CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 7 – Introduction to C: The C Level of Abstraction

(Thanks to Hal Perkins)

Welcome to C

Compared to Java, in rough order of importance

- Lower level (less for compiler to do)
- Unsafe (wrong programs might do anything)
- Procedural programming — not "object-oriented"
- "Standard library" is much smaller
- Many similar control constructs (loops, ifs, ...)
- Many syntactic similarities (operators, types, ...)

• A different world-view and much more to keep track of; Java-like thinking can get you in trouble

Our plan

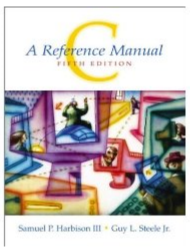
A semi-nontraditional way to learn C:

- Learn how C programs run on typical x86 machines
  - Not promised by C's definition
  - You do *not* need to "reason in terms of the implementation" when you follow the rules
  - But it does help to know this model
    - To remember why C has the rules it does
    - To debug incorrect programs
    - To write better programs (performance, portability…)
- Learn some C basics (including "Hello World!")
- Learn what C is (still) used for
- Learn more about the language and good idioms
- Towards the end of the quarter: Some C++ (C with classes and other conveniences of a modern language)

Some references

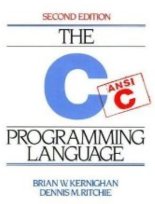
There's a lot on the web, but here are some primary sources

C: A Reference Manual, Harbison & Steele (now 5th ed.)



- The best current reference on C and its libraries; includes information about recent versions of the C standard

The C Programming Language, Kernighan & Ritchie



- "K&R" is a classic, one that every programmer must read. A bit dated now (doesn't include C99 or C11 extensions), but the primary source

Essential C, Stanford CS lib, http://cslibrary.stanford.edu/101/EssentialC.pdf

Good short introduction to the language

Why C?

- small language (i.e., a minimum of features) makes it relatively easy to write a compiler for C (contrast with C++)
- provides low level control over the computer, closer to that of assembly (machine) language
- Still used in:
  - embedded programming
  - systems programming
  - high-performance programming (lots of fast libraries for nicer languages are written in C)
- Additional reason for CSE 374: programming in C will help us understand better how computers work

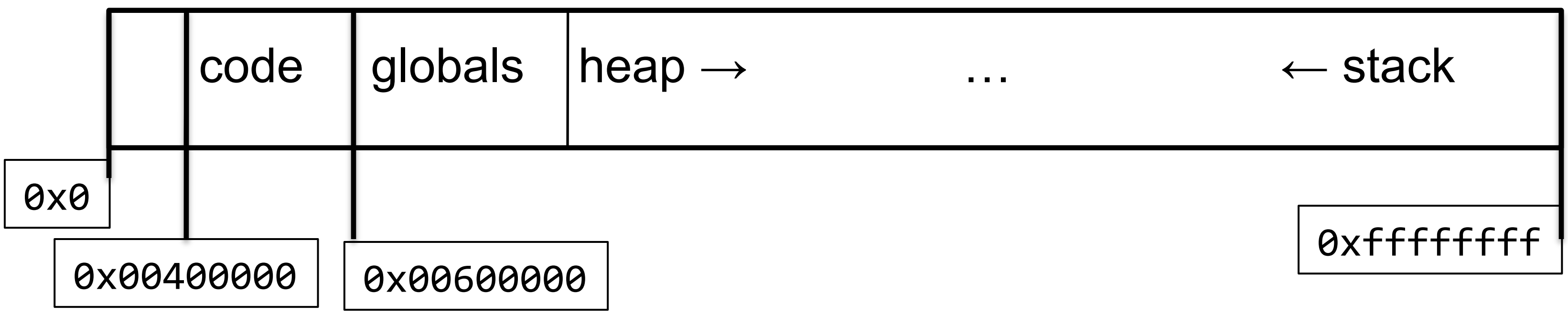
Address space

Simple model of a running process (provided by the OS):

