Program Style

Computer programs are meant for two audiences: the *computer* that compiles and runs it, and the *people* who must read, modify, or evaluate the program. In this course, the latter includes not only the instructor, teaching assistants, and graders, but also you a week after you wrote the code.

Most programming shops have "in-house standards" for programming style, because a standard style tends to make programs more readable. ECS 36A is no exception; its rules are listed in this section. Part of your homework grade reflects how well you meet these standards.

Underlying all style guides is one precept: *use common sense*. Remember that this style guide is only a guide. You want to be sure that others can read and understand your program. If you think extra comments or a particular program organization will make the program clearer, go for it. But be careful — what seems logical or clear to you may not be logical or clear to others. So if you want to deviate too far from these guidelines, please talk to us.

1 Write Self-Documenting Code

"Self-documenting code" means that the program itself gives information about what it is doing. Some ways to do this are:

- The structure of the program should reflect the way you would solve the problem naturally.
- Choose meaningful names for variables, functions, function arguments and parameters, and constants. For example, a variable that stores an amount of money should be named `amount` or `amountMoney`, not `x`.
- Use macros or named constants to represent constants. Avoid sprinkling numbers throughout your code. This makes changing the number easier—you change it once, where it is assigned a name or in the macro, rather than having to find each time it occurs and changing each occurrence. A good rule of thumb is that, if a number appears more than once in a program and means something special (like the size of an array or the maximum size of an input number), name it and use the name, not the number.
- Avoid repeating code whenever possible. Use a function instead.

2 Capitalization of Names and Keywords

C is sensitive to capitalization; the variable names `amountMoney` and `amountmoney` refer to two different variables. Keywords are *never* capitalized; it's always **`while`**, never **`WHILE`** or **`WhIlE`**.

In general, variable names are in lowercase, the sole exception being if the name consists of more than one word. In that case, you can either keep everything in lowercase, separate the words with an underscore '_' or capitalize the first letter of each word (this last is sometimes called "camel case"). So, any of the variable names `amountmoney`, `amount_money`, `amountMoney`, or `amount_Money` is fine. Pick the convention that seems best to you. But be consistent—use the same one throughout your program.

The names of macros, defined types, and named constants should be entirely UPPERCASE. So the ratio of the circumference of a circle to its diameter would be written as **`PI`**, not **`Pi`** or **`pi`**.

3 Code Indentation and Formatting

A block of code refers to the statements between the open brace '{' and the close brace '}'. After the opening brace, indent the contents of the block by a consistent amount (usually 4–8 spaces or one horizontal tab). If there is a block within a block, indent the inner block twice as much as the outer block. As an example:

```
int main(int argc, char *argv[])
{
    int inpnumber;
    int sum = 0;
    int numread = 0;

    while (scanf("%d", &inpnumber) == 1){
        printf("I read %d\n", inpnumber);
        sum += inpnumber;
        numread++;
    }
    printf("The sum of the %d numbers I read is %d\n", numread, sum);
    return(0);
```

```
}
```

Also, note how the braces are placed. The first brace is always on a line before any statements in the block, but in the above it may be on the same line as the **`while`** statement or on the line immediately after. In the latter case, line it up with the beginning of the **`while`**:

```
while (scanf("%d", &inpnumber) == 1)
{
```

Again, be consistent. But you should always put the opening top-level braces (such as the one under the line with *`main`*) on a separate line.

Line up the closing brace with the matching opening brace. The closing brace always goes on its own line.

If a statement is split across multiple lines, indent the second and following lines an equal amount from the first line:

```
printf("this is a big print %d %s %d\n",
                  first_integer, first_string,
                  second_integer);
```

4 Exit Codes

When a C program terminates, it returns a value to its caller (usually but not always the shell). The code indicates how the program terminated. If the exit code is 0, the program ran without error. If the exit code is not 0, then the program is saying something happened—usually an error.

If you need to terminate your program within a function other than *`main`*, use the function *exit*(3). If you need to do this in *`main`*, you can use either *exit* or **`return`**.

You can express an exit code in one of two ways.

- To indicate success, you can use either the integer **`0`** or the macro **`EXIT_SUCCESS`** (and if you use the latter, don't forget to include the header file **`stdlib.h`**).
- To indicate a generic failure (that is, the program failed but you don't want to indicate anything else), you can use either the integer **`1`** or the macro **`EXIT_FAILURE`** (again, if you use the latter, don't forget to include the header file **`stdlib.h`**). If you want to indicate which of several problems caused the failure, use an integer error code.

5 Comments

Comments should be used to make programs absolutely clear. Use them liberally but not so much that the meaning gets submerged. Begin a comment with `/*` and end it with `*/`.

You should *comment your code as you write it*. It is actually much faster than going back later, and comments are very useful when debugging. (If what the comment says the code does is different than what it really does, you found a bug.) We reserve the right to refuse to look at uncommented programs.

ECS 36A "house standards" call for five kinds of comments:

- Start-of-file comments
- Start-of-function comments
- Paragraph comments
- Variable declaration comments
- Line comments

We will describe the requirements for each kind in turn.

