C++ Programming Style*

Scott D. Anderson

1 Principles

Programming is part engineering and part communication.

Programming is engineering, because the result (the program) is an artifact that *works*—it does something that is useful in its own right. Getting a program to work is hard, particularly when you're a beginner, and you should feel proud of yourself when you get it working. That's also why the majority of your grade on a programming assignment is based on just whether the program works at all. In my class, a working program will receive a minimum of about a 70—a passing score for a working program. The rest of the points come from the efficiency of the program (another aspect of the engineering task) and its style, which is important because programming is communication.

Programming is communication, because programs are not just read by the compiler, but are read by other programmers. In school, they are read by your professor or your teaching assistant, and in industry, they are read by co-workers, review committees, bosses and people who replace you when you leave. In fact, the person reading the code may well be yourself, six months later, when you have to go back and fix a bug or add a feature. All of those people need to be able to understand the program without having to ask you. In school, it may seem silly to explain to the professor what the program does, but it's no more silly than explaining the War of 1812 to your history professor. The educational goals are to improve your communication skills and to ensure that you understand the material. The task for you is to pretend that the person reading your paper or program doesn't know anything about what it's supposed to do.

There is, of course, a balance to be struck, between explaining too much and explaining too little. In your history paper on the War of 1812, you could probably omit that Britain is a small country northwest of France, and the U.S. is a large country in North America. Similarly, in explaining a program, you can assume that the reader knows the programming language. Therefore, for example, you must never do the following:

```
i++;          // increase i by one
```

You should assume your readers know enough C++ so that they know what `++` does, so don't explain the obvious. However, your readers may not know, depending on what `i` is, *why* you are increasing `i`. Keep *purposes* in mind when explaining your code.

Rule 1 *Don't explain the obvious.*

*Parts of this essay have benefited from the contributions of Mark Pauley at the University of Nebraska at Omaha, `Mark_Pauley/IST/UNO/UNEBR@unomail.unomaha.edu`

What is style? In literature, style is what distinguishes one author from another, such as Hemmingway from Joyce. If those two authors described the same event, the two stories would be remarkably different. Maybe we would enjoy reading both, find both equally understandable and equally valuable as a contribution to literature; maybe not. In programming, there are still stylistic differences among authors, but the differences are smaller, and the standards are more exacting, because of the engineering task (two programs to do the same thing will have to be pretty similar) and because of the overwhelming emphasis on clarity. Therefore, programming style is not an opportunity for you to be creative; indeed, many companies have a policy on proper programming style that is rigidly enforced.

Programming style is a time for you to step back from your program, take a deep breath, and try to write it and re-write it until it is as clear as possible. This document will review a number of issues and aspects of programming style and give guidelines and suggestions.

2 Names

In order to tell a story, you have to have names: Jack and Jill, the bartender and the patron, or whatever. If you're going to describe how a program works, you have to name the parts, both the data and the functions. Names are a critical foundation upon which the story of your program is built.

Rule 2 *All names should be* ***descriptive*** *and* ***accurate****.*

Here are some examples of how to name things.

- If a function finds the biggest element in a linked list, don't call it `sort`. It should be called something like `findMax()` or `getBiggest()`.[1] You could even call it `largest`, omitting the "find" part. That would read nicely in calls like:

  ```
  swap(current,biggest(list));
  ```

  The preceding code obviously swaps the current list element with the biggest in the list.

- There are circumstances under which you might name a variable `prev` that will point to elements of linked list. You would do this when there is another variable called `current` or `elt` or something, and `prev` is the previous value of that other variable or is the preceding list element. But if that's not what the variable is, you have just *misled* your reader. This is terrible. I have seen it happen.

- Similarly, I have seen code to find the smallest element in a list in which the name for the current element of the list was `biggest`. It wasn't the biggest *anything*. Again, I was misled.

- By "descriptive," I mean that the name of the variable should talk about the purpose or use of the variable. For example, you would never name a variable `anInt`; surely there is some better description available.

