C Coding Style and Conventions

2015

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.

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 paramenters 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 declaration ensures that no arguments can be passed to `func()` at all.

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 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 recommendations contained within this document.

4 Program Organization

An important organization 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 defitions 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.

[1] Really.

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, we strongly recommend that you absorb and follow them so that you can develop a consistent C style.

5.1 Function Length

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.

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 differnce between these two ways of writing a while loop — the C compiler treats all whitespace as a delimiter, without discrimitating between spaces, tabs, or newlines. Use whichever convention you prefer, but please do so consistently. Any support code you receive from CS033 will place the opening brace on the same line as the function declaration or reserved word.

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).

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.

Choice of convention is up to you. Note that using underscores may lend your code additional consistency with the conventions used by the C standard libraries and staff-provided support code. Regardless of what you choose, please write consistent and readable code.