CS31: Introduction to Computer Systems

Week 1, Class 2
Introduction to C Programming
01/25/24

Dr. Sukrit Venkatagiri
Swarthmore College
Where are we?

C (programming language)

Compiled

x86 Assembly (instruction set architecture)

Assembled

Binary (logic / bits)

CPU / memory (logic / bits)

Logic gates, circuits (voltage)
### Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>”</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>”</td>
</tr>
<tr>
<td></td>
<td><strong>Spring Break</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>”</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>”</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>”</td>
</tr>
</tbody>
</table>

**Diagram:**

- **C**: Programming language
- **x86 Assembly**: Instruction set architecture
- **Binary**: Logic / bits
- **CPU / memory**: Logic / bits
- **logic gates, circuits**: Voltage

The diagram illustrates the relationship between different levels of abstraction in computer programming and hardware. The process begins with **C**, a high-level programming language, which is compiled into **x86 Assembly**, an assembly language that is closer to the hardware level. **Binary** represents the logic at the bit level, and **CPU / memory** deals with the interaction between the processor and memory. The **logic gates, circuits**, which are the fundamental building blocks of digital electronics, operate at the lowest level, using **voltage** to represent logical states.
### Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Spring Break</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- **C**: Programming language
- **x86 Assembly**: Instruction set architecture
- **Binary**: Logic / bits
- **CPU / memory**: Logic / bits
- **logic gates, circuits**: Voltage
<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

Diagram:

```
C  \(\text{compiled}\)  x86 Assembly  \(\text{assembled}\)  Binary  CPU / memory  logic / bits  logic / bits  logic gates, circuits  voltage
```
Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td>Spring Break</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

C


compiled

x86 Assembly

assembled

Binary

CPU / memory

logic / bits

logic / bits

logic gates, circuits

voltage

programming language

instruction set architecture

logic / bits

voltage
### Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td><strong>Spring Break</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

---

![Diagram](attachment:image.png)

- **C**: programming language
- **x86 Assembly**: instruction set architecture
- **Binary**: logic / bits
- **CPU / memory**: logic / bits
- **logic gates, circuits**: voltage
<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td><strong>Spring Break</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

Where are we?

- C: programming language
- x86 Assembly: instruction set architecture
- Binary: logic / bits
- CPU / memory: logic / bits
- logic gates, circuits: voltage

Diagram:

- `C`
  - `compiled`
    - `x86 Assembly`
      - `assembled`
        - `Binary`
          - `CPU / memory`
            - `logic gates, circuits`
              - `voltage`
Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td></td>
</tr>
</tbody>
</table>

C:
- programming language

x86 Assembly:
- instruction set architecture

Binary:
- logic / bits

CPU / memory:
- logic / bits

logic gates, circuits:

OS:
- voltage
## Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Intro to C</strong></td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

### Diagram:

- **C**: compiled programming language
- **x86 Assembly**: instruction set architecture
- **Binary**: logic / bits
- **CPU / memory**: logic / bits
- **logic gates, circuits**: voltage
CS31: Introduction to Computer Systems

Week 1, Class 2
Introduction to C Programming
01/25/24

Dr. Sukrit Venkatagiri
Swarthmore College
Agenda

• Basics of C programming
  • Comments, variables, print statements, loops, conditionals, etc.
  • NOT the focus of this course
  • Ask questions if you have them!

• Comparison of C vs. Python
  • Data organization and strings
  • Functions
The First “Computers”: Women

ENIAC was developed 10 mi from here, at UPenn

Fun Fact

ENIAC was developed 10 mi from here, at UPenn

History

When Computer Coding Was a 'Woman's' Job | HISTORY

Computer programming used to be a 'pink ghetto'—so it was underpaid and undervalued.
What is C?

Dennis Ritchie worked at Bell Labs. C is a programming language created for systems programming back in 1972. C was created to write Unix. C was created to write Unix. C was created for systems programming back in 1972. First transistor, solar cell, compilers, C, C++, Unix, deep learning, + more!
Why C in this course?

Have you watched the Wizard of Oz?
What was going on behind the curtains?
More than what you would think!
The mystery revealed!
Python versus C: Paradigms

Python and C follow different programming paradigms.

- **C:**
  - is procedure-oriented
  - breaks down to functions

- **Python:**
  - follows an object-oriented paradigm (as do C++ and Java)
  - allows Python to break down objects and methods
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 that exposes the details of the hardware is really important!

