Programming Style Sheet Details for CSCE 240

1 Introduction

Mature, adult, professional programmers ought to know how to write code that will be taken as the output of mature, adult, professionals. This style sheet is an attempt to define what will be taken as proper coding style for CSCE 240. For anything approaching a complicated program, there are usually many ways to write the program so that it executes "correctly". Correct execution on a homework assignment, however, is a *minimal standard*, and by the time you get to CSCE 240 we will be expecting more than that bare minimum.

Students should remember when they think of writing code that the bulk of the expense of a software project (estimates range from 60% to 80%) is the cost of maintaining the code, not the initial cost of writing the code. If you do not write clear, simple, readable, code, or if you do not provide meaningful documentation, you will add to that maintenance cost. That's a bad thing.

For many of the strictures detailed here, you will almost certainly be able to find a style guide elsewhere that says that it is permissible to write code in a different way. That, however, is not relevant to your grade in 240. Each of the strictures detailed here can be justified as good practice. Our experience is that without what might seem at first to be an arbitrary set of standards, our students tend to write code according to their own standard, or no standard, or even according to multiple standards within the same program. This is simply bad form; this is not what mature, adult, professional programmers would do. Writing code according to one particular style guide does not mean that you will have written code that conforms to all possible style guides, but it will help prevent you from writing sloppy code. And based on our collective experience, code that follows a standard–any standard–is less likely to be incorrect due to sloppy mistakes than is code written to no standard. It is also more likely to be readable and maintainable by someone other than the original programmer.

We require you to follow these standards, because we will be grading according to these standards. We really do not like to take points off for reasons such as these. Points taken off for bad logic are not hard for the

faculty to accept. But given that we have, in this document, told you exactly how not to have points taken off for style, it is baffling and frustrating for us as faculty to understand why students seem deliberately to choose not to write to the standard and thus deliberately choose to require us to deduct points.

One final thing: REMEMBER YOUR AUDIENCE.

For the most part, in your professional lives, and as preparation for that in your academic life, you are not writing code for yourself, although you will no doubt see your code. You are also not writing code for your instructor, although she or he will see your code. You are not writing documentation for an expert who made the assignment in the class.

You are writing code for an unknown audience of people who will see, use, maintain, or modify your code at some unspecified time in the future. Those readers will not have participated in your coding process, and they are unlikely to know what design decisions you made as you wrote the code. Whether it is a supervisor who is too busy to know the details of your life, or a subordinate newly-tasked with maintaining code you wrote last year, you should assume that the reader of your written work and programs knows less than you do about the details. It is your job, then, to make those details clear.

As such, then, your writing and code must stand alone, readable by a more naive user who may well not have configured all the software tools in the way that you have.

2 Code As If You Really Meant It

Some basic guidelines for the process of writing code:

- As just mentioned above, remember that from now on you are writing not for yourself or for your instructor, but for the world. It is not the purpose of your code to meet the specifications of an assignment. The purpose of your code is to perform some computational function. You should try to design and write code as if someone else were going to take it, independent of the fact that it happens to have been written for an assignment in a class, and then use that code.

  To the extent that it is feasible, therefore, your code should be independent, portable, and general-purpose. And, most importantly, it should

be written in such a way that changes to the code are possible without undue hassle.

- Build out a skeleton of stubs first. That is, write stub functions that merely take in their input parameters and return the appropriate output parameters, with perhaps an appropriate output "I got here" line to be able to trace the execution.

  Two positive benefits can result. One is that you can have a program that actually executes while you build out the functions with the real code. It is always easier to debug a program that executes but does not yet produce the correct output than it is to debug something that doesn't yet execute. (See elsewhere about the scientific method for experiments.)

  Second, this will often uncover logic issues in passing parameters or in detecting error conditions. For example, if a function is passed a parameter that might generate an exception, then you will either have to deal with that exception inside the function (output an error line? dump code and die?) or else you will have to return a `boolean` flag variable to tell the calling program that the exception occurred. This is the kind of complication that can often be overlooked, because we tend to write code to do what we want it to do, not to deal with exceptions. (See elsewhere on coding defensively.)

  Hint: You can always comment out code you don't yet want to execute, or in C++ hide it with an `ifdef` to keep it from being compiled at the moment.

- Build out your code breadth-first, not depth first. The code in a given function should be roughly at the same level of complexity.

