The Official CS50 C Style Guide

This document outlines a set of style rules used by many programmers. It is based on KNF (Kernel Normal Form) and is the style used in the Berkeley derivative of the Unix operating system. You will find that a great deal of freely available software is written in this style. If you write in this style, your code will look natural to other C programmers and will be easy to read.

The sections on comments, constants and macros, file inclusion, and characters and strings are essential rules. You should follow them regardless of how much programming you have already done. If you have never programmed before, you should follow the rest of this document as well. If you have programmed extensively, you might have your own style; we will not force you to follow our style, but you should read through the following sections carefully so that you understand the components of good style. Most code reading assignments and all code for assignments will be written in this style, so you may find them easier to understand if you write code in this style as well.

When we grade for style, we consider the following properties:

1. Readability – is the code easy to read? Does the reader have to struggle to figure out where blocks of code begin and end or it is apparent from the formatting? Is the control flow apparent from the indentation?
2. Consistency – are things always done the same way? All statements of the same type (e.g. **if** statements) should look the same. If you have to write a statement that spans multiple lines, you should format it consistently each time.
3. Proper indentation – human factors analysis dictates that 4-character indentation is the minimum that the eye can discern easily. You must use *at least* 4-character indentation and we recommend 8 (a full tab stop). If your code is indented so much that it's difficult to read, think about restructuring it. After 3 or 4 levels of indentation, your design can likely be improved.
4. Modifiability – would it be easy for someone to come in and modify your code? This means that all constants are defined with **#define** or **enum**, the program is decomposed into logical units that can be modified or replaced, and appropriately placed comments clearly describe what is going on.

If you follow the guidelines below, you should get superior scores on style. If you use your own style but follow the basic principles we outline, you should also get full credit, but if you make up your own style and it does not meet our criteria, you will lose credit.

1. Comments

Comments are among the most important parts of any program: if another programmer reading your code can't understand it, there's something seriously wrong, no matter how well the program works.

Single-line comments are used to clarify a tricky line of code or state the function of a loop within a function. A comment on a line by itself lines up with the first line of code it describes. If it shares a line with code, it lines up with surrounding comments. The */ that ends a single-line comment follows immediately after the end of the text.

```
/* Most single-line comments look like this. */

int  i;     /* counter variable */
char mybuf; /* temporary buffer */
```

Any comment extending for more than one line or very important comments should be opened and closed at the left margin. Comments like these should be complete sentences.

```
/*
 * This is a VERY important single-line comment.
 */

/*   Comments that take up more than one line should be
 * carefully formatted for easy readability. No one wants
 * to have to wade through comments as difficult to read
 * as the code they attempt to clarify.
 *   Format longer comments in sentences and paragraphs;
 * write them just as you would any other English text.
 */
```

Do not use redundant or obvious comments. You should place a comment at the beginning of each function stating what it does and a comment about the program before the main() function. Each global variable should have a comment explaining its role in the program. Comment consistently and clearly, and your code will shine.

2. Variables

The proper choice of variables is one of the most important parts of program design; a perfect algorithm is useless without well-planned data structures. It thus makes sense that the naming and use of these variables in the written code is also extremely important. Use descriptive names for variables. If you name your functions and variables well, you will often find that your code is self-commenting.

Use the appropriate variables for the job. If you can't justify using a new variable, you probably don't need it. On the other hand, don't use too few variables; programmers who attempt to save a few bytes of memory by re-using variables within a function open their programs to myriad bugs.

Choose the scope of a variable appropriately. If every function in the entire program needs to utilize a particular data structure, then a global variable might be appropriate. Sometimes a variable is used by every function in a particular file, and functions outside this file need not access this variable directly. In this case, a variable declared in the file before any functions will do the job. When declaring variables with a large scope (i.e. more than one function), justify why using such a variable is better than passing the data structure as an argument to the functions using it. You may find that your program makes more sense if the data structure is passed from function to function, and the functions are certainly more portable.

When choosing names for variables, avoid cryptic abbreviations and names several characters long. A variable named s_tr is succinct, but not nearly as clear as saved_tree. However, don't use names like original_saved_binary_tree — such long names are unwieldy and not a benefit to clear coding. Conventional short names like i and j for loop indices are preferable as long as their scope is limited.

