The C Programming Language

C is

- Compiled
- Static Typed
- Low Level
- Sequential
- Everywhere
- Safe
- Evolving

History

- Developed in early ’70s by Denis Ritchie at Bell Labs
- Ultimately used to redevelop UNIX kernel
- K&R C book release in 1978
- ANSI C released 1989 (C89)

Hello, World

- Given the following code:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    if (puts("Hello, World!\n") != EOF) {
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

- The code can be compiled into executable format:

```
make hello-world
```

- The executable has the following output:

```
Hello, World!
```

Integers

- C has a bunch of standard integer types
- They map to data understood at a hardware level

```
#include <stdint.h>
int foo = 1;
unsigned int bar = 2;
long baz = 3;
long long qux = 4;
int16_t quux = 5;
int64_t waldo = 6;
int16_t corge = 0xFFFFFFFF;
```

Floats

- C also defines some binary floating point types
- The use of IEEE754 as an in memory format has been the source of billions of dollars of mistakes

```
float one_point_five = 1.5;
float point_three = 0.3;

double one_e_123 = 1e123;
```

Pointers and Strings

- A pointer is a value that describes where data can be found in memory

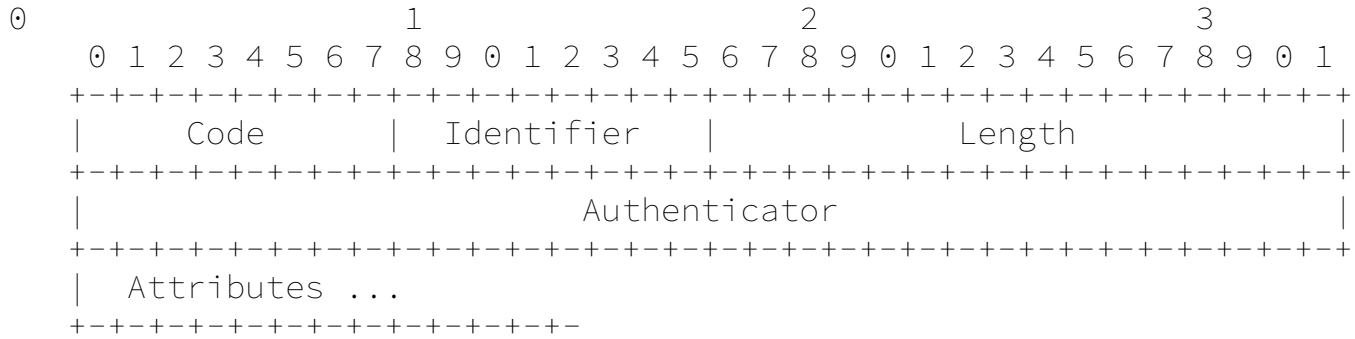
```
int int_value = 123;
int *int_pointer = &int_value;
```

- The use of a null terminator in c-strings has been the source of trillions of dollars of mistakes

```
char* string = "Hello, World!";
```

Custom Types

- C supports structs and unions
- Typedefs can be used to create aliases
- Data packed manually into memory



(RADIUS protocol)

Allocation

- In C, the stack and heap are first class concepts

```
int value_on_the_stack = 42;

int *value_on_the_heap = (int*)malloc(1 * sizeof(int));
*value_on_the_heap = 42;
```

The Stack And Heap

- The stack:
  - Fast efficient linear memory allocation
  - Very explicit lifetime bounds
- The heap:
  - Slightly less efficient (but still fast) memory allocation
  - Manual lifetime management

Addition

- Given the following function:

```
int add(int l, int r) {
    return l + r;
}
```

- Compiled like this:

```
clang -O3 -c add.c
```

- The machine code generated is "optimal":

```
$ otool -vax add.o
add.o:
(__TEXT,__text) section
_add:
0000000000000000        add     w0, w1, w0
0000000000000004        ret
```

Vector maths


```c
#include <stdio.h>
#include <arm_neon.h>

int main(){
    uint8x16_t v1, v2;
    for (int idx = 0; idx < 15; ++idx) {
        v1[idx] = 2 * (v2[idx] = idx);
    }
    uint8x16_t res = v1 * v2;
    char *sep = "";
    for (int idx = 0; idx < 15; ++idx) {
        printf("%s%d", sep, res[idx]);
        sep = ", ";
    }
    printf("\n");
}
```

Function invocation

- Nothing too surprising, this code

```c
#include <stdio.h>
#include <stdlib.h>

int foo(int in) {
    return in * 2;
}

int bar(int in) {
    return in * 3;
}

int main(void) {
    if (printf("%d", foo(bar(5))) != EOF) {
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

- . . . prints this

```
30
```

State

Local to function

```
int foo() {
    int state = 1;
    ...
}
```

Local to module

```
    static int state = 1;
    int foo() {
    ...
}
```

Global

```
    int state = 1;
    int foo() {
    ...
}
```

Shared across processes

```
int foo() {
    shmget(...);
    ...
}
```

Quirks

- Instruction Reordering
- Signal handling
- Jumps (goto considered harmful)

Everywhere-ish

- Linux (~24million SLOC)
- PostgreSQL (~2 million SLOC)
- Python
- unixODBC
- Sqlite3
- ...

Safe

- Overflow
- Buffer Overrun
- Thread safety
- Aliasing
- Signal handling
- Memory Leaks
- Memory Ownership

Evolving

- C99
- C11
- C17