CS354: Machine Organization and Programming

Lecture 2
Friday the September 4th 2015

Section 2
Instructor: Leo Arulraj
Class Announcements

1. Are your mailing lists working? Did you receive the Welcome email yesterday evening?

2. Question about Midterm 1 conflict: How many are taking ECE 353?

3. Code, slides, etc. will be shared with you but it won’t be timely. So, take notes in class!

4. Good job in Piazza (Anon. posts are okay!)

5. **Project 0 is due Sep 14th before 9 AM.** Partner details for projects.
Five Realities you must embrace

1. Ints are not Ints and Floats are not reals.
2. You have to know assembly.
3. Memory hierarchy matters
4. Performance is not just algorithmic complexity
5. Computer do more than just load-store-execute! They do I/O, networking with other computers etc. for example.
Reality 3

Great Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
Memory Hierarchy

Figure 1.9  An example of a memory hierarchy.

Smaller, faster, and costlier (per byte) storage devices

L1:
L1 cache (SRAM)

L2:
L2 cache (SRAM)

L3:
L3 cache (SRAM)

L4:
Main memory (DRAM)

L5:
Local secondary storage (local disks)

L6:
Remote secondary storage (distributed file systems, Web servers)

CPU registers hold words retrieved from cache memory.

L1 cache holds cache lines retrieved from L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote network server.
Memory Mountain

The Memory Mountain

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache

- L1
- L2
- L3

Copyij
Copyji
Introduction to C Programming

- Operators
- Types and Declarations
- Statements
- Functions
- C-Preprocessor Directives
- Simple I/O
Operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Miscellaneous Operators
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example A=10,B=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds two operands</td>
<td>A + B will give 30</td>
</tr>
<tr>
<td>-</td>
<td>Subtracts second operand from the first</td>
<td>A - B will give -10</td>
</tr>
<tr>
<td>*</td>
<td>Multiplies both operands</td>
<td>A * B will give 200</td>
</tr>
<tr>
<td>/</td>
<td>Divides numerator by de-numerator</td>
<td>B / A will give 2</td>
</tr>
<tr>
<td>%</td>
<td>Modulus Operator and remainder of after an integer division</td>
<td>B % A will give 0</td>
</tr>
<tr>
<td>++</td>
<td>Increments operator increases integer value by one</td>
<td>A++ will give 11</td>
</tr>
<tr>
<td>--</td>
<td>Decrements operator decreases integer value by one</td>
<td>A-- will give 9</td>
</tr>
</tbody>
</table>
## Relational Operators

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example A=10, B=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Checks if the values of two operands are equal or not, if yes then condition becomes true.</td>
<td>(A == B) is not true.</td>
</tr>
<tr>
<td>!=</td>
<td>Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.</td>
<td>(A != B) is true.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt; B) is not true.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt; B) is true.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt;= B) is not true.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt;= B) is true.</td>
</tr>
</tbody>
</table>
# Logical Operators

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Called Logical AND operator. If both the operands are non-zero, then condition becomes true.</td>
<td>(A &amp;&amp; B) is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.</td>
<td>!(A &amp;&amp; B) is true.</td>
</tr>
</tbody>
</table>
# Bitwise Operators

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| &   | Binary AND Operator copies a bit to the result if it exists in both operands. | A(60) = 0011 1100  
B(13) = 0000 1101  
(A & B) will give 12, which is 0000 1100 |
| |  |  |
| | Binary OR Operator copies a bit if it exists in either operand. | A | B) will give 61, which is 0011 1101 |
| |  |  |
| ^   | Binary XOR Operator copies the bit if it is set in one operand but not both. | A | B) will give 49, which is 0011 0001 |
| ~   | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61, which is 1100 0011 in 2's complement form. |
| <<  | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >>  | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |
## Assignment Operators 1

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Simple assignment operator, Assigns values from right side operands to left side operand</td>
<td>C = A + B will assign value of A + B into C</td>
</tr>
<tr>
<td>+=</td>
<td>Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand</td>
<td>C += A is equivalent to C = C + A</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand</td>
<td>C -= A is equivalent to C = C - A</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand</td>
<td>C *= A is equivalent to C = C * A</td>
</tr>
<tr>
<td>/=</td>
<td>Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand</td>
<td>C /= A is equivalent to C = C / A</td>
</tr>
</tbody>
</table>
## Assignment Operators 2