Variables are declared following the compiler directives (#include, #define) at the beginning of a file or immediately following an open curly brace (**{**). Write the variable type (or keyword, if applicable), a tab, and then the rest of the line with one space between each element. The only exception to this is when initializing several variables. In this case, line up the assignment operators (**=**) with tabs. To declare a pointer, place the asterisk immediately before the pointer name for each pointer declared. If there are too many variables of one type to fit on one line, use a semicolon at the end of the line and redeclare the type.

Arrays are a special type of variable whose internal representation we are not concerned with at this point. They nevertheless raise some important style issues. With the possible exception of strings (arrays of chars), arrays declared as such should be referenced later in the program as arrays, not pointers. The brackets surrounding the array index should immediately follow the name of the array, and have no extraneous spaces within. Computations of array indices should be enclosed in parentheses inside the brackets for clarity.

It is also possible to initialize arrays by following the array declaration with an assignment operator and a set of values formatted in the same manner as an enumerated type. It goes without saying that the cardinality of the initialization set should be equal to the size of the declared array. If the initialization spans multiple lines, line up the first value in each line, and if possible, line up the elements themselves.

To make a variable global, declare it in a .c file and add an identical declaration preceded by the extern keyword in the corresponding .h file.

```
/* foo.h */

#define FOO_LENGTH       50
#define FOO_ODD          7
#define FOO_EVEN         8

extern   char     foo_lr[(FOO_ODD + FOO_EVEN)];
extern   int      foo_array[FOO_LENGTH];

-----------------------------------------------

/* foo.c */

#include "foo.h"

int      foo_lr[(FOO_ODD + FOO_EVEN)] =
    { 1, 3, 5, 7, 9, 11, 13,
      0, 2, 4, 6, 8, 10, 12, 14 };
int      foo_array[FOO_LENGTH];
```

```
int main(int argc, char *argv[])
{
        static  int unchanging;
        char    *buff, weak, *strong, long_char_var;
        char    another_char, *a_string;
        ...
-----------------------------------------------
/* bar.c */

#include "foo.h"

int barf_unction (int level, char str[], int len)
{
        int     i; /* Quintessential counter */

        for (i = 0; i < len; i++) {
                int     anointing_index = i;
                double  precision       = BAR_PRECISION;

                printf("Eww! Barf unction? Oh, a typo!"
                    "Should be bar_function! ;-)\n");
                ...
        }
}
```

3. Including other files

The inclusion of other files is an integral part of C programming; it might seem that such a simple item is hardly worth mentioning here. There are nevertheless a few important points to make about the #include compiler directive:

- Include files always go at the top of the file; it makes clear exactly what is being included and helps avoid bugs from multiply defined data.
- Never #include any .c file. If the functions in that file are important enough to use in another file, you should write a header (.h) file for them and include it.
- Comments for an include file are placed in a standard single-line comment immediately above the #include line.
- Standard include files (generally found in /usr/include) are enclosed in angle brackets; local include files (those you use from the current working directory) are in double quotes.

```
/* Comments for an included file go here. */
#include <stdio.h>

/* For the weird library I wrote */
#include "weird.h"
```

4. Constants and macros

The C programming language, like most modern languages, provides for giving constant (i.e.,unchanging) data a special name for use within programs, allowing for easy identification and modification of all instances. It is poor style not to make use of this, because not only are literal "magic" numbers difficult to change, but they also make the code hard to understand. Consider:

```
/* The next line is bad style. What's that @#$% number? */
total = (subtotal * 0.06125);
```

Someone reading the program might not know whether the number was a surcharge, a tax rate, or even a mistake. This is called "hard coding" a number, and should be avoided. Use the #define compiler directive, a tab, the name of the constant in capitals, and another tab. If defining several constants together, line up the values with tabs.

```
/* The next line goes at the beginning of the program. */
#define SALES_TAX_RT    0.06125
#define DISCOUNT        0.1
...
/* Much better style! Now it's unmistakably clear. */
total = ((subtotal - (subtotal * DISCOUNT)) * SALES_TAX_RT);
```

Another note on constants: when the preprocessor encounters a #define, it literally replaces every instance of the item immediately following the #define compiler directive with the remainder of the line. This can be a problem. Do not terminate #define statements with semicolons:

```
#define PI 3.14159;    /* disaster! */
```

since the semicolon will cause an errors whenever PI is included in an expression. Similarly, make sure that any comment in a #define is properly terminated:

```
#define FOOBAR "What do foo and bar mean?" /* Watch out
                                              for this */
printf(FOOBAR "\n");
expands to
   printf ("What do foo and bar mean?" /* Watch out "\n");
```

When FOOBAR is replaced with the rest of the line later on in the program (as shown in italics), it could end up adding an unwanted open comment, which would at least cause an error and could invisibly render part of your code useless.

