CS 31: Introduction to Computer Systems

04: C Programming
January 30
Introduction to C programming

I could restructure the program's flow
or use one little 'goto' instead.

Eh, screw good practice. How bad can it be?
goto main_sub3;

*compile*

Dinosaur sitting on a chair:

Slide 2
Announcements

• Midterm Exam in Lecture Hall – SCI 199
• Please let me know your accommodations TODAY!
• Homework 1 is out
  – Due in class next Thursday (Feb 6th)
  – Can work in groups of up to 3
  – All HWs worth 3%
Reading Quiz
Today

• Basics of C programming
  – Comments, variables, print statements, loops, conditionals, etc.
  – Ask questions if you have them!

• Comparison of C vs. Python
  – Data organization and strings
  – Functions
# How a Computer Runs a Program

<table>
<thead>
<tr>
<th>Binary Program</th>
<th>How instructions &amp; data are encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Libraries</td>
<td>OS Abstractions, Resource management</td>
</tr>
<tr>
<td>Operating System</td>
<td>How underlying HW organized &amp; works</td>
</tr>
<tr>
<td>Computer Hardware</td>
<td></td>
</tr>
</tbody>
</table>

What we know so far:

- Binary Arithmetic, and overflow rules
  - Two’s complement Subtraction can be reframed as addition
- Know **encodings and sizes for different C types**
  - char, unsigned char, int, unsigned int, ...
- Know how to perform **binary operations** (Add, Sub)
  - Don’t yet know how the Add HW circuit works (next week)
A Main Course Goal:

Understand how program written in a high-level language is run on the underlying System (OS/HW)

Understand the details all the way down

C Program

main() {
    int x;
    x = 6 + 7;
    printf("x \%d", x);
}

Binary Program

Software Libraries

Operating System

Computer Hardware

gcc

0101011010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
What is C?

Dennis Ritchie worked at Bell Labs.

C was created for systems programming back in 1972.

C was created to write Unix.
Evolution of C – K&R, ANSI C

- **K&R** (Brian Kernighan & Dennis Richie)
  - Bell Labs
  - “The C Programming Language”, 1\textsuperscript{st} Edition (1978)

- **ANSI C** (American National Standards Institute)
  - Defines both language and standard C library

- **ISO C90** (International Standards Organization) $\cong$ ANSI C in 1990

- **ISO C99** (International Standards Organization)
  - New data types, moving C to 64 bit processors
  - Support for text strings with characters not in the English language

- **C18**: Current International Standard
Why C in this course?

• Did you ever see the wizard of Oz?
What was going on behind the curtains?
More than what you would think!
The mystery revealed!

What the...
So, why C in this course?

- **Closed Curtains**
  - A safe place for programmers!
  - Python hides certain aspects of reality
  - This is good!

- **Behind The Curtains Programming**
  - Most Operating Systems (OS)
  - Access to memory and memory management
  - Dangerous, but necessary
  - Important to understand how the real system works
  - Makes you a **better** programmer!
Python and C follow different programming paradigms.

- **C:**
  - is more procedure-oriented.
  - breaks down to functions.
- **Python**
  - follows an object-oriented paradigm.
  - allows Python to break down Objects.
Python versus C: Paradigms

- Python is an “interpreted” language: interpreter does lexing, parsing, compiling and interpreting!
- Python does not allow you to access memory directly.
- Python provides automatic memory management. It is a “garbage collected” language.
- Python types “expand to available memory”.
- Python provides “exceptions”: if your program has an error at run-time it will throw an exception.

- C is compiled language: compiles to machine code that is “executed” by the underlying processor.
- C allows you to access memory directly, interpret that memory in any way you wish, and shoot yourself in the foot.
- C provides manual memory management.
- Types in C are dependent on the machine you are running on.
- No built-in error handling, if you are lucky (and smart) your program will check “error conditions” and fail gracefully.
Intro to C Programming

• is closer to the machine: see relationship between C code and computer execution
• can write faster code!
• want abstraction? out of luck – DIY implementation of dictionary
• It gives you access to aspects of the machine that are not accessible in Python.
So, the point(er) is….?

• Programming Languages Are Tools
  – Python is one language and it does its job well
  – C is another language and it does its job well

• Pick The Right Tool for the Job
  – C is a good language to explore how the system works under-the-hood.
  – C is the Language of Systems Programmers: Fast running OS code is really really really important!

