What we will cover

- A crash course in the basics of C
- You should read the K&R C book for lots more details

Like Java, like C

- Operators same as Java:
  - Arithmetic
    - +, -, *, /, %,
  - Relational and Logical
    - <, >, <=, >=, ==, !=
    - &&, ||, &|, !

- Syntax same as in Java:
  - if ( ) { } else { }
  - while ( ) { }
  - do { } while ( );
  - for(i=1; i <= 100; i++) { }
  - switch ( ) {case 1: ... }
  - continue; break;
Simple Data Types

datatype        size   values
char             1      -128 to 127
short            2      -32,768 to 32,767
int              4      -2,147,483,648 to 2,147,483,647
long             4      -2,147,483,648 to 2,147,483,647
float            4      3.4E+/−38 (7 digits)
double           8      1.7E+/−308 (15 digits long)

Java programmer gotchas
(1)

{  
    int i
    for(i = 0; i < 10; i++)
    
        ...

NOT

{  
    for(int i = 0; i < 10; i++)
    
        ...

Java programmer gotchas
(2)

• Uninitialized variables
  - catch with −Wall compiler option

#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;
    factorial(i);
    return 0;
}

Java programmer gotchas
(3)

• Error handling
  - No exceptions
  - Must look at return values
“Good evening”

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    /* print a greeting */
    printf("Good evening!\n");
    return 0;
}
```

$ ./goodevening
Good evening!
$

Breaking down the code

- #include <stdio.h>
  - Include the contents of the file stdio.h
  - Case sensitive – lower case only
  - No semicolon at the end of line
- int main(...)  
  - The OS calls this function when the program starts running.
- printf(format_string, arg1, ...)
  - Prints out a string, specified by the format string and the arguments.

```
#define FIFTEEN_TWO_THIRTEEN "The Class That Gives CMU Its Zip\n"
```

int main(int argc, char* argv[]) {
    printf(FIFTEEN_TWO_THIRTEEN);
    return 0;
}

format_string

- Composed of ordinary characters (not %)
  - Copied unchanged into the output
- Conversion specifications (start with %)
  - Fetches one or more arguments
  - For example
    - char %c
    - char* %s
    - int %d
    - float %f
- For more details: man 3 printf

C Preprocessor
After the preprocessor (gcc -E)

```c
int main(int argc, char* argv)
{
    printf("The Class That Gives CMU Its Zip\n");
    return 0;
}
```

Conditional Compilation

```c
#define CS213

int main(int argc, char* argv)
{
    #ifdef CS213
    printf("The Class That Gives CMU Its Zip\n");
    #else
    printf("Some other class\n");
    #endif
    return 0;
}
```

After the preprocessor (gcc -E)

```c
int main(int argc, char* argv)
{
    printf("The Class That Gives CMU Its Zip\n");
    return 0;
}
```

Command Line Arguments (1)

- `int main(int argc, char* argv[])`
- `argc`
  - Number of arguments (including program name)
- `argv`
  - Array of char* (that is, an array of ‘c’ strings)
    - `argv[0]`: program name
    - `argv[1]`: first argument
    - ...
    - `argv[argc-1]`: last argument
# Command Line Arguments (2)

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    printf("%d arguments\n", argc);
    for(i = 0; i < argc; i++)
        printf(" %d: %s\n", i, argv[i]);
    return 0;
}
```

$ ./cmdline The Class That Gives CMU Its Zip
8 arguments
0: ./cmdline
1: The
2: Class
3: That
4: Gives
5: CMU
6: Its
7: Zip
$

# Command Line Arguments (3)

```c
char foo[80];
- An array of 80 characters
  - sizeof(foo) = 80 _ sizeof(char) = 80 _ 1 = 80 bytes

int bar[40];
- An array of 40 integers
  - sizeof(bar) = 40 _ sizeof(int) = 40 _ 4 = 160 bytes
```

# Arrays

- **char** foo[80];
  - An array of 80 characters
    - sizeof(foo) = 80 _ sizeof(char) = 80 _ 1 = 80 bytes

- **int** bar[40];
  - An array of 40 integers
    - sizeof(bar) = 40 _ sizeof(int) = 40 _ 4 = 160 bytes

# Structures

- **Aggregate data**

```c
#include <stdio.h>

struct name {
    char* name;
    int age;
}; /* <= DO NOT FORGET the semicolon */

int main(int argc, char* argv[]) {
    struct name bovik;
    bovik.name = "Harry Bovik";
    bovik.age = 25;

    printf("%s is %d years old\n", bovik.name, bovik.age);
    return 0;
}
```
Pointers

- Pointers are variables that hold an address in memory.
- That address contains another variable.

Using Pointers (1)

```c
float f;            /* data variable */
float *f_addr;      /* pointer variable */

f_addr = &f;        /* & = address operator */
```

Pointers made easy (2)

```c
*f_addr = 3.2;       /* indirection operator */

float g = *f_addr;  /* indirection: g is now 3.2 */
/* but g is still 3.2 */
```
**Function Parameters**

- Function arguments are passed “by value”.
- What is “pass by value”?  
  - The called function is given a copy of the arguments.
- What does this imply?  
  - The called function can’t alter a variable in the caller function, but its private copy.
- Three examples

**Example 1: swap_1**

```c
void swap_1(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: Let x=3, y=4, after swap_1(x,y);
   x =? y=?
   - A1: x=4; y=3;
   - A2: x=3; y=4;

**Example 2: swap_2**

```c
void swap_2(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: Let x=3, y=4, after swap_2(&x,&y);
   x =? y=?
   - A1: x=3; y=4;
   - A2: x=4; y=3;

**Example 3: scanf**

```c
#include <stdio.h>
int main() {
  int x;
  scanf("%d\n", &x);
  printf("%d\n", x);
}
```

Q: Why using pointers in scanf?
A: We need to assign the value to x.
Dynamic Memory

- Java manages memory for you, C does not
  - C requires the programmer to explicitly allocate and deallocate memory
  - Unknown amounts of memory can be allocated dynamically during run-time with `malloc()` and deallocated using `free()`

Not like Java

- No `new`
- No garbage collection
- You ask for $n$ bytes
  - Not a high-level request such as “I’d like an instance of class `String`”

malloc

- Allocates memory in the heap
  - Lives between function invocations
- Example
  - Allocate an integer
    ```c
    int* iptr = (int*) malloc(sizeof(int));
    ```
  - Allocate a structure
    ```c
    struct name* nameptr = (struct name*) malloc(sizeof(struct name));
    ```

free

- Deallocates memory in heap.
- Pass in a pointer that was returned by `malloc`.
- Example
  ```c
  int* iptr = (int*) malloc(sizeof(int));
  free(iptr);
  ```
- Caveat: don’t free the same memory block twice!