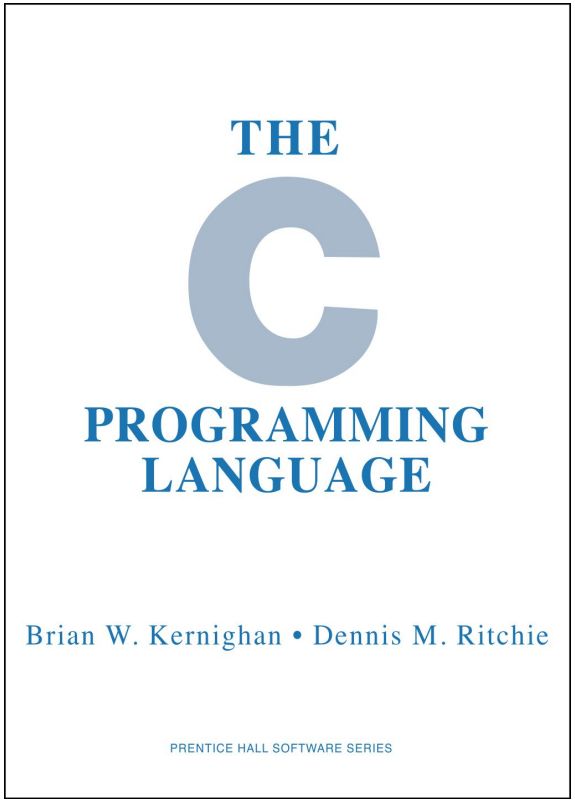

MONTANA
STATE UNIVERSITY

Functions In C

Encapsulating Logic in the C Programming Language

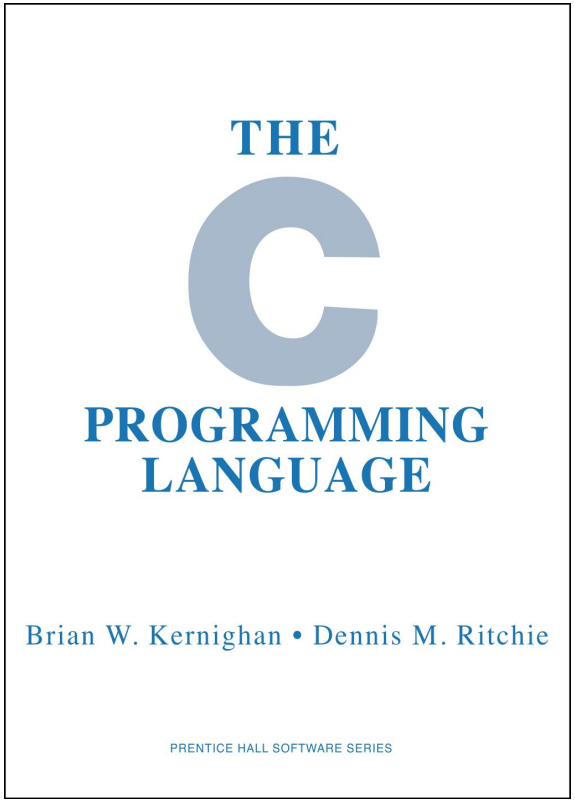
The C Programming Language

- C is a *functional* programming language
- Not really thought of this way because of how janky it is
- Most snooty functional programming languages have extremely complex type systems



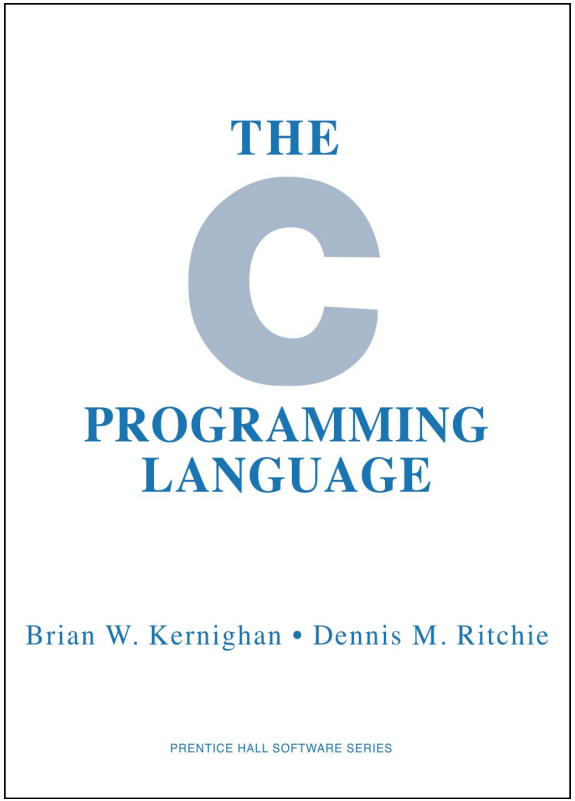
The C Programming Language

- None the less C *is* functional
- The primary component of a C project is its functions and, to a lesser extent, the structs it works with



The C Programming Language

- Functions break large computing tasks into smaller tasks
- Allow you to break out and reuse common logic
- Function calls in C are *fast*
  - Especially on x64 due to calling conventions



