C Coding Style and Conventions

2012

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; while some of these conventions are more mandatory than others (we will make clear which of them are most necessary; for the most part, the document is organized in order of decreasing necessity), they are all important, and we highly recommend that all students internalize and follow the guidelines set forth by this document. Most likely, you will find that it helps prevent mistakes that would later turn into bugs of the most difficult and annoying nature; at worst, you will have written a program that not only runs, but also looks good to a perusing eye (and impressing your grader is always a good thing!).

2 Major Issues; Program Organization

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 that even a novice could 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 layed out below.

2.1 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();
```

This should behave just like the Java equivalent, right? Unfortunately, that is not the case. A C function declared without any parameters is actually declared as a function that can take any number of arguments, and the compiler will not check to make sure you have passed in the proper number (in this case, none) when the function is subsequently called. While in most cases, this won't lead directly to bugs — if too many arguments are passed in, extra space will simply be allocated on the stack, but never accessed — it might lead to confusion later on, should you forget that the function doesn't take arguments, or accidentally confuse it with a similarly-named function that does. And confusion most certainly can lead to very strange bugs, that will be difficult to track down.

To avoid this issue and ensure that the compiler checks the argument structure of functions that take no arguments, we instead declare functions without arguments with the parameter `void`, as follows:

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

Now, if we attempt to call this function with arguments, the compiler will catch our mistake, and we won't be left scratching our heads, wondering where that incomprehensible and very incorrect value is coming from.

2.2 Function Length

In C, as in Java or any other programming language, functions should be kept to a reasonable length, to improve readability and encourage modularity and separation of function (pun intended!). There will be, of course, cases in which a function must be quite long (especially when doing parsing/string manipulation, or in the main interactive loop of a large program), but these should be kept to a minimum, and they should be easily broken up into discrete units. In general, though, you should be able to view the entirety (or close to it) of a function without using the scroll bar of your editor (assuming, of course, that the window is fairly large or full-screen). We won't be using any sort of absolute cut-off for function length when grading; however, we will take the length of each function into account when assessing its stylistic acceptability, and overly long functions will be viewed more negatively than shorter ones with similar functionality.

This doesn't, however, mean that you have to go out of your way to make every function as short as humanly possible, especially if that would be at the expense of readability — you need not apply every trick of C syntax to shave lines off of your functions. If a slightly longer function turns out to be significantly more readable, then by all means write it that way — extremely compact but unreadable code will also lose you points.

2.3 Sorting Functions into Files

Another organizational issue that must be dealt with is the organization of C functions into files. In Java, file organization is straightforward — each class gets its own file, and the methods belonging to

that class, naturally, must be placed in the same file as the class. In Java, there are no independant functions that are not associated with a class; thus, there is no need to worry about which file to put a given function into.

In C, on the other hand, there are no classes whatsoever, and thus the organization of functions in files becomes a little bit less clear. C programs are organized around files — there is no other unit of access control, unlike in Java where the class forms the basis of function or variable access. Generally, one groups functions that have a common functionality — such as the set of functions used to manimpulate data in a particular struct or other data structure — into a single file, with the definition of the data structure(s) or struct(s) at the top of the file. This convention is akin to that of Java, although one should avoid putting too much weight on that analogy, since C structs and Java classes are completely different animals. However, what the two languages do have in common is that they both seek to group functions with a common functionality together.

2.3.1 Header Files

In C, functions can only be called after they have been declared, and below their declaration in a given file. You've seen forward declarations in the C primer document — they look exactly like function definitions, only without the body — and you've probably been exposed to the practice of grouping them together in a *Header File* for easy inclusion in a program (by means of the `#include` statement). However, it bears mentioning that a function should only be included in a header file if it will be called outside of the file in which it is defined. Separation of functionality demands that functions which should not be called elsewhere should not be available elsewhere. Thus, helper functions (defined as those functions used only within a file or library) should not be included in header files, nor should they be forward-declared unless absolutely necessary (e.g., if they will be utilizing mutual recursion). Also, you will need to think long and hard about whether or not you want to put the definition of a struct in the header file or a *.c* file — sometimes, it is beneficial or necessary to be able to access the fields of a the struct directly in another file, in which case you'll want to define the struct in the header file. However, if that isn't the case, it's usually better to hide the internal workings of the struct inside of the *.c* file containing the functions that will need access to its fields. To allow a struct, but not its inner workings, to be used in other files, it is common practice to place a `typedef` statement in the header file, but keep the struct itself hidden from view.

3 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 find avoid bugs before you accidentally type them. Nevertheless, we strongly recommend that you absorb and follow them, as it wil make your (and our!) lives easier.

3.1 The 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. Most programmers seems to agree that there should be one star (or one group of stars) and one space between the type and the variable or function name, but the order of the space and the star is the source of many an argument. One camp argues that the star should come first, as it is part of the variable's type (there is also precedence for this in C++). This is probably the most intuitive style for Java programmers (at first at least), who are used to treating all non-primitive types as pointer types. Code following this convention would declare a pointer to an integer thus:

```
int* ptr;
```

However, the other camp argues that the space should come first, with the star on the right, touching the variable or function name, like this:

