C Programming Helper

Prof Bill - Mar 2018

This is a little helper document for quickly plunging into the C Programming Language.

The sections are:

A. Introduction
B. The Command Line
C. C Coding
D. My Program #1 Stuff

thanks...yow, bill

A. Introduction

The C programming language is 45 years old (gasp), and it's the language I used in my first real nerd job out of school (gasp gasp). It's still important and used today.

The **book**:

The C Programming Language, ANSI C
by Brian Kernighan and Dennis Ritchie

Here's a pretty nice PDF of the original K&R book. The page numbers are off, but still…

The C programming Language (PDF)

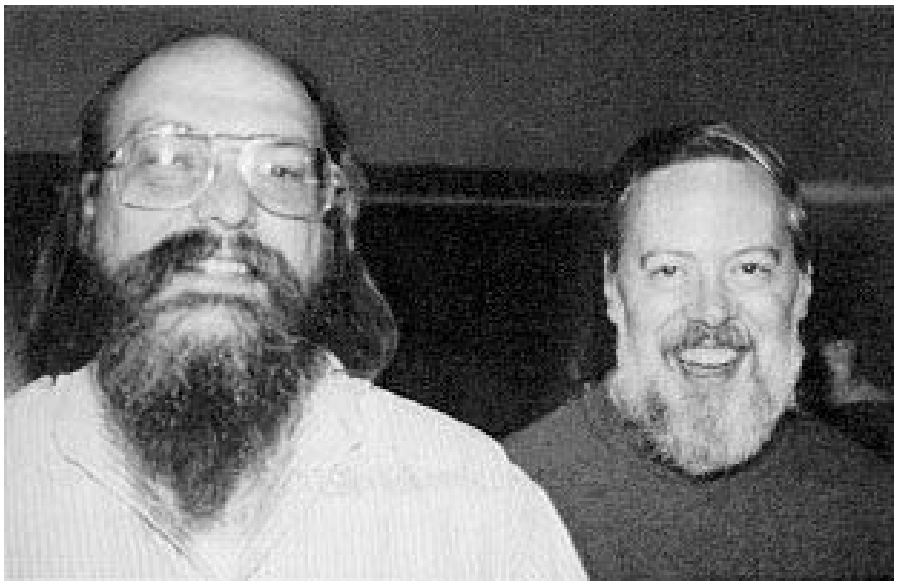
The 2nd edition should work. It's here in a bunch of formats:

archive.org/details/CProgrammingLanguage2ndEditionByBrianW.KernighanDennisM.Ritchie

A little **history**: "The origin of C is closely tied to the development of the Unix operating system" and "Also in 1972, a large part of Unix was rewritten in C.[13] By 1973, with the addition of struct types, the C language had become powerful enough that most of the Unix's kernel was now in C."

You get a nice overview from Wikipedia:

en.wikipedia.org/wiki/C_(programming_language)



C is a lower-level programming language than Java. It doesn't have objects or classes or exceptions or interfaces or abstract thingies. A C program is a collection of functions. There's no garbage collection; you allocate and free memory yourself. C has pointers. We'll learn a lot about pointers in our programming assignment.

Fwiw, here's a big comparison table: introcs.cs.princeton.edu/java/faq/c2java.html
thanks… yow, bill

B. The Command Line

IDE's like Eclipse and NetBeans are great. Hey, so are windows and the mouse and GUI's and… But typing commands at a **command line** or shell is an important skill for a programmer. Here are two ways to get started.

- On Windows, run **cmd**
- On Mac, run the **Terminal** application in Utilities

From the command line, file system from the command line:

- cd <folder> - change directory
- ls - list files

In Windows, you can use Notepad++ to edit your C programs.

Compile C programs with the Gnu C compiler: gcc.gnu.org/

- The **gcc** compiler is available on our school computers. So is the debugger, **gdb**.
- If you have a Mac at home (like I do), gcc is included in Mac's Xcode library. I couldn't find gdd on my Mac, but (google google) a program called **lldb** was available and worked fine: lldb.llvm.org.