• It is the right tool for the job we need to accomplish in this course!
GNU Compiler Collection

$gcc -g -o prog prog.c
GNU Compiler Collection

```
$gcc -g -o prog prog.c
```

compile programs using the GNU C Compiler
GNU Compiler Collection

$gcc -g -o prog prog.c

compile programs using the GNU C Compiler

Command line options
-g: generates debugging information
-o: build to output file prog
## Hello World

<table>
<thead>
<tr>
<th><strong>Python</strong></th>
<th><strong>C</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code># hello world</code></td>
<td><code>// hello world</code></td>
</tr>
<tr>
<td><code>import math</code></td>
<td><code>#include &lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td><code>def main():</code></td>
<td><code>int main( ) {</code></td>
</tr>
<tr>
<td><code>    print &quot;hello world&quot;</code></td>
<td><code>    printf(&quot;hello world\n&quot;);</code></td>
</tr>
<tr>
<td><code>main()</code></td>
<td><code>    return 0;</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
## Hello World

### Python

```python
# hello world
import math

def main():
    print "hello world"

main()
```

### C

```c
// hello world
#include <stdio.h>

int main() {
    printf("hello world\n");
    return 0;
}
```

- `#:` single line comment
- `//:` single line comment
## Hello World

### Python

```python
# hello world
import math

def main():
    print "hello world"

main()
```

### C

```c
#include <stdio.h>

int main() {
    printf("hello world\n");
    return 0;
}
```

<table>
<thead>
<tr>
<th>Python</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td># hello world</td>
<td>// hello world</td>
</tr>
<tr>
<td>import math</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td>def main():</td>
<td>int main() {</td>
</tr>
<tr>
<td>print &quot;hello world&quot;</td>
<td>printf(&quot;hello world\n&quot;);</td>
</tr>
<tr>
<td>main()</td>
<td>return 0;</td>
</tr>
<tr>
<td>#: single line comment</td>
<td>//: single line comment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>import libname: include Python libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include&lt;libname&gt;: include C libraries</td>
</tr>
</tbody>
</table>
# Hello World

## Python

```python
# hello world
import math

def main():
    print "hello world"

main()
```

## C

```c
// hello world
#include <stdio.h>

int main( ) {
    printf("hello world\n");
    return 0;
}
```

### Single Line Comment

- Python: `#` (single line comment)
- C: `//` (single line comment)

### Include Libraries

- Python: `import libname`: include Python libraries
- C: `#include<libname>`: include C libraries

### Blocks: Indentation

- Python
- C: `{ }` (indentation for readability)
“White Space”

• Python cares about how your program is formatted. Spacing has meaning.

• **C compiler does NOT care.** Spacing is ignored.
  – This includes spaces, tabs, new lines, etc.
  – **Good practice (for your own sanity):**
    • Put each statement on a separate line.
    • Keep indentation consistent within blocks.
Are these the same program?

```c
#include <stdio.h>

int main() {
    int number = 7;
    if (number > 10) {
        do_this();
    } else {
        do_that();
    }
}
```

```c
#include <stdio.h>

int main() {
    int number = 7;
    if (number > 10) {
        do_this();
    } else {
        do_that();
    }
}
```

Yes – but one is harder to debug than the other
## Hello World

<table>
<thead>
<tr>
<th>Python</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># hello world</code></td>
<td><code>// hello world</code></td>
</tr>
</tbody>
</table>
| `import math` | `
#include <stdio.h>` |
| `def main():` | `int main( ) {` |
| `    print "hello world"` | `    printf("hello world\n");` |
| `main()` | `    return 0;` |
| `#: single line comment` | `#: single line comment` |
| `import libname:` | `
#include<libname>` |
| `include Python libraries` | `include C libraries` |
| Blocks: **indentation** | Blocks: `{   }` (indentation for readability) |
| `print:` | `printf:` |
| `statement to printout string` | `function to print out format string` |
| `statement:` | `statement:` |
| `each on separate line` | `each ends with ;` |
| `def main():` | `int main( ):` |
| `the main function definition` | `the main function definition` |
| | `(int specifies the **return type** of main)` |
Types

- Everything is stored as bits.

- Type tells us how to interpret those bits.

- “What type of data is it?”
  - integer, floating point, text, etc.
Representation Matters!

• No self-identifying data  
  – Looking at a sequence of bits doesn’t tell you what they mean  
  – Could be signed, unsigned integer  
  – Could be floating-point number  
  – Could be part of a string  

• The machine interprets what those bits mean!
Types in C

• All variables have an explicit type!