  If you are writing the basic function to process the data, which might perhaps be called simply `doTheWork`, then the main steps are likely to be

  ```
  get the input data
  process the input data
  output the results
  ```

  Do not clutter up these three basic steps by expanding the `get the input data` code inline; call a function instead. It is sufficient at the

level of the `doTheWork` function that `doTheWork` knows that input data has been provided to the function and that the data has been verified and validated to meet the input specs. In a program of any reasoanble size, the `doTheWork` function does not have a need to know what that validation process has been. Create a function to input data, and return a `boolean` to inform `doTheWork` whether or not the input data has been obtained. In your `getInputData` function, then, you can focus on what may be a number of tedious tiny details of validating input.

Similarly, there are many instances in which formatting the output can be detailed and tedious. Write a function for this specific purpose.

In a main program, if a file is to be opened, you will want to check that the file name is valid and that the file has in fact been opened. The code to do this is simple and only takes a few lines, but this is really a utility function. Create such a function, pass it the file name, and let the function do the error checking so you don't clutter up the train of thought in your main program.

- Code so that testing is easy.

  If it will be necessary to test your code on multiple data inputs, then write your code so that this will be easy. If you are doing something delicate, don't try to do five different things on one line of code; break the steps up. If the program specs require you to have a `toString` function that does nice formatting on that part of the data that the user is supposed to see, then you may well want to have another function `toStringRaw` that dumps all the data from a given class.

  This advice goes along with the principle that functions should have essentially one purpose, that your code should be written in obvious and identifiable chunks, and that you should be able to examine the bread crumbs from one execution step to the next as the program executes.

- Code defensively.

  John Mashey used to give a great lecture on the fact that almost everything done by human beings–art, music, sports, start up business, etc.–turns out to be a failure.

  Do not design and code your programs under the assumption that things are going to work correctly. Instead, design and code with the idea that you want to make it harder to fail.

If you are doing a program that deals with a tree data structure, you might well have to write a function that displays the data in the tree. Even though it is more work, you should think about writing that function so it displays all the data starting from an input parameter value as the root of the tree.

- Design and code using the basic principle used for scientific experiments– you want to test one variable at a time. If you make one change to a program and it stops working, it's a reasonable bet that the change is what has triggered the error. If you make two changes, then either Change A is an error, or else Change B is an error, or perhaps both of them are. If you make three changes, there are seven possible situations.

- Do not expect that diagnosis of errors and debugging of programs requires you to think about the philosophical nature of your program. It doesn't. There may be some errors that can be found by asking the question, "Why is this not working?" but most errors are not found this way. Remember that whether the program is working correctly or not, it is in fact *doing something*. If that "something' is not The Right Thing, then the first step in diagnosis and debugging is the empirical process of finding out *what* that "something" is. If you know *what* the program is doing, and if what the program is doing is not the right thing, then almost certainly one of two things is true: either you have a specific error in your logic (like running a loop one too many times), or else you have misunderstood what the language construct or library function is doing. Either way, answering the "what" question comes before knowing "why" the error exists.

- Most of the cost in a software package is not the cost of creating it in the first place. Some 60-80% of the total cost of a software package is in maintaining and updating the package. Given that that is the case, you should view your code as something that will never actually be "finished" and design and code accordingly. Don't write code just to meet the minimum specs. Write the code so that if the specs suddenly change, you don't have to start from scratch.

- Your code should be abstracted as much as possible, but not too much, especially in the initial stages. You have a search tree to be manipu-

lated? Then you will probably want a function to display the tree. But don't hard code the variable for the tree in the function; pass that in as a parameter instead. Don't hard code the root of the tree. Pass that in as a parameter so you can start the display from anywhere, because eventually you are going to want to do something like that.

At some point, of course, you need to quit abstracting. In the Iron Age, there was an IBM utility called `USERDEBE` that was invoked with different parameters for almost any kind of file action. I don't know what the true acronym was for `USERDEBE`; the standard response if someone asked was that the last part stood for "Does Everything But Eat". It isn't necessary that *all* your functions have *total* generality.

- Similarly, your code should be as portable as possible, with the limitation that making code truly portable is a huge task in and of itself. This CSCE 240 class uses the Linux platforms in the computer labs as the reference machines. Code turned in must run on these machines without changes and without errors. There is an incentive, in making code portable, to making it as vanilla as possible. If a function you want is in the standard libraries on the lab machines, it is probably fair game to use it, but you do need to be aware of what might be standard and what might well not be standard.

  This admonition is especially relevant to those who might be writing their assignment programs on machines other than those in the labs. You might have a different version of Linux installed on the machine on which you are developing code, and it might have slightly different compilers or libraries.

  You should also be careful not to use Microsoft-only features. Just as with IBM-only features in an earlier era, these are generally not found on Linux machines. If you are going to develop on a Windows machine or a Mac, you need to be doubly careful to verify that your code runs on the lab machines.

  You do not need to write code that is infinitely portable, but part of really knowing about writing programs is knowing what parts of your code might not be portable and thus need to be written with care and with a proper admonition to the user.

