Oregon State University
Intro to programming
Source Code Style Guide

Modified from:
OREGON INSTITUTE OF TECHNOLOGY
COMPUTER SYSTEMS ENGINEERING TECHNOLOGY
Modified by: Joseph Jess

Programming Style Guide v1.1

For CS 161, 162, 165, and possibly other classes, as requested

Examples in this are currently given in C, but other examples will become available as needed for courses or upon request.

Table of Contents

Programming Style Guide

The purpose of this document is to set forth guidelines for the formatting of programs. The goal is to create programs that are easier to read, understand, and maintain.

1 Documentation

Take credit for your work. Every program that you write should have a leading comment block that identifies the author and the purpose of the program.

1.1 Main Documentation Header Block

This header block will be the first part of your source file. The header block will consist of the following sections:

Author:	Your name
Date Created:	Starting date of the project
Last Modification Date:	Last time the project was changed
File name:	The name of the source code file (useful if printed)

Overview:

This section specifically states what the program does.

Input:

This section will state where the input for the project is coming from, what the input represents, and any limitations placed on the input. If the input is coming from a file, the filename, type of file (such as text or binary), and a sample format of the file will also be given.

Output:

This section states what your program produces as output, form of the output, examples, and where the output is going.

Example:

```
/************************************************************
* Author:                       Todd Breedlove
* Date Created:                 12/17/96
* Last Modification Date:       1/5/01
* Filename:                     Lab1.c
*
* Overview:
*     This program will read in a name and four grades and
*     calculate the average.  The average will then be used to
*     find the number of each letter grade (i.e. 5 A's, 6 B's,
*     etc.).  The program will also find the highest average and
*     the lowest average in the class.
*
* Input:
*     The input will consist of a name and four scores for a
*     varying number of people and is read from the file
*     c:\cst116\lab1.dat.  The data file is a text file in the
*     expected format of:
*     first_name last_name score1 score2 score3 score4
*
*     Example input file contents:
*     Todd Breedlove 90 90 90 90
*     Bill Jones 85 65 65 65
*
* Output:
*     The output of this program will be the cumulative
*     number of each letter grade.  The program will also
*     display the name of the person with the lowest and the
*     highest average for the class and their averages.  The
*     output will be to the screen and will have the form:
*
*     Number of A's: 10
*     Number of B's: 5
*     Number of C's: 10
*     Number of D's: 5
*     Number of F's: 10
*
*     Todd Breedlove has the highest average of: 90
*     Bill Jones has the lowest average of: 75
************************************************************/
```

2 Naming Conventions

Describe **what** a variable or function means or does, rather than **how** it works!!

Keep in mind that many languages are case sensitive (including C)!

2.1 Variables

Variable names must be descriptive of their purpose. All variable names must start in lowercase. Multiple words will be either camel case with upper character for the next word or all lower case separated by an underscore. Either way is acceptable but the naming convention must be consistent throughout the program.

All variable declarations should have a comment explaining the purpose of the variable. Some example variable declarations and comments are provided below.

Example:

```
int fileSize = 25;          // size of the input file
char selection;             // users selection of menu item
const int MAX_SIZE = 50;    // constant maximum size of file_name
char file_name[MAX_SIZE]; // the file name of the output file
int i = 0, j = 0;           // index variables for loops
```

Defining variables with names like x, y, or z, gives no insight into how the variable is to be used or the types of values that can be assigned, unless taken from a well-known mathematical equation, which you should reference explicitly.

Although it may take a little longer to type out a descriptive variable name, doing so can save hours of frustration when trying to correct a program that doesn't work or when trying to modify a program that you haven't worked on for awhile.

One common exception to the descriptive naming rule is that variables used solely for iteration purposes (such as an index in a for loop), may be named with single letters. Preferred letters for this purpose are i, j, and k. A comment should exist where these variables are declared that explains that they are for use as index variables. Such variables should never be used for any other purpose.

2.2 Constants

Constants should be given names in ALL_CAPITAL_LETTERS and USE_UNDERSCORES to separate words. Constants that apply only to a specific “translation unit” should be defined as a static constant within the file, to ensure that programmers wishing to reuse your class do not overlook them.

Example:

```
const int MAX_SIZE = 15;
static const int MAX_LENGTH = 63;
```

2.3 Functions, Methods, Subroutines, whatever your languages calls these...

2.3.1 Function Names

Function names must be descriptive of their purpose. Since functions are orders to do something, function names should start with strong verbs. Function names should start with uppercase letters and have uppercase letters at word boundaries. This distinguishes your functions from standard library functions and also prevents variable name and function name conflicts.