5.1 Start-of-file comments

```
/*
 * CHANGE -- figure out how many quarters, dimes, nickels, pennies
 * are in some amount of cash
 *
 * Usage: change
 *
 * Inputs: User enters an integer for change
 * Outputs: number of quarters, dimes, nickels, pennies in the change
 * Exit Code: EXIT_SUCCESS (0) if valid integer entered
```

```
 *            EXIT_FAILURE (1) otherwise
 *
 * written for ECS 36A, Fall 2019
 *
 * Matt Bishop, Sep. 1, 2019
 *  original program written
 */
```

The comments at the beginning of a file, collectively called its "header," must contain your name in addition to other general information about the program and what it does. The date when the program was written and a history of major modifications are required.

5.2 Start of function comments

Each function must have its own header. Here's an example for a function called *take_off_queue*:

```
/*
 * take an element off the front of an existing queue
 *
 * parameters:   QTICKET qno  ticket for the queue involved (in)
 *               int *value   value removed from the queue (out)
 *                            (assumes it's a valid int pointer
 *                            but checked for NULL)
 * returns: int  0            success!
 *               QE_BADPARAM  bogus parameter because:
 *                            * parameter refers to deleted, invalid,
 *                              or unallocated queue
 *                            * pointer points to NULL address for
 *                              returned element
 *               QE_INTINCON  queue is internally inconsistent
 *               QE_EMPTY     no elements so none can be retrieved
 * exceptions:   none
 */
int take_off_queue(QTICKET qno)
{ ... }
```

The reader should be able to determine what a function does and how to use it by reading only this header information. Things like global variables referenced or modified, assumptions about the input, or anything else that the reader should know before "lifting" a function and using it in another program should go here. So should the algorithm used, if it's not absolutely clear.

You also need to list the parameters to the function, their type, and what they mean. It's also helpful to say whether they contain information passed into the function ("in"), returned from the function ("out"), or both ("in/out").

Then list the possible return values (if any) and say what they mean. In the above, the return value may be one of 4 values, three of which are macros defined elsewhere. The header states what each return value means. If the function does not return a value, say "none" or "void" here.

Also note any unusual or exceptional conditions—for example, if the function handles messages from other programs (these are called "interrupts" or "traps". For this class, you simply need to include this and say "none", because we don't think any programs or functions will have to handle these.

You should also state any input and output assertions. Input assertions is a fancy way of saying "I assume this about the input." For example, for a square root routine *sqrt(x)*, the usual input assertion is $x > 0$. In the above, look at the comment about "value": "(assumes it's a valid int pointer but checked for NULL)". That's an assertion, because it says the function doesn't check that "value" points to an integer. Output assertions are similar—they describe what is true after the function has been called. It is a good habit to get into writing these assertions. It helps you specify exactly what the routine does, and this is precisely the information that someone needs to know in order to lift your routine and use it elsewhere.

If you borrow code from another source, you must cite the author, where you got it from—a book or web site—and any other information that will help another reader can find the reference.

5.3 Paragraph comments

When you write pseudo-code or describe how a function accomplishes its task, you usually break it down into a series of steps. For example, to insert an item into an array, you find where it belongs, move everything after it down one place, and finally copy the item into the correct position. These steps sometimes become functions with descriptive names, but often they become several lines of code. In this case, it helps to write a comment that explains the purpose of the next section of code. For example, here is a segment of code that checks an error condition:

```
/*
 * check that qno refers to an existing queue;
 * readref sets the error code
 */
if ((cur = readref(qno)) != GOOD)
    return(cur);
```

Paragraph comments are generally not necessary in short functions.

5.4 Variable declaration comments

These comments typically go at the end of a line. They explain what the variable is. Here's an example, from the *take_off_queue* function:

```
int cur;      /* index of current queue */
QUEUE *q;     /* pointer to current queue structure */
```

You can also do this:

```
int cur;      // index of current queue
QUEUE *q;     // pointer to current queue structure
```

5.5 Line comments

These comments usually go at the end of a line. They explain what the line of code does, and are used when the meaning of the line is not clear. Do *not* say what the line does; say why it is there. As an example:

```
q->queue[(q->head+q->count)%MAXELT] = n; /* append element to end of queue */
q->count++;                              /* one more element in the queue */
```

or

```
q->queue[(q->head+q->count)%MAXELT] = n; // append element to end of queue
q->count++;                              // one more element in the queue
```

I prefer to put these comments above the lines and indented, because it is easier (for me) to read:

```
    /* append element to end of queue */
q->queue[(q->head+q->count)%MAXELT] = n;
    /* one more element in the queue */
q->count++;
```

Both styles are widely used, so you should use whichever you like. But do *not* write this type of line comment:

```
q->count++;  /* add 1 to q->count */
```

This just translates the C into English. A useful comment would tell the reader why you are adding 1 to `q->count`.

6 Credit

These guidelines are adapted from Sean Davis' guidelines for ECS 30, and the style guide of Texas A&M University at `http://faculty.cs.tamu.edu/welch/teaching/cstyle.html`.