JardinIoT

C Coding Style and Conventions

Fall 2019

1 Introduction

This document shall serve as a brief introduction to C coding style, according to the standards that we will be following in this class. As in other languages, proper style is not enforced by the compiler, but is necessary in order to write clear and human-readable code. All support code will be written according to these standards outlined in this document. The first set of issues that we will touch upon are those that present a significant threat to the correctness of your C code. While code that violates these conventions will still compile, and may well run correctly, it comes with a much higher risk of bugs, and will be more susceptible to programmer mistakes. You should be careful to heed the warnings given here, and make sure your code conforms to these specifications, lest you accidentally introduce bugs that take you an inordinate amount of time to track down. While we won't be covering software engineering in this course, the ability to organize a program in a clear and logical manner transcends all course and professional boundaries. Organization can mean the difference between an incomprehensible,

unmaintainable monster of a program, and a lean, clear, easy-to-read masterpiece far easier to debug. Thus, we will be placing significant weight on organization, even though it isn't the main focus of this class. Please code accordingly, and pay attention to the tips and conventions laid out below.

2 Functions Without Arguments

Function arguments in C are passed just as they are in Java, in a comma-separated list enclosed by parentheses. If a function is called without arguments, the parentheses are still required, but nothing is placed between them. When declaring a function, parameters are specified according to the same syntax, with each parameter name preceded by a type name. In this respect, C functions behave like those of Java. However, when declaring a function that does not take any arguments, things get a bit more complicated. In Java, one simply omits the argument list, leaving the function declaration with a pair of empty parentheses, as below:

```
public int func();
```

In C, it is entirely possible to do exactly the same thing, leaving out the parameters in the declaration:

```
int func();
```

However, a C function that is declared in such a manner does not behave the same way. The compiler interprets the declaration as that of a function which can take any number of arguments, and will not check to ensure that you have passed in the proper number of arguments. Instead, declare zero-argument functions with the following:

```
int func(void);
```

This prevents confusion and ensures that you cannot call **`func()`**with additional arguments, which could cause debugging confusion down the line.

3 Stars and Spaces Forever

One of the more hotly debated issues in the realm of C coding style is the question of where to put the “star” character (**`*`**) when declaring a variable or function with a pointer type. This issue divides C programmers into two ideologically-distinct[1] groups. One group argues that the star should come first (i.e. **`int* ptr`**), as it is part of the variable's type (there is also precedence for this in C++); the other favors association of the star with the variable's name rather than its type (i.e. **`int *ptr;`**). Java programmers may leap to agree with the first group, as in Java all

[1] Really.

non-primitive types are effectively pointer types; however, there are compelling reasons to follow the second style, as CS033 will. This argument is based on a feature of C syntax: when multiple variables are declared in a single statement, the type declaration "distributes" over the variable names; hence, the following declaration will produce two new integer variables:

```
int i, j;
```

The same, however, is not true of the star operator. Consider the following declaration:

```
int* i, j;
```

Such a declaration is commonly interpreted to produce two variables which are both pointers to integers. However, this is not the case; instead, only **`i`** is declared as a pointer to an integer, with **`j`** being declared as merely an integer. To declare two pointers to integers, we must give each variable its own star:

```
int *i, *j;
```

This syntactic feature provides a compelling argument in favor of space-star declarations. For many data types, making a mistake can change the behavior of a program entirely. Consequently, this is among the more important stylistic guidelines contained within this document.

4 Program Organization

An important organizational issue that you must tackle as a C programmer is how to organize your program into files. In Java, file organization is straightforward --- each class gets its own file, and all definitions corresponding to that class belong in that file. A C program, however, does not have classes; each function belongs to the entire program. Consequently how functions should be organized into files is less clear. In general, you should group functions of similar or interdependent functionality into the same file --- for example, functions which operate on a particular data structure should all be grouped together.

4.1 Header Files

C programs use *header files* to share functions or other definitions between different parts of a program. These files should provide *only* functions which other parts of your program will need --- helper functions should not be declared in a header file.
Header files are sometimes also an appropriate location for **`struct`** definitions. If other parts of the program will make direct use of the **`struct`** fields, then the definition of that **`struct`** necessarily must appear in the header file. If that is not the case, it is better to hide the **`struct`**

definition in a *.c* file. A common practice is to use a **`typedef`** statement in a header file to declare a type for other parts of the program, and hide the definition of that type. For example:

```
typedef struct my_struct my_struct_t;
```

A **`typedef`** statement allows you to refer to objects of the first type with the second type --- in this example, a reference to a **`my_struct_t`** becomes the same as a reference to a **`struct my_struct`**. By convention, **`_t`** is appended to the new type name.