3 Files, File Types, File Names

One of the complications in life that must be dealt with by any techno-wizard is the multitude of file types, formats, and names. These complications aren't going to go away any time soon, so it is necessary that you be able to deal with them.

As the techno-wizard, remembering your audience, it is **your job** *to minimize the extent to which your less-sophisticated readers have to deal with these complications, not their job to figure out what you might have done.* You are the expert making life easier for them; you have to accommodate their lack of knowledge, not force them to acquire your knowledge.

First: Unix/Linux file names are case-sensitive; a file `onefilename` is different from a file `OneFileName`. In Windows, although you can use upper and lower case to make file names (possibly) more readable, the actual file name is not case sensitive. This can be seen by trying to create two files with the above names; Windows won't let you do this. If you do go ahead and create two such files on the shared file system using Linux, then Windows will take these as the same file.

As the techno-wizard, it is your job to ensure that file names either are or are not case sensitive depending on what the naive user downstream will know to expect.

Second: Unix/Linux files have no inherent extension to the file name and no meaning attached to a file name. In Unix/Linux, a file is a file is a file, and it consists of a collection of bytes. In contrast, windows won't let you create a file without an extension, because it insists that each file have a "meaning" attached to it. There is, as far as Unix/Linux is concerned, no "type" associated with a file, and thus a file name can be any legal string.

As the techno-wizard, it is your job to ensure that file names either do or do not have file extensions, as appropriate or not, and it is your responsibility to ensure that the extension, if mandated, is included in your instructions. Do not tell a user to create a file `MyFile` if what you really mean is that a file `MyFile.txt` should be created.

Third: Unix/Linux files consist of exactly the bytes you insert into the file. There is nothing added by the operating system. Thus, a program edited under Unix/Linux has lines that end with a newline character (ascii decimal 10), because you the user hit "enter" at the end of the line. In Windows, there is a carriage return (ascii decimal 13) inserted before the newline. This can be a big deal if you are reading files character by character.

As the techno-wizard, it is your job to ensure that files either do or do not have the extra carriage return.

4 Function and Variable Names

4.1 Names for Functions and for Ordinary Variables

Variable names for functions and for ordinary variables should be written in "camel case" with the first letter not capitalized but with the first letter of subsequent embedded words capitalized. Thus `countBeforeSumming` and `countAfterSumming` with the `A`, `B`, and `S` capitalized but the initial `c` not capitalized.

Function and variable names should be meaningful, but they need not be overly long.

Be crisp in your use of function and variable names and avoid ambiguity. For example, in a loan amortization program, a variable called `interestRate` is ambiguous. Is this the yearly rate? the monthly rate? Be clear.

Similarly, do not change the meaning of a variable in the middle of a unit of code. Do not, for example, use a variable `interestRate`, read in a yearly interest rate, and then divide by 12 and use the same variable as the monthly interest rate.

4.2 Names for Constants

Variable names for constants should be written all in caps so the casual reader can easily see them to be constants. Thus `LINESPERPAGE` or `LINES_PER_PAGE` for something like a 55 lines per normal page fixed value for formatting output.

5 Indentation

Code should be indented properly and consistently. There are multiple "correct" ways to indent a block of code. Any of

```
for(int i = 0; i < 10; ++i)
{
  sum += i;
```

```
  for(int j = 0; j < 10; ++j)
  {
    sum += j*3;
  }
}
```

or

```
for(int i = 0; i < 10; ++i) {
  sum += i;
  for(int j = 0; j < 10; ++j) {
    sum += j*3;
  }
}
```

or

```
for(int i = 0; i < 10; ++i)
  {
    sum += i;
    for(int j = 0; j < 10; ++j)
      {
        sum += j*3;
      }
  }
```

will be viewed to be acceptable according to one style sheet or another, although some style sheets will demand one of the three. The first form is the most symmetric, but it takes more lines (and thus more screen space) than the second. The second takes fewer lines on the screen, but also requires the reader to look carefully at the line with the `for` loop in order to determine that the object of the loop is more than one line of code. (The loop on `j`, for example, might simply be indented incorrectly.) The third is symmetric, and it is what comes out of some automatic formatting programs, but the raggedness of the indentation for the last two closing braces requires a closer scrutiny than is needed for the first example.

What is likely to viewed as incorrect by all style sheets is to have, in the same piece of code, instances of more than one of these. Therefore, **choose one of the above, and use that choice exclusively in any given**

homework assignment. It is ok to change styles from one assignment to the next if you decide you don't like your previous choice. But don't change styles in the middle of one assignment.