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%=</td>
<td>Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.</td>
<td>C %= A is equivalent to C = C % A</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left shift AND assignment operator</td>
<td>C &lt;&lt;= 2 is same as C = C &lt;&lt; 2</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Right shift AND assignment operator</td>
<td>C &gt;&gt;= 2 is same as C = C &gt;&gt; 2</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise AND assignment operator</td>
<td>C &amp;= 2 is same as C = C &amp; 2</td>
</tr>
<tr>
<td>^=</td>
<td>bitwise exclusive OR and assignment operator</td>
<td>C ^= 2 is same as C = C ^ 2</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bitwise inclusive OR and assignment operator</td>
</tr>
</tbody>
</table>
## Miscellaneous Operators

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof()</td>
<td>Returns the size of an variable.</td>
<td>sizeof(a), where a is integer, will return 4.</td>
</tr>
<tr>
<td>Unary &amp;</td>
<td>Returns the address of an variable.</td>
<td>&amp;a; will give actual address of the variable.</td>
</tr>
<tr>
<td>Unary *</td>
<td>Value of a pointer</td>
<td>*a; will value stored in the address a.</td>
</tr>
<tr>
<td>? :</td>
<td>Conditional Expression</td>
<td>If Condition is true ? Then value X : Otherwise value Y</td>
</tr>
</tbody>
</table>
Example C program on Operators
Operator Precedence

\[ a > b + c \& \& d \]

This expression is equivalent to:

\[ ((a > (b + c)) \& \& d) \]

Why not this?: \[ ((a > b) + (c \& \& d)) \]
## Operator Precedence 1

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary scope resolution</td>
<td>left to right</td>
<td>::</td>
</tr>
<tr>
<td>Primary</td>
<td>left to right</td>
<td>()  [  ]  .  -&gt; dynamic_cast typeid</td>
</tr>
<tr>
<td>Unary</td>
<td>right to left</td>
<td>++  --  +  -  !  ~  &amp;  *  (type_name)</td>
</tr>
<tr>
<td>C++ Pointer to Member</td>
<td>left to right</td>
<td>sizeof new delete</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>left to right</td>
<td>*  /  %</td>
</tr>
<tr>
<td>Additive</td>
<td>left to right</td>
<td>+  -</td>
</tr>
<tr>
<td>Bitwise Shift</td>
<td>left to right</td>
<td>&lt;&lt;  &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>left to right</td>
<td>&lt;  &gt;  &lt;=  &gt;=</td>
</tr>
<tr>
<td>Operator Name</td>
<td>Associativity</td>
<td>Operators</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Equality</td>
<td>left to right</td>
<td>== !=</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>left to right</td>
<td>&amp;</td>
</tr>
<tr>
<td>Bitwise Exclusive OR</td>
<td>left to right</td>
<td>^</td>
</tr>
<tr>
<td>Bitwise Inclusive OR</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>Logical AND</td>
<td>left to right</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Logical OR</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>right to left</td>
<td>? :</td>
</tr>
<tr>
<td>Assignment</td>
<td>right to left</td>
<td>+= -= *= /= &lt;= &gt;= &gt;&gt;= %= &amp;= ^=</td>
</tr>
<tr>
<td>Comma</td>
<td>left to right</td>
<td>,</td>
</tr>
</tbody>
</table>
Example C program on Operator precedence
Types

- Integer types
- Floating point types
- The void type
- Type Qualifiers
- Strings in C
## Integer Types

The actual size of integer types varies by implementation. Standard only requires size relations between the data types and minimum sizes for each.

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 bytes</td>
<td>-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 or 4 bytes</td>
<td>0 to 65,535 or 0 to 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>
### Floating Point Types

The value representation of floating-point types is implementation-defined

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 byte</td>
<td>1.2E-38 to 3.4E+38</td>
<td>6 decimal places</td>
</tr>
<tr>
<td>double</td>
<td>8 byte</td>
<td>2.3E-308 to 1.7E+308</td>
<td>15 decimal places</td>
</tr>
<tr>
<td>long double</td>
<td>10 byte</td>
<td>3.4E-4932 to 1.1E+4932</td>
<td>19 decimal places</td>
</tr>
</tbody>
</table>
# Void type

<table>
<thead>
<tr>
<th></th>
<th>Function returns as void</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, <code>void exit (int status);</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Function arguments as void</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, <code>int rand(void);</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Pointers to void</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A pointer of type <code>void *</code> represents the address of an object, but not its type. For example a memory allocation function <code>void *malloc( size_t size );</code> returns a pointer to void which can be casted to any data type.</td>
</tr>
</tbody>
</table>
Strings in C

- Strings in C are one dimensional arrays of characters terminated with a null character.

