Today

• 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
What is C?

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

• **ANSI C** (American National Standards Institute)
  – Defines both language and standard C library
Evolution of C – ISO C90, ISO C99

• **ISO C90** (International Standards Organization)
  – Mostly the same as 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 found in the English language
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!
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!
So, what are some differences between Python and C?
Python versus C: Paradigms

- Python and C follow different programming paradigms.
- C breaks down to functions and is more procedure-oriented.
- Python follows an object-oriented paradigm that allows Python to break down Objects.
Python versus C: Native

- Python is an “interpreted” language
- C is a “compiled” language

The Python interpreter does lexing, parsing, compiling and interpreting!

C is compiled to machine code that is “executed” by the underlying processor.
Python versus C: High/Low

• Python is a high-level language
• C is a low-level language

C is closer to the machine. It gives you access to aspects of the machine that are not accessible in Python.

Types in C are dependent on the machine you are running on. Python types “expand to available memory”.
Python versus C: Memory

• Python does not allow you to access memory directly
• C allows you to access memory directly, interpret that memory in any way you wish, and shoot yourself in the foot.

In Python, you have references to objects in memory. You can pass these references to methods, invoke methods.

In C, you have references to data in memory. In C we have mechanisms to allow you to manipulate memory in arbitrary ways.
Python versus C: Memory Management

- **Python provides automatic memory management.** It is a “garbage collected” language.
- **C provides manual memory management.** You are responsible for managing that memory yourself!

In Python, you can dynamically allocate objects in a special part of memory called the heap. When you are finished with that object you simply forget about it.

In C, you can dynamically allocate variables in a special part of memory called the heap. You use the malloc library function. When you are finished with that variables you must use the free library function. If you do not, you will have a *memory leak*.
Python versus C: Error Handling

- Python provides “exceptions”
- C does not provide “exceptions”

In Python, if your program has an error at run-time it will throw an exception. You get a nice “stack” trace of where the error occurred to help you find the problem.

In C, if your program has an error at run-time it will not throw an exception. If you are lucky (and smart) your program will check “error conditions” and fail gracefully. If you are not lucky your program will crash and simply tell you: segfault.
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.
  - 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

### 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

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

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

- `#: single line comment`
- `//: single line comment`
- `import libname: include Python libraries`
- `#include<libname>: include C libraries`
# 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**
- **import libname:** include Python libraries
- **#include<libname>:** include C libraries
- **Blocks: indentation**
- **Blocks: {   }** (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.
These are 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();
    }
}
```
### 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><strong>#</strong>: single line comment</th>
<th><strong>//</strong>: single line comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>import libname</strong>: include Python libraries</td>
<td><strong>#include&lt;libname&gt;</strong>: include C libraries</td>
</tr>
<tr>
<td><strong>Blocks: indentation</strong></td>
<td><strong>Blocks: {   }</strong> (indentation for readability)</td>
</tr>
<tr>
<td><strong>print</strong>: statement to printout string</td>
<td><strong>printf</strong>: function to print out format string</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;
}
```

<table>
<thead>
<tr>
<th>#: single line comment</th>
<th>//: single line comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>import libname</strong>: include Python libraries</td>
<td><strong>#include&lt;libname&gt;</strong>: include C libraries</td>
</tr>
<tr>
<td><strong>Blocks: indentation</strong></td>
<td><strong>Blocks: {   }</strong> (indentation for readability)</td>
</tr>
<tr>
<td><strong>print</strong>: statement to printout string</td>
<td><strong>printf</strong>: function to print out format string</td>
</tr>
<tr>
<td><strong>statement</strong>: each on separate line</td>
<td><strong>statement</strong>: each ends with ;</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;
}
```

<table>
<thead>
<tr>
<th>Python Comment</th>
<th>C Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>#: single line comment</td>
<td>//: single line comment</td>
</tr>
<tr>
<td><strong>import libname</strong>: include Python libraries</td>
<td><strong>#include&lt;libname&gt;</strong>: include C libraries</td>
</tr>
<tr>
<td>Blocks: <strong>indentation</strong></td>
<td>Blocks: { } (indentation for readability)</td>
</tr>
<tr>
<td><strong>print</strong>: statement to printout string</td>
<td><strong>printf</strong>: function to print out format string</td>
</tr>
<tr>
<td><strong>statement</strong>: each on separate line</td>
<td><strong>statement</strong>: each ends with ;</td>
</tr>
<tr>
<td><strong>def main()</strong>: the main function definition</td>
<td><strong>int main()</strong>: 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.
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)
Numerical Type Comparison

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

Floating Point (float, double)
- Example:
  ```c
  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
}