- Calling a variable `count` is little improvement. What is it counting? Why is it counting? If the explanation is too long for a variable name, it can be put in the documentation, but struggle to improve the variable name. Names like `bucketCount` or `loopCount` may need further explanation, but once explained, they are easy to understand and remember.

[1] In text, it's common to denote that something is a function or method by putting empty parentheses after its name. When possible, a `fixed-width` font is used.

- There are, of course, variables that are truly generic. One example is indices used in `for` loops or for accessing arrays. We are all familiar with code like the following:

```
for( int i=0 ; i<arraySize; i++ )
   array[i] = 0;
```

  There is no need to get clever trying to name `i`, for two reasons. First, it is simple and needs no explanation. Secondly, we can see every use of the variable right before our eyes; mnemonic names are most important when a variable will be seen a long way from its initialization and documentation. For such variables, a generic name like `i` or `x` is acceptable. Note that this is not an exception to the rule: a generic name is, in fact, accurate and descriptive.

- Sometimes a more generic name is better for functions and their arguments as well. Imagine a function that swaps two pointers. If we are using it in a program to sort students, we might call it `swapSmallest`, since it will be used to swap the smallest student with one at the beginning of the list. However, this function could also be used in shuffles, permutations, and other unanticipated uses. Therefore, the second declaration is better than the first, because the second is too specific:

```
void swapSmallest( studenPtr & curr, studentPtr & small )

void swap( studentPtr & x, studentPtr & y)
```

- Data members need to be appropriately named as well. The only difference between them and variables is that data members will be accessed via a variable, so the name of the data member is always in a context. For example, the following is probably too cumbersome:

```
class Student {
public:
   char studentName[20];           // The name of the student
}

Student bestStudent;               // The student with best GPA
cout << bestStudent.studentName;   // This seems too long and unwieldy
```

  Everything seems okay until we get to that last line. Then we realize that, since the `studentNname` data member is found in an object of type `Student`, it's redundant to include the "student" in the name. We might go with `best.studentName` but `bestStudent.name` seems better, and even `best.name` is fine. Descriptive names needn't be incredibly long.

Rule 3 *Be consistent with capitalization and punctuation of names. Decide on a scheme and stick with it.*

Because C and C++ are case-sensitive, you can distinguish different kinds of things with capitalization, but be careful about this, because humans are *not* always case-sensitive. How many times have you had to ask whether it's "String.h" or "string.h"? People, particularly computer-people, can train themselves to be case-sensitive, but it's always easy to overlook.

Here are some rules that are in common use or that are reasonable.

- Constants are in all capital letters. For example, `NULL` is in all caps. Very often, `TRUE` and `FALSE` are done that way; C++ seems to be changing to lower case for these. Code that is

substantially uppercase is ugly, so using all caps for variables that occur only every so often is best. Examples that might occur in your programs are `ARRAYMAX` or `MAX_STUDENTS`.

- User-defined types, which includes names of structures and classes, start with a capital letter. Thus, you might name your class `Student` or `Node`.
- Variables, functions, and data members start with a lower-case letters. Thus, you might have variables named `best` or `head` or whatever.

Creating new types—classes and structures—is more rare and important than naming a variable or function. Therefore, it's reasonable to signal this by a capital letter. Constants are usually even more rare, so the uppercase names will be few. At least, this is a sensible view, even if there are exceptions.

You can use other schemes if you prefer. For example, *Foundations of Computer Science*, by Aho and Ullman, typically uses uppercase for new types, so they would write `LIST` or `NODE` as the type for such data structures. This is acceptable as well. However, do not forget to be *consistent*. Don't name one variable `CURR` and the next `prev` because they will look like completely different kinds of things, and the reader will be confused and then annoyed.

Note that a good name often has two or three words in it. The issue of how to separate them always arises. For example, a variable representing the smallest list element should not be named `smallestlistelement`. Instead, it could be named `smallest_list_element`, separating the word by underscores, or `smallestListElement`, indicating word separation by capital letters. Both ways are commonly used, so use whichever you prefer. I think underscores are easier to read, but either is acceptable. As always, be *consistent*.