- There is one address space (an array of bytes)
  - Most common size today for a typical machine is 2^{32} or 2^{64}
  - For most of what we do it doesn't matter
  - 2^{64} is way more RAM than you have, you might have 2^{32} (4GB) or more (OS maintains illusion that all processes have this much even if they don't – may lead to slowness)
  - pointing to an element of this array takes 32 or 64 bits
  - Something's address is its position in this array
  - Trying to read a not-used part of the array may cause a "segmentation fault" (immediate crash)
  - In contrast, in Java *every* call to `new` provides an isolated object
- All data and code for the process are in this address space
  - Code and data are bits; program "remembers" what is where
  - O/S also lets you read/write files (stdin, stdout, stderr, etc.)

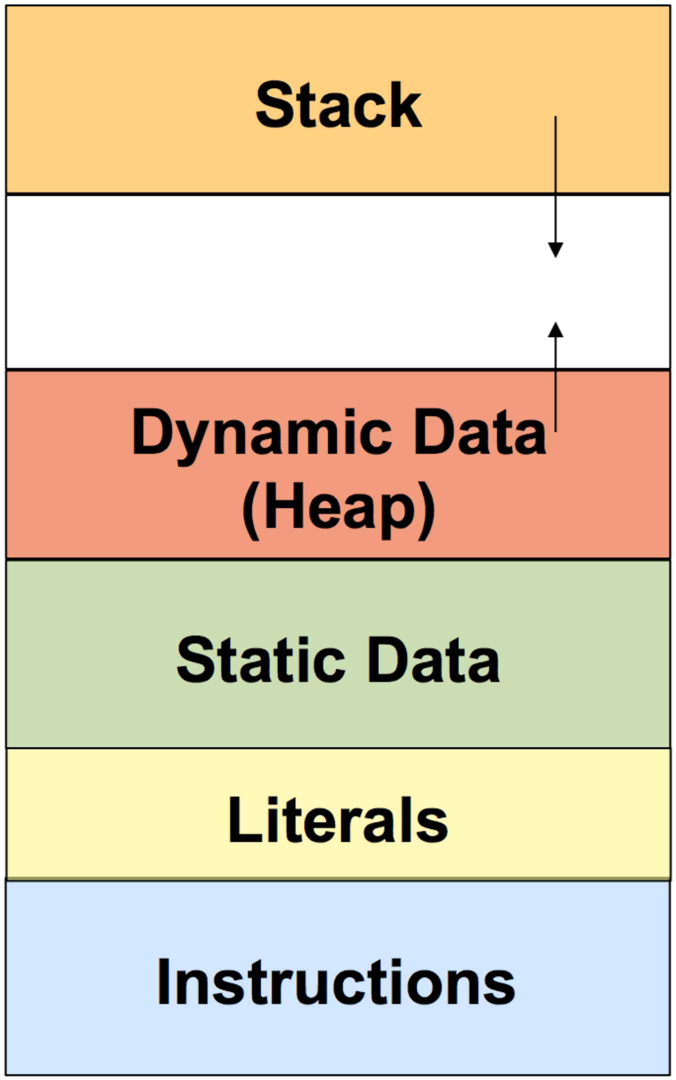
Address-space layout

- The following can be different on different systems, but it's one way to understand how C is implemented:



- So in one array of 8-bit bytes we have:
  - Code instructions (typically immutable)
  - Space for global variables (mutable and immutable) (like Java's static fields)
  - A *heap* for other data (like objects returned by Java's new)
  - Unused portions; access causes a "seg-fault"
  - A call-*stack* holding local variables and *code addresses*
- ints typically occupy 4 bytes (32 bits); pointers 4 or 8 (32 or 64) depending on underlying processor/OS (64 on our machines)

Address-space layout



Stack: read/write; not executable — local variables, return addresses for function calls (managed "automatically" by compiler)