Example:

```
SelectionSort();
GetMenuSelection();
```

Functions should be written to do one purpose and should not be longer than one page, including appropriate documentation. Exceptions will be made to this rule if deemed necessary or appropriate. Check with instructor.

When printing source code, more than one function may be on any given page as long as there are no functions that are “broken” by a page break.

2.3.2 Function Declarations

Comments need to be clear, concise and should specifically describe **what** the function does, not how it works. There are two types of declaration comments:

1) If the declaration appears in the .c file, then you are required to use a one or two-line comment above the function declaration to explain the purpose of the function.

```
Example:
    // This function will read the input data for each plant
    void ReadInputData( int plant_array[], int array_size);
```

2) If the declaration appears in a separate .h file, then you are required to use a full comment block listing each function, its purpose, entry conditions, and exit conditions.

```
Examples:
    /*****************************************************************
    *  void ReadInputData(int plant_array[], int array_size);
    *
    *
    *     Purpose: This function reads the input data for each plant.
    *
    *     Entry: array_size is the declared size of the array plant_array.
    *
    *     Exit: For plant_number = 1 through array_size:
    *           plant_array[plant_number - 1] equals the total
    *           production for plant number plant_number.
    *
    *  int GetTotal(void);
    *
    *     Purpose: This function reads non-negative integers from the
    *           keyboard and accumulates the total production per plant.
    *
    *     Entry: none.
    *
    *     Exit: Accumulate the total units produced by each department
    *           of a plant and end input with a sentinel input number ( -1 ).
    *
    *  void clear_buffer(void);
    *
    *     Purpose: Cleaning the input buffer of left over data.
    *
    *     Entry: none.
    *
    *     Exit: (hopefully) leaves the standard input buffer empty.
    *****************************************************************/
    void ReadInputData( int plant_array[], int array_size);
    int GetTotal(void);
    void clear_buffer(void);
```

2.3.3 Function Definitions

Each function definition should have a comment block above the function that describes the entry and exit conditions that apply to the function. If no entry condition exists, then state “none.” The exit condition should explain **what** the function does, **what** it returns (if it is a non-void function), and which, if any parameters are modified.

Example:

```
/************************************************************
*   Entry:   array_size is the declared size of the array
*                  plant_array.
*
*    Exit:   For plant_number = 1 through array_size:
*                  plant_array[plant_number - 1] equals the total
*                  production for plant number plant_number.
*
* Purpose:   Read the input data for each plant
*
************************************************************/
void ReadInputData(int plant_array[], int array_size)
{
       int plant_number;

       for (plant_number = 0; plant_number < array_size; plant_number++)
       {
              printf("\nEnter production data for plant number %d\n", plant_number + 1);

              plant_array[plant_number] = GetTotal();
       }
}
```

3 Indentation

To improve the readability of programs, and to aid you in correcting your programs, there should be indentation or tab that indicates different blocks of statements. An example of a properly indented program is provided below.

```
void main()
{
	int num;

	printf("Please enter a number: ");
	scanf("%d", &num);

	if (num == 0)
		printf("Zero\n");
	else
	{
		printf("The number is greater than 0, or ");
		printf("the number is less than 0");
	}
	printf("\ndone");
}
```

The curly braces should be placed immediately beneath the line, lined up at the same indentation level. It is easy to see that every opening brace has a closing brace at the same level of indentation. This makes it easier to determine which block a brace is closing. Also, it is **MUCH** easier to verify that every closing brace has a matching opening brace. You may think you have started a block but may have inadvertently left off the opening brace at the end of a statement.

4 Parentheses and Precedence

The operator precedence order is a rather tricky thing. Rather than relying on rules of precedence, mathematical and relational expressions should make use of some parentheses to clarify how the expression is to be evaluated (to be clear to programmers what your intended order was).

Example:

```
x = (3 * x + y) / z  + (3 * a) / b;
```

Although the program will compile and run properly, you or someone required to maintain your program may forget the order of evaluation, or simply make a mistake, and cause the program to stop functioning properly. The following statement is a little less easy to understand:

Example:

```
x =  3 * x + y  / z + 3  *  a / b;
```

5 White Space

Add white space (i.e., blank lines and or spaces) to improve readability. In general:

- Use one blank line to separate logical chunks of code. Avoid using more than one blank line, but DO NOT double space lines of code!

- Put a blank line before and after each control structure.

- Place a space after each comma (,) to make programs more readable.
  (such as variable lists, argument lists, initializer lists, and comments)

- Place a space on either side of a binary operator. This makes the operator stand out and the program easier to read (but avoid spaces by unary operators such as !, *, &, and possibly others).

