CS 210 - Fundamentals of Programming I
C Programming Style Guideline

Most of a programmer's efforts are aimed at the development of correct and efficient programs. But the readability of programs is also important. There are many "standards" in use. Since following them will not guarantee good code, these "standards" are actually guidelines of style. Each one has its proponents and detractors. The textbook advocates one particular style for C programs. This document collects the style comments in the textbook into one place and adds some allowable alternatives as well as giving the documentation requirements for this course. The essential point is that a program is a medium of communication between humans; a clear, consistent style will make it that much easier for you to communicate.

Program Documentation

Code File Headings

Every code file should begin with a comment block of the following form and with the information filled in as appropriate:

```
/* File: <name of file, e.g., main.c>
 * <Description of what is in this file.>
 *
 * ----------------------------------------------------------------
 * Class: CS 210                    Instructor: Dr. Deborah Hwang
 * Assignment:                      Date assigned:
 * Programmer:                      Date completed:
 */
```

The description should be short, but meaningful, at a minimum the following should given:

Preprocessor Section

In the preprocessor section, include statements for header files should have comments indicating the types, constants, variables, or functions used from the header if they are not commonly used. For example,

```
#include <math.h>       /* sin, cos */
#include "vector.h"     /* vector struct and operations */
```

Constant Macros

All define macros for constant definitions should appear together after the include statements.

Function Prototypes

Include a short description for each function (not including **`main`**) in a comment block before each function prototype. Place each function parameter on a separate line and annotate with its analysis (i.e. movement) and a short description. For example:

```
/* Function: sample
   Returns: <description of what is computed> OR
   <description of what function does if no returned value>
*/
type1 sample (type2 arg1,  // RECEIVED: description of obj1
              type3 *arg2, // PASSED BACK: description of obj2
             ...);
```

"RECEIVED" may be abbreviated "REC'D" and "PASSED BACK" may be abbreviated "P'BACK".

Alternatively, an analysis table can be used. For example:

```
/* Function: sample
   < A short description of what function does >
   Objects                Type     Movement        Name
   ------------------------------------------------------
   Description of obj1    type2    received        arg1
   Description of obj2    type3    passed back     arg2
   Description of result  type1    returned        result
*/
type1 sample (type2 arg1, type3 *arg2, ...);
```

Identifiers

Identifiers should be chosen to be self-documenting. Abbreviations, except where standard, should be avoided. In addition, comment variables with descriptions from the problem statement and when usage is restricted. (E.g., an integer used to represent a calendar month, so its valid values are 1-12.)

Two styles of identifiers are acceptable for this course. However, **only one style must be used** consistently within **all files** comprising a project. Using both styles in one project will incur grading penalties. The styles are as follows:

- The style used by the textbook and most current professional programmers is the original style advocated by the inventors of C. All identifiers except constant identifiers are in all lowercase with underscore between each word. E.g., **`find_min_index`**, **`point`**, or **`first_name`**. Constant identifiers should be written in all uppercase with words separated by an underscore (_). E.g., **`MAX_STRING_SIZE`**.

- Another style (often used by the instructor) is a style that was in vogue in the 80's and 90's and often is called "camel case". Each word in a function identifier or structure type identifier should start with an uppercase letter. E.g., **`FindMinIndex`** or **`Point`**. Each word except the first one in an variable identifier should start with an uppercase letter. E.g., **`firstName`**. No underscores should be used in function, structure, or variable identifiers. Constant identifiers should be written in all uppercase with words separated by an underscore (_). E.g., **`MAX_STRING_SIZE`**.

Commenting

Local variables in functions should be commented when usage is not apparent. That is, when the identifier does not completely describe the purpose of the variable.