5 Stylistic Conventions

Now that we've covered the most important facets of proper style, we will be moving on to several less essential (but still important!) stylistic conventions of the C language. These conventions are primarily motivated by readability and clarity of code, and will not directly affect the correctness of your program, although they may well help you avoid bugs before you accidentally type them. Nevertheless, developing a consistent C style is important, so we will be grading on this.

5.1 Layout and Spacing

Functions should be of a reasonable length: you should not have to scroll down through your editor of choice to view an entire function body. Sometimes this may be unavoidable; in such cases, ensure that your functions are easily broken up into discrete units. You should not, however, sacrifice readability for length.

Specifically on the `main` function, you should strive to have no more than 200 lines of code. Having a long `main` function indicates a lack of abstraction, meaning that your code has not been sufficiently split into logical, smaller functions.

Further, your line lengths should be no more than 120 characters. This includes not only code, but also comments. Indentation should be consistent - please use a 4-space indent. Do not indent by a single space; this makes indentation extremely hard to follow.

Last, be sure to use a readable amount of horizontal spacing. For example, you should put spaces between operators.

```
(x+11)/(y%5)-z
```

Is far less readable than

```
(x + 11) / (y % 5) - z
```

5.2 Brace Yourselves...

Another stylistic choice is the placement of the opening curly brace around the code block which forms the body of a function, conditional, or loop. Here, there also seem to be two commonly-used options: one may either place the opening brace immediately after the function name or reserved word (with or without a space), or insert a newline before opening the code block. These styles, respectively, are shown below:

```
while (1) {
    ...
}

while (1)
{
    ...
}
```

As with the space star versus star space debate, there is no syntactic difference between these two ways of writing a while loop --- the C compiler treats all whitespace as a delimiter, without discriminating between spaces, tabs, or newlines. Any support code you receive from CS033 will place the opening brace on the same line as the function declaration or reserved word. We request that you do the same, as this is generally the preferred style in C programs.

There are, however, some rules about the placement of curly braces. For example, in the following block of code:

```
if (...)
    if (...)
        // this is a comment
```

Technically, if the code within the inner if statement is a single line, it will be executed as expected (if both conditions are met). However, this is *bad*. If you need to add more code to execute when both conditions are met, you will likely forget that you need curly braces. In part for this reason, you should instead do the following for nested if statements:

```
if (...) {
    if (...) {
        // this is a comment
    }
}
```

You may omit the curly braces in the case of a simple, non-nested, single-line if statement. However, we do not encourage this (for similar reasons as in the above example). In addition, you should never have nested loops without brackets.

5.3 Naming Conventions

Different programming languages observe different naming conventions for programs, functions, and variables. In Java, for example, the “CamelCase" convention is used:

```
public int countSomeItem(...) {
    ...
}
```

This convention is often employed by C programmers - for example, some of the lecture slides employ this convention. However, another common convention frequently found in C programs is to name program constructs using *underscores*:

```
int count_some_item(...) {
    ...
}
```

The C standard libraries abide by this convention, as do the support code files you will receive throughout this course. For example, the **<stdio.h>** library names types and functions using underscores (such as the **size_t** type and **rand_r()** function). In naming functions, we encourage you to use the underscoring conventions, but as long as you are consistent, CamelCase is acceptable as well. `For JardinIoT, use CamelCase.`

Preprocessor macros are often named differently still, combining the two conventions.

```
#define NUM_ROWS 10
```

Macros are typically defined using all uppercase characters, with underscores separating different words. As with the underscoring convention, C library functions abide by this convention - the **NULL** pointer defined in **<stdlib.h>**, for example, is actually a macro.

You will also find yourself often using structs in this class, which you will learn about very near the beginning of the semester. When naming struct types, similar to the example under section 4.1, you should follow a descriptive name with `_t`. For example, you might have a struct representing a calendar date named `date_t`.

As in all of your coding endeavors, you should strive to name variables clearly and descriptively, so that you can easily see what is going on in your code. Variables such as `foo`, `bar`, `a`, `b`,

`myvar123`, etc should not appear in any programs you write. And further, you should not name your variables with a single capital letter (`int M = 2`)[2].

You should follow the conventions laid out above. Note that this will lend your code additional consistency with the conventions used by the C standard libraries and staff-provided support code.