C Function Definitions

- You should be familiar with the basics of a C function definition
  - Return type, name, arg list, function body
  - Return statement (not required!) returns a value

```c
int my_int_function(int *pointer_to_int) {
    return *pointer_to_int;
}
```

C Function Definitions

- If you omit a return type, it will default to int
  - Int is sort of a universal value in C
- If your function returns nothing, you can set the return type to void

```
void my_int_function(int *pointer_to_int) {
    *pointer_to_int;
}
```

C Function Arguments

- Function arguments are listed with comma separated values
  - <type def> <name>
  - Can be confusing with pointers because the * can be placed anywhere

```
void my_int_function(int *pointer_to_int) {
    *pointer_to_int;
}
```

C Function Arguments

- This is the *same* argument type, with the pointer * on the int, rather than on the symbol
- Personally, I find this far clearer
- Unfortunate, this is not the standard

```
void my_int_function(int* pointer_to_int) {
    *pointer_to_int;
}
```

C Function Arguments

- C also supports *variable* arguments

```c
#include <stdarg.h>
void my_va_int_function(int* pointer_to_int, ...) {
    va_list vaList;
    va_start(vaList, l: 10);
    for (int i = 0; i < 10; ++i) {
        va_arg(vaList, l: int*);
    }
    va_end(vaList);
}
```

C Function Arguments

- Uses the stdarg library
- Requires that you *know* how many arguments have been passed
- Printf figures it out by parsing the first argument string
- Pretty insane, you won’t need this for the project

C Function Definition

- All functions must be *declared* before they are used

```
void func_1() {
    func_2()
}

void func_2() {
    func_1();
}
```

C Function Definition

- All functions must be *declared* before they are used

```
void func_2();
void func_1() {
    func_2();
}

void func_2() {
    func_1();
}
```

C Function Definition

- This is *deeply* annoying to me
- There is no good reason the C parser couldn't scan files in two passes
  - First a declaration pass
  - Then a definition pass
- This is exactly what the vast majority of modern languages do
  - E.g. Java
- Because of this limitation, the C language developed the *header file*

C Functions

- Recursion *is* allowed
- Nested functions are *not* allowed
- No notion of closures in C
- There are however function pointers
  - Needed for the pthread API we will be using

C Header Files

- Header files contain function declarations
- Often used as a “public” api for a given module

```
#ifndef INC_366_DEMOS_MAIN_H
#define INC_366_DEMOS_MAIN_H

void func_2();

#endif //INC_366_DEMOS_MAIN_H
```

C Header Files

- Note the #ifndef statement
- Part of the C pre-processor system

```
#ifndef INC_366_DEMOS_MAIN_H
#define INC_366_DEMOS_MAIN_H

void func_2();

#endif //INC_366_DEMOS_MAIN_H
```

C Header Files

- This statement prevents the header file from being included twice by the compiler
- Not exactly world beating compiler tech...

```
#ifndef INC_366_DEMOS_MAIN_H
#define INC_366_DEMOS_MAIN_H

void func_2();

#endif //INC_366_DEMOS_MAIN_H
```

C Header Files

- Using the header file involves the *#include* preprocessor statement
- Note that we use quotes rather than angle brackets

```c
#include "main.h"
void func_1() {
    func_2();
}
```

C Header Files

- Quotes are “local” headers: it will look for local .h files first
- Angle brackets are “global” headers, compiler will look in system library directories

```c
#include "main.h"
void func_1() {
    func_2();
}
```

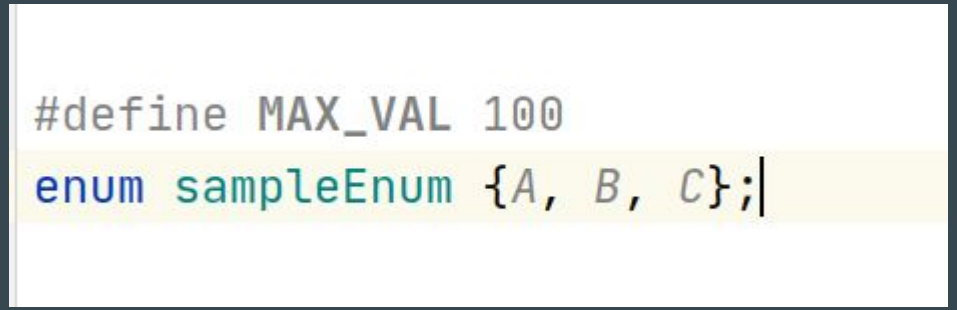
C Header Files

- Important to understand that this is *literal* string substitution, the source file is being included directly in the file

```
#include "main.h"
void func_1() {
    func_2();
}
```