Dynamic Data (Heap): read/write; not executable — allocated memory like objects (managed by programmer)

Static Data: read/write; not executable — global variables (initialized when process starts)

Literals: read-only; not executable — constants (initialized when process starts)

Instructions: read-only; executable — code (initialized when process starts)

[Thanks to CSE 351]

The stack

- The call-stack (or just stack) has one part, or “frame”, for each active function (cf. Java method) that has not yet returned

Stack-based languages

- Languages that support recursion
  - e.g., C, Java, most modern languages
  - Code must be re-entrant
    - multiple simultaneous instantiations of a single function
  - need some place to store state of each instantiation
    - arguments
    - local variables
    - return address (index into code for what to execute after the function is done)
- stack discipline
  - state for a given procedure needed for a limited time
    - starting from when it is called
    - ending when it returns
  - callee always returns before the caller does
- stack allocated in frames
  - state for a single procedure instantiation

[Thanks to CSE 351]

Call chain example

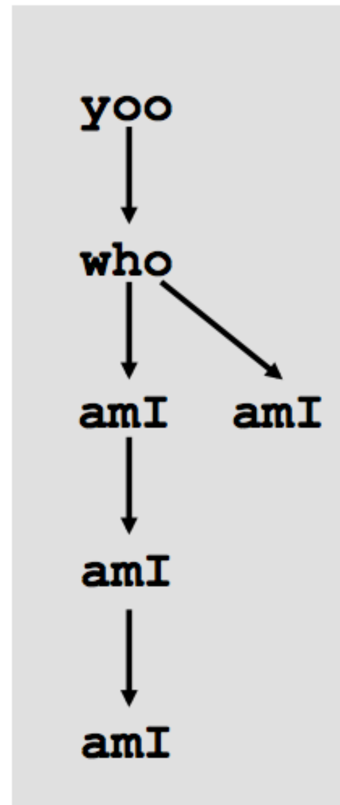
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

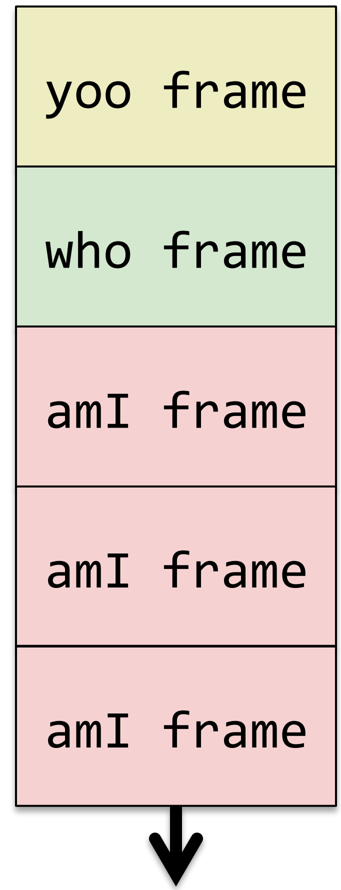
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

procedure `amI` is recursive
(calls itself)

example
call chain



example
stack



What could go wrong?

- The programmer needs to think about bits even though C deals in terms of variables, functions, data structures, etc. (not bits)
  - If arr is an array of 10 elements, arr[30] accesses some other undefined thing
  - Storing 8675309 where a return address should be makes a function return start executing stuff that may not be code
  - . . .
- Correct C programs can't do these things, but nobody is perfect
- On the plus side, there is no "unnecessary overhead" like keeping array lengths around and checking them!

Hello, World!