```
int *ptr;
```

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. Imagine we include the following declaration in a C program:

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

One might (logically) assume that the result would be two pointers to integers, but that is not the case. Instead, `i` is declared as a pointer to an integer, but `j` is merely an integer, which is not what we wanted at all! To declare two pointers to integers, we must give each variable its own star:

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

Space star advocates argue that, since the star does not "distribute" in the same manner as the type name, it should be grouped with the variable name, rather than with the type. Because of this syntactic argument, we have decided to follow the space star convention in all support code for this class. While you are free to follow whichever standard best suits your liking, we highly recommend space star over star space, as it may help avoid bugs similar to the one depicted above. This probably the most important of the style recommendations that we will make in this document, due to the potential for weird bugs, so please keep that in mind when writing your first assignments.

3.2 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 and newlines (or tabs, but that's for later!). Thus, you are welcome to use either of these conventions in your own code, as long as you use only one consistantly. All support code will place the opening brace on the same line as a function name or the reserved word introducing a loop or conditional (with a space between the two), as this seems to be the dominant style among the course staff, but you should not feel obligated to do so — there isn't a clear syntactic argument for either convention, and neither seems particularly likely to cause or help avoid bugs. However, following this convention will make your code more readable to the TAs, which can only be a good thing.

3.3 Keeping Tabs on Spaces

Now that we've covered the basics of C style, we will open another, less obvious can of worms: the infamous tabs versus spaces debate. When indenting code, one must choose between indenting by tabs ('\t' characters) and indenting by spaces (usually inserted in groups of four). If you've previously done most of your coding in an editor (such as Eclipse) that handles indentation for you, you may not be aware of this contentious issue. For example, look at these two functions. The top one uses tabs for indentation; the bottom, spaces:

```
int incr(int n) {
    return n + 1;
}

int incr(int n) {
    return n + 1;
}
```

See the difference? No? Well, there isn't one that you can see, and that's part of the problem. Assuming that the correct number of spaces are used, tabs and spaces look exactly the same to the

user. Thus, readability may seem not to be at stake here. However, there is an additional twist: different editors use different tab sizes, allowing the same program to have different amounts of indentation in different environments. Confounding this all is the fact that most editors may be configured to insert spaces rather than a tab character when the TAB key is pressed, meaning that the user may not be aware of which convention they are using. All course documents will use tab characters for indentation, and we gently recommend that students do the same, as we consider them somewhat more intuitive than spaces. However, this is one of the least important decisions that you will have to make; as long as you use only one convention in a given program, it doesn't really matter which one you choose.

If you're wondering which convention your editor is following, go to "Preferences" and look for an option that allows you to choose between inserting spaces and inserting tab characters. Some editors default to inserting spaces, whereas others default to using tab characters. Here, we provide instructions on how to configure tabs versus spaces in a few common editors.

3.3.1 Gedit

If you use Gedit as your main editor, you can configure it to use tabs thus: go to *Edit* → *Preferences* → *Editor* and make sure the option to "insert spaces instead of tabs" is unchecked.

3.3.2 Kate

Kate users should go to *Settings* → *Configure Kate...* → *Editing* and uncheck the box marked "Insert spaces instead of tabulators".

3.3.3 Vim

Vim, unfortunately, is a bit more complicated than the editors we were just discussing. Most likely, if you haven't configured it otherwise, vim will be set to use tabs rather than spaces, but it never hurts to double check. If you do need to reconfigure vim to use tabs, it is usually enough to run the command `:set noexpandtab` in vim. However, this won't persist beyond your current session; if you want to make the configuration permanent, you'll want to add the line `set noexpandtab` to your *.vimrc* file.

3.3.4 Emacs

Can someone who knows emacs do this? And can someone who knows vim better than me double-check the last section?

3.3.5 Checking Your Editor

If you're unsure about which convention your editor is using, a failsafe way of checking is to type a few tab characters into an empty file using the editor, then close the editor and run `hexdump -c <filename>`. This will print out the file to the terminal as its raw characters, which can then be inspected to determine if they are indeed tabs, or instead spaces masquerading in their place. As a brief aside, `hexdump` is a very useful utility to know about, especially when you're wondering

just what, exactly, is actually going into a file when you save it with another program. You should check out its man page at some point when you have spare time.

3.4 Underscores

Finally, we shall deal with another issue specific to the C language: the use of underscores in multi-word function and variable names. In other languages, such as Java and OCaML, names consisting of multiple English words are handled by capitalizing the first letter of each word, which is often referred to as "Camlcase" (presumeably after OCaML). However, this convention is frowned upon in C, for reasons that nobody is certain of. Instead, C programmers separate words with underscores, and don't capitalize them. For example, look at how the following function would be named in Java:

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

and in C:

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

While a C program will, of course, still compile if you violate this convention, it will look strange to experienced C programmers, and many of the same find violations of this convention annoying or even insulting. The support code that you receive will always use underscores, and you should do so as well, although we won't be grading you on it.

4 Final Words

Above all, be consistant, especially within a given program or assignment. Mixing styles looks terrible, and can become a nightmare to read, especially when dealing with pointers to pointers, large numbers of functions declared both in header files and in your *.c* files, and so on. With regards to stylistic concerns, inconsistancy in your coding style will likely lose you style points on an assignment, whereas your choice of convention probably won't. However, it is advisable to learn the conventions now, since small things like coding style can greatly affect how you are perceived by industry recruiters. Additionally, the major issues discussed at the top of this document are very important for proper functionality, and in software engineering. Although this is not a software engineering course, we believe that proper design techniques should be used from the very beginning, and thus organization and style will be assigned a not-insignificant value. Hopefully, you will find this information helpful, both in writing your assignments for this course, and in all your computer science experiences to come.