The Java

3 Structure/Modularity

If we think of programming as writing, the best analogy to a paragraph is a function. In good writing, a paragraph should be clear, focused and propel the argument forward. Similarly, in programming, a function should do one thing and only one thing. A function that reads in a list and sorts it does too much: it should be broken up into two functions, one that does the reading and the other that does the sorting. Later, when we discuss documentation, you'll learn that each function should be preceded by a paragraph describing its use and purpose. If you ever feel tempted to use two or more paragraphs in that description, the function is definitely too long.

Rule 4 *Functions should do just one thing.*

This rule can be confusing: what counts as "one thing?" If a function finds the smallest element in a list is that one thing? If it finds the smallest element and swaps it with the first element, is that one thing? If it repeatedly swaps the next element with the smallest element in the rest of the list, is that one thing? We call the last algorithm "selection sort," so it seems like one thing.

In fact, any of the preceding is acceptable modularity, as long as the function is properly named. Just as some paragraphs are short and others long, some functions are short and do very little, while others are longer and accomplish a lot. You would prefer the smaller functions if they could be called independently, thereby providing a general service, like our `swap` function above. If the smaller parts are useless and the single function doesn't become excessively long, you would prefer the larger function.

What is the proper size of a function? Is there a minimum or maximum number of lines in a function? No, no firm limits can be set, but I can give general guidelines. The most important is determined by the size of the screen: it's annoying to have to scroll back and forth to read a function; it's nice to be able to put all the lines of a function on the screen at once. I would use that limit for any programmer; beginners, however, should aim for shorter functions. Just as beginning writers use shorter sentences with simpler structure, beginning programmers write smaller, shorter programs with smaller, shorter functions. I would advise sophomores to avoid writing functions that are longer than two dozen lines. Juniors and seniors can go longer, particularly if the function is conceptually simple.

Please take the guideline in the previous paragraph with a pinch of salt: some short functions are very conceptually complex, while some long functions are conceptually simple:

```
// Like strncmp, except it ignores case in the comparison.
int strncmp_ic(char *s, char *t, int n)
{
  int diff;

  while( 0 == (diff=toupper(*s++)-toupper(*t++)) & (n--)) ;
  return(diff);
}

void print_info(char *name, char *address, char *phone)
{
  cout << endl
       << ``name = `` << name << endl
       << ``address = `` << address << endl
       << ``phone = `` << phone << endl;
}
```

In the second code segment, I could go on for two dozen more lines of formatting and output, and never run the slightest risk of confusing anyone. The code on the top, however, is only three lines long and shows C at its nightmarish zenith of expressive power. I wouldn't wish five consecutive lines of that kind of stuff on anyone. All lines are not created equal.

When writing a function, we sometimes see that it has conceptually distinct phases or parts. When that happens, it is acceptable, even helpful, to put a blank line between these phases. Even better is a blank line followed by a documentation line that explains what is in store in the next phase. This helps the reader see the sub-structure of the function. However, note that one blank line is sufficient. There is never any reason for two consecutive blank lines within a function, and rarely even in a file. Remember that is is useful to be able to see lots of the code on the screen, so keep it relatively compact.

4 Indentation and Braces

If functions are the paragraphs of the program, the statements are the sentences of the program, and they need to be properly punctuated. In fact, the task is even more demanding than mere punctuation (semi-colons after each statement, for example), because of the way that people read programs. The reader skips from place to place, jumping up to find variable declarations, jumping forward from a loop to the statement following it, and so forth. To make this easy, we must

make the syntactic structure of the program visually evident. The compiler can count braces and semi-colons, but the reader's eye is best guided by indentation. In short:

Rule 5 *Indentation is important. Code must be properly indented to show the syntactic structure of the program.*

The syntactic structure of a program is a tree: statements have parts, and those parts can be statements in their own right, just as English sentences can contain multiple embedded clauses. Here's an example from psycholinguistics:

> The rat the cat the dog chased bit died.

Sounds like gibberish, doesn't it? Even adding the restrictive relative pronoun doesn't help much:

> The rat that the cat that the dog chased bit died.

What if I introduce some indentation:

> The rat
> that the cat
> that the dog chased
> bit
> died.

