CS 31: Intro to Systems
C Programming

Martin Gagné
Swarthmore College
January 26, 2017
Agenda

• 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
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…
What values will we see for x, y, and z?

/* a multiline comment:
   anything between slash-star and star-slash */
#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
}
Operators: need to think about type

• **Arithmetic**: +, -, *, /, % (numeric type operands)
  
  /: operation & result type depends on operand types:
  
  • 2 int ops: 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;`
Operators: need to think about type

- **Relational** (operands any type, result int “boolean”):
  - $<$, $<=$, $>$, $>=$, $==$，！=
  - $6 \neq (4+2)$ 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

• 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

• Questions?

• “Which non-zero value does it use?”
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

### Clicker choices

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

**Basic if statement:**

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

**With optional else:**

```
if (<boolean expr) {
  if-true-body
} else {
  else body(expr-false)
}
```

**Chaining if-else if**

```
if (<boolean expr1>) {
  if-expr1-true-body
} else if (<bool expr2>) {
  else-if-expr2-true-body (expr1 false)
} ...
} else if (<bool exprN>) {
  else-if-exprN-true-body
}
```

**With optional else:**

```
if (<boolean expr1>) {
  if-expr1-true-body
} else if (<bool expr2>) {
  else-if-expr2-true-body (expr1 false)
} ...
} else if (<bool exprN>) {
  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>;
```
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>;
```
While Loops

- Basically identical to Python while loops:

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

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

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

Avoid this whenever possible
For loops: different than Python’s

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

<nest 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)

for(i=1; i <= 10; i++) {  // example for loop
    printf("%d\n", i*i);
}
What can/can’t while loops and for loops do in C?

a) Anything you can compute with a C while loop (and more) can be computed with a C for loop.

b) Anything you can compute with a C for loop (and more) can be computed with a C while loop.

c) C’s while loops and for loops can perform completely disjoint sets of computations.

d) C’s while loops and for loops can perform partially overlapping sets of computations.

e) C’s while loops and for loops can perform exactly the same computations.
while loop → for loop

while(condition) {
    body;
}

for(;condition;;) {
    body;
}

for loop → while loop

for(init; cond; step) {
    body;
}

init;
while(cond) {
    body;
    step;
}
Data Collections in C

• Many complex data types out there (CS 35)

• C has a few simple ones built-in:
  – Arrays
  – Structures (struct)
  – Strings

• 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
  – Need to 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`
    
  – Often accessed via a loop:
    
    `for(i=0; i < 5; i++) {
      arr[i] = i;
      rates[i] = (arr[i]*1.387)/4;
    }
    `
Array Characteristics

```c
int myarray[31];  // array of ints
```

- 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.
- C does **NOT** do bounds checking.
  - Asking for `myarray[35]`?
    - Python: error
    - C: “Sure thing, boss”
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?
  – ASCII Standard says so.
# Characters and Strings

<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 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>
Recall Arrays

• int myarray[31];

  “myarray”
  Location of [0] in memory.

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

↑ Array bucket indices.

• A string is an array of characters (type char)
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 name[20] = "Martin";
```

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?).
Characters and Strings

<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 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>

Special stuff over here in the lower values.
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:

  ```c
  char name[20] = “Martin”;
  ```
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 different field values of the struct variable
Struct Example

- Suppose we want to represent a student type.

```
struct student {
    char name[20];
    int grad_year;
    float gpa;
};

struct student bob;

strcpy(bob.name, “Robert Paulson”);
// Set name (string) with strcpy()
bob.grad_year = 2016;
bob.gpa = 3.1;

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

struct student {
    char name[20];
    int grad_year;
    float gpa;
};

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 = 2014;
classroom[0].gpa = 4.0;
strcpy(classroom[1].name, "Bob");
classroom[1].grad_year = 2017;
classroom[1].gpa = 3.1
strcpy(classroom[1].name, "Cat");
classroom[1].grad_year = 2016;
classroom[1].gpa = 3.4
```

<table>
<thead>
<tr>
<th>classroom:</th>
<th>Alice</th>
<th>Bob</th>
<th>Cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>grad_year</td>
<td>2014</td>
<td>2017</td>
<td>2016</td>
</tr>
<tr>
<td>gpa</td>
<td>4.0</td>
<td>3.1</td>
<td>3.4</td>
</tr>
</tbody>
</table>
Functions: example from lab 2

```c
void open_file_and_check(char *filename);

int main (int argc, char *argv[]) {
    //...
}

void open_file_and_check(char *filename){
    int ret;
    ret = open_file(filename)
    if(ret == -1) {
        printf("bad error: can't open %s\n", filename);
        exit(1);
    }
}
```
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() {
    int x, y;  // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

```
Stack

main:
  x: 4
  y: 7

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

Note: This doesn’t change!
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() {
    int x, y;  // declare two integers
    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]);
}
```

What will this print?
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]);
}
```

What will this print?
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]);
}
```
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]);
}
```

What will this print?
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]);
}
```

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