Here are some common gcc commands/options:

```
# compile hello.c, creating executable output a.out
gcc hello.c

# compile hello.c, creating executable output hello
gcc hello.c -o hello

# compile hello.c, with extra files for debugging
gcc hello.c -g -o hello
gdb hello

#compile hello.c to create hello.o, but no executable
gcc -c hello.c
```

Here's a nice, short summary of gdb debugging commands:
www.tutorialspoint.com/gnu_debugger/gdb_commands.htm

The gdb commands I use most are:

- ➔ l - list code, so you can see where to set breakpoints
- ➔ l <line-num> - list code at line number
- ➔ b <line-num> - set breakpoint at line-num
- ➔ run - run program to the next breakpoint
- ➔ n - next, execute one line
- ➔ s - step, execute one line of code, but step into function calls
- ➔ p <variable> - print value of a variable
- ➔ help
- ➔ q - quit

/* When I installed gcc on my MacBook, gdb was not there. (google google) So, on my Mac, I use lldb instead. The interface is nearly identical. */

C. C Coding

There are a million online **tutorials** and other resources. I'll leave you to your googling. Of the stuff I've looked at, I think I liked this one best. We'll use this in class some.

> www.learn-c.org

Here are two different, easy summaries of functions in the C standard library:

> www.tutorialspoint.com/c_standard_library/index.htm
> en.wikipedia.org/wiki/C_standard_library

The most important of these are:

- ➢ stdio.h - always include this
- ➢ stdlib.h - you can always include this one too; includes malloc() and free()
- ➢ string.h - string functions

Naming conventions

The main thing here: be consistent. I chose this style from a stackoverlfow post.

> The most important thing here is consistency. That said, I follow the GTK+ coding convention, which can be summarized as follows:
>
> 1. All macros and constants in caps: MAX_BUFFER_SIZE, TRACKING_ID_PREFIX.
> 2. Struct names and typedef's in camelcase: GtkWidget, TrackingOrder.
> 3. Functions that operate on structs: classic C style: gtk_widget_show(), tracking_order_process().
> 4. Pointers: nothing fancy here: GtkWidget *foo, TrackingOrder *bar.
> 5. Global variables: just don't use global variables. They are evil.
> 6. Functions that are there, but shouldn't be called directly, or have obscure uses, or whatever: one or more underscores at the beginning: _refrobnicate_data_tables(), _destroy_cache().

For example, in Program #1 I have a ItemList struct. I have functions like read_wheel_file(). And a constant like MAX_WHEEL_SIZE.

Pretend Objects

C isn't object-oriented. At all. But we can pretend… well, sort of.

Let's pretend to create an object (a "class" in Java) called Professor. A professor is a struct. We'll define the professor-related struct, a typedef name, and function definitions in one header file: **professor.h**.

```
#ifndef PROFESSOR_H
#define PROFESSOR_H

/* Part 1. Define the professor struct */
struct Professor {
     char * name;
     int dept_code;
     struct College *employer;
};
typedef struct Professor *Professor; /* Part 2. a nicer name */

/* Part 3. extern the prof-related functions */
extern Professor *new_professor( char *nm, int code, struct
college *emp);
extern void grade_programs( Professor *p, ProgramList
*programs)
extern int get_tenure( Professor *p)

#endif
```

The **#ifndef, #define and #endif statements** are there to guard the header file. They prevent the header file from being compiled more than once.

Part 1. The professor **struct** and its fields are defined here.

Part 2. A **typedef** makes it a little nicer to reference a professor as "Professor", rather than "struct professor".

Part 3. The **extern** allows functions in other files to call your Professor functions.

Notice the big difference here between C and Java. This object setup is all optional in C. In Java, it's all part of the language.

Some K&R notes

I read K&R and took some supplemental notes. These things struck me as important.

Ch 1 Tutorial Intro

- C printf is like printf in Java
- Use #define for constants; #define MAX_LINE 100
- All function arguments are pass-by-value. You can use the "address of" operator (&) to pass the address of a variable into a function. This is also called a "pointer to" the variable. See swap() function example in Chapter 5!
- In C, char array (char []) or char pointer (char *) is used to represent a string. You must allocate space for the string characters if using char *. Unlike JAva, these C strings are mutable.

Ch 2 Types, Operator, Expressions

- Only 4 basic built-in types in C: char, int, float, double

Ch 3 Control Flow

Ch 4 Functions

Ch 5 Pointers and Arrays

- See swap() function for an excellent example of pointers and the address of operator (&). Use & to effectively pass-by-reference.
- Pointers and arrays are very similar. char * is like char [].
- In C, you can define a pointer to a function. Cool. We won't need this though.

