CS 521: Systems Programming

Functions, Arguments, and Pointers

Lecture 4

Before we Start

- Finishing I/O (writing)
- Quiz 1

Quiz 1

- Quiz 1 is next Tuesday
  - 15 minutes
  - Multiple choice, short answer
  - Covers lectures and lab material
    - Concepts from the first three weeks of class
- To study: look over the slides, note anything you don't remember/understand, and go through the labs
- For things you don't know: check the reading for more in-depth explanation, or ask on Campuswire

Today's Schedule

- Functions and Data Types
- Pointers
- Argument Passing Conventions

Today’s Schedule

- **Functions and Data Types**
- 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)

Argument Types

- There are two types of function arguments in C, called *formal* and *actual*
- **Formal** arguments are specified in your function:

```
void location(int x, int y);
```

- **Actual** arguments are the actual (raw) values passed into the function: `location(2, 4);`

Calling a Function

What happens when we call a function?

1. System makes a note of the return address
2. Storage is set up for formal arguments
3. Actual arguments are copied into formal arguments
4. Control branches to first statement of function
5. The function executes!
6. Copy return value into memory
7. Jump back to the return address

C Data Types

- When defining arguments and variables, the following data types are possible in C:
  - `char`
  - `int`
  - `float`
  - `double`
- Wait… that’s it?! Yeah! Well, there are a few *modifiers*:
  - `short`, `long`, `signed`, and `unsigned`

Sizing

- `short` and `long` modify the data type's size
- The C standard specifies the *minimum* size for each type. You can determine the sizes (in bytes) with `sizeof`:
  - `sizeof(char) = 1`
  - `sizeof(short int) = 2`
  - `sizeof(int) = 4`
  - `sizeof(long int) = 8`
- …but these can be platform-specific. Don't make assumptions!
  - One thing can be certain: `char` is **guaranteed** to be 1 byte

Demo: Data Type Sizes

(you can do this one on your VM, or local machine if you have a C compiler!)

Signed Data Types

- Integer types can be **signed** or **unsigned**
  - Signed integers use one bit as a *sign bit* to determine whether the number is negative or positive
- Java doesn't have unsigned `ints`. What might they be useful for?
  - Enforce a particular variable to always be positive
  - Use that extra bit to store larger positive numbers
- Related: integer overflow is undefined behavior (UB)

Today’s Schedule

- Functions and Data Types
- **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:

```c
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` returns 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;`

Demo: Writing swap()

Today’s Schedule

- Functions and Data Types
- Pointers
- **Argument Passing Conventions**

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:

```c
/* 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

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:

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

While on the other hand:

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

Next Time: Strings

…and if we have time now, the phases of C compilation!

Phases of C Compilation

1. **Preprocessing**: perform text substitution, include files, and define macros. The first pass of compilation.
   - Directives begin with a #
2. **Translation**: preprocessed code is converted to machine language (also known as *object code*)
3. **Linking**: your code likely uses external routines (for example, `printf` from stdio.h). In this phase, libraries are added to your code

Stepping Through Compilation

- When we compile our source code, we get an output binary that is ready to run
  - The steps are mostly invisible to us
- We can ask the compiler to only execute a subset of its compilation phases
  - Let’s do just that!

Preprocessing

- We can ask gcc to only perform the preprocessing step using the `-E` flag:
  - `gcc -E my_program.c`
- This will print the preprocessed file to the terminal
- We can write this output to a file by redirecting the stdout (**standard output**) stream:
  - `gcc -E my_program.c > my_program.pre`
- …And view it with a text editor

Translating to Assembly Code

- We can also view the **assembly** code generated by the compiler:
  - `gcc -S my_program.c`
  - Produces my_program.s
- This representation is very close to the underlying **machine code**
- For a reference on x86-64 processor assembly:
  - https://web.stanford.edu/class/cs107/guide_x86-64.html

Producing Object Code

- Finally, we can produce the **machine code / object code** representation of the program
- `gcc -c my_program.c`
  - Produces my_program.o
- We can view this with a **hex dump**
  - `hexdump -C my_program.o`