What type of language is C?

- C is **function-oriented** (rather than object-oriented)
- C (like Java) is **statically typed**, so variables must be declared with a type and remain that type for the duration of the program's execution
- C has **no garbage collection**: the programmer must manage all dynamically allocated memory themselves

Hello, World!

```
#include <stdio.h>

int main(int argc, char* argv[]) {
      printf("Hello world!");
}
```

- `#include <stdio.h>` tells the pre-processor (a programmer that runs before the compiler) to include the stdio (standard input/output) C library when compiling this program
- stdio contains (among other things) the definition for the function `printf()`
- When you run a C executable, it will start execution from the `main()` function
- The main function takes two arguments:
  - `argc`: the number of arguments <u>including the name of the executable</u>
  - `argv`: an array of all of the command line arguments as strings
  - `./a.out foo 6 7`
    - `argc == 4`
    - `argv` will hold a pointer to memory that looks like this:

0xFFFF0000	0xFFFF0006	0xFFFF000A	0xFFFF000C	NULL

(an array of pointers to memory that might look something like this:)

0xFFFF0000						0xFFFF0006				0xFFFF000A		0xFFFF000C	
a	.	o	u	t	\0	f	o	o	\0	6	\0	7	\0

- - - Strings are arrays of characters (8-bit numbers between 0 and 255 that represent ASCII values) that end in a null terminator (the \0 character)
- `printf()` will print its contents to STDOUT (standard output)
  - Strings in C are indicated by " " double quotes
  - Single quotes ' ' are for characters

Basic Syntax

Functions and Variable Definitions

```
int foo (int baz, size_t bar) {
      int retval = 5;
      return retval * bar + baz;
}
```

- Functions are declared in the format

return_type function_name(**variable_type** variable_name, ...) { function_body }

- If you declare a variable inside a function, it is limited in scope to that function (you cannot access it from outside that function)
- Functions are pass by value

Loops and Conditionals

```
if (a > *b) {
      return bar;
} else if (a == *b) {
    return foo;
} else {
      return baz;
}

int i; // After C99, you can also declare in for loop
for (i = 0; i < n; i++) {
      array[i] = i;
}

while (n != 42) {
      n += array[i];
}
```

sizeof()

- What does sizeof() do?
  - Tells you the size in bytes of the variable type that you pass in. Characters are *always* one byte, in every system.
  - Everything else can vary in size! Do NOT rely on an integer being 4 bytes. It is true that for numbers, it goes short <= int <= long <= long long.

In a 64-bit system with 4-byte integers…

```
int x;            // sizeof(x) == 4
int * y;          // sizeof(y) == 8
int ** z;         // sizeof(z) == 8
int arr[7];       // sizeof(arr) == 28
```

- Takeaways:
  - Pointers are always the same size, which depends on architecture
  - Calling sizeof() on an array returns the number of bytes in the whole array

Pointers

- A pointer is **an address in memory where a variable lives**

```
int x = 0x61c;
int y = 0x2a;
int * p = &y;        // p is a pointer to y
int ** pp = &p;      // pp is a pointer to p
```

- & is the address operator; putting it before a variable will give you the address where that variable is stored in memory
- * is the dereference operator; putting it before a variable will treat the variable as an address and give you what value is stored at that address in memory
- Pointers are how we get pass by reference functionality from a pass by value language
- Every programming language uses pointers that are actually addresses; C is unique in that it very directly exposes them to the programmer

Pointer Arithmetic

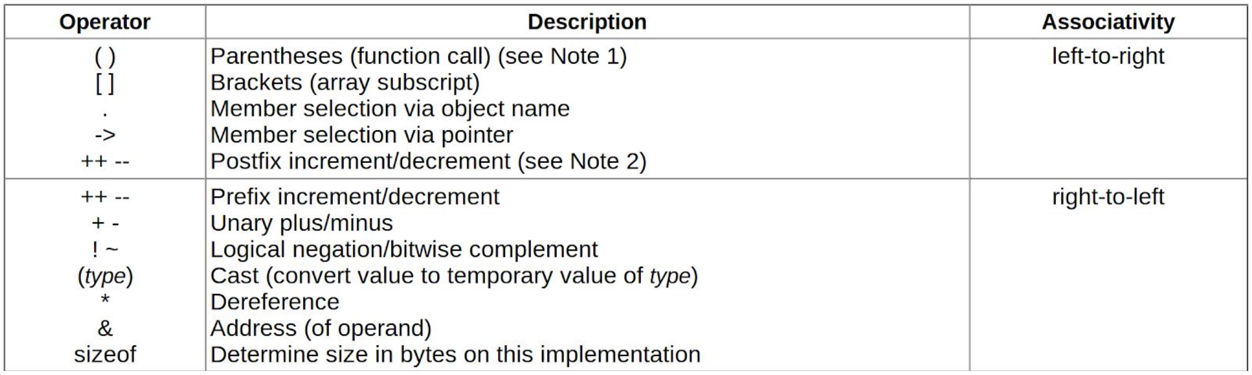
- You can perform addition and subtraction operations on pointers
  - Can add or subtract a number from a pointer
  - Can subtract one pointer from another
  - Do not add two pointers together!