- It’s the right tool for the job we need to accomplish in this course!
## 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;
}
```
## Hello World

<table>
<thead>
<tr>
<th>Python</th>
<th>C</th>
</tr>
</thead>
</table>
| # hello world  
import math  
def main():  
    print "hello world"
main()        |
| #: single line comment |
|              | //: single line comment |

```python
# hello world
import math

def main():
    print "hello world"
main()
```

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

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

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

<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>
<tr>
<td>import libname:</td>
<td>#include&lt;libname&gt;:</td>
</tr>
<tr>
<td>include Python libraries</td>
<td>include C libraries</td>
</tr>
<tr>
<td>Blocks: indentation</td>
<td>Blocks: { } (indent for readability)</td>
</tr>
</tbody>
</table>
To Blank Space or Not to Blank 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.
# 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;
}
```

<table>
<thead>
<tr>
<th>Python</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#</code> single line comment</td>
<td><code>//</code> single line comment</td>
</tr>
<tr>
<td><code>import libname:</code> include Python lib.</td>
<td><code>#include&lt;libname&gt;</code>: include C libraries</td>
</tr>
<tr>
<td>Blocks: indentation</td>
<td>Blocks: <code>{ }</code> (indent for readability)</td>
</tr>
<tr>
<td><code>print:</code> statement to printout string</td>
<td><code>printf:</code> function to print out format string</td>
</tr>
<tr>
<td>statement: each on separate line</td>
<td>statement: each ends with <code>;</code></td>
</tr>
<tr>
<td><code>def main():</code> : the main function definition</td>
<td><code>int main( ) :</code> the main function definition (int specifies the return type of main)</td>
</tr>
</tbody>
</table>
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.
Type 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:
  ```c
  int humidity;
  float temperature;
  humidity = 20;
  temperature = 32.5
  ```
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...
Operators: consider the type

• **Arithmetic:** +, -, *, /, % (numeric type operands)
  
  /: operation and result type depends on operand types:
  - Two int operands: int division truncates the result → 3/2 is 1
  - One or two float or double operands: floating-point 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
Operators: consider the type

- **Shorthand operators:**
  - var = var \textcolor{red}{op} expr; \quad \textcolor{red}{\rightarrow} \quad var \textcolor{red}{op}= expr;
    \begin{align*}
    x & += 4 \quad \text{is equivalent to} \quad x = x + 4 \\
    \end{align*}
  - \textcolor{red}{\text{int}} y = 4; \\
  - y *= 2; \quad \text{What is the value of } y?
  - var = var+1; \quad \text{\textcolor{red}{\rightarrow}} \quad var++;
  - var = var+1; \quad \text{\textcolor{red}{\rightarrow}} \quad var--;
  - x++ \quad \text{is same as} \quad x = x + 1 \quad \text{\textcolor{red}{x--} \quad is same as} \quad x = x - 1;
  - ++x \quad \text{and} \quad -x \quad \text{are different from} \quad x++ \quad \text{and} \quad x-- \quad (\text{we’ll talk about this later})
Boolean (true/false) values in C

• There is no “boolean” type in C!

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

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

• Questions?

• “Which non-zero value does it use?”
Operators: consider the type

- **Relational** (operands any type, result integer “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 integer “boolean”):
  - `!` (not): `!6` is 0 (false)
  - `&&` (and): `8 && 0` is 0 (false)
  - `||` (or): `8 || 0` is non-zero (true)
## Conditional Statements

### Basic if statement:

<table>
<thead>
<tr>
<th>Conditional Statements</th>
<th>With optional else:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If</strong> (&lt;boolean expr&gt;) {</td>
<td></td>
</tr>
<tr>
<td>if-true-body</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong> (&lt;boolean expr&gt;) {</td>
<td></td>
</tr>
<tr>
<td>if-true-body</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>else</strong> {</td>
<td></td>
</tr>
<tr>
<td>else body(expr-false)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