Macros[1] also use the #define compiler directive; they are essentially constants with arguments and can be used as miniature functions. Resist the temptation to use them for anything but the most simple C functions; this is poor style.

Macros are generally useful if you have a specific formula that you use frequently in your program. If a macro spans more than one line, use a space before a backslash (\) at the end of each line's text. Do not use comments inside macros. Enclose all arguments within parentheses, avoid side-effects (altering data beyond the scope of the macro), and

1. For a more detailed discussion of macros, consult *The C Programming Language* (K&R), pp. 89-91.

capitalize the macro name to remind the programmer to watch out for the pitfalls macros often create.

```
#define DENSITY         50.1995
#define MASS(h, w)      ((h) * (w) \
                             * (DENSITY))
```

5. Type definitions, structs and enums

A great way to confuse anyone reading your code is to use complicated and poorly formatted type definitions. The simplest type definition is simply renaming an existing type for easier readability in the code. Here, we create the answer_t type from the existing char type, then define an array of our new type. The _t is a convenient method of indicating a defined type.

```
#define ANSWER_LEN 50;

typedef char answer_t;
typedef answer_t[ANSWER_LEN] AnswerArray;
```

Structs are very useful for creating complex data structures, and make data exchange in a program much cleaner when used judiciously. When defining a struct, use one variable per line, and put a single tab after the first word on each line. If you intend to use a struct only once, simply define the struct:

```
struct foo {
        int     first_foo;
        char    second_bar;
};
```

However, if you plan to use several instances of a defined struct, use a typedef for clarity. To keep things straight while defining the struct (particularly in self-referential structures), use an underscore before the actual type name appearing at the end of the definition:

```
/* Our linked list structure definition */
typedef struct  _llist_t {
        struct  _llist_t *next;
        struct  _llist_t *prev;
        int     data;
} llist_t;
```

Enums allow you to use mnemonic names for variables of type int. As opposed to #define, enums should be used when the *names* of several items are important rather than their having predefined values. You could use #define for this, but enum provides a much cleaner method. Write the name of the new type with the _t suffix, and separate the type from the **{** with a tab. Capitalize all elements, and keep one space between the elements and the curly braces, and use one space *after* the comma between elements. If an enum spans multiple lines, line up the first element on each line.

```
enum color_t { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET };
```

Now, anywhere in the program you can use variables of type color_t, knowing that the compiler will keep track of values for them. If the values are important, they can be specified with the syntax NAME = value. If only the first value is specified, the compiler will assign values to the succeeding elements in counting order (here 2, 3, ...).

```
enum month_t { JAN = 1, FEB, MAR, APR, MAY, JUN,
               JUL, AUG, SEP, OCT, NOV, DEC };
```

6. Functions

Every C program uses functions, and proper formatting of function declarations and definitions is imperative for understandability of your code. CS50 uses the ANSI C standard function definitions, but some C compilers do not support this standard. Thus, your function definition will begin in one of two ways:

ANSI C	Original C

`int main(int argc, char *argv[])`
`{`
	`int main(argc, argv)`
	`    int argc;`
	`    char *argv[];`
	`{`

Both are correct, though the ANSI standard is more compact and must be used for all assignments. We prefer to put the return type on the same line as the function. Another popular and acceptable style puts the type on the immediately preceding line. In either case, the opening curly brace appears on its own line after the function declaration.

When writing functions, you will call other functions as part of your program. It is important to make the distinction between functions (e.g., printf, exit, and malloc) and keywords (e.g., if, for, and return). When calling or declaring a function, use no space between the function name and the parentheses enclosing its arguments. Put one space between argument and its type, and one space after the commas separating the arguments. If your argument list extends over one line, indent the next line half of the normal indentation, to indicate one unit split over several lines.