You might be aware of the `indent` program available on Unix/Linux systems. This program works reasonably well on files created on Unix/Linux systems with standard Unix/Linux editors. You should be aware, though, that it does some mysterious things when confronted with an input file with the extra Microsoft carriage returns and with a mixture of tabs and blank spaces for indentation. The `indent` program can be used, but it is not magical and you should examine the output afterwards to ensure that its output is really what you want.

6 Line Wrap

It is very painful to try to read a chunk of code like the following (this is a real example).

```
cout << "\nFor a loan of $" << principal << endl;       //following
statements echo the user
cout << "with monthly payments of $" << paymentPerMonth << endl;
  // input variables
cout << "and an interest rate of " << interestRate << " percent" <<
endl;

cout << "The payment schedule is as follows: \n" << endl;

while(newPrincipal > 0){//loop prints out payment schedule for each
month


        if(interest > payment){//determines if there is an error(pa
yment insufficient for loan)
                cout << "Error: Payment is insufficient to pay off
the loan" << endl;
                        <<"\tthe initial interest is: $" << intere
st << endl;
                exit(1);
        }
```

```
        if(counter == 1){       //if statement prints out headers f
or the schedule
                snprintf(s, "%10s        %5s             %5s             %
5s              %5s\n", "Month", "Int", "Pmt", "Old Pr", "New Pr");
                cout << s;
        }
        //print out payment schedule now
        snprintf(s, "%10d        %6.2f   %15.2f  %14.2f  %14.2f\n",
counter, interest, payment, oldPrincipal, newPrincipal);
        cout << s;

        oldPrincipal = newPrincipal;    //recalculate all variables
for next months payment
        interest = ((interestRate * .01)/12) * oldPrincipal;
        payment = paymentPerMonth - interest;
        newPrincipal = oldPrincipal - payment;
        ++counter;
        //end while loop
}
sprintf(s, "%10d        %6.2f   %15.2f  %14.2f  %14.2f\n", counter,
interest, payment, oldPrincipal, newPrincipal);
```

This happens because students use a variety of different editors and IDEs to write their source code, and then they do not adjust the code so as to allow it to be displayed conveniently with virtually any IDE or editor.

Many window sizes are still set up as if the standard display was of an ordinary sheet of paper. Do not let your lines wrap past 80 columns. In fact, you should probably stop sufficiently short of 80 columns in order to make sure that you don't inadvertently go past 80 columns. If you are careful, then your code will be readable on an 80-column screen window as well as being printable on a standard piece of paper. You are writing code for the world; make it readable without extra effort. Just as you would write web applications to make them display properly on any of the standard browsers, so should you be writing code so as to make it display in a readable way on any of the standard editors or IDEs. Code lines absolutely should not wrap at random places beyond an 80-column line; force a line break at a semantically meaningful place. The code above (which is a compression of lines of code

from more than one routine) is more readable if written as below.

Note also that the first line provides a clear visual cue about how many columns have been used.

```
//34567890123456789012345678901234567890123456789012345678901234567890
// Echo the user input.
cout << "\nFor a loan of $" << principal << endl;
cout << "with monthly payments of $" << paymentPerMonth << endl;
cout << "and an interest rate of " << interestRate << " percent"
     << endl;

cout << "The payment schedule is as follows: \n" << endl;

// Loop to print the payment schedule.
while(newPrincipal > 0) {
        // If the initial interest is greater than the planned
        // payment, then the loan will never be paid off.
        if(interest > payment){
                cout << "Error: Payment is insufficient to "
                     << "pay off the loan" << endl;
                     <<"\tthe initial interest is: $"
                     << interest << endl;
                exit(1);
        }

        // Print the header the first time through.
               if(counter == 1){
                snprintf(s1, "%10s        %5s", "Month", "Int");
                snprintf(s2, "%5s        %5s", "Pmt", "Old Pr");
                snprintf(s3, "%5s\n", "New Pr");
                cout << s1 << s2 << s3;
        }
        //Print out payment schedule now
        snprintf(s, "%10d        %6.2f   %15.2f  ",
                     counter, interest, payment);
        snprintf(s, "%14.2f  %14.2f\n",
                     oldPrincipal, newPrincipal);
        cout << s;
```

```
        // Recalculate all variables for next month's payment.
        oldPrincipal = newPrincipal;
        interest = ((interestRate * .01)/12) * oldPrincipal;
        payment = paymentPerMonth - interest;
        newPrincipal = oldPrincipal - payment;
        ++counter;
        //end while loop
}
snprintf(s, "%10d        %6.2f   %15.2f  ",
             counter, interest, payment);
snprintf(s, "%14.2f  %14.2f\n",
             oldPrincipal, newPrincipal);
```

7 Tab Characters

You will no doubt have noticed by now that some development environments use tab characters for indentation. You should probably view this as A Decidedly Bad Thing, for the following two reasons.