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

<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>

Do the following statements evaluate to True or False?

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

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

Basic if statement:

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

With optional else:

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

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>
</table>
| `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`  
  `}` | `if(<boolean expr1>) {`  
  `  if-expr1-true-body`  
  `} else if (<bool expr2>) {`  
  `  else-if-expr2-true-body`  
  `}`  
  `...`  
  `} 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:

  ```python
  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

```c
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: \texttt{print} "\%d \%s\t\%f" \%(6, "hello", 3.4)
C: \texttt{printf}("\%d \%s\t\%f\n", 6, "hello", 3.4);

\texttt{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)

\texttt{a'}: in Python is a string, in C is a \texttt{char}

"a": in Python is a string, in C is a \texttt{string}

C: string and char are different types

Slide 48
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>]`;

    ```c
    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

int january_temps[31]; // Daily high temps

• Indices start at 0! Why?

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

int january_temps[31]; // Daily high temps

“january_temps”
Location of [0] in memory.

• 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
```

1. `scanf("%d", january_temps);`
2. `scanf("%d", &january_temps[1]);`
3. None of the above
int january_temps[31]; // Daily high temps

“january_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”
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.
<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>
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[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?).
<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>”</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 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”;
  ```
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
Suppose we want to represent a student type.

```c
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 = 2018;
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;
};

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>class:</th>
<th>'A'</th>
<th>'l'</th>
<th>'i'</th>
<th>'c'</th>
<th>'e'</th>
<th>' \0'</th>
<th>‘e’</th>
<th>‘l’</th>
<th>‘i’</th>
<th>‘c’</th>
<th>‘e’</th>
<th>‘ \0’</th>
<th>‘e’</th>
<th>‘l’</th>
<th>‘i’</th>
<th>‘c’</th>
<th>‘e’</th>
<th>‘ \0’</th>
<th>‘e’</th>
<th>‘l’</th>
<th>‘i’</th>
<th>‘c’</th>
<th>‘e’</th>
<th>‘ \0’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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;
}
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);
}
```

```
<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>x:</td>
</tr>
<tr>
<td>y:</td>
</tr>
</tbody>
</table>

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

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]);
}
```

The output will be: 
```
0, 5, 10
```
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`.
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 code will print `0, 5, 10`.

**Explanation:**
- The `main` function initializes `x` to 0 and defines `values` with two elements.
- It then calls `func` with `x`, `x`, and `values`.
- Inside `func`, `y` is set to 1, `my_array[a]` to 0, and `my_array[y]` to 8.
- The function returns `y`, which is 1.
- In `main`, the value of `x` is updated to the return value of `func`, which is 1.
- The `printf` statement outputs the values of `x`, `values[0]`, and `values[1]`.

**Result:**
- The output is `0, 5, 10`.

**Diagram:**
- The diagram shows the stack with `x` being 0, `values` being `[5, 10]`, `y` being 1, and `my_array` having elements `[0, 8]`.
- The values are labeled as follows:
  - `x`: 0
  - `values[0]`: 5
  - `values[1]`: 10
  - `y`: 1
  - `my_array[0]`: 0
  - `my_array[1]`: 8
- The stack is used to store these values during function calls and local variable assignments.
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: `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