C Programming Part 1: Overview

ECEN 330: Introduction to Embedded Programming

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Hello World , #include


```c
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

All C programs start with “main”

- Return 0 if no error

#include

- Before compilation the included file is literally copied into this location.
- <> ← Look in system directories
- “ “ ← Look in user’s program directories

- In C, each “.c” file is compiled separately, and linked together later

- .h files contain information about what is available in other compiled files:
  - Function declarations (like “printf”)
  - Data types (like uint32_t)

Console


```c
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

- Where does printf go?
  - Terminal / Console
- This is called “stdout” (**Standard Out**)
- You can also get characters that the user types into the terminal
  - This is called “stdin” (**Standard In**)

Compiling with GCC

There are lots of different C compilers out there, but GCC is open-source and the most popular.

```
gcc main.c
```

This will produce an executable named “a.out”, and we can run it like this:

```
./a.out
```

If we want the executable to be named differently:

```
gcc main.c -o myExe
```

For the labs, we use Cmake/Make (which calls gcc behind the scenes)

Statements


```
int i = 6;
```

- C code is made up of **statements**
- Statements are ended by semicolons

```
/* this declares the variables 'i', 'test', 'foo', and 'bar'
    note that ONLY the variable named 'bar' is set to six! */
int i, test, foo, bar = 6;
```

Blocks


```
int main(void)
{
    /* this is a 'block' */
    int i = 5;

    {
        /* this is also a 'block', nested
       inside the outer block */
        int i = 6;
    }

    return 0;
}
```

A block is a set of executable statements

Blocks are normally started with functions, loops, if statements, etc.

…but you can make new blocks wherever you want (this isn't very common)

Scope


```
int i = 5; /* this is a 'global' variable, it can be accessed from anywhere in
              the program */


/* this is a function, all variables inside of it are "local" to the function. */
int main(void) {
  int i = 6;          /* 'i' now equals 6 */
  printf("%d\n", i); /* prints a '6' to the screen, instead of the global
                        variable of 'i', which is 5 */

  return 0;
}
```

- **Global scope:** Accessible throughout the entire file
  - Also accessible from other files (unless declared **static**)
- **Local scope:** Only accessible within the *block* it was declared in.

Scope


```c
int main(void)
{
    int i = 6; /* this is the first variable of this 'block', 'i' */

    {
        /* this is a new 'block', it has its own scope */

        int i = 5;
        printf("%d\n", i); /* prints a '5' onto the screen */
    }

    /* now we're back into the first block */

    printf("%d\n", i); /* prints a '6' onto the screen */

    return 0;
}
```

Executable Code Outside Functions


```c
int main() {
    printf("Hello world\n");
    return 0;
}
printf("This is outside a function\n");
```

- In C, you **cannot** have executable code outside of a function
  - (Some languages allow this, but not C)

```c
int x1 = 10;
int x2 = x1 * 2;

int main() {
  printf("x1 = %d, x2 = %d\n", x1, x2);
  return 0;
}
```

- This code also **won't compile**

```c
#define X1 10
#define X2 (X1 * 2)

int main() {
  printf("X1 = %d, X2 = %d\n", X1, X2);
  return 0;
}
```

- This works!
  - One of the benefits of using #define

C Standard Library

- <assert.h>
- <complex.h> (C99)
- <ctype.h>
- <errno.h>
- <fenv.h> (C99)
- <float.h>
- <inttypes.h> (C99)
- <iso646.h> (C95)
- <limits.h>
- <locale.h>
- <math.h>
- <setjmp.h>
- <signal.h>
- <stdalign.h> (C11)
- <stdarg.h>
- <stdatomic.h> (C11)
- <stdbool.h> (C99)
- <stddef.h>
- <stdint.h> (C99)
- <stdio.h>
- <stdlib.h>
- <stdnoreturn.h> (C11)
- <string.h>
- <tgmath.h> (C99)
- <threads.h> (C11)
- <time.h>
- <uchar.h> (C11)

- The Standard Library provides functions for tasks such as input/output, string manipulation, mathematics, files, and memory allocation.
  - Eg, **printf**
- The Standard Library does not provide functions that are dependent on specific hardware or operating systems, like graphics, sound, or networking.

- ***(From Wikipedia)*** Many other implementations exist, provided with both various operating systems and C compilers. Some of the popular implementations are the following:
  - The BSD libc, various implementations distributed with BSD-derived operating systems
  - GNU C Library (glibc), used in GNU Hurd, GNU/kFreeBSD and Linux
  - Microsoft C run-time library, part of Microsoft Visual C++
  - dietlibc, an alternative small implementation of the C standard library (MMU-less)
  - µClibc, a C standard library for embedded µClinux systems (MMU-less)
    - uclibc-ng, an embedded C library, fork of µClibc, still maintained, with memory management unit (MMU) support
  - Newlib, a C standard library for embedded systems (MMU-less)[5] and used in the Cygwin GNU distribution for Windows
  - klibc, primarily for booting Linux systems
  - musl, another lightweight C standard library implementation for Linux systems[6]
  - Bionic, originally developed by Google for the Android embedded system operating system, derived from BSD libc