First: One tab, at a usual eight spaces, is ok. Two tabs is somewhat less ok–that's a lot of indentation. Three tabs, indenting 24 spaces in, will leave relatively little space to the right for you to write code in without having to worry about line wrap.

Second: Even if *you*, in your editor or IDE, choose to reset the tabs down to two or three characters, you cannot guarantee that everyone who reads your code will have done this, and you cannot expect them to make that extra effort willingly.

Don't force other readers to go to extra work just to read your code. Indent your code not with tabs but with two spaces or with three spaces consistently. One space is not enough, but two or three are ok. Four is also ok, but starts leaving too little room to the right for the actual code. But *be consistent in a given assignment.*

Remember also that the simple Unix command

```
sed '1,$s/ /  /g' oldFileName >newFileName
```

where the first white space in between slashes is a tab character, and the second white space in between slashes is two blank spaces, will substitute

two blank spaces for every tab character throughout the file `oldFileName` and will output the changed file as a new file named `newFileName`.

(Remember also that one of the evil things about tab characters is that a space followed by a tab is usually not the same thing as a tab followed by a space. But since both are white space characters, you don't actually "see" the tab character, so you don't really know what you have for formatting unless you use a feature in your editor that will display the ASCII code for a tab (a decimal 9). No such confusion can exist if you stick to blank spaces; a blank space is a blank space is a blank space.)

REMEMBER YOUR AUDIENCE: You will know what settings you have on your IDE or editor. Your instructor will no doubt be able to change settings so as to match yours. But your job is to write in such a way that these changes are not necessary. If you have five people on a project, and they each write with different settings, it will be a nightmare trying to merge the code into a coherent unit readable by all. Standards exist to diminish these kinds of incompatibilities.

8 Spacing

There was a time when memory was scarce on the early toy computers of the late 1970s and early 1980s. Code of that era was often written in the most dense, and thus most unreadable, form. The world has changed, and memory space for source code is not an expensive commodity.

You should space uniformly before and after arithmetic operators, equals signs used for assignment, commas after parameters in function declarations, and the like. This will make the code more readable.

Spacing before and/or after the parentheses in a `while`, `if` or `for` statement is a matter of religious debate. Whatever you do, do it consistently.

You might be forgiven for adapting the spacing around arithmetic operators if an adapted spacing actually makes the expression more rather than less readable. For example, one could certainly argue that

```
x = a*x*x + b*x*y + c*y*y;
```

is more readable than

```
x = a * x * x + b * x * y + c * y * y;
```

because it more clearly indicates a sum of products. Do this only after thinking about it; don't do this randomly and inconsistently.

9 Documentation

Every class, function, etc., should have documentation that explains what the function does, what the input and output of the function are, and what special things must be remembered in order to understand the function and maintain the code.

At the top of every file (header, class implementation, etc.) should be a statement of the general purpose of the code, the author of the code, a copyright notice, and a modification date. For example,

```
/*************************************************************
 * Homework 5 Main program
 * Author/copyright: Duncan Buell
 * Date last modified: 12 October 2009
 *
 * This program will create a simulated memory structure,
 * allocate memory blocks 20 times, and then deallocate
 * memory blocks 7 times.  The 20 and 7 are magic numbers
 * and could be anything, depending on how rigorously one
 * wants to test this code.
 *
 * Again for testing purposes there is a magic constant 'MEM'
 * currently set at 200.  The random number is taken modulo 'MEM'
 * as the size of the block to be allocated.  This could be
 * changed for more thorough testing, and the size of the memory
 * block could be made to vary if one wanted to be more realistic.
 *
 * Input: none
 *
 * Output: only a log of what takes place
**/
```

You should have the author/copyright/date information in all your files submitted, both header and implementation. In many instances, a header file

(and sometimes a main program file) will be provided for you by your instructor. In these cases, you should include your name on the file for completeness.

For example,

```
/*****************************************************************
 * Homework 5 Main program
 * Author/copyright: Duncan Buell
 * Submitted by: J. Random Programmer
 * Date last modified: 12 October 2009
**/
```

Sometimes you will need to (or want to) modify the file. In that case, something like the following would be appropriate. You should always be concerned about preserving the copyright of intellectual property that is entrusted to you, and thus the attribution to the original programmer is important. You should also be concerned about preserving your rights to any modifications.

```
/*****************************************************************
 * Homework 5 Main program
 * Author/copyright: Duncan Buell
 *                   modifications by J. Random Programmer
 * Date last modified: 12 October 2009
**/
```

The first statement that documents a function, or class, or header, should be a top level statement about the general purpose of the function.