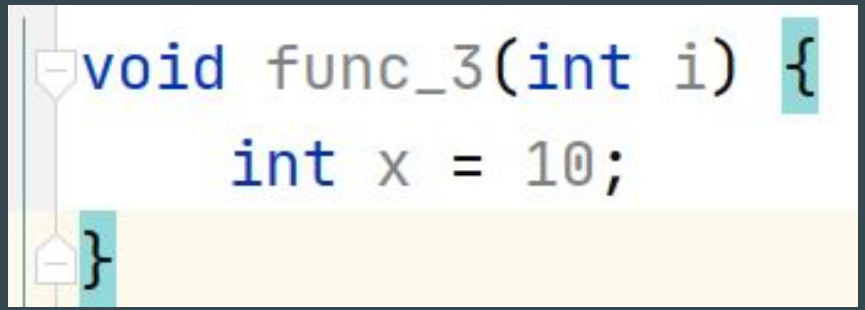
C Header Files

- Header files don’t just contain function declarations
  - May include constants (usually macros)
  - May include enumerations
  - May include structs



C Scoping Rules

- C has a mix of sensible and insane scopes
- Let's start with the sensible:
  - Variables local to a function (sometimes called *Automatic* variables) follow the sensible scoping rules you expect: the variable is available for the body of the function, and exits scope afterwards
  - Function parameters also follow the rules you would expect



C Scoping Rules

- OK, still pretty sensible:
  - Variables declared *outside* a function (extern) are initialized once and available to all functions
  - Kind of a singleton or global
  - They are shadowable in functions (bad idea!)

```
int x = 10;
void func_3(int i) {
    int x = 12;
    printf("x is %d", x);
}
```

C Scoping Rules

- Turns out there is also an extern keyword for this
- *extern* in this case means truly global
- This can be dangerous, relatively rare

```
extern int x = 10;
void func_3(int i) {
    int x = 12;
    printf("x is %d", x);
}
```

C Scoping Rules

- Let’s get to the less sane stuff: static variables

```
void func_3() {
    static int x = 12;
    x++;
    printf( format: "x is %d", x);
}
```

C Scoping Rules

- Believe it or not, this function will print 13, 14, 15
- Static variables are initialized once and reused between function calls

```
void func_3() {
    static int x = 12;
    x++;
    printf( format: "x is %d", x);
}
```

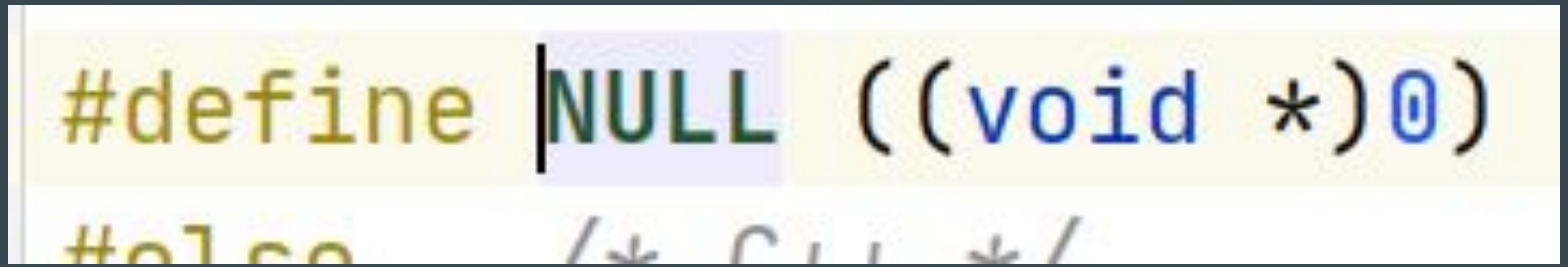
C Scoping Rules

- Madness. No one uses this, right?
- False. strtok, the standard string tokenizing function does!

```
char *str ="A Sample String";
char * token;
token = strtok (str, delim: " ");
while (token != NULL)
{
    printf (format: "%s\n", token);
    token = strtok (s: NULL, delim: " ");
}
```

C Scoping Rules

- Side note: how is NULL defined?
- With a macro of course!
- Value is equal to zero, with a void * type (the C equivalent of “anything”)



C Scoping Rules

- Madness: register scoping
- Stores the variable in a register
  - We will discuss registers later
- Fast, but you can’t get its address
- Rarely used

```
void func_5() {
    register int foo = 10;
    int *foo_ptr = &foo;
}
```

C Macros

- We have already seen some macros
  - #include
  - NULL
- Start with # character
- Macros are expanded before compilation
- Used for
  - Conditional compilation
  - Constants
  - General sleaziness

C Macros

- Defined with the #define keyword
- What does this print?

```
#define square(x) x*x
void func_6() {
    printf( format: "%d", square( x: 2 + 2));
}
```

C Macros

- Corrected version

```c
#define square(x) (x)*(x)
void func_6() {
    printf( format: "%d", square( x: 2 + 2));
}
```

C Macros

- Normal day to day use is for constants and that’s about it
- Unless you enjoy tormenting people
- Or participating in the The International Obfuscated C Code Contest
  - https://www.ioccc.org/

Review

- C function syntax
  - Not too dissimilar from Java, etc.
- Header files
  - Contain declarations
  - Shouldn’t be necessary
- Various scopes
  - Some sane, some less so
- Macros
  - Generally insane


MONTANA
STATE UNIVERSITY