Ch 6 Structures

- A struct defines related fields, like a class in Java. No methods though!

```
struct Person {
  char *name;
  int dept_code;
  double hourly_salary;
};
```

- Access a field for struct with dot (.): p.name. Access field for a struct pointer using two-char arrow (->): p2->name. We usually deal with pointers to structs!
- structs can be self-referential, as in list nodes.
- typedef struct XXX XXX… to rename the struct and make code a little cleaner.

Ch 7 Input Output

- #include <stdio.h>... I always do this too: #include <stdlib.h>
- you can redirect stdin with < in command line. Redirect stdout with >.
    - program1 < test1.txt # test.txt is now stdin
    - program1 < test1.txt > test1_out.txt
- File access. Google fopen().
    - FILE *fp;
    - fp = fopen(“test.txt”, “r”);
- In Unix, programs return integers. exit(0) on success. exit(1), or any non-zero value on error.
- Use malloc or calloc for memory allocation. See examples. On diff: calloc zeroes out memory that has been allocated. Use free() to free up space.
    - int array1[100]; // static array
    - int *array2; // dynamic array
    - ip = (int *) calloc(100, sizeof(int));

Ch 8 Unix

We’re running on PC/Windows systems at school and won’t need this chapter.

D. My Program #1 stuff

I'm writing Program #1 too. I'll add notes from my implementation here.
My advice to you: Start early. Stay late. Email me if you get stuck.
This is a challenging program in a new programming language. Also, C doesn't have a lot of modern conveniences that we may be user to. Beware! thanks… yow, bill

Fri Mar 30

Together, we identified 3 classes in P1:

- dll.c - doubly-linked list
- wheel.c - the wheel
- program1.c - the program, includes main()

What kinds of functions should be in these classes?

- ➔ dll.c - linked list operations like: create, add, get, copy, remove, print...
- ➔ wheel.c - operations like our command set: first, last, reverse, spin…
- ➔ program1.c - main(), command_loop(), others?

Example: What should your main() look like? Here's some pseudocode.

```
/* functions in main.c */
main() {
        print fancy program #1 greeting
        create a new empty wheel
        call command loop to read command, execute it
        print fancy program #1 goodbye
}
command_loop() {
        /* coming soon… */
}
```

thanks… yow, bill

Sat Mar 31

You'll find "some help" in my common_area folder on the k: drive.

- some_help.[ch] - some I/O and string functions that are kind of nasty
- some_help_test.c - examples of how to call these functions

Copy my files if you want to use these functions. Some notes about this:

- ➔ Include them in your gcc command. Example: if you are working on program1.c:

```
$ gcc program1.c some_help.c
$ a
```

- ➔ Notice that you don't include the test driver (some_help_test.c) in your gcc command. It has a main() and this will cause your link step to fail. I wouldn't even bother copying the test driver… just peek at it to see how functions are called.
- ➔ Once compiled and linked, this will create an executable called a.exe, which is the default. While your coding, a.exe is fine… and fast to type, too.
- ➔ Notice how all my functions use a prefix: sh_. I do this so I won't conflict with other function names. I do this throughout: dl_, wheel_, etc.

My process. After some puttering, I have decided to start my Wheel (wheel.c) later. I am currently focusing on:

- program1.c - my main(), intro(), and then command loop (some help functions are nice here)
- dl_list.c - my doubly-linked list; I have dl_list_test.c which has its own main(); I add a function and then try calling it in my test driver

Once I'm satisfied that my program1 and dl_list basics work, then I'll add the meat to the sammich, wheel.c.

Changed my files to: my_list.c, my_list.h, and my_list_test.c. Much nicer.

- ❏ My MyList and Node structs are defined in my_list.c.
- ❏ I have a typedef in my_list.h to make MyList available; this also loses the struct in front of everything.

```
typedef struct MyList MyList;        /* lose the "struct" */
```

- ❏ With this, the first three functions in my header are:

```
extern MyList *ml_create();
extern void ml_delete( MyList *the_list);
extern MyList *ml_copy( MyList *the_list);
```

- ❏ Notice that I have to pass in a list to my functions. There's no object.method() in C. Also notice, I always use "MyList *". This means "a pointer to MyList". Without it, my functions expect the struct itself. this is a major difference between Java and C. In Java, everything is a pointer.