The next statements about a function should explain in more detail how it is that that function is implemented. Remember that the reader will be able to look at the code, so explain the implementation in terms that are relevant to the purpose of the code, not just specifics of the variable names. If the purpose of the function is a recursive search through a tree structure, do *not* say, "we visit left child, current node, right child." That much is probably understandable by reading the code. Instead, write "we do an inorder traversal of the tree."

If the purpose of the code is to walk through a two-dimensional array, do not say "we run an outer loop on `i` and an inner loop on `j`." Instead, say, "we run an outer loop on rows and an inner loop on columns."

Documentation should be written in the third person, or in passive voice, but not in the first person, and not in the colloquially-common but grammatically-incorrect "you" mode. Use the editorial "we". You are not writing for the instructor of 240 as the grader of the program. You are writing for the unknown maintainer of your code some months or years in the future.

All functions must document the input parameters, the returned values, and any output to the console or a file. This should be done in bulleted fashion similar to the style that is mandated for use by Javadoc.

For example:

```
/*************************************************************
 * Function to deallocate memory.
 *
 * We run a loop on the subscript and shadow that with an
 * increment to an iterator.  This is because the 'erase' works
 * not with subscripts but with iterators.  This is a bit of a
 * kluge and could be improved in a rewrite.
 *
 * For debugging and tracing execution, we go ahead and dump the
 * block data up to the vector entry that we are going to delete.
 *
 * We don't do any error checking in this function.  We assume
 * that error checking is done before calling this function.
 *
 * Parameters:
 *   which - the subscript in the 'vector' to be deallocated
 *
 * Output: none except the tracing/logging information that
 *         could disappear in a "real" implementation
 *
 * Returns: none
**/
void Memory::deallocate(long which)
{
  vector<MemBlock>::iterator iter;

  iter = theMemory.begin();
  for(long i = 1; i <= which; ++i)
```

```
  {
    ++iter;
    cout << TAG << "block " << i << " "
         << (*iter).toString();
  }
  theMemory.erase(iter);
} // void Memory::deallocate(long which)
```

9.1 Inline Documentation

The documentation mentioned above is what should appear at the top of any function. If you do this right, there is usually not much need for inline documentation except to explain non-obvious or clever things that you have done. In any event, do not document every line of code. Assume that the reader has some understanding of programming. A line or two at the beginning of major blocks of code might well be enough to guide the reader.

9.2 "Gotchas"

One of the most important parts of documenting a function in the preamble to the function is to mention any "gotchas" that a casual reader might overlook or not immediately understand.

In the example just above, the comment about error checking is an example of this. A careful reader will know to think about error checking, but since this module does not do error checking, it is important to point that out.

Another example: Some algorithms dealing with binary trees are more easily explained if the nodes in the binary tree are numbered 1 for the root, 2 and 3 for the next level, 4, 5, 6, and 7 for the next level, and so on. If you do this, and you use an array for the nodes, you will have to decide whether to ignore the zero-th subscript and to index all the way to the end, which wastes one memory location but makes the code for subscripting easy to write:

```
for(int i = 1; i <= numberOfNodes; ++i)
{
  // code in which node i has subscript i
}
```

or to index zero up in the usual fashion of Java and C++ and adjust all subscripts accordingly (which will be tedious to code):

```
for(int i = 0; i < numberOfNodes; ++i)
{
  // code in which node i has subscript i-1
}
```

Either way, you should be careful to point this out to the reader of your code. This is the kind of "gotcha" that must be documented.

You cannot expect a less-experienced user to understand exactly what your code does without careful reading, but you can at least point out to the reader that there is something that needs careful reading. When we read code, we may well gloss over the `for` loop because it's a pattern, and we may think we see the pattern to be what we have seen many times before, so being told that it's not "the usual pattern" is important.

9.3 Spelling of Jargon Terms

We all make typing mistakes, and occasional misspellings or typos will creep into your code and documentation. However, misspelling jargon terms will lead a reader to question your competence. An interviewer might ask the question, "If you have read the word 'parameter' in five different classes, and still consistently spell it 'paramiter', then how can we believe that you paid attention to anything else that went on in class?"

This is something that you should be especially careful of in naming variables. If a variable name derives from a jargon term, then be careful to spell it correctly.

You do not want something as silly as misspelling to lead someone to question your actual technical skills, and yet it will. So spell jargon terms correctly.

9.4 Fonts

You should have noticed by now that writing text about computer programs can be awkward. In this document, for example, the computer text itself is done not in the default font for text but in fixed-width "typewriter font". Those who write papers and books about computing invariably have to put

in some extra work in order to make sure that words and letters with meaning beyond standard English are displayed in a font that distinguishes that extra meaning. (In writing mathematics, italic font is used for mathematical symbols.)