It's still hard to understand, but we can see from the indentation that "died" belongs with "the rat" and it was the cat that bit the rat, and the dog that chased the cat.

Fortunately, English is rarely so heavily embedded as that toy sentence. Unfortunately, programs usually are. We think nothing of code like the following:

```
for( int i=0; i<n-1; i++ ) {
   small=i;
   for( int j=i; j<n; j++ ) {
      if( A[j]<A[small] )      // line 1
         small=j;              // line 2
   }
   int temp=A[i];              // line 3
   A[i]=A[small];
   A[small]=temp;
}
```

The indentation is crucial to showing that line 2 is part of (contained in) the statement starting on line 1. Similarly, the indentation is crucial to showing that line 3 is not contained in line 1, but is at the same level—both are contained in the outer `for` loop.

Would you rather read the following code?

```
for( int i=0; i<n-1; i++ ) {
small=i;
for( int j=i; j<n; j++ ) {
if( A[j]<A[small] )
small=j;
}
```

```
int temp=A[i];
A[i]=A[small];
A[small]=temp;
}
```

The thought is too horrible to contemplate.

Rule 6 *A statement that belongs to (is part of) another statement should be indented relative to the containing statement. Statements that are at the same level in the syntax tree should be indented the same amount.*

Most students know that they should indent, but they forget or they find it time-consuming. Here are a few hints:

- If you're a "vi" person, use tabs to move over. A couple of tabs are less tedious to type than the equivalent number of spaces. The only drawback is that each tab is often equivalent to eight spaces, which can mean that you quickly find your code starting halfway across the page. However, you can set tab stops to be any size in "vi"; check the man pages to find out how. There is also a pretty-print function in "vi".

- If you're an Emacs person, use the tab key once, and Emacs will automatically indent your code by the correct amount. An important benefit of this is that if Emacs indents your code differently from the way you think it should, it's likely that your code isn't quite right. For example, suppose I forget the opening brace in the inner `for` loop, above:

```
for( int i=0; i<n-1; i++ ) {   // line 1
   small=i;
   for( int j=i; j<n; j++ )    // line 2
      if( A[j]<A[small] )
         small=j;
} // Emacs puts the brace here, closing line 1, not line 2.
```

  We expected Emacs to put the brace under the "f" in the inner `for`, but it put it under the "f" in the outer `for`, because we forgot the opening brace for the inner `for`. Thus, Emacs gives us immediate feedback on syntax errors.

Either of these indentation schemes will result in two other important properties of good indentation: it's *visible* (indenting only 1–2 characters is hard to notice; 3–4 is better) and it's *consistent* (if you sometimes indent 4 spaces and sometimes 8, I can't tell what level some statement is at.

Indentation is defined by syntactic structure, and syntactic structure is essentially defined by *braces*,[2] because braces enclose a sequence of statements where normally only one can occur. Thus, we are led to a discussion of brace placement, over which blood can easily be shed.

One standard method is as follows:

```
for( int i=0; i<n; i++ )
{
  A[i]=0;
}
```

[2]For reasons I don't understand, many people call these characters {} "curly braces" and these [] "square brackets." Since those are the only kind of braces I know, I just call them braces, and similarly I call the other things brackets. There is no reason I can think of for the extra adjectives.

Emacs does this automatically using the tab key, so this way is easy to follow.

Others like to do the following, presumably because the brace is contained in the `for` statement, as well as containing the assignment statement.

```
for( int i=0; i<n; i++ )
  {
    A[i]=0;
  }
```

You can do whichever you like, as long as you're consistent. (It's possible to customize Emacs to do this for you.)

Another way of doing the braces is to begin at the end of the previous line:

```
for( int i=0; i<n; i++ ) {
  A[i]=0;
}
```

This produces a more compact style of code, which I happen to favor, as you've seen in previous examples. However, *warning*, I am in the minority on this. *Most* C and C++ programmers put braces on lines by themselves, and you should probably adopt that style. When in Rome, do as the Romans.