- All addresses are byte addresses, but arithmetic doesn’t work in units of bytes but in units of `sizeof(variable pointed to)`

```
Example: assume sizeof(int) == 4, sizeof(int *) == 8

      int x = 5;
      int * y = &x;         // y = 0x4000
      y = y + 2;            // y = 0x4008, added 8

      char * z = “bears!”   // z = 0x2000
      z = z + 2;            // z = 0x2002, added 2

      int ** q = &y;        // q = 0x6000
      q = q + 2;            // q = 0x6010, added 16
```

Operator Precedence

- The order in which you apply operators in C depends on operator precedence
- Not something to memorize! Look at a table:



Operator	Description	Associativity
( ) [ ] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left

- `*p++` is equivalent to `*(p++)`
- `++*p` is equivalent to `++(*p)`
- `*++p` is equivalent to `*(++p)`
- `&p->next` is equivalent to `&(p->next)`

Structs

- A struct in C is an ordered grouping of variables
- Syntax: don’t forget the semicolon after the last bracket!

```
struct foo {
    int bar;
    char baz;
    float qux;
};
```

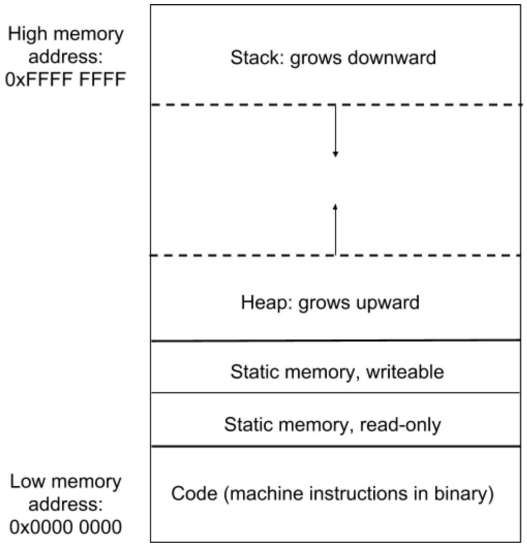
- You can assume that if we have a `struct foo x, x->bar` will have the lowest memory address, `&(x->baz)` will be the next greatest, and `&(x->qux)` will be the highest
- When we say we’re using <u>32-bit word-aligned architecture</u>, this means every variable with size >= 4 will start at an address that is a multiple of 4, and if necessary, there will be some empty space in the middle of the struct
    - For the struct above, if a float is 4 bytes, and int is 4 bytes, and a char is 1 byte, if we were using 32-bit word aligned architecture, we’d have `sizeof(struct foo) == 12`

typedef

- Giving one defined type another name

```
typedef uint32_t u_int32_t; // When we use u_int32_t it will refer to uint32_t
typedef uint8_t ONE_BYTE;   // Now when we use ONE_BYTE it will refer to uint8_t
typedef struct node {
    int value;
    struct node * next;
} ll_node;                  // Now when we use ll_node it will refer to this struct
```

Heap, Static, Stack & Code


High memory address: 0xFFFF FFFF
Stack: grows downward
Heap: grows upward
Static memory, writeable
Static memory, read-only
Low memory address: 0x0000 0000
Code (machine instructions in binary)

Stack: function local variables, strings allocated as arrays (e.g. `char bears[10] = “Go Bears!”`)

Heap: dynamically allocated memory (with `malloc, calloc, realloc`)

Static: global variables, statically allocated strings (e.g. `char * = “Go Bears!”`, the string literal (“Go Bears!”) is stored in read-only static memory in the compiled executable

Code: the 1’s and 0’s that represent the compiled machine instructions for your program

Dynamic Memory Allocation

- Memory is allocated on the heap with the dynamic memory allocation functions:
    - `void* malloc(size_t size)`
        - Allocate `size` bytes of space in the heap, return a pointer to it
        - What if we try to allocate more memory than is available in the heap? Returns NULL
    - `void* calloc(size_t number_items, size_t size_items)`
        - Allocate `size_items` * `number_items` bytes of space in the heap, initialize it to zeroes, return a pointer to it
    - `void* realloc(void* ptr, size_t bytes)`
        - Move data stored in `ptr` to new space of size `bytes`
        - What if the new "bytes" argument is more than originally allocated? Have garbage at end
        - What if the new "bytes" argument is less than originally allocated? Lose some data
    - `void free(void* ptr)`
        - De-allocate memory pointed to by `ptr`
        - What happens if we try to free a pointer that's already free? Double free()s cause heap corruption

CHANGELOG:
-- `argv` table on page 1 updated!
-- Fixed typo in function signature for `free()`