### Chaining if-else if

<table>
<thead>
<tr>
<th>Conditional Statements</th>
<th>With optional else:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong> (&lt;boolean expr1&gt;) {</td>
<td></td>
</tr>
<tr>
<td>if-expr1-true-body</td>
<td></td>
</tr>
<tr>
<td>} else if (&lt;bool expr2&gt;) {</td>
<td></td>
</tr>
<tr>
<td>else-if-expr2-true-body</td>
<td></td>
</tr>
<tr>
<td>(expr1 false)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong> (&lt;boolean expr1&gt;) {</td>
<td></td>
</tr>
<tr>
<td>if-expr1-true-body</td>
<td></td>
</tr>
<tr>
<td>} else if (&lt;bool expr2&gt;) {</td>
<td></td>
</tr>
<tr>
<td>else-if-expr2-true-body</td>
<td></td>
</tr>
<tr>
<td>(expr1 false)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>...</strong></td>
<td></td>
</tr>
<tr>
<td><strong>else if</strong> (&lt;bool exprN&gt;) {</td>
<td></td>
</tr>
<tr>
<td>else-if-exprN-true-body</td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td>else body</td>
<td></td>
</tr>
<tr>
<td>(all exprX’s false)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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;
}

x = 20;
while (1) { // while true
    y = y + x;
    x += 4;
    if (x >= 100) {
        break; // break out of loop
    }
}
```

```java
<next stmt after loop>
```

<next stmt after loop>
For loops: different than Python’s

for (<initialize>; <condition>; <step>) {
    for-loop-body-statements
}

<nex stmt after loop>;

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

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

• Similar to Python’s formatted print statement, with a few differences:
  • C: need to explicitly print end-of-line character (\n)
  • C: **string and char are different types**
    ‘a’: in Python is a string, in C is a (single) char
    “a”: in Python is a string, in C is a **string**

Python: print “%d %s\t%f” % (6, “hello”, 3.4)
C: printf(“%d %s\t%f\n”, 6, “hello”, 3.4);

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>int placeholder (-13)</td>
</tr>
<tr>
<td>%f or %g</td>
<td>float or double 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>
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)

```
<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]*1.5)/4;
}
```

Get/Set value using brackets [] to index into array.
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. (The memory address)
  – january_temps” (without brackets!) Location of january_temps[0] in memory.
  – Keep this in mind, we’ll return to it soon (functions).
Array Characteristics

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

- Indices start at 0! Why?
- The index refers to an offset from the start of the array
  - e.g., [3] means “three integers forward from the starting address”

Location of [0] in memory.
Array Characteristics

int january_temps[31]; // Daily high temps

“january_temps”
Location of [0] in memory.

[0] [1] [2] [3] [4] ... [29][30] [35]
Array bucket indices.

• Asking for january_temps[35]?

C does NOT do bounds checking.
• Python: error
• C: “Sure! I don’t care ..” <ominous silence while bad things happen>
Your TODO List

• **Now**: Submit partner survey

• **Now**: Buy an iClicker

• **Before lab tomorrow**: Complete Lab 0

• **By 11:59pm Thursday**: Lab 1 is due

• **By 11:59pm Friday**: Complete HW1, submit to gradescope

• **The next 13 weeks**: Read the readings before class
Characters and Strings

• A character (type `char`) is 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?
  – American Standard Code for Information Interchange (ASCII) standard says so.
<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>@</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 transmission</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>Vertical tab</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>Form feed</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>Carriage return</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>Shift out</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 in</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>Data link escape</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>Device control 1</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 2</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 3</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 4</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 5</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>Device control 6</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 acknowledge</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 of transmission</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>0</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>

$ man ascii

119 = w
Characters and Strings

• A character (type `char`) is 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?

```
name: Pizza[6]
```

```c
[0][1][2][3][4]
```
0 is the “Null character”

Special stuff over here in the lower values.

<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>@</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td></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></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></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></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></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></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></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></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></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 tab</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>0</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>

Characters and Strings

$ man ascii
Characters and Strings

• A character (type char) is 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”;

```
P  i  z  z  a  \0  ...  
[0] [1] [2] [3] [4] [5] [6] [7] [18][19]
```
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!
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 field values of a 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

```
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:**

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 'A' | 'l' | 'i' | 'c' | 'e' | \0 | ... | 'B' | 'o' | 'b' | \0 | ... | 'C' | 'a' | 't' | \0 ...
| 2019 |   |   |   |   |   |   | 2020 |   |   |   |   |   | 2021 |   |   |
| 4.0 |   |   |   |   |   |   | 3.1 |   |   |   |   |   | 3.4 |   |   |
Functions: Specifying Types

Need to specify the return type of the function, and the type of each parameter:

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

// 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);
}
```
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
```
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

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

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

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

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);
}
```

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);
}
```

Stack

```
main:
  x: 4
  y: 2
```

```
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**
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 logically, use a whiteboard, whatever else!
  – Translate to C later
  – Eventually, you’ll start to “think in C”
Up next…

• Bits, Bytes, Binary (data representation)