Mon Apr 2

I'll elaborate on these two **"extra topics"** in class:

- I added a Makefile to my program1 folder. The make command simplifies your tight loop: edit, compile, run.
    - Simple make example here: stackoverflow.com/questions/21548464/how-to-write-a-makefile-to-compile-a-simple-c-program
    - My example is on the k: drive, common_area/program1.
- If it's for you tough to get on campus, here are 2 online gcc environments. They both have "issues", but still…
    - www.c9.io - Cloud9 is great. You work in the cloud, but it's command + gcc and gdb, just like at school. The only issue: you need a credit card to signup. It's free; they just want to verify that you're a real person.
    - https://www.onlinegdb.com/online_c_compiler - Thanks to Maia P for this one. I haven't used it much. It looks fine for trying small things.
    - Regardless of where you work or how you get it done: 1) get it done, and 2) ALWAYS run your program here at school before turning it in.

More on my process…

- ➢ Get my doubly-linked list working first: my_list.[ch]
- ➢ Then, get my command loop with one command going: program1.c. I think my first command actually was help.
- ➢ Then I started in on my wheel functions: wheel[ch]. I did this one at a time. **Critical!** It's absolutely critical to code and test each small feature before you move on.
- ➢ My first feature: create an empty wheel in program1 main(), add the size command to the loop.
- ➢ Once your infrastructure is in place, just keep picking off commands one at a time. Start with the simplest and work to the big ending: spin!
- ➢ I'm leaving my file commands for last.

C stuff that I'm using:

- ➔ To generate **random** things, you'll use the C functions srand() and rand(). https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm
- ➔ I include **<stdio.h>** and **<stdlib.h>** to get all the standard C library functions
- ➔ Use **malloc()** to allocate memory, like a Node or MyList. Use **free()** to delete.
- ➔ Some handy string functions (char * is a string) include **strlen()**, **strncmp()**, **strcpy()**.

- ➔ There's some good file stuff in my some_help functions: **fgets()**, **scanf()**. But I wrote some_help, so that you could avoid some of the tough patches.
- ➔ Java only has pointers. C has structs directly or pointers. In our app, we'll always use pointers, so variables will almost always have a "*" in them. For example, in my_list.c:

```
Node *n;
```

my_list.c notes:

- the Node struct is defined in my_list.c. There's no extern or anything in the my_list.h header file because it's only used internal to my_list.c
- copy item strings when they are added to the lists; use my some_help function here.
- when you delete a list, you must free the memory associated with each Node!
- when you reverse your list… do it in place. No creating a whole new list and adding tail-first. Efficiency!

Design - Your doubly-linked list should have **no** wheel references in it. This is just a general-purpose linked list. The Wheel is just one of many potential users of this code. In my P1, the Wheel has-a MyList. This means that MyList knows nothing about Wheel.

wheel.c notes:

- Wheel has-a MyList. That means that MyList is a member of the Wheel struct:
- Many of these functions are very thin… a line or two. They are mostly just calls to a MyList function that does the heavy lifting.

Design - Your Wheel should have **no** references to program1 in it. A good test of this: Could a gui use the wheel.c code you written, just as program1 uses it for a console app?

Thu Apr 5

Last on my TODO list… files.

file write - aka save your wheel; use fopen() and fprintf(); here's the setup:

```
char *file_name = "test.txt";
FILE *fp = fopen( file_name, "w");
if( fp == NULL) {
    printf( "Error: can't open file=%s", file_name);
}
else {
    fprintf( fp, "Hello, Bill\n");
    // more code...
    fclose( fp);          // must close your file!
}
```

file read - aka read your wheel; use fopen() and my some help function; go:

```
#define MAX_LINE 256

char line[MAX_LINE];
char *file_name = "test.txt";

FILE *fp = fopen( file_name, "r");
if( fp == NULL) {
    printf( "Error: can't open file=%s", file_name);
}
else {
    int ret;
    ret = sh_get_line( line, MAX_LINE, fp);
    // if ret is 0, the EOF, else more code...
    fclose( fp);          // must close your file!
}
```