In a computer program, it is hard to convey that extra meaning, because only one font is used. If you look at the Javadoc for a Java program, though, you will notice that the computer words themselves come out in typewriter font. This happens because proper Javadoc requires that computer words, like variable names and method names, be written in the comments section of source code as

`<code>variableName</code>` or `<code>methodName</code>`

in an HTML-like fashion. If the `<code>` and `</code>` tags are used, then the Javadoc converts those words to typewriter font.

You could accomplish something like the same result by enclosing computer words (like variable names) in single quotation marks. If you have a function whose signature is

```
void setWhich(MyObject that, long which)
```

then a sentence like

```
// We must be careful to distinguish which which is being used,
// because this has a which, that has a which, and we know which
// which to set based on the comparison of the this which with the
// that which.
```

can be written slightly less confusingly as

```
// We must be careful to distinguish which 'which' is being used,
// because 'this' has a 'which', 'that' has a 'which', and we know
// which 'which' to set based on the comparison of the 'this'
// 'which' with the 'that' 'which'.
```

9.5 Say What You Mean

Be crisp, clear, and unambiguous in your documentation. Do not speak in generalities–your code does specific, not general, things, and your documentation should say what it does.

Remember that words have usual meanings, and jargon terms have specific meanings. Even though you might use the `snprintf` command to produce a formatted string, do not say that the function that uses this command "prints" something. The word "print" is usually associated with producing output on a printer; the `snprintf` produces a formatting string suitable for actual printing.

If the function reads

```
void myFunc(int howMany)
{
  int myArray[howMany];

  for(int i = 0; i < howMany; ++i)
    myArray[i] = i * i;
}
```

then do not use as documentation the following, which is vague and not very informative (and is an actual example from student work).

```
/*************************************************************
 * This function creates an amount of memory and fills it
 * with values relevant to the computation.
**/
```

Instead, the following tells the reader specifics about the code.

```
/*************************************************************
 * This function creates an 'int' array of length equal to the
 * input parameter and then fills the 'i'-th location with the
 * value 'i * i'.
**/
```

Attention should especially be paid to one detail of writing: since Java and C++ subscript zero-up, you should be careful when referring to "the first" item in an array. Is the first item the item at subscript 0 or the item at subscript 1?

It is possible that you can really understand the code but do not know how to write about it in precise terms. However, it is unlikely that you will be able to write about the code in precise terms without understanding it. (This is a corollary to the theorem that one never really understands something until one tries to teach that material.)

10 Variables

10.1 Global Variables

Global variables are variables whose scope is an entire compilation step, or an entire file. Global variables are identifiable in C++ because they are defined outside any open-close-brace pair.

Global variables are almost always evil, period. Do not use global variables. You may define a *constant* globally (such as `LINESPERPAGE`, mentioned above). But you should never, ever, ever, define an ordinary variable except inside a relevant set of braces.

10.1.1 Class Variables and Local Variables

Similarly, you should be careful to define variables *in the most limited scope possible that will still get the job done.* Variables for loop control should be defined inside the function that uses them. Variables used as temporaries should be defined inside the function that uses them. The only justification for declaring a temporary variable as a class variable is if the program is a computationally intensive application for which the creation and deletion of the temporary would be an unnecessary addition to the run time. You are not going to be in this situation in CSCE 240, so use local variables whenever possible to allow C++ to do your housekeeping for you.

10.2 Variable Declarations in the Middle of Code

Both Java and C++ allow you to declare variables in the middle of a block of code. This is a practice that you should avoid, and this is a practice that will lead to points being deducted. Declare all your variables at the top of a block of code or the top of a function, before any of the executable statements in the function. The rationale for this stricture is as follows.

First: A variable's scope in a function is from the point at which the variable is declared. The program segment below will not, therefore, compile, because when the line `sum = 10` is encountered, there is no variable `sum` declared (yet).

```
int main( int argc, char *argv[])
{
  sum = 10;
```

```
  int sum;
  sum = 20;
  return 0;
}
```

If you declare all your variables at the top of a block of code, then the scope of those variables will be the entire block of code.
Second: If you consistently declare all variables at the top of a block of code, then a reader always knows exactly where to look to find the type of a variable. If what the reader encountered was code like

```
int main( int argc, char *argv[])
{
  many lines of code
  int sum = 10;
  many lines of code
  sum = 20;
  many lines of code
  return 0;
}
```

then in order to determine the type of `sum` after having read the `sum = 20;` line, one would have to scan upwards fifty lines or so before finding the `int sum = 10;` line where the type was named. Do not place this burden on the reader of your code.