• You (programmer) must declare variable types.
  – Where: at the beginning of a block, before use.
  – How: <variable type> <variable name>;

• Examples:
  
  int humidity;       float temperature;
  humidity = 20;      temperature = 32.5
We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn’t C?
We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn’t C?

A. C is old.
B. Explicit type declaration is more efficient.
C. Explicit type declaration is less error prone.
D. Dynamic typing (what Python does) is imperfect.
E. Some other reason (explain)
We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn’t C?

A. C is old (maybe?)
B. Explicit type declaration is more efficient.
C. Explicit type declaration is less error prone.
D. Dynamic typing (what Python does) is imperfect.
E. Some other reason (explain)
### Numerical Type Comparison

#### Integers (int)
- Example:
  ```
  int humidity;
  humidity = 20;
  ```
- Only represents integers
- Small range, high precision
- Faster arithmetic
- (Maybe) less space required

#### Floating Point (float, double)
- Example:
  ```
  float temperature;
  temperature = 32.5;
  ```
- Represents fractional values
- Large range, less precision
- Slower arithmetic

I need a variable to store a number, which type should I use?

**Use the one that fits your specific need best...**
An Example with Local Variables

/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h>  // C's standard I/O library (for printf)

int main() {
   // first: declare main’s local variables
   int x, y;
   float z;

   // followed by: main function statements
   x = 6;
   y = (x + 3)/2;
   z = x;
   z = (z + 3)/2;

   printf(...)  // Print x, y, z
}

Slide 56
An Example with Local Variables

/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h> // C’s standard I/O library (for printf)

int main() {
   // first: declare main’s local variables
   int x, y;
   float z;

   // followed by: main function statements
   x = 6;
   y = (x + 3)/2;
   z = x;
   z = (z + 3)/2;

   printf(...) // Print x, y, z
}

Clicker choices

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>4.5</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>4.5</td>
<td>4.5</td>
</tr>
</tbody>
</table>
An Example with Local Variables

/* a multiline comment:
   anything between slashdot and dotslash */

#include <stdio.h> // C’s standard I/O library (for printf)

int main() {
  // first: declare main’s local variables
  int x, y;
  float z;

  // followed by: main function statements
  x = 6;
  y = (x + 3)/2;
  z = x;
  z = (z + 3)/2;

  printf(...); // Print x, y, z
}

Clicker choices

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>4.5</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>4.5</td>
<td>4.5</td>
</tr>
</tbody>
</table>
Operators: need to think about type

**Arithmetic**: +, -, *, /, % (numeric type operands)

/: operation and result type depends on operand types:

- Two **int** operands: int division truncates: **3/2 is 1**
- 1 or 2 **float or double**: float or double division: **3.0/2 is 1.5**

%: mod operator: (only int or unsigned types)

- Gives you the (integer) remainder of division: **13 % 2 is 1, 27 % 3 is 0**
- Shorthand operators:
  - `var op = expr;` (var = var op expr):
  - `x += 4` is equivalent to `x = x + 4`
  - `var++; var--;` (var = var+1; var = var-1):
    - `x++` is same as `x = x + 1`
    - `x--` is same as `x = x - 1;`
Boolean values in C?

• There is no “boolean” type in C!

• Instead, integer expressions used in conditional statements are interpreted as true or false

• Zero (0) is false, any non-zero value is true
  – Use this to always check return value of the function

• Questions?
• “Which non-zero value does it use?
• E.g., int x = 10>5. what is x?

  The value of x is compiler specific don’t rely on the output to be a certain value
Operators: need to think about type

• **Relational** (operands any type, result int “boolean”):
  • <, <=, >, >=, ==, !=
  • 6 != (4+2) is 0 (false)
  • 6 > 3 some non-zero value (we don’t care which one) (true)

• **Logical** (operands int “boolean”, result int “boolean”):
  • ! (not): !6 is 0 (false)
  • && (and): 8 && 0 is 0 (false)
  • || (or): 8 || 0 is non-zero (true)
Boolean values in C

- Zero (0) is **false**, any non-zero value is **true**
- **Logical** (operands int “boolean”->result int “boolean”):
  - ! (not): inverts truth value
  - && (and): true if both operands are true
  - || (or): true if either operand is true

Do the following statements evaluate to True or False?

