Getting Better Grades On Computer Science II Projects

Arthur G. Werschulz
Department of Computer and Information Sciences
Fordham University
agw@dsm.fordham.edu

September 8, 2017

As with most courses at Fordham University, CISC 2000 (Computer Science II) has a required lecture component. As important as these lectures are, you will probably find yourself focusing on the programming projects that will be assigned throughout the semester. These assignments will go a long way towards helping you to internalize the ideas that the text and the lectures are trying to transmit. As a result, a large part of your grade on the courses CISC 2000 and its companion course CISC 2010 (Computer Science II Laboratory)[1] will be affected by how well you do on the programming assignments.

Your projects will be graded according to the following standards:

- documentation: 20%
- correctness
  - correctness of the algorithm: 20%
  - correctness of the program: 20%
- input and output quality: 20%
- program style: 20%

Some of these may seem self-explanatory; some may not. After reading this handout, I hope that you'll have a better idea of what each of these categories mean, which should help you to get better grades on these assignments.

1 Documentation

Many programmers consider documentation to be a pain. However, a program needs to be understandable to all those who might read it. This includes both other people (the poor souls who have to maintain your code after you leave or are assigned to other projects) and you (when you're asked to modify the code after being away from it for six months). So we insist that you adhere to a consistent documentation style.

1. Your solution to a particular assignment will consist of one or more computer files. Each file should start with a block comment, which should look something like Figure 1. (See pp. 157–158 of the text for additional discussion.)

   In more detail:

   - The first line should give the project number and its title.

[1] You will receive the same grade for both courses. This will probably work to your benefit, since most people get better grades on their programming assignments than they do on their exams.

```
/*************************************************************************
 *
 * Programming Project #42: The Traveling Salesperson Problem
 *
 * This program solves the Traveling Salesperson Problem in polynomial
 * time.
 *
 * Lorem ipsum dolor sit amet, consectetur adipiscing elit.  Morbi
 * ipsum nibh, tempor eu ultrices ut, mattis a tortor.  Ut risus sem,
 * molestie at molestie ac, varius eu dolor.  Donec feugiat elit vel
 * lacus ultrices aliquet.
 *
 * Author: Harry Q. Bovik <harry@bovik.com>
 * Date:   30 February 9872
 *
 *************************************************************************/
```

Figure 1: Typical block header

- This should be followed by a description of what the program does. For clarity's sake, use a one-line description, followed by a more detailed description (if necessary).
- At the end, give your name, email address, and the date that the program was completed.

If you're using the `emacs` editor,[2] you can create a row of 70 asterisks by typing the keystroke[3] C-u 70 * (see the `emacs` reference card or the `emacs` tutorial for details, as well some other `emacs`-related material that you can find at `http://www.dsm.fordham.edu/ agw/emacs.html`. Also note that you should put a blank between the asterisk that starts a line and the content of the line, which improves readability.

2. Every function should be preceded by a descriptive comment. (This includes the member functions of a `class`.) For the `main()` function, you can simply say

```
// the usual main() function
int main()
{
    .
    .
    .
}
```

The documentation of the other functions should make the following information clear to the reader:

- what the function does
- the purpose of its parameters, and
- the function's return value, if any.

Sometimes it will be a good idea to be a bit more pedantic, giving the following information:

- the function's preconditions and postconditions, and

[2] If you're not using `emacs`, I have a question for you: why not? Although some editors (e.g., `pico` or `nano` might be easier to learn, `emacs` has a a lot of goodies that make it a good programmer's editor. Ask me, and I'll demonstrate.

