

Math 210

C Style Sheet

Introduction

Programming style is not important to the computer. The computer doesn't care that code is indented logically, that variables have intelligent names, that the design of the program is modular, or that the code is commented (accurately or not).

Programming style is important to programmers. Programmers by-and-large don't write programs. They spend most of their times maintaining, upgrading, and debugging existing code. Even when programs are called to create a new program, rarely do they work in a vacuum. Code from other programs routinely end up being incorporated into "new" programs.

Programming is hard to do well, even with good style. Without good style, it is quite difficult. However, there is no right style, and no wrong style (although the book comes close). There are ideas that are represented by every good style, but their implementation will be different. This is not **The One and Only Style Sheet for C**. It is the style sheet for this course.

Consistency is the most important element of style. The style inside a program should not change from one section to the next, or from one program to the next if the programs make up a system.

Comments

General information

Well-documented code is the second step to a good program (a good design faithfully represented is the first). The most common method of documenting a program is via comments. Comments give mundane information (programmer's name, version history, etc), as well as the reasons for decisions made during the design or implementation.

Bad attitudes

Many student programmers (and some "professional" programmers) consider comments a nuisance at best, an unnecessary requirement of simple-minded professors at worst. To them, working code is most important thing; comments are something to be done, haphazardly, once the program works. I have found that writing comments before code focus my thinking, allowing me to catch many mental mistakes before writing any code.

Maintaining comments

Comments must be maintained along with the code ... otherwise they are detrimental. It is very frustrating to discover an outdated (or plain wrong) comment has misled you for the past 2 hours.

Block comments

Block comments are long comments that give a lot of detail. Block comments stick out, so they should only be used when you really need to say something important. I personally reserve block comments for the program's overall comment at the top of the code, plus a block comment before each function.

Block comments can take multiple forms. I think the most attractive are:

```
/* **** */          /* **** */
/*          */      OR  *
/* comment */      * comment
/*          */      *
/* **** */          **** */
```

The first style looks nice, but is a royal pain to maintain. I prefer to use the second in my programs.

Program comment block

Each program should have a comment block at its beginning, giving the programmer's name, a short (10 word max) program name/description, date program created, assignment/project number, general description of the algorithm used, and a narrative giving the purpose of the program. Out in the "real world," you'll also find version history listing the changes made, the date the change was made, and the programmer's name that made changes, among other information.

Function comment blocks

A comment block should precede each function, giving the name of the function, an explanation of the parameters, an explanation of the return value (for non-void functions), and an explanation of the algorithm used (if appropriate). Other useful information: Does the function error trap the parameters? What happens if the function encounters an error? Does the function use a constant (#define value)?

Multi-line comments

Comments placed inside code to explain a group of statements should stand out from the general comments. The following illustrates one way of doing multi-line comments.

```
/*
 *
 * comment
 *
 */
```

A common beginners mistake is to use these comments next to code. This style of comment doesn't mix well within code. Consider the following:

```

exams      = exam1+exam2+exam3;          /*
projects   = proj1+proj2+proj3;         * Compute Final Average
average    =  0.75*(exams/3)            *
           + 0.25*(projects/3);        */

```

The second and third assignment statements are commented out. It isn't trivial to see this.

Single line comments

A single line comment comments a single line. It is best to place the comment to the right of the code. If it won't fit, it can be placed before the line (but not after ... I personally dislike comments after code).

Comments for variables

Single line comments should be used for all but the simplest variable declarations (like single letter counter variables). Don't repeat information already in the declaration (like x is an integer, y is a pointer to a character). Give information about the variable that isn't apparent from the declaration: What units are represented? Do special values represent something?

Examples

```

/*
 * The following code assumes that month has a correctly
 * spelled month. February isn't spelled Febuary!
 *
 */

/* Compute the total owed, including shipping and tax */
total_cost = sub_total + tax + shipping;

i = i + 1;          /* Next element in the array */

int  num_students, /* Number of students taking course */
     TotalCost;    /* Cost of all items, in pennies */
char name[35];     /* Name of user. If empty string, */
                  /* give read only access */
int  print_heading; /* 0 -> Don't print heading */
                  /* 1 -> print heading */

```

Variables

General information

Variables (other than counter and temporary pointer variables) must have meaningful names, but should not be obnoxiously long. Abbreviations can be used, but they should make sense (num for number, etc). If a variable is made up of words, either an underscore should be used to separate the words, or capitalize the first letter of each word. If you can't pronounce the variable name out loud, choose another name.