**#1:** (!10) || (5 > 2)

**#2:** (-1) && (!5) > -1

<table>
<thead>
<tr>
<th></th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>D</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Boolean values in C

• Zero (0) is **false**, any non-zero value is **true**

• **Logical** (operands int “boolean”->result int “boolean”):
  - `!` (not): inverts truth value
  - `&&` (and): true if both operands are true
  - `||` (or): true if either operand is true

---

Do the following statements evaluate to True or False?

<table>
<thead>
<tr>
<th></th>
<th>#1</th>
<th>#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>D</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

#1: `(!10) || (5 > 2)`

#2: `(-1) && ((!5) > -1)`
Conditional Statements

<table>
<thead>
<tr>
<th>Basic if statement:</th>
<th>With optional else:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if(&lt;boolean expr&gt;) {</code></td>
<td><code>if(&lt;boolean expr&gt;) {</code></td>
</tr>
<tr>
<td>if-true-body</td>
<td>if-true-body</td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>else {</code></td>
</tr>
<tr>
<td></td>
<td>else body(expr=false)</td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Very similar to Python, just remember `{ }` are blocks, no curly braces, only the next line will be executed! **Always use curly braces.**
Conditional Statements

<table>
<thead>
<tr>
<th>Chaining if-else if</th>
<th>With optional else:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if(&lt;boolean expr1&gt;) {</code></td>
<td><code>if(&lt;boolean expr1&gt;) {</code></td>
</tr>
<tr>
<td><code>if:expr1-true-body</code></td>
<td><code>if:expr1-true-body</code></td>
</tr>
<tr>
<td><code>} else if (&lt;bool expr2&gt;)</code></td>
<td><code>} else if (&lt;bool expr2&gt;)</code></td>
</tr>
<tr>
<td><code>else-if:expr2-true-body</code></td>
<td><code>else-if:expr2-true-body</code></td>
</tr>
<tr>
<td><code>(expr1 false)</code></td>
<td><code>(expr1 false)</code></td>
</tr>
</tbody>
</table>
| `}`                                       | `}
| ...                                       | `...
| `} else if (<bool exprN>)`             | `} else if (<bool exprN>)`
|   `else-if:exprN-true-body`             |     `else-if:exprN-true-body`                                 |
| `}`                                       | `}

|                                 | `else {`                                                        |
|                                 |     `else body`                                                 |
| `}`                                 |     `(all exprX’s false)`                                      |

Very similar to Python, just remember {} are blocks
While Loops

• Basically identical to Python while loops:

```java
while(<boolean expr>) {
    while-expr-true-body
}
```

```java
x = 20;
while (x < 100) {
    y = y + x;
    x += 4;  //  x = x + 4;
}<next stmt after loop>;
```

```java
x = 20;
while(1) {  // while true
    y = y + x;
    x += 4;
    if(x >= 100) {
        break;  // break out of loop
    }
}<next stmt after loop>;
```
For loops: different than Python’s

for (<init>; <cond>; <step>) {
    for-loop-body-statements
}

<next stmt after loop>;

1. **Evaluate `<init>` one time**, when first eval `for` statement
2. Evaluate `<cond>`, if it is false, drop out of the loop (`<next stmt after>`)
3. Evaluate the statements in the for loop body
4. Evaluate `<step>`
5. **Goto step (2)**

```c
for(i=1; i <= 10; i++) {
    // example for loop
    printf("%d\n", i*i);
}
```
printf function

Python:  print "%d %s\t%f" % (6, "hello", 3.4)

C: printf("%d %s\t%f\n", 6, "hello", 3.4);

printf(<format string>, <values list>);

<table>
<thead>
<tr>
<th>%d</th>
<th>int placeholder (-13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f or %g</td>
<td>float or double (higher-precision than float) placeholder (9.6)</td>
</tr>
<tr>
<td>%c</td>
<td>char placeholder  (‘a’)</td>
</tr>
<tr>
<td>%s</td>
<td>string placeholder  (&quot;hello there&quot;)</td>
</tr>
<tr>
<td>\t \n</td>
<td>tab character, new line character</td>
</tr>
</tbody>
</table>

Formatting Differences:

C: need to explicitly print end-of-line character (\n)

‘a’: in Python is a string, in C is a char
“a”: in Python is a string, in C is a string

C: string and char are different types
Data Collections in C

- Many complex data types out there (CS 35)

- C has a few simple ones built-in:
  - Arrays
  - Structures (struct)
  - Strings (arrays of characters)

- Often combined in practice, e.g.:
  - An array of structs
  - A struct containing strings
Arrays