7. Characters and Strings

C provides for other constants' use in programs as well. Unlike numbers, string and character constants are frequently used explicitly within a program.

To use a string in C, enclose it in double quotes ("). Neighboring strings are concatenated at compile time, so it is possible to have strings span several lines in your program. Remember to include newlines at the appropriate places without surrounding spaces. Indent following lines with half a normal indentation:

```
printf ("This is a very long string, because I need "
    "it to span multiple lines.\nIf I didn't need it "
    "to do so, it wouldn't be as long.\n");
```

It is possible to define constant strings. You may recall that the **#define** directive replaces the word with the rest of the line, and wonder how to define a multiple line string constant. As with macros, the backslash (\) will allow this. Line up these strings.

```
#define LONG_STRING "Because this string extends over " \
                    "two lines, I have broken it up.\n"
```

Character constants in C are simply shorthand for the ASCII[1] numeric representation of a character. If you wanted to check to see if the user typed a certain character, rather than looking up the number in a table, you can use the built-in character constant by surrounding the character in single quotes (’):

```
if (input == ’y’) {
```

Do not confuse `’y’` with `"y"` — the former is a number that represents the character y in ASCII; the latter is a string of length one consisting of the character y and a terminating NULL (`\0`) character. You may also use the backslash codes within single quotes for nonprintable characters, e.g., `’\0’` or `’\n’`.

8. Control flow

C uses certain keywords to help the programmer implement control flow in a program. These include if, for, while, do, and switch. When using these looping constructs, put a single space between the keyword and any parentheses that follow it. The keywords if, for, while, and do are frequently seen with curly braces surrounding one or more statements. The braces are not necessary if the keyword applies to only one statement, but they are always correct and make the code easier to revise later; we recommend them. Consider the following code fragment:

```
if (foo == bar)
        printf("Hey, foo equals bar!\n");
return bar;
```

Three weeks later, someone else edits this file and makes a change by adding another line after the if statement, properly indented:

```
if (foo == bar)
        printf("Bar equals foo, hey!\n");
        foo = 0;
return (foo + bar);
```

However, without curly braces, only the *first* statement is controlled by the condition, and the statement foo = 0 is executed regardless of whether foo is equal to bar. But when a human reads the second code fragment, it appears from the indentation that both statements are governed by the if statement. Had the original programmer used curly braces around the single statement, this unfortunate and difficult-to-find bug could have been avoided.

```
if (foo == bar) {
        printf("Hey, foo equals bar!\n");
```

1. American Standard Code for Information Interchange

```
}
return;
```

When using the do-while construct, place the opening brace on the line with the do and the closing brace at the beginning of a line followed by a space, the while condition and a semicolon. Always use braces with do-while.

```
do {
        ...
} while (foo != bar);
```

The switch statement is often difficult to format, because of the number of cases you might have for each code segment or the current level of indentation. There are a few tricks to making it less obtrusive: first, use half an indent for the case labels and another half indent for the code (a full indent between the switch and the code). If cases are related, place them on the same line to save space, but if they aren't, be sure to use a separate line for each case.

Always include a default case, unless you are switching on an enumerated type. In addition, if you utilize the "feature" of fall-through in switch statements (where if you don't explicitly break, the code in the following case is executed) be sure to comment this.

```
if (foo == bar) {
        switch (foo) {
            case 'a': case 'A':
                do_this();
                break;
            case 'b': case 'B':
                do_that();
                /* Fall through to next case */
            case 'c':
            case FOO_FINAL:
                do_it_all();
                break;
            default:
                complain_bitterly();
                exit(1);
        }
}
```

Finally, the `return` keyword is particularly confusing – remember that it is *not* a function, but a keyword with one operand. If you are returning a single value, no parentheses are necessary, but you should always enclose any computation in parentheses. Use return; at the end of a function with a void return value.

It is also good style to avoid returning from more than one point inside a function — innumerable bugs creep in when a function with more than one return statement is modified, and the programmer forgets to modify the code around every return statement. A better method is to declare a variable for the return value of the function and design a flow of control that obtains the correct return value and returns it in a single return statement at the end of the function. This method clearly shows where the function begins and ends, and prevents mishaps of omission in revision. The only exception to this rule is that it is occasionally appropriate to return prematurely on an error condition.