Another issue is whether to enclose single statements in braces. For example, the code above could have been written as follows:

```
for( int i=0; i<n; i++ )
  A[i]=0;
```

This eliminates the brace issue completely, and produces nice, compact code. The drawback, however, is a serious one. Suppose you decide that, just before the assignment statement, you'd like to print out the array element. So, you go add a statement:

```
for( int i=0; i<n; i++ )
  cout << A[i];
  A[i]=0;
```

Now you're in big trouble, because the assignment statement is no longer in the loop! In fact, you will be lucky in this case, because the compiler will probably complain that:

```
foo.cc:8: warning: name lookup of ‘i’ changed for new ANSI ‘for’ scoping
foo.cc:6: warning:   using obsolete binding at ‘i’
```

After thinking about that for a long while, you'll realize that, since the assignment statement is outside the loop, the `i` variable no longer exists.

You could also be saved if you have Emacs re-indent your code; you'd notice that the indentation became:

```
for( int i=0; i<n; i++ )
  cout << A[i];
A[i]=0;
```

and all would be clear.

However, many programmers prefer to avoid any risk of that occurring, and so they always use braces, whether for single statements or multiple statements. Either is acceptable.

Finally, there are annoying details, such as whether keywords can start after braces and such. That comes up with the little-used `do...while` construct. Choose one of the following, depending on what you like, and use it consistently:

```
do {
  cin >> response;
}
while ( response != 'N' );
```

```
do {
  cin >> response;
} while ( response != 'N' );
```

Personally, I prefer the one on the left, with the `while` keyword starting at the correct indentation level (and being colored correctly by Emacs), but either is acceptable.

While there is latitude in using braces, some things are simply not done. For example, an open brace is always the last thing on a line. No one would do the following:

```
for( int i=0; i<n; i++ )
  { A[i] = 0;
    B[i] = 1; }
```

That goes double for statements on a single line. I might do the following if I was putting in debugging code, but that's *only* because I would delete such code before anyone else saw it:

```
for( int i=0; i<n; i++ ) { cout << A[i]; A[i] = 0; }
```

However, that doesn't mean that proper code always starts a new line. Compact code is nice, and so single-statement blocks can be put on the same line as their containing statement:

```
for( int i=0; i<n; i++ ) A[i] = 0;
```

Contrast this with another acceptable style, which takes four lines instead of one:

```
for( int i=0; i<n; i++ )
{
   A[i] = 0;
}
```

We've already discussed the advantages of the latter, sparse style of coding, namely that it's easy to insert additional lines of code.

When functions, loops, and if statements get long, as can easily happen, it can become hard to visually match up the closing brace with the opening brace and, more importantly, the containing statement. If that is likely to happen, it is thoughtful to help the reader out, by saying what the brace is closing.

```
while (true)

   ...
   Many lines of code
   ...
 // end of while loop
```

We would probably not do this if the loop were short, say only 5–6 lines of code, because it clutters up the code and makes it harder to read. On the other hand, if the code in the loop is very long, we would probably try to introduce functions to make it more clear and compact. If that's not possible, then this marker is very important, despite the risk of visual clutter. After all, our goal in every case is to make the structure of the program easy to see.

5 Documentation

If you obey all the rules above, your program will have come a long way towards being clear and understandable. Nevertheless, many aspects of a program don't come out clearly in the code, and so we must *document* our code. What the documentation usually describes is the purpose, role, structure or technique of the code. This is because, often, the reader misses the forest for the trees: the code has all the details but none of the outline. Like an pointillist painting, we must step back from the code in order to see it clearly.

Rule 7 *Document the program as a whole. What does it do, and what are its inputs and outputs. Say who wrote it, and when.*

At the top of your program file, you should have a longish comment that explains what the program does. This needn't be long and elaborate, but it should get the reader started. Think of it as a title and abstract for the paper. For example:

```
/*  This program reads in an array of integers, with the number of
  integers specified by the user, up to a maximum of 100.  It then
  sorts those numbers and prints them in order, one per line.

Written by Scott D. Anderson
October 23, 1997
*/
```

That would be a bare minimum for this first chunk of documentation. Additional documentation would describe the structure of the program, in terms of what data are defined, what functions are called, and so forth.