[3] C-something means "control-something", and M-something means "meta-something".

```
// read chars from cin and compose a Token
void Token_stream::get() { ... }

// put a Token back into the buffer of a Token_stream
void Token_stream::putback(Token t) { ... }

// iteratively compute the n-th Fibonacci number
Fib_type fib(int n) { ... }

// is num a prime number?
bool is_prime(int num) { ... }

// precondition: n >= 0
// return value: base raised to the power n
// throws Illegal_exponent if n < 0
int pow(int base, int n) { ... }

// precondition: data is sorted (in increasing order)
// return value:
//   if (data[i] == target) for some i in the range [0..data.size()),
//       return value is i
//   otherwise (i.e., target not contained in data)
//       return value is -1
int binary_search(const vector<double>& data, double target) { ... }
```

Figure 2: Examples of function documentation. (The function bodies are compressed to save space.)

- exceptions that the function throws.

See Figure 2 for some examples of function documentation.

Sometimes functions are defined in one place, but declared in another place. Why?

- If your program is contained in one source file, you might want to put all the declarations at the beginning of the file, before the `main()` function. This allows somebody who's reading the program to see `main()` as soon as possible, which (in turn) allows her to see the high-level logic of the program as soon as possible.
- If your program is contained in several files, you will probably have header files (such as "`foo.h`") and implementation files (such as "`foo.cc`").

You should not put the documentation in both places, since it's pretty hard to keep them in synch with each other. If your instructor doesn't tell you where the documentation should go, you should choose one or the other, but *be consistent* (i.e., don't document one function at its point of declaration, but another at its point of definition).

3. You should document every important variable, unless you're 100% sure that its name unambiguously gives this information to the reader. For example, you might be surprised to learn that

   ```
   int length;
   ```

   may not be sufficiently self-documenting; you might need to do something like

   ```
   int length;                                // length of a furrow, in pixels
   ```

   instead. I would *not* recommend the verbose name

```
int furrow_length_in_pixels;
```

simply to avoid placing a comment. But if you're using several different lengths, you might want to use

```
int furrow_length;                          // in pixels
```

and if you're using different units for the same length, you might consider

```
// furrow lengths
int length_pixels;
int length_microns;
```

Consistency and common sense should guide you here.

4. Don't document unimportant variables. The main problem here is to figure out which variables are unimportant. For example, loop control variables are often (but not always!) unimportant, so it's fairly likely that you can write

```
for (int i = 0; i < blivit.size(); i++) {
    .
    .
    .
}
```

5. You should use loop invariants to document loops.

6. Do not include useless documentation. In particular, I don't want to see code translated into English, such as

```
// add a to b, giving c
c = a + b;
```

7. Do not include incorrect documentation, such as

```
// add a to b, giving c
a = b + c;
```

8. Finally, note that documentation is a form of communication. One thing that it communicates is your level of professionalism. Thus, you will lose points for misspellings and for ungrammatcial usage.

2 Correctness

Your program needs to be correct. At the bare minimum, it should produce the correct output for any sample input data sets that I give you. Failure to check your program against any and all such data sets will cause you to lose points.

The odds are pretty good that this is the extent to which I'll check your programs; however, I reserve the right to run your programs against other data sets.

I will assess the correctness of both the algorithm and the program, each having an equal weight. Roughly speaking, correctness of the algorithm will mean that the general idea underlying your program is correct, whereas correctness of the program will mean that all the details are correct.

If you find that you can only implement (say) 80% of the program features, don't despair. I am quite happy to give partial credit. Please don't fall into the trap of spending hours of time working on that final 20%, the end result being that you fall behind in your work on the other assignments.

A warning: Correctness only counts for 40% of your grade. However, I reserve the right to give a zero to a program that does not (at least partially) solve the problem described in the project handout. In other words, you can't turn in a "Hello, world!" program and expect to get a 60%. By the way, I will often give you a preliminary version of a program (perhaps a "stub version") that you are suppose to extend in some way or another; if you simply submit this original version, even with beautiful documentation and style, you can expect to get a zero.

Having said all this, let me give you one final word here: relax.[4] Most people tend to get a near-perfect score on the correctness portion of their programming assignments.

3 Input and output quality

It should be easy for a user to submit input to your program; it should be easy for her to read the results that your program produces.