6 Commenting

6.1 Header Comments

Similar to the various introductory sequences you may have taken before CS0330, we require that you document your functions using header comments. Header comments should include a concise description of what the corresponding function does, an explanation for each argument, and an explanation for the output of the function, as in the (simple) example below:

```
/*
 * This function counts the number of instances of
 * an int in an array.
 *
 * item - the int to be found in the array
 * my_arr - the array to search in
 * returns the number of times item appears in my_arr
 */
public int countSomeItem(int item, int *my_arr) {
  …
}
```

This may seem a bit redundant in this example, but as your functions become more complex, documenting each function will become imperative to debugging and understanding your own code.

6.2 Inline Comments

As well as a header comment for each function, you should write inline comments to help explain particularly complicated sections of code. Of course, it is best to write code that is clear to begin with, but sometimes it is not possible to do so, especially in the more complex projects you will be implementing in this class. However, you should not find yourself writing inline comments on every other line of your code– if you find yourself doing so, you may want to reevaluate your logic and see if there is a clearer way to write the code.

---

[2] Single letters are not descriptive, so they should not appear as variables whether they are capitalized or lowercase!

6.3 Extraneous Comments

There are some comments you should *not* leave in your code. You might find yourself using printlines or other debugging code in your programs. Before handing in, you should *delete* these statements, not just comment them (and do not leave debug code in the file!). In fact, you should not have commented code anywhere in your files. Many source files we provide contain commented TODOs to help guide you along the way. These TODOs should also be deleted.

7 README

The README is where you document a high-level overview about how your program functions, list which students helped you debug (by login), and discuss any known bugs in your program. The README is not a place to detail every single function you wrote; rather, it is a summary of how those parts work together. Failing to document an obvious bug (i.e “the program segfaults immediately”) will result in a deduction, as will an overly brief project summary. Don’t feel as though you should spend an inordinate amount of time on this, though. Write as much as you feel would be necessary for another programmer to understand your project on a high level.

8 Logic/Pointers

8.1 Logic

There are various indicators of poor logic in a program. Each of these is considered a style mistake; we’ll briefly list them below:

I. Using `while(1)` and then breaking out of the loop once a certain condition is met. What you should do instead is `while(/*condition is not met*/)`. This is acceptable in very specific cases, such as in REPLs, but otherwise it is generally a symptom of poor program logic. And definitely do not do this with a `for(...)` loop.
II. Empty brackets in an `if`/`else` or an `ifdef`, such as `if(...){}else{...}`.
III. Using `assert(false)` or similar for the purpose of exiting a program early.
IV. Creating a variable just to return it such as `int ret = 3 * mul + 7; return ret;` Instead you should simply have `return (3 * mul + 7);`
V. Creating multiple bit masks when one would suffice.
VI. Using an `if`/`else` statement to return the value of the condition i.e. `if(cond) return true; else return false;` instead you should return `cond`.
VII. Excessive/unnecessary global variables.
VIII. Redundant code; this includes not creating a function for something you do more than one time, or doing the same calculations multiple times unnecessarily.
IX. Using `goto` at any point.

```
Important: In some of these examples, the single space is missing between
keywords. Do not use them as an example of how you should code.
```

8.2 Pointers and Syntactic Sugar

There are some style don'ts relating to pointers as well, which are outlined here:

I. Passing copies of structs instead of using pointers. This is just not the C way of doing things; pointers are your friends!
II. Misusing pointers such as making a pointer to an array, an array of pointers, etc. You'll learn about the proper usage of pointers at the beginning of the semester, and will likely never feel the need to misuse pointers in this way.
III. Using `(*p).val` instead of `p->val`. Manually dereferencing the pointer and then accessing its field is more work than simply using the syntactic sugar available to access the field; use the `->` syntax!
IV. Using array variables as pointers rather than indexing in, i.e. doing `*(arr + i)` rather than `arr[i]`, can lead to mistakes, since in the former way of accessing an element, `i` changes based on what type of array `arr` is. `arr[i]` is much cleaner and clearer.

8 The Reformat Script

There is a script available to help you format your files. It's found in the `/course/cs0330/bin` folder, and you can run it as follows:

```
cs0330_reformat <file1> <file2> ...
```

Note that this will only work on `.c` and `.h` files. Unfortunately, because of `clang-format`[3] limitations, it cannot address all style requirements outlined in this guide. However, at a minimum you should run `cs0330_reformat` on all your `.c` and `.h` files before handing in. Note that the script makes some additional changes not described in this style guide, such as operator spacing and comment formatting.

To reformat all `.c` and `.h` files in the current directory, run the following command:

```
cs0330_reformat *.c *.h
```

[3] `clang-format` is a C/C++ formatter that's part of the LLVM compiler infrastructure project