- C’s support for **collections of values**
  - Array buckets store a single type of value
  - **Specify max capacity** (num buckets) when you declare an array variable (single memory chunk)

```c
<type> <var_name>[<num_buckets>];
int arr[5];  // an array of 5 integers
float rates[40]; // an array of 40 floats
```
Arrays

- C’s support for collections of values
- Often accessed via a loop:

```c
int arr[5]; // an array of 5 integers
float rates[40]; // an array of 40 floats
for (i=0; i < 5; i++) {
    arr[i] = i;
    rates[i] = (arr[i]*1.387)/4;
}
```

Get/Set value using brackets [] to index into array.
Array Characteristics

```c
int january_temps[31];  // Daily high temps
```

- Indices start at 0! Why? Computing the offset from the start of the array

```
[0] [1] [2] [3] [4]  ...  [29][30]
```

↑ Array bucket indices.  ↑

- `january_temps`` (without brackets!) Location of [0] in memory.
Array Characteristics

```c
int january_temps[31]; // Daily high temps
```

- **Indices start at 0! Why?**
- **Array variable name means, to the compiler, the beginning of the memory chunk. (address)**
- Keep this in mind, we’ll return to it soon (functions).
- Index number is an offset from beginning.
Given what we know about arrays, how can we add a temperature reading second element in the array?

```c
int january_temps[31]; // Daily high temps
```

“january_temps”

**Location of [0] in memory.**

1. `scanf("%d", january_temps);`
2. `scanf("%d", &january_temps[1]);`
3. None of the above
Given what we know about arrays, how can we add a temperature reading second element in the array?

```c
int january_temps[31];  // Daily high temps
```

```
"january_temps"
```

**Location of [0] in memory.**

1. `scanf("%d", january_temps);`
2. `scanf("%d", &january_temps[1]);`
3. None of the above
Array Characteristics

```
int january_temps[31]; // Daily high temps
```

Location of [0] in memory.

- Array variable name means, to the compiler, the beginning of the memory chunk. (address)
- Index number is an offset from beginning.

C does NOT do bounds checking. Asking for `january_temps[35]`?
- Python: error
- C: “Sure! I don’t care”
Chars and Strings

A character (type `char`) is a numerical value that holds one letter.

```
char my_letter = 'w';  // Note: single quotes
```

What is the numerical value?
- `printf("%d  %c", my_letter, my_letter);`
- Would print: 119  w

Why is ‘w’ equal to 119?
- ASCII Standard says so.
- American Standard Code for Information Interchange
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audible bell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
</tbody>
</table>

119 = w
Characters and Strings

- A character (type `char`) is a numerical value that holds one letter.
- A string is a memory block containing characters, one after another...
- Examples:
  
  ```c
  char food[6] = "Pizza";
  ```

Hmm, suppose we used `printf` and `%s` to print name.

How does it know where the string ends and other memory begins?
How can we tell where a string ends?

A. Mark the end of the string with a special character.

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it’s contained within (e.g., char name[20] must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).
How can we tell where a string ends?

A. Mark the end of the string with a special character. (what we do in C)

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it’s contained within (e.g., char name[20] must be of length 20) –
   • inconvenient and inflexible

D. All of these could work (technically true)

E. Some other mechanism (such as?).
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
</tbody>
</table>

0 = Null

Special stuff over here in the lower values.
Characters and Strings

- A character (type `char`) is a numerical value that holds one letter.
- A string is a memory block containing characters, one after another, with a null terminator (numerical 0) at the end.
- Examples:
  ```
  char name[20] = “Pizza”;
  ```

![Diagram showing string storage]
Strings in C

• C String library functions: `#include <string.h>`
  – Common functions (strlen, strcpy, etc.) make strings easier
  – Less friendly than Python strings

• More on strings later, in labs.

• For now, remember about strings:
  – Allocate enough space for null terminator!
  – If you’re modifying a character array (string), don’t forget to set the null terminator!
  – If you see crazy, unpredictable behavior with strings, check these two things!
Functions: Specifying Types

specify the **return type** of the function, and the **type of each parameter**

```c
<return type> <func name> ( <param list> ) {
    // declare local variables first
    // then function statements
    return <expression>;
}
```

```c
// my_function takes 2 int values and returns an int
int my_function(int x, int y) {
    int result;
    result = x;
    if(y > x) {
        result = y+5;
    }
    return result*2;
}
```

Compiler will yell at you if you try to pass the wrong type!
Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

```
Stack
main:
  x: 4
  y: 7
```
• Arguments are **passed by value**
  – The function gets a separate copy of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

**Stack**

<table>
<thead>
<tr>
<th>main:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 4</td>
</tr>
<tr>
<td>y: 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>func:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 9</td>
</tr>
<tr>
<td>b: 7</td>
</tr>
</tbody>
</table>

Note: This doesn’t change!