Single letter variables

Single letter integer variables named i, j, or k, can be used as counter variables in a for loop. Single letter pointer variables can be used to walk an array or linked list. Usually the letters p, q, and s are used (r is rarely used for some reason).

Declaring Variables

Each variable should be on its own line (single letter counter and pointer variables can be an exception). There should be a comment after each variable to give any information that the variable name doesn't (for example, units: inches, square feet, pennies, etc). When declaring a pointer, the asterisk should be next to the variable name.

Examples

```
int  num_students, /* Number of students taking course */
     TotalCost;    /* Cost of all items, in pennies */
char *name;        /* Name of user */
int  print_heading; /* 0 -> Don't print heading */
                       /* 1 -> print heading */
```

Constants

General Information

Constants created via the preprocessor directive #define should be in UPPER case, with an underscore character separating words. Numeric constants should be aligned so the digits are aligned (ones digits under ones digits, tens under tens, etc). Use MAX and MIN prefixes for maximum and minimum values.

Examples

```
#define  MAX_LINES      32
#define  PI              3.1415927
#define  BLOCK_SIZE    256
```

Magic numbers

The only numeric constants that should appear in your program are 0 or 1 (possibly 2). All other constants should be represented by a #define.

Expressions

General Information

Expressions should make sense when read out loud (they should "read well"). Good variable names are the key. Don't be afraid to change a variable name if the name doesn't sound right in an expression.

White Space

Binary operators should be surrounded by spaces, but sometimes fitting an expression on a line can supercede this rule. Unary operators should be next to their operand.

Long Lines

Avoid long expressions by using temporary variables to hold partial results. If a long expression is unavoidable, break the expression along operators, with the operator indented inside the assignment operator.

Examples

```
total_cost =  sub_total
              + sales_tax(sub_total)
              + shipping_charge(sub_total);

/* vs. */
tax = sales_tax(sub_total);
shipping = shipping_charge(sub_total);
total_cost = sub_total + tax + shipping;

record_ptr = &record;
if (value < 0)
    value = -value;
```

Indentation

General Information

Consistent indentation is easy on the eyes, and highlights what statements are part of a block of code. All code between an open curly and a close curly should be indented two, three, or four spaces. Once you choose a value, BE CONSISTANT!

if/while/for statements

The opening curly should be on the same line as the if/while/for statement, OR on the next line by itself, under the first letter of the statement. The close curly should line up under the first letter of if/while/for. The else clause should appear on its own line after the close curly of the “then” portion, with the open curly either on the same line, or the next line under the first e in else.

if/else if/else if sequences

Do not indent each subsequent if statement. Instead, place the next if statement on the same line as the else. Do not use long if/else if/else sequences if a switch statement can be substituted.

Examples

```
if (num_matches == 0)
{
    printf("No matches found!\n");
    exit(0);
}
```

```
}
/* or */
if (num_matches == 0) {
    printf("No Matches found!\n");
    exit(0);
}

if (score >= 1000)
    score += 25;
else if (score >= 2000)
    score += 200;
else if (score >= 3000)
    score += 800;
```

Functions

General Information

The placement for an open curly of a function should match the open curly placement for an if/while/for statement. Code in a function should be indented the same amount as code in if/while/for statements. The function's return type should be on the same line as the function's name. A void function's name should state what it does; a non-void function's name should indicate what it returns.

Prototypes

Prototypes for all functions should appear before the main function, but after any #include, #define, and structure definitions. I encourage you to include the parameter names with the prototypes, as it helps describe what the parameters to the function are.

Parameters

Parameter names need not match the names of the variables being passed to the function. It is sometimes better to have a different name to emphasize the *pass by value* nature of parameters in C. If possible, have the function's parameters on the line with the name of the function. If they won't fit, break the list between declarations, indenting the parameters inside the open parenthesis. If a function has no parameters, put void inside the parentheses. Do **not** leave the list blank.

Length

A function should accomplish one task. A function should fit on a page (or two). This includes the function main. If it doesn't, then the function probably does more than one task. Break it into separate functions accordingly.

Examples

```
float sales_tax(float sub_total) {
    return sub_total*TAX_RATE;
}

void print_heading(void)
{
    print("XYZ Corporation\n");
    print("Financial Report\n\n");
}

double withholding(double salary,
                   int num_dependents,
                   int lives_in_state) {

    double result;          /* Computed withholding */
    /* ... */
    return result;
}
```