CSE 303 Concepts and Tools for Software Development

Richard C. Davis
UW CSE – 10/11/2006

Lecture 7 – Introduction to C

Administravia

- Extra Office Hours
  - Richard: Thursday (10/12), CSE 210

Tip of the Day

- ACM Student Chapter Tutorials
  - Accessible from CSE 303 "Comp. Resources"
  - Fast way to find way around Linux
    - Printing
    - Lists of common commands
    - Emacs tutorials
    - Language tutorials
      - Sed, Awk, Python
      - C, C++
    - Tool Tutorials (

Where are We?

- Before: Shell
  - Files
  - Users
  - Processes
- Now: C and C++ (for "processes")
  - Numbers
  - Memory

Welcome to C

- Comparing to Java:
  - ***Lower Level***: good for compiler, bad for you
  - ***Unsafe***: wrong programs can do anything
  - "***Standard Library***" is smaller
  - ***Similar Syntax***: pan help or confuse
  - ***Procedural***: Not "object oriented"
- Java-like thinking can get you in trouble!

Plan for Learning C

- Learn "C Style" Programming
  - Syntax
  - Pointers & Memory Management
  - Good idioms
  - Why use C?
- Later
  - "C++ Style" Programming
  - Development Tools

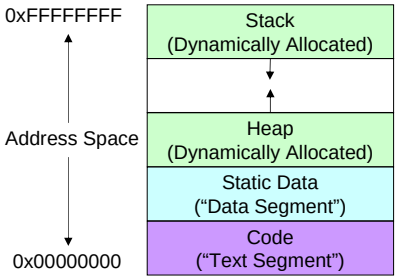
Learning Method

- We'll Look at How C Programs Run
  - Not promised by C's definition
  - Allows reasoning about the implementation
    - Know why C does what it does
    - Avoid this in general!
    - But sometimes need to debug programs
- Note: Text does not cover this in detail
  - Notes will be important

Today

- C Memory Model
- Simple C Programs
- Introduction to Pointers

Address Space of Unix Process



Address space is an array of 8-bit bytes

An index into this array is called an *address*

Typical total size is 2^{32}

We'll assume that an integer is 4 bytes

More on the Address Space

- Read an unused part of memory?
  - May cause "segmentation fault" (crash)
- ***Code***: instructions for program (read only)
- ***Static Data***: usually global variables
- ***Stack***: local variables and code addresses
  - Grows and shrinks as program executes
- ***Heap***: data (like objects from Java's new)
  - Managed manually

Hello World

```
#include <stdio.h>

/*
 * First C program
 */
int main() {

  // Print to the stream "stdout"
  fputs("Hello World\n",stdout);

  return 0;
}
```


Testing Hello World

- Compiling and running hello.c
  - Compile: `gcc -g -Wall -o hi hello.c`
  - Run: `./hi`
- Compile Command Meaning
  - `gcc` : Gnu C Compiler
  - `-g` : include debugging information
  - `-Wall` : show all warnings (good for learning)
  - `-o hi` : specifies program name
    - Omit? Program is named `a.out`

Header Files

- `#include <stdio.h>`
  - Directive to C Preprocessor (more later)
  - Finds `stdio.h` and includes *entire* contents
  - stdio.h is a "header" file
  - stdio.h describes `fputs` and `stdout`

Functions

- `fputs`
  - Function that sends characters to a stream
  - `\n` is an escape sequence for "newline"
- `main`
  - Every C program starts execution here
- Functions like Java methods, but…
  - Not part of a class
  - Not associated with an object (no "this)

How Processes Operate

- Load Code/Static Data into memory
- Store current "Stack Frame"
  - Address of current instruction
  - Local variables
- Calling "Functions" (like Methods in Java)
  - Creates new stack frame
  - Current instruction becomes "return address"
- On exit, delete all memory

Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int integer2;
4       int sum;
5       integer1 = 10;
6       integer2 = 20;
7       sum = integer1 + integer2;
8       printf("\nSum is %d", sum);
9       return 0;
   }
```

Stack after line 4

integer1	XXXX
integer2	XXXX
sum	XXXX

Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int integer2;
4       int sum;
5       integer1 = 10;
6       integer2 = 20;
7       sum = integer1 + integer2;
8       printf("\nSum is %d", sum);
9       return 0;
   }
```

Stack after line 5

integer1	10
integer2	XXXX
sum	XXXX

Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int integer2;
4       int sum;
5       integer1 = 10;
6       integer2 = 20;
7       sum = integer1 + integer2;
8       printf("\nSum is %d", sum);
9       return 0;
   }
```

Stack after line 6

integer1	10
integer2	20
sum	XXXX

Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int integer2;
4       int sum;
5       integer1 = 10;
6       integer2 = 20;
7       sum = integer1 + integer2;
8       printf("\nSum is %d", sum);
9       return 0;
   }
```

Stack after line 7

integer1	10
integer2	20
sum	30

Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int integer2;
4       int sum;
5       integer1 = 10;
6       integer2 = 20;
7       sum = integer1 + integer2;
8       printf("\nSum is %d", sum);
9       return 0;
   }
```

Stack during execution of printf

integer1	10
integer2	20
sum	30
activation record for printf	

Activation Records

Return Address (code)
Address for returned value
Argument 1
Argument 2
…
Local Variable 1
Local Variable 2
…

Introduction to Pointers

- Pointer = variable that holds an address
- Declaring a pointer
  - `int *mypointer;`
- Assigning an address to a pointer
  - `mypointer = &integer1;`
- Accessing data pointed to by a pointer
  - `*mypointer`

Pointer Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int* mypointer;
4       integer1 = 10;
5       mypointer = &integer1;
6       printf("\nValue is %d", integer1);
7       printf("\nValue is %d", *mypointer);
8       return 0;
9  }
```

Stack after line 3

integer1	XXXX
mypointer	XXXX

Pointer Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int* mypointer;
4       integer1 = 10;
5       mypointer = &integer1;
6       printf("\nValue is %d", integer1);
7       printf("\nValue is %d", *mypointer);
8       return 0;
9  }
```

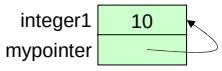
Stack after line 4

integer1	10
mypointer	XXXX

Pointer Execution Example

```
   #include <stdio.h>
1  int main() {
2       int integer1;
3       int* mypointer;
4       integer1 = 10;
5       mypointer = &integer1;
6       printf("\nValue is %d", integer1);
7       printf("\nValue is %d", *mypointer);
8       return 0;
9  }
```

Stack after line 4



Why C is Dangerous

- C deals with variables, functions, *etc*…
- But Programmer must keep *bits* straight!
  - arr is array of 10 elements: arr[30] is what?
    - Could be anything
  - Writing 3263827 to a return address
    - Program could start running from anywhere
- Why would anyone work like this?
  - Fast: No "unnecessary" overhead
  - Need to for embedded/operating systems

Summary

- C Memory Model
- Simple C Programs
- Introduction to Pointers

Reading

- Skim Sections that look familiar!
  - Author does not assume you know Java
  - C++ book does assume you know Java
- Programming in C
  - Chapter 1: Introduction
  - Chapter 2: Fundamentals
  - Chapter 3: Compiling and Running
  - Chapter 11: (pp235-240) Pointers

Next Time

- C Program Structure
- Expressions
- Local Variables
- Left vs. Right expressions
- Dangling Pointers