I will often provide you with an input and/or output format, as follows:

- This may appear in the project handout, under the rubric of "here is a sample run of the program".
- I might also make a executable version of the program available to you (say, in the project's `share` directory), in which case you are *highly* encouraged to run my version of the program, to see the I/O behavior.

(I might give you both.) Should this be the case, one of two situations will occur:

1. Your program will exactly follow my input/output format. In that case, you'll get a perfect score under this section.
2. Your program's input/output format will differ from mine. Should this be the case, you'll need to convince me that your format is at least as good as mine if you want a perfect score under I/O. You probably won't be too surprised to learn that I'm not easily convinced.

So you can either bet on a sure thing or take your chances. It's your call.

4 Program style

In 1918, William Strunk and E. B. White wrote *The Elements of Style*, which is an invaluable guide to clear writing.[5] In 1974,[6] Brian Kernighan wrote *The Elements of Programming Style*, which may be thought of as a "Strunk and White" for programmers. I won't attempt to give an exhaustive list of rules for programming style in this short document, but here are some points that you should keep in mind (your text will have additional rules, as well). The whole point of these rules is that your programs should be easy to read, which will make them easier to understand and debug, which (in turn) will improve the odds of their being both correct and maintainable. You might want to think of this section as being an extension to Section 5.9 of the text.

1. Break your code into small functions, each expressing a logical action.
2. Avoid complicated code sequences.
3. Use library facilities rather than your own code when you can.
4. Use a consistent indentation style. This can be done automatically if you're using the `emacs` editor, as follows.

   One way to for an `emacs` user to ensure that her code is always properly indented would be for her to get into the habit of always auto-indenting each program file before you save it. You do this by issuing the commands C-x h (which marks the entire buffer as being the `emacs` region) and M-C-\ (which indents the region).

[4] If you want two final words, fine: "Don't panic."

[5] If you weren't been forced to read same when you took ENGL 1100 (or its equivalent), drop everything right now, get a copy, and study it intently. It will make you a better writer, and it might just even make you a better programmer.

[6] P. J. Kernighan was a co-author of the 1978 edition.

```
;;; automatically indent buffer upon save (but ask first)
;;; steals ideas from
;;; Andreas Politz <politza@fh-trier.de>
;;; Eric James Michael Ritz <Eric@cybersprocket.com>

(defun indent-buffer-ask()
   (when (y-or-n-p "Indent buffer before saving? ")
     (indent-region (point-min) (point-max))))

(defun indent-buffer-no-ask()
  (indent-region (point-min) (point-max)))

(setq c++-mode-hook
      '(lambda ()
         (c-set-style "cc-mode")
         (define-key c++-mode-map "\C-c\C-c" 'compile)
         (define-key c++-mode-map "\C-c\C-e" 'next-error)
;;;      (add-hook 'before-save-hook 'indent-buffer-ask nil t)
;;;      (add-hook 'before-save-hook 'indent-buffer-no-ask nil t)
))
```

Figure 3: Automatic indentation when saving in `emacs`.

If you'd rather not remember this set of keystrokes everytime you save your work, add the code found in Figure 3 to your `~/.emacs` file. [7] (This is a file in your home directory named `.emacs`, which `emacs` consults for customization information whenever `emacs` starts up. It is written in the Emacs Lisp programming language.) Uncomment whichever `add-hook` gives you the auto-indenting behavior you want; note that the semicolon is the comment character in Emacs Lisp. Please note that changing your `~/.emacs` file changes the behavior of `emacs` for future sessions, and not the current session. Instead of going through the steps necessary to make it work on the current `emacs` session, it's easier to simply quit the current `emacs` session and fire up a new one.

Now do yourself a favor and *look* at the overall indentation of the program; if the indentation looks incorrect, you're probably missing an important character (such as brace or a semicolon) someplace.