- Code:

```
#include<stdio.h>
int main(int argc, char**argv) {
   printf("Hello, World!\n");
   return 0;
}
```

  - Compiling: gcc -std=c11 -o hello hello.c (normally add -Wall -g)
  - Running: ./hello
- Intuitively: main gets called with the command-line args and the program exits when it returns
- But there is a *lot* going on in terms of what the language constructs mean, what the compiler does, and what happens when the program runs
- We will focus mostly on the language

Quick explanation

```
#include <stdio.h>
int main(int argc, char**argv) {
   printf("Hello, World!\n");
   return 0;
}
```

- #include finds the file stdio.h (from where?) and includes its entire contents (stdio.h describes printf, stdout, and more)
- A function definition is much like a Java method (return type, name, arguments with types, braces, body); it is not part of a class and there are no built-in objects or “this”
- An int is like in Java, but its size depends on the compiler (it is 32 bits on most mainstream Linux machines, even x86-64 ones)
- main is a special function name; every full program has one
- char** is a long story…

Pointers

& "address of"
* "value at address" or "dereference"

- Think address, i.e., an index into the address-space array
- If argv is a pointer, then *argv returns the pointed-to value
- So does argv[0]
- And if argv points to an array of 2 values, then argv[1] returns the second one (and so does *(argv+1) but the + here is funny)
- People like to say "arrays and pointers are the same thing in C". This is not true. The two are very closely related but are different.
- Type syntax: T* describes either
  a. NULL (seg-fault if you dereference it)
  b. A pointer holding the address of some number of contiguous values of type T
- How many? You have to already know somehow; pointers have no length primitive (e.g., argc is number of char* argv points to)

Pointers, continued

- So reading right to left: argv (of type char**) holds a pointer to (one or more) pointers to (one or more) char
- Fact #1 about main: argv holds a pointer to j pointers to (one or more) char(s) where argc holds j
- Common idiom: array lengths as other arguments
- Fact #2 about main: For 0 ≤ i ≤ j where argc holds j, argv[i] is an array of char(s) with last element equal to the character ’\0’ (which is not ’0’)
- Very common idiom: pointers to char arrays ending with ’\0’ are called *strings*.
  - The standard library relies on this idiom (e.g., strnlen)
  - The language relies on this idiom (e.g. string constants like “Hello”)

(question from class)

- If two individual pointees happen to be adjacent, can I just access either pointee with either pointer?
- No, this would be an incorrect C program (it might work sometimes but behavior is undefined by the standard and it will probably break)
- e.g.

```
char* g = “ab”;
char* h = “xy”;
g[2];  // okay
g[3]; // BUG! although it might return ‘x’
```

‘a’	**‘b’**	**‘\0’**	**‘x’**	**‘y’**	**‘\0’**	**…**

Let's draw a picture of "memory" when hello runs.

- ./hello -n 374
- assume 64-bit machine

address	data	# bytes
0x04	(char*) 0x10	8
0x0c	(char*) 0x22	8
	…	
0x10	'-'	1
0x11	'n'	1
0x12	'\0'	1
	…	
0x22	'3'	1
0x23	'7'	1
0x24	'4'	1
0x25	'\0'	1
	…	
0x50	(argc) 2	4
0x54	(argv) 0x04	8

Rest of the story

```
#include<stdio.h>
int main(int argc, char**argv) {
   printf("Hello, World!\n");
   return 0;
}
```

- printf is a function taking a string (a char*) (and often additional arguments, which are formatted according to codes in the string)
- "Hello, World!\n" evaluates to a pointer to a global, immutable array of 15 characters (including ‘\n’ and the trailing ‘\0’)
- printf writes its output to stdout, which is a global variable of type FILE* defined in stdio.h
  - How this gets hooked up to the screen (or somewhere else) is the library’s (nontrivial) problem
- return in main is the program’s exit code; (caller can check, e.g. in shell scripts with $?)

But wait, there's more!

- More features will be explored, starting in hw4
  - Accessing program command-line arguments (argc and argv)
  - Other I/O functions (fprintf, fputs, fgets, fopen, …)
  - Strings – much ado about strings
    - Strings as arrays of characters (local and allocated on the heap)
    - Updating strings, buffer overflow, '\0'
    - String library (<string.h>)