- Declare each variable on a separate line. This format allows for placing a descriptive comment next to each declaration. Unless the variables are homogeneous (same data type) and related to the same calculations, then you can place them on the same line.

Example:

```
double testAverage(90.0);                  // Average score for tests
int firstNumber(3), secondNumber(5);       // User enters two numbers
float quizAvg, labAvg;                     // Decimal point averages
```

6 Example Program

```
/*************************************************************
* Author:                       Naomi West
* Modified by:                  Joseph Jess
* Date Created:                 1/24/03
* Last Modification Date:  3/05/13
* Lab Number:              CS161 Lab 1
* Filename:                     Lab1.c
*
* Overview: This is a menu driven lab that prompts
*           and gets the user input of four plants
*            For each plant in the array, the program
*            prompts and gets the number of units from
*            each department per plant.
*
*    Input: There are four manufacturing plants. The
*            user input is a list of numbers entered by the
*           user giving the production for each department
*            in that plant. The list is terminated with a
*            negative number that serves as a sentinel
*            value.
*
*   Output: The number of the manufacturing plant, the
*            total number of units produced within each
*            plant. The output to the screen will be in the
*            form of:
*
*            Enter the production data for plant number 1
*            Enter the number of units produced by each department.
*            Append a negative number to the end of the list.
*            1 2 3 -1
*            Total = 6
*************************************************************/

#include <stdio.h>

static const int NUMBER_OF_PLANTS = 4;

// This function reads the input data for each plant.
void ReadInputData(int plant_array[], int array_size);

// This function reads non-negative integers from the
// keyboard and accumulates the total sum per plant.
int GetTotal(void);

//clears the input buffer as best as I know how
void clear_buffer(void);

int main()
{
      // Array of total production for each of the four plants
      int production[NUMBER_OF_PLANTS];
      char run_again;     // user input answer
```

```
	do	// Menu to input another plant or exit
	{
		int i; // loop driver

		ReadInputData(production, NUMBER_OF_PLANTS);  // Input plant data

		printf("\nTotal production for each of 4 plant\n"); // Display production
information

		// Loop through all the plants
		for( i = 0; i < NUMBER_OF_PLANTS; i++ )
		{
			printf("Plant production for plant %d: %d\n", (i + 1), production[i]);
		}

		printf("Test Again? ( Type y or n and return key ): ");    // Continue or exit
		scanf("%c", &run_again);

		clear_buffer();

	} while( ( run_again != 'N' ) && ( run_again != 'n') );

	printf("\n\nNow exiting.");

	return 0;
}

/*************************************************************
*   Entry:   array_size is the declared size of the array
*                  plant_array.
*
*    Exit:   For plant_number = 1 through array_size:
*                  plant_array[plant_number - 1] equals the total
*                  production for plant number plant_number.
*
* Purpose:   Read the input data for each plant
*
*************************************************************/
void ReadInputData(int plant_array[], int array_size)
{
	int plant_number;

	for (plant_number = 0; plant_number < array_size; plant_number++)
	{
		printf("\nEnter production data for plant number %d\n", plant_number + 1);

		plant_array[plant_number] = GetTotal();
	}
}

/*************************************************************
*   Entry:   none.
*
*    Exit:   Accumulate the total units produced by
*                  each department of a plant and end input
*                  with a sentinal number (-1).
*
* Purpose:   Reads non-negative integers from the keyboard
*                  and places their total in sum per plant.
```

```
*************************************************************/
int GetTotal()
{
	int total = 0;		// total units per plant
	int next;	// unit of a plant department (added to total each iteration)
	int count = 1;		//number of departments

	// prompt for and get user input
	printf("Enter the number of units per dept.\nEnter a negative number to the end the
list.\n");

	printf("\n\tenter the units produced by department #%d: ", count);
	scanf("%d", &next);

	clear_buffer();

	// accumulate units per plant
	while(next >= 0)
	{
		total += next;
		++count;

		printf("\n enter the units produced by department #%d: ", count);
		scanf("%d", &next);

		clear_buffer();
	}

	// output total to display
	printf("Total = %d\n", total);

	return total;
}


/*************************************************************
*   Entry:  none.
*
*    Exit: (hopefully) leaves the standard input buffer empty.
*
* Purpose: Cleaning the input buffer of left over data.
*************************************************************/
void clear_buffer()
{
	char ch; //for clearing input buffer

	//clear buffer for any later input
	while((ch = getchar()) != '\n' && ch != EOF)
	{}
}
```