5. Don't use overly-wide indentation. Usually three or four spaces per level is about right. Eight spaces (the default provided by `emacs`) is definitely far too many, since it causes your program to eat up horizontal space. If you're using `emacs`, you can adjust this by putting

   ```
   (setq c-basic-offset 4)
   ```

   in your `~/.emacs` file. Play around with various values to see what you like best; maybe you'd like an even smaller value (`2` or `3`).

6. Use a consistent bracing style. In other words, be consistent where you put opening braces. I like to distinguish between two kinds of braces:

   - The opening brace for a block that defines a `class`, `struct`, or function body should appear on its own line, e.g.,

     ```
     int foo (int bar)
     {
     ```

[7] A variant of this code may already be there, towards the end of your `~/.emacs` file. This will save you a lot of error-prone typing. If it's not there, you should be able to do a cut-and-paste from this document into your `~/.emacs` file.

```
        .
        .
        .
    }
```

- The opening brace for a block that appears as the body of a control statement (`if`, `while`, `try`, etc.) should appear on the same line as the header for the control statement, e.g.,

```
while (baz > 0) {
   bar = quux;
   snap = crackle;
}
```

Some people like to have all opening braces appear on their own lines. If you *really* want to do this, please make sure that the braces for control statements don't get indented. In other words,

```
while (baz > 0)
{
   bar = quux;
   snap = crackle;
}
```

is okay, but

```
while (baz > 0)
   {
      bar = quux;
      snap = crackle;
   }
```

eats up too much horizontal space; it's even worse when combined with overly-wide indentation:

```
while (baz > 0)
        {
                bar = quux;
                snap = crackle;
        }
```

If you imagine a similarly-indented `if`-statement within such a `while`-loop, and you'll see what is sometimes referred to as the "accordion effect".

You will lose points on style if your code looks like either of the previous two examples.

If you really insist on putting the opening brace of a control structure on its own line, you'll find that the default indentation that `emacs` produces is exactly the indentation given in the last two examples. The good news is that this is fixable.[8] The fix is to simply put

```
(setq c-set-style "stroustrup")
```

in the `~/.emacs` file.

[8] The behavior of virtually anything in `emacs` can be reconfigured.

7. Identifiers are used to name things (variables, constants, classes, functions, etc.). Here are some thoughts regarding identifiers:

   - Use mnemonic identifier names where appropriate. In other words, the name of an identifier should be almost self-documenting (but see the earlier discussion under documentation). However, don't make this into a fetish. As a rule of thumb, the more "global" an identifier is, the more important it is for that identifier to have a good name. Hence the names of functions, classes, and class members should be carefully chosen. At the other extreme, the scope of a loop control variable tends to be fairly small. If your loop control variable is of integer type (as is often the case), so you can get by with names such as `i`, `j`, ..., `n` for same.[9] In other words, don't use `loop_control_variable` for the name of a loop control variable; use (e.g.) `i` instead. Moreover, if you have an embedded loop, this would allow you to use `j` for the inner loop control variable, e.g.,

     ```
     for (int i = 0; i < num_rows; i++) {
         // stuff
         for (int j = 0; j < num_columns; j++) {
            // more stuff
        }
        // still more stuff
     }
     ```

   - This brings us to the issue of identifier names that are compound nouns. There are two schools of thought here. To separate the words in such an identifier, you can either use "camel casing" (e.g., `getUserName()`) or you can use underscores (e.g., `get_user_name()`). The choice of which to use is a "religious argument". My general policy when teaching a course is to follow the text's choice (e.g., underscores with Stroustrup's *Programmming: Principles and Practice Using C++* and camel casing with Carrano's *Walls and Mirrors*. In any case, be consistent, i.e., don't use both styles within a given program.

   - The name of a datatype should start with a capital letter; the name of anything else should start with a lower-case letter. For instance:

     ```
     class Employee { ... };
     .
     .
     .
     Employee emp;
     ```

     This will help the reader to know whether a given identifier names a datatype or something else. (Note that this is not quite a standard; in particular, the Standard Template Library doesn't follow this rule.)