10.3 Type Casting

As you will notice in the text, both the old C type casting in the last line of

```
double x = 10.5;
n = (int) x;
```

and the new version C++-specific type casting

```
double x = 10.5;
n = static_cast<int>(x);
```

are legal and will work. However, you should not use the old version. You are writing code in C++, not in C, and you should use the new techniques for casting.

In general, if there is a new-version C++ way of doing something and the new version is different from the old C version, you should be making every effort to use the newer C++ version.

10.4 Limiting Buffer Overflow Issues

It has been argued that there is nothing in computer security that we need to do that we have not know for 25 years that we need to do. Nonetheless, insecure code continues to be written. In C++, the old and insecure string handling functions `strcpy`, `sprintf`, etc., can still be used. However, these are *strongly deprecated*, and you will lose points for the use of these functions. If you must use a version of these old C-style functions, then you must use the versions that control the number of characters: `strncpy`, `snprintf`, and so forth.

10.5 Magic Numbers

A "magic number" is a number, hard-coded into the text of a program, that has some specific meaning to the program. For example, a routine that formats output for a page printer might very well use the fact that typewriter output is usually done at 55 output lines per page. Writing code such as

```
while(more output)
{
  if(lineCount > 55)
  {
    issue a page throw
  }
}
```

is absolutely forbidden, because the 55 is a magic number. If this is what you want to write, then you should declare

```
#define LINESPERPAGE 55
```

or

```
const int LINESPERPAGE=55
```

somewhere near the top of the program or in a header file and then write:

```
while(more output)
{
  if(lineCount > LINESPERPAGE)
  {
    issue a page throw
  }
}
```

You *will* lose points if your code has magic numbers. There are two basic reasons for our insistence that your code not use magic numbers.

First, the meaning given to the number 55 is made more clear with a symbolic reference to `LINESPERPAGE`.

Second, it is often the case that you will use such a magic number in more than one place in your program. If you have a symbolic reference instead of an actual number, and you have to change that value, you only have to change that value in one place (where the symbol is defined) and not possibly in many places in many classes spread over many files.

11 Side-Effect-Free Programming

As a programmer, you should be writing functions that have no side effects. A side effect is something that happens to be done, perhaps as an afterthought, perhaps because the programmer viewed the side effect as an added feature of the program module. A good function, in contrast, will do essentially only one thing and will have a function name that reflects its purpose.

Side effects are almost always bad and should be avoided. Side effects are bad because it is a bad idea to have some otherwise-irrelevant function changing the value of a variable from another function (in a probably-undocumented way). It is dishonest, underhanded, and sneaky; if you are going to change the value of a variable, then your function names and documentation should admit that fact.

A program function that, say, is supposed to read data from a file should not also pause to send a message to the nearest Starbucks to order coffee. In Java, the strong typing and the syntactic constraints on access to variables by program modules make it much harder to have side effects, but in C++ it is easier to create undesirable side effects (largely due to the presence of pointers).

If your function is named `readData`, then it should be written as a function to read data, and not also to allocate memory, initialize variables, or some other function.

Most of the strictures listed here and mentioned in the texts and elsewhere are intended to make it harder to have side effects. If you always give your variables the minimum scope, then it is harder to have side effects because no function will have access to those variables except those functions that *must* have access to the variables. If you have no global variables, then you will not be able to have "the brilliant idea" of choosing to change the value of one of these global variables inside a function serving some other purpose.

Back Matter

Programming Style Check List for CSCE 240

1. Author name, copyright, and date are present in each file and are correct.
2. The case sensitivity of file names, and the differences in case sensitivity between Unix/Linux and Windows, has been taken into account.
3. File names and extensions are as required by the assignment, and files are either Unix/Linux or Microsoft format as required.
4. Variable and function names mean what they are supposed to mean.
5. Each function has one purpose and executes without side effects.
6. Variables are declared only at the top of blocks of code, not in the middle of code, and they have scope only as far as is needed.
7. All files/classes have header documentation that includes
   (a) an opening sentence or two about the basic purpose of the class;
   (b) a description of the implementation of the class.
8. All functions have header documentation that includes
   (a) an opening sentence or two about the basic purpose of the function;
   (b) a description of the implementation of the function, if it's anything more complicated than an accessor/mutator;
   (c) in bulleted form the input and output parameters, returned values, and external output (e.g., writing to `cout`) for the function.
9. The program is properly and consistently indented, preferably using spaces and not tabs, and without line wraps.
10. The program lines are consistently spaced horizontally for readability.
11. The program lines are consistently spaced vertically in blocks so that blocks of code that correspond to execution units are identifiable.