Notice, by the way, that I didn't have to start every line with the double-slash comment characters, `//`, because I used C comment characters, which comment out everything from `/*` to `*/`. These are very nice for long comments, especially paragraph comments where you might want to fill the paragraph when you're done (M-q in Emacs).

Rule 8 *Document functions. Each function should be preceded by a brief paragraph explaining what it does, how it works (if necessary) and the meaning of its arguments and return values.*

Each function is, essentially, a small program, so just as you'd document the inputs, outputs and purpose of a program, you should document each function. If that paragraph starts getting long and cumbersome, the function is probably doing too much; consider breaking it into pieces.

Rule 9 *Document variables and data members. Explain the purpose and use of the data and any non-obvious aspects of its implementation.*

We have seen this issue before, when we discussed how to name a variable. Often, the name is not sufficient to explain the variable, in which case the documentation helps the reader. For now, you should probably document all but the generic variables (like `i` or `temp`). For example, here are some class definitions:

```
class Student {
public:
  char name[MaxName+1];         // The student's name
  int number;                   // The student number, used for sorting
  float gpa;                    // The student's grade point average
};

class Link {
public:
  Student *elt;                 // A pointer, to make swapping efficient
  Link *next;                   // Pointer to next list element, or null.
};
```

You could argue that none of this documentation is particularly necessary, but it can be helpful, particularly for names that are abbreviated, such as "gpa" or possibly ambiguous, such as "number."

Notice, by the way, that the documentation is nicely lined up and separate from the code, making the code easy to read. This formatting is easy to do and worthwhile. In "vi," you'd have to tab over the correct number of times, and in Emacs, the command M-; will move you to the correct column and insert the comment symbols.

Having documented the program, each function, and all the variables and data members, you might think we're done. Not quite. The documentation we've discussed so far concentrates on the high-level view: the purpose and plan of the code. Sometimes, the implementation is a little tricky, and so we document those tricky parts of the code, explaining what's going on.

Rule 10 *Document any code that is not obvious.*

Trickiness, of course, is a matter of judgment. What is complex to a beginner is obvious to an experienced programmer. For example, it wasn't that long ago that you appreciated the following comment:

```
cur = cur->next;                // Go to next list element
```

Now, that code is starting to be idiomatic to you—you see it as a single idea, like `i++`. Indeed, such code is sometimes called programming *idioms* or *clichès*. Soon, you'll be omitting documentation for code like that. But there will still be many things to document:

```
(*tail) = new Node;             // Add new node to end of list.
tail = &((*tail)->next);        // update tail ptr to addr of next in last elt
```

For now, you should document any code that isn't obvious to you. If you're not sure, err on the side of documenting, because it's good practice, and because something that is obvious while you're writing the code may not be so obvious to a reader, even to yourself after a week's time.

Rule 11 *Use proper spelling and grammar, except when brevity is more important.*

Programming style is a way of showing respect and concern for your reader: you're going to go to the extra effort to make your code clear and understandable. Not bothering to write proper English undercuts that message. It makes your reader work harder to understand you. It's more likely to be ambiguous or misleading. It makes you look uneducated, which can make your reader doubt you and your code. Finally, every once in a while, the reader will be some pedant who just becomes irrational when forced to read poor spelling and grammar.[3]

If you think about it, spelling and grammar are the analog of the struggle we go through to write a program that means and does what we want. We have to spell all the keywords, types and identifiers correctly and use the syntax of the language properly to get our program to work. We just need to put in the same effort on our documentation; less, actually, because English is both more familiar and less demanding than programming languages.