Comment the ending curly brace with **`// end <description>`** whenever the body is more than 10 lines long. This applies to function bodies, selection bodies, and loop bodies. For example:

```
/* Function: try
   Attempt to do lots of things...
 */
void try (...) {
   ...
   /* Do lots of things */
   ...
} // end try function
```

A sufficient number of design steps must be given in comments so that the reader can determine WHAT is being done or WHY it is being done without reading the actual code (which describes HOW something is being done). For very short or very simple functions, the function description will suffice. The code that implements a step should immediately follow the comment for the step. In addition, comment complex code to improve clarity. Note that a comment should not be the meaning of the code written in English. For example,

```
   /* 3. Adjust i to point to the end of the previous word */
   i = i - 1;
```

says why the computation is being made whereas the follow does not add any information

```
   /* 3. Subtract 1 from i */
   i = i - 1;
```

Comments should be in good English. Grammar and spelling should be correct.

Abbreviations in comments should rarely be used, and then only those that would be found in a standard dictionary.

Library Files

Library function declarations (i.e. prototypes) and definitions (i.e., implementations) should be divided into multiple files so that we can reuse them easily. By convention, the function prototypes are stored in a *header* file which has the extension "**.h**". The functions definitions are stored in a *source* file with the extension "**.c**". The main program usually is stored in a separate source file that includes the header files for each library used.

Every library header file should use compiler directives **#ifndef**, **#define**, and **#endif** to ensure a header file is included only once. The symbol should be the name of the header file in all capital letters with an underscore (_) replacing the dot. For example, for the header file **vector.h**, the directives would be:

```
#ifndef VECTOR_H_
#define VECTOR_H_

/* Function: vector_add
   Add elements of two vectors together pairwise
*/
void vector_add (struct vector v1,       // REC'D: left operand
                 struct vector v2,       // REC'D: right operand
                 struct vector *result); // P'BACK: result
```

```
/* Function: vector_multiply
   Multiply elements of two vectors together pairwise
*/
void vector_multiply (struct vector v1,       // REC'D: left operand
                      struct vector v2,       // REC'D: right operand
                      struct vector *result); // P'BACK: result
...
#endif  /* VECTOR_H_ */
```

Constants and Variables

Constants

Constant values in your algorithm should be replaced by constant identifiers in your program. Exceptions should be made only when the constant conveys its own meaning, such as 0 as an initial value for a sum or to start a count, or is part of a constant mathematical formula, such as 2 in $2\pi r$.

All constants should be defined after any include statements and before any function prototypes and the main program function, even if they are used only in one function.

Variable use

Each variable identifier that is used in a function should be **local** to that function - that is, declared in the function's header or in the function's body.

1. If the variable may have its value changed in the body of the function and that new value will be needed back in the calling program (i.e., it is passed back), then the variable should be declared as a reference formal parameter. (Sometimes data is both received and passed back.)

2. If the variable gets its initial value from the calling program but does not send a different value back (i.e., it is only received), then the variable should be declared as a value formal parameter, except for C arrays, which are automatically passed as reference parameters. Received-only arrays should be declared as **`const`**. For example,

   ```
   /* Function: function1
      Returns: <description>
   */
   int function1 (const double scores[]) // REC'D: score array
   {
      ...
   } /* end function1 */
   ```

3. If the variable does not get its initial value from the calling program and does not pass its value back (via a parameter), then the variable should be declared as a local variable of the function. This generally includes the returned object, if any. When possible, variables should be initialized when declared.

NEVER use a global variable in this course unless explicitly told otherwise. While there are valid reasons for having global variables, they generally should be avoided and **will not** be tolerated in this course.

Program Formatting

Any preprocessor statements (**#include**, **#define**, etc.) should be at the beginning of a file (after the program file heading comment). Any typedef type declarations should follow. Finally, function prototypes should appear just before the main function. No other statements should appear outside a function body unless explicitly allowed. The main function is to be the first function definition.

There should be a blank line between each function definition. A blank line should be used to separate logical sections within a program or function. In general, blank lines should be used wherever their use will improve readability.

Indenting should be used to convey structure. Indentation should be at least 3 spaces, and generally no more that 6 spaces unless otherwise noted. Indenting should be consistent. Use an IDE indenter prior to submitting assignments, whenever possible.