Examples:

```c
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

char greeting[6] = "Hello";

char* greeting = "Hello";
```

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>H</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>\0</td>
</tr>
<tr>
<td>Memory Address.</td>
<td>0x88321</td>
<td>0x88322</td>
<td>0x88323</td>
<td>0x88324</td>
<td>0x88325</td>
<td>0x88326</td>
</tr>
</tbody>
</table>
Type Qualifiers

• **const**: means that something is not modifiable, so a data object that is declared with `const` as a part of its type specification must not be assigned to in any way during the run of a program.

• **volatile**: tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference.

• **restrict**: Has to do with pointers. Later!
## Storage Classes

<table>
<thead>
<tr>
<th>#</th>
<th>Storage Specifier</th>
<th>Storage place</th>
<th>Initial / default value</th>
<th>Scope</th>
<th>Life</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>auto</td>
<td>memory</td>
<td>Garbage value</td>
<td>local</td>
<td>Within the function only.</td>
</tr>
</tbody>
</table>
| 2 | extern            | memory         | Zero                    | Global| Till the end of the main program.
Variable definition might be anywhere in the C program |
| 3 | static            | memory         | Zero                    | local | Retains the value of the variable between different function calls. |
| 4 | register          | Register memory| Garbage value           | local | Within the function                          |
Fundamental to understanding programming languages is the concept of variables. Variables are placeholders in memory that can store values. Understanding the difference between global variables and local variables is crucial.

**Global Variable**: A global variable is a variable that is declared outside all functions. Its scope is the entire program. Any function in the program can access a global variable.

**Local Variable**: A local variable is a variable that is declared inside a function. Its scope is limited to the function in which it is declared. Once the function ends, the variable is no longer accessible.

**Examples**:

```c
const int foo = 10;  // foo is const integer with value 10

char foo;  // foo is a char

double foo();  // foo is a function returning a double
```
Explicit Type Conversions

double da = 3.3;
double db = 3.3;
double dc = 3.4;
int r1 = (int)da + (int)db + (int)dc; // r1 == 9
int r2 = (da + db + dc); // r2 == 10
Example C program on Types, sizes
If Statement

```java
if(boolean_expression){
    /* statement(s) will execute if the boolean expression is true */
}
```
If-else Statement

if(boolean_expression) {
    /* statement(s) will execute if the boolean expression is true */
} else {
    /* statement(s) will execute if the boolean expression is false */
}
Else-if Statement

```java
if(expression){
    /*Block of statements;*/
}else if(expression){
    /*Block of statements;*/
}else{
    /*Block of statements;*/
}
```
Example C program on if, if else, else if statements
Switch

```java
switch(expression){
    case constant-expression1:
        statements1;
        [case constant-expression2: statements2;]
        [case constant-expression3: statements3;]
        [default: statements4;]
}
```
While loop

```java
while (expression) {
    Single statement or Block of statements;
}
```
For loop

```
for(expression1;expression2;expression3){
    Single statement or
    Block of statements;
}
```

You can also skip expression1, expression2, expression3.

What does this do? `for(;;){printf("a\n");}`
Do while loop

do{
    Single statement  or
    Block of statements;
}
while(expression);
C provides two commands to control how we loop:

• **break** -- exit form loop or switch.
• **continue** -- skip 1 iteration of loop.
Goto and Labels

goto label;

............

............

............

label: statement

You can have better label names
( e.g. mycalc, complexcalc etc. )
Example C program on for, while loops, switch statements
Functions 1

**Function Prototype (Declaration):**
```
return_type function_name(
    type(1) argument(1),...,type(n) argument(n));
```

**Function Definition:**
```
return_type function_name(
    type(1) argument(1),..,type(n) argument(n))
{
    //body of function
}
```
Functions 2