8. Horizontal spacing can either add to or detract from your program's readability.

   Here's where I like to use a single horizontal space:

   - With includes, e.g., "`#include␣<iostream>`".
   - After `for`, `while`, `if`, etc., e.g., "`while␣(i␣<␣0)`" or "`if␣(x␣>␣0)`".
   - On either side of an assignment operator, e.g., "`x␣=␣a␣-␣b`" or "`x += 2`".
   - After a comma operator or a semicolon, e.g., "`enum␣Color␣=␣{␣RED,␣BLUE,␣GREEN};`" or "`for␣(int␣i␣=␣0;␣i␣<␣foo.size();␣i++)␣{`".
   - For arithmetic (or logical) expressions, I like to follow mathematical typesetting rules that go back hundreds of years, and put a single horizontal space
     - on either side of an additive operator, e.g., "`a␣+␣b`" or "`a␣||␣b`", and

[9] These names are fairly conventional for loop control variables. If you would like a mnemonic device to help you remember them, simply recall that `i` and `n` are the first two letters of the word "integer" (or, if you prefer, the C++ reserved word `int`.

– on either side of a relational operator, e.g., "`a␣==␣b`" or "`a␣<␣b`".

I generally don't use horizontal spaces on either side of multiplicative operators. So the algebraic expression "$a + bc$" would be written as "`a␣+␣b*c`".

9. Vertical spacing can also affect your program's readability. You should use a blank line to separate the chunks of your program. For example:

   - After a file's initial block comment.
   - Between the `#include` section and the function declaration section.
   - Before the (comment that precedes the) definition of any function.
   - Between major sections of a function.

   By the way, it wouldn't hurt to put a descriptive comment before each of these sections.

10. Most students quickly learn that expressions should not be under-parenthesized. As a result, they often tend to err in the opposite direction, writing expressions such as "`(((a + ((b))*c)))`", which are nearly unreadable.

    It's probably okay to assume that everybody knows the following rules (which are based on the corresponding rules in algebra):

    - Multiplicative operations have a higher priority than additive operations, which means that you should use "`a/b + c`" instead of "`(a/b) + c`".
    - Relational operations have a lower priority than either arithmetic operations or logical operations, which means that you should use "`b*b < 4*a*c`" rather than "`(b*b) < (4*a*c)`".

    Of course, when in doubt, use parentheses!

11. Avoid "magic numbers," i.e., explicit numbers that appear in the code with no apparent explanation. (Sometimes people call these "manifest constants".) For example, consider the statement

    ```
    cost = price*1.07;
    ```

    What's the purpose of the `1.07`? It would be better to use

    ```
    const double tax_rate = 0.07;
    .
    .
    .
    cost = price*(1 + tax_rate);
    ```

    Similarly, in the statement

    ```
    x_coord = length*sin(1.5708*theta);
    ```

    the role of `1.5708` is a mystery. If you use

    ```
    const double pi = 3.1416;
    const double pi_by_2 = pi/2;
    .
    .
    .
    x_coord = length*sin(pi_by_2*theta);
    ```

    things will be clearer. In all honesty, I probably would do eliminate `pi_by_2` and simply use

```
const double pi = 3.1416;
.
.
.
x_coord = length*sin(pi*theta/2);
```

As with all things, this can be overdone. Certain numbers (such as `0`, `1`, `2`) can be used as is. It may take a bit of experience to learn the when you should use manifest constants (such as `0`, `1`, `2`) and when you should use named constants (such as `tax_rate` or `pi`).

One last note: Some folks like to use `ALL_UPPER_CASE` to define `const`s, and some don't. (There are historical reasons for why some people do this.) My advice is to follow the standard used in your text. If your text has no particular standard, I don't care whether you use upper case or not, as long as you're consistent.