The only time when spelling and grammar rules can be broken is in end-of-line comment when abiding by the rules would cause visual clutter because the comment would have to be continued on the next line. Consider the following:

```
(*tail) = new Node;          // Add new node to end of list.
tail = &((*tail)->next);     // Update the tail pointer to be the
                             // address of the ``next'' data member
                             // in the last element of the list
```

Is this clearer than the original? Probably a little clearer. Yet it took three lines, resulting in either (1) blank lines in our program, which breaks the visual rhythm of the code or (2) putting the documentation at the end of other lines of code, which would be confusing. Neither of these is worth it. Abbreviate and telegraph as much as reasonable when using end of line comments. Of course, you can't sacrifice clarity: if the documentation isn't clear, it's not worth it. So, if the end-of-line comment just can't be done, go to an in-code block comment, as follows:

```
(*tail) = new Node;          // Add new node to end of list.

// Update the tail pointer to be the address of the ``next''
// data member in the last element of the list
tail = &((*tail)->next);
```

I sometimes like to precede such comments with a blank line, so that the reader knows to shift gears when reading the code. However, with modern font coloring by Emacs, the comment will be in a different color and so the reader will be easily able to distinguish code from comments.

Rule 12 *Remember that screens and printers have finite width. Stick to an 80-character line.*

Back in the olden days, screens and printers weren't bitmapped and there was only one, fixed-size font. Screens were exactly 80 characters wide and 24 lines long, and printers put 80 characters on a line (unless you had a line-printer, which could go to 132 characters) and 66 lines on a page. The 80 comes from the really ancient times, when programs were written in FORTRAN on 80-column punched cards.

Why should you, who are fortunate enough to live in modern times, care about keeping below 80 characters on a line? There are two related reasons. First, maybe you can select a smaller font so that more characters can fit on a screen or the page, but you and others don't really want to read a font that small. Secondly, there are psychological and physiological studies of human

[3]Unluckily for you, I'm one of those people.

reading that shows that the human eye doesn't track so well from line to line (typically in the fast movement from the end of one line to the beginning of the next) when the line is very long. This is not so much an issue with code, since the lines aren't that long, but with documentation it is. Since modern times haven't made eyes any better or paper any wider, keep your line width in documentation to 80 characters, and do the same for your code, too.

You can check your width in Emacs with C-x =.

6 Rule Summary

This summary just recapitulates all the rules.

1. Don't explain the obvious.
2. All names should be **descriptive** and **accurate**.
3. Be consistent with capitalization and punctuation of names. Decide on a scheme and stick with it.
4. Functions should do just one thing.
5. Indentation is important. Code must be properly indented to show the syntactic structure of the program.
6. A statement that belongs to (is part of) another statement should be indented relative to the containing statement. Statements that are at the same level in the syntax tree should be indented the same amount.
7. Document the program as a whole. What does it do, and what are its inputs and outputs. Say who wrote it, and when.
8. Document functions. Each function should be preceded by a brief paragraph explaining what it does, how it works (if necessary) and the meaning of its arguments and return values.
9. Document variables and data members. Explain the purpose and use of the data and any non-obvious aspects of its implementation.
10. Document any code that is not obvious.
11. Use proper spelling and grammar, except when brevity is more important.
12. Remember that screens and printers have finite width. Stick to an 80-character line.

7 Conclusion

We have looked at a great many examples, and we have seen some general rules, but our work has really only begun. We have started to try to think about code from the point of view of our reader, the person we are really trying to communicate with, rather than thinking about the compiler. But writing is hard, and rules change over time. For example, there was a time when I would be castigated for starting the sentence before this with "but," because "but" is a coordinating conjunction, and there is no clause that I was coordinating with. Fortunately, that rule is less enforced nowadays, and I felt I had a good rhetorical reason for breaking it. In other words, I was

willing to break a rule in order to strive for better communication. The rules serve us, not vice versa.

Because writing is hard, even though you've had many years of practice, you should expect that proper programming style is difficult, too. Indentation, braces, and simple stuff like that you'll pick up quickly. What takes years of practice—and we're all still learning—is trying to anticipate what the reader will and won't understand, and describing the code in ways that make it clear and accessible. Even choosing good names for variables and functions is cause for thought and consideration. If you start feeling sorry for yourself, remember that, compared to writing a good essay, documentation is easy.

Work at it, practice, and, if you're in doubt, ask your professor. We don't all have the same style, and our answers won't necessarily agree, but we do have more experience and we'll do our best to help. Becoming a good programmer includes good documentation, because even the most brilliant code becomes obsolete if no one can understand it. Finally, even if it's not your life's ambition to be a great programmer, learning to express your ideas clearly in English is important.