**Function Call:**
function_name(argument(1),.....argument(n));

**Return Statement:**
return (expression);

C always passes arguments `by value`: a copy of the value of each argument is passed to the function; the function cannot modify the actual argument passed to it.
C always passes arguments `by value': a copy of the value of each argument is passed to the function; the function cannot modify the actual argument passed to it.

```c
#include <stdio.h>
int add(int a, int b);
int main(){
    ...
    sum=add(num1,num2);
    ...
}
int add(int a, int b){
    ...
}

Here,
    a=num1
    b=num2
```
Working of Functions

```c
#include <stdio.h>
int add(int a, int b);
int main()
{
    ............
    sum = add(num1, num2);
    ............
    return type of function
    { int add; 
      ............
      return add;
    }
    sum = add
```
C Preprocessor

File Inclusion

#include <file> - used for system header files. File is looked for in standard list of system directories

#include “file” - used for local header files in program.
C Preprocessor

Macro substitution

#define [identifier name] [value]

Eg. #define PI_PLUS_ONE (3.14 + 1)

#define MACRO_NAME(arg1, arg2, ...) [code to expand to]

Eg. #define MULT(x, y) x * y
C Preprocessor

Conditional Inclusion:

Simple example is:

```c
#ifdef MACRO
    controlled text
#endif /* MACRO */
```

More versions with else, ifndef etc. allowed.
Example C program illustrating C Preprocessor
**int printf(const char *format, ...)** function writes output to the standard output stream `stdout` and produces output according to a format provided.

**int scanf(const char *format, ...)** function reads input from the standard input stream `stdin` and scans that input according to format provided.
int b, a; long int b; char s[10], float d;

printf("%d\n",b);

scanf("%d", &a);

printf("%3d\n",b);

printf("%3.2f\n",d);

printf("%ld\n",b);
I/O Redirection and Pipes

I/O Redirection:

prog <infile >outfile
infile will be stdin and outfile will be stdout

Pipes: |

With pipes, the standard output of one command is fed into the standard input of another.
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i or %d</td>
<td>int</td>
<td>12345</td>
</tr>
<tr>
<td>%c</td>
<td>char</td>
<td>y</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
<td>“sdfa”</td>
</tr>
<tr>
<td>%f</td>
<td>Display the floating point number using decimal representation</td>
<td>3.1415</td>
</tr>
<tr>
<td>%e</td>
<td>Display the floating point number using scientific notation with e</td>
<td>1.86e6</td>
</tr>
<tr>
<td>%E</td>
<td>Like e, but with a capital E in the output</td>
<td>1.86E+06</td>
</tr>
<tr>
<td>%g</td>
<td>Use shorter of the 2 representations: f or e</td>
<td>3.1 or 1.86e6</td>
</tr>
<tr>
<td>%G</td>
<td>Like g, except uses the shorter of f or E</td>
<td>3.1 or 1.86E6</td>
</tr>
</tbody>
</table>
## Format String 2

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Length Modifier</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int, unsigned short int</td>
<td>h</td>
<td>short int i = 3; printf( &quot;%hd&quot;, i );</td>
</tr>
<tr>
<td>long int or unsigned long int</td>
<td>l</td>
<td>long int i = 3; printf( &quot;%ld&quot;, i );</td>
</tr>
<tr>
<td>wide characters or strings</td>
<td>l</td>
<td>wchar_t* wide_str = L&quot;Wide String&quot;; printf( &quot;%ls&quot;, wide_str );</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
<td>long double d = 3.1415926535; printf( &quot;%Lg&quot;, d );</td>
</tr>
</tbody>
</table>
Example C program with simple I/O

Format Specifiers have a ton more details
Comments in C

• Single Line Comments:

  // this is a single line comment

• Multi Line Comments:

  /* this is a multi line comment */
Undefined Behavior

The C FAQ defines “undefined behavior” like this:

Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.
Undefined Behavior Example

As a quick example let’s take this program:

```c
#include <limits.h>
#include <stdio.h>

int main (void)
{
    printf ("%d\n", (2147483647+1) < 0);
    return 0;
}
```
## Undefined Behavior

### Allowed Results

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$ ./test</code></td>
<td>1</td>
</tr>
<tr>
<td><code>$ ./test</code></td>
<td>0</td>
</tr>
<tr>
<td><code>$ ./test</code></td>
<td>42</td>
</tr>
<tr>
<td>And this:</td>
<td></td>
</tr>
<tr>
<td><code>$ ./test</code></td>
<td>Formatting root partition, chomp chomp</td>
</tr>
</tbody>
</table>
See you in Next Lecture

• Read Chapter 1 in K&R (C Programming Language Book)
• Read more of K&R (Ch 2-7)
• Try out some examples on your own, understand what they do line by line
• **Start early on Assignment 0!**