For variable declarations, generally each identifier should be declared on a separate line with appropriate comments following. The type identifier should be indented, and the variables of that type should line up under each other. Use commas to separate variable identifiers of the same type.

For example, in the main function, we might declare

```
#define MAX_SIZE 100
#define PI 3.14159

int main ()
{
   int i, j,                 // Loop indexes
       score,                // Input: a test score
       tests[MAX_SIZE];      // Array of test scores
   double average;           // Output: average score
   struct point p1;          // Point to test
   ...
}  // end main
```

Comments should explain the purpose of each variable where appropriate, and should line up with the code being described.

A statement should not be longer than a screen's width (about 80 characters). If a non-function call statement must be continued on more than one line, the second line should be indented to convey the continuation and successive continuation lines should be aligned with the second line of the statement. For example, we might write:

```
   /* Find the next non-alphabetic character */
   while ((('a' <= line[i]) && (line[i] <= 'z'))
         || (('A' <= line[i]) && (line[i] <= 'Z')))
      i++;
```

A long function call statement should be broken up so that the function arguments lined up. For example, we might write

```
sample (argument1, argument2, argument3,
        argument4, argument5, argument6);
```

For the statement that follows **if**, **else**, **while**, **for**, **do**, and **switch**: the statement should start on the next line and be indented. For example,

```
/* first and last indexes have crossed, so everything matched */
if (first => last)
   found = true;
```

Each line of the body of a compound statement should be indented within the curly braces surrounding it. Two styles of curly brace alignment are acceptable for this course. However, **only one style must be used** consistently within **all files** comprising a project with one exception given below. Using both styles in one project will incur grading penalties. The styles are as follows:

- The style used by most current professional programmers is the original style advocated by the inventors of C. The opening curly brace appears at the end of the line introducing the compound statement, and the closing curly brace is lined up under the first line. For example,

```
/* test for the target item */
if (a[middle] == item) {
   item     = a[middle];
   found    = true;
   position = middle;
}
```

- Another style (often used by the instructor and textbooks) is a style that was in vogue in the 80's and 90's that places the opening curly brace under the line introducing the compound statement. For example,

```
/* test for the target item */
if (a[middle] == item)
{
   item     = a[middle];
   found    = true;
   position = middle;
}
```

The only exception to consistent use of one of these styles is that function definitions may use the latter style regardless of the style used for executable statement bodies as is done in the textbook. However, **all** function definitions must use the same style, otherwise grading penalties will be incurred.

Column alignment should be observed for each set of reserved words **if** and **else**. This include multi-branch constructs, for example:

```
/* Assign a grade using a straight scale */
if (x > 90)
   grade = 'A';
else if (x > 80)
   grade = 'B';
else if (x > 70)
   grade = 'C';
else if (x > 60)
   grade = 'D';
else
   grade = 'F';
```

Comments that describe one or more statements should be immediately above and aligned with the statement or collection of statements which they describe. For example,

```
/* Consider items starting at index i */
j = i;
/* Swap elements until finding the right place or end of list */
while ((j > 1) && (a[j - 1] > a[j])) {
   /* a[1..j-1] is unsorted and a[j..i] sorted: */
   swap_ints (a[j], a[j - 1]);
   j = j - 1
}
```

Function headers should start at the left edge. The main curly braces for functions should line up with the corresponding heading. For example,

```
void find_minimum_index (...) {
   ...
}  // end find_minimum_index
```

At least one space should be used in the following locations within C code (this does not apply within comments and character strings):

1. before and after **=**, **/***, any relational, logical, arithmetic, or assignment operator
2. before **(** when not preceded by another **(**
3. after a comma in argument lists, and after semicolon in for-loop headers

A function (including the main program function) should fit on one screen (about 25-30 lines) if possible and must fit on a listing page (about 50 lines). Ideally, the analysis and design will not produce code that is more than a page long, but if the code does not fit initially, introduce one or more new functions to split up the work in logical places.