function

Pointer Syntax

- `int *x;` – defines a *pointer*. Note that this doesn't create an integer, it creates a **pointer to an integer**.
  - To make life a little easier, focus on the fact that it's a pointer. Don't worry about its data type for now.
- `&` – 'address of' operator. `&a` retrieves a pointer to `a`.
  - When a function takes a pointer as an argument, you need to give it an address
- After passing the value of the pointer (memory address), we can **dereference** it (`*` operator) to retrieve/change the data it points to:
  - `*x = 45;`

Pointing Somewhere Else

- Let's say we have a pointer, `int *p`.
- If we assign a value to `p`, we are modifying the memory address `p` points to.
- However, if we ***dereference*** `p`, then we change the actual memory it points at.

Demo: Writing swap()

Today's Schedule

- Pointers
- **Argument Passing Conventions**

Defining a Function

Functions are defined in C like this:

```
<return type> <function name>(<argument list>)
{
    ...
}
```

- If the function does not return a value, the return type should be void
- If there are no arguments, then the argument list is void (not required)
- Let's dig a bit deeper into this…

Argument Conventions [1/2]

- Coming from the Java or Python world, we're used to passing inputs to our functions
- The result (output) of the function is usually given to us in the return value
  - In Python you can even return a tuple. Nice!
- This is **not** the case with C.
  - In many cases, both the function inputs **and** outputs are passed in as arguments
  - The return value is used for error handling

Argument Conventions [2/2]

Here's an example:

```
/* Here's a function that increments an integer. */
void add_one(int *i)
{
   *i = *i + 1;
}

int a = 6;
add_one(&a); /* a is now 7 */
```

“In/Out” Arguments

- In C, some of the function arguments serve as **outputs**
- Or in the example we just saw, the function argument is **both** an input and an output!
- Some API designers even label these arguments as “in” or “out” args (example from the Windows API):

```
BOOL WINAPI FindNextFile(
 _In_ HANDLE hFindFile,
 _Out_ LPWIN32_FIND_DATA lpFindFileData
);
```

That's Weird… Why?!

- **Reason 1**: C does not have exceptions
  - Problem in a Java/Python function? Throw an exception!
  - Exceptions are a bit controversial among programming language designers
- In C, the return value of functions often indicates success or failure, called a ***status code***
- Functions don't ***have*** to be designed this way, but it's a very common convention

Efficiency

- **Reason 2**: Speed!
- Return values have to be copied back to the calling function
  - Say my function returns a bitmap image. The entire thing is going to get copied!
- In a language that focuses on speed and efficiency, updating the values directly in memory is faster
- Imagine transferring lots of large strings, objects, etc. around your program, copying them the whole time

Arguments/Return Values: How to Know?

- The return value **might** indicate a status code… and it might not.
- To be sure, use the `man` (manual) pages
  - (You could also google it, but that can occasionally lead you to the wrong documentation / advice)
- The C documentation is in section **3** of the man pages:
  - `man 3 printf`
- Each man page will explain how the arguments and return values are used

Error Messages

- Many C functions return a status code **and** set `errno`
  - Global variable that contains the last **error number**
- You can use the `perror()` function to convert this number into plain English (or your local language)
- Pass in a string prefix to help you trace your code:
  - Call `perror("open");` after `open(...)` call
  - Result: `open: No such file or directory`
    - (assuming the file being opened didn't actually exist)

void Argument [1/3]

- In C, there's a difference between `function()` and `function(void)`
- void arg: the function takes no arguments
- Empty arg list: the function may or may not take arguments
  - If it does, they can be of any type and there can be any number of them

void Argument [2/3]

- Why is this important?
- First, to understand older code
- From the C11 standard:
  - "The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature."
- Second, this may lead to incorrect function prototypes or passing incorrect args in your code

void Argument [3/3]

So, to sum up:

```
/* Takes an unspecified number of args: */
void function();
```

And:

```
/* Takes no args: */
void function(void);
```