**no impact on values in main!**
Function Arguments

• Arguments are **passed by value**
  — The function gets a separate **copy** of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

**Output:** 4, 2
What will this print?

```c
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);

    printf("%d, %d, %d", x, values[0], values[1]);
}
```

A. 0, 5, 8
B. 0, 5, 10
C. 1, 0, 8
D. 1, 5, 8
E. 1, 5, 10

Hint: What does the name of an array mean to the compiler?
What will this print?

```c
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];
    x = 0;
    values[0] = 5;
    values[1] = 10;
    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
```

A. 0, 5, 8  
B. 0, 5, 10  
C. 1, 0, 8  
D. 1, 5, 8  
E. 1, 5, 10

Hint: Still accessing the same memory location of array in `func`
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
What will this print?

int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
What will this print?

```c
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
```

The program will print `0, 5, 10`. The diagram shows the stack and the state of variables after the function call `func(x, x, values)`. The values `0` and `1` are pushed onto the stack, and the function modifies them, leaving `0` and `1` on the stack. Finally, these values are printed without modification.
What will this print?

```c
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
```

The output will be:
```
0, 5, 8
```
What will this print?

```c
int func(int a, int y, int my_array[]) {
    y = 1;
    my_array[a] = 0;
    my_array[y] = 8;
    return y;
}

int main() {
    int x;
    int values[2];

    x = 0;
    values[0] = 5;
    values[1] = 10;

    x = func(x, x, values);
    printf("%d, %d, %d", x, values[0], values[1]);
}
```
structs

• Treat a collection of values as a single type:
  – C is not an object oriented language, no classes
  – A struct is like just the data part of a class

• Rules:
  1. Define a new struct type outside of any function
  2. Declare variables of the new struct type
  3. Use dot notation to access the different field values of the struct variable
Suppose we want to represent a **student** type.

```c
struct student {
    char name[20];
    int grad_year;
    float gpa;
};
// Variable bob is of type struct student
struct student  bob;
// Set name (string) with strcpy()
strcpy(bob.name, “Robert Paulson”);
bob.grad_year = 2019;
bob.gpa = 3.1;

printf(“Name: %s, year: %d, GPA: %f”, bob.name, bob.grad_year, bob.gpa);
```
Arrays of Structs

```c
struct student {
  char name[20];
  int grad_year;
  float gpa;
};
//create an array of struct students!
struct student classroom[50];

strcpy(classroom[0].name, "Alice");
classroom[0].grad_year = 2014
classroom[0].gpa = 4.0;

// With a loop, create an army of Alice clones!
int i;
for (i = 0; i < 50; i++) {
  strcpy(classroom[i].name, "Alice");
classroom[i].grad_year = 2014;
classroom[i].gpa = 4.0;
}
```
Arrays of Structs

```c
struct student classroom[50];

strcpy(classroom[0].name, "Alice");
classroom[0].grad_year = 2019;
classroom[0].gpa = 4.0;

strcpy(classroom[1].name, "Bob");
classroom[1].grad_year = 2020;
classroom[1].gpa = 3.1

strcpy(classroom[2].name, "Cat");
classroom[2].grad_year = 2021;
classroom[2].gpa = 3.4
```
# Struct: Layout in Memory

**classroom:**

|   | A | l | i | c | e | \0 | ... | B | o | b | \0 | ... | C | a | t | \0 | ... |
|---|---|---|---|---|---|----|-----|---|---|---|----|-----|---|---|---|----|-----|-----|
| [0]| 2019 | 4.0 |   |   |   | ... | 2020 | 3.1 |   |   | \0 | ... | 2021 |   |   |   | \0 | ... |
| [1]|   |   |   |   |   |   |     |   |   |   |   |     |   |   |   |   |   |   |
| [2]|   |   |   |   |   |   |     |   |   |   |   |     |   |   |   |   |   |   |

Slide 100
Fear not!

• Don’t worry, I don’t expect you to have mastered C.
• It’s a skill you’ll pick up as you go.
• We’ll revisit these topics when necessary.

• When in doubt: solve the problem in English, whiteboard pictures, whatever else!
  – Translate to C later.
  – Eventually, you’ll start to think in C.
Up next…

• Digital circuits