Developing Good Style

As we will stress all quarter, a working program is only half the challenge—constructing an elegant and well-engineered solution is your ultimate goal. Developing a good sense of style takes practice and work, conveniently we've got plenty of both coming up this quarter! Your section leader's feedback on your code will be invaluable in helping you learn how to design and organize your code to form a readable, well-engineered whole. As with any complex activity, there is no one "right" way, nor is there an easy-to-follow checklist of do's and don'ts. But that said, this handout will try to identify some qualities that can contribute to readable, maintainable programs to give you an idea what to strive for.

Commenting

The motivation for commenting comes from the fact that a program is read many more times that it is written. A program should be readable, and not just to the programmer who wrote it. A program expresses an algorithm to the computer. A program is clear or "readable" if it also does a good job of communicating the algorithm to a human. Given that C++ is a rather cryptic means of communication, an English description is often needed to understand what a program is trying to accomplish or how it was designed. Comments can provide information that is difficult or impossible to get from reading the code. Examples of information you might include in comments:

- General overview. What are the goals and requirements of this program? this function?
- Data structures. How is the data stored? How is it ordered, searched, accessed?
- Design decisions. Why was a particular data structure or algorithm chosen? What other strategies were tried and rejected?
- Error handling. How are error conditions handled? What assumptions are made? What happens if those assumptions are violated?
- Nitty-gritty code details. Comments are invaluable for explaining the inner workings of particularly complicated (often labeled "clever") paths of the code.
- Planning for the future. How might one make modifications or extensions later?
- And more... (This list is by no means exhaustive)

It is a good convention to begin each file with an overview comment for the program, interface, or implementation contained in the file. The overview is the first thing being read and usually lays out a roadmap— giving a high-level sketch of the structure, pointing out the important features, and noting the relationship of this file to the others. Essentially, the overview contains all the information that is helpful for understanding the module as a whole. The overview comment should also contain author and version information: who worked on this file and when.

Each noteworthy function deserves a comment describing the function's purpose and its inputs and outputs. You might also include information about the expected state ("this function expects the pen to be positioned at the …"). It's a good idea to mention if the function relies directly on any defined constants. Additionally, you should describe any special cases or error conditions the function handles (e.g. "...prints out an error message if divisor is 0", or "...returns the constant NOT_FOUND if the word doesn't exist").

Remember to consider the viewpoint of your readers. For comments in an interface (.h) file, you are telling the client how to use the functions. Therefore it is appropriate to describe the parameters, return value, and general behavior of a function in the interface. It is <u>not</u> the place to go into details of how the function is implemented. Such specifics on the inner workings, (algorithm choice,

calculations, data structures, etc.) should be included in the comments in the corresponding .cpp file, where your audience is a potential implementer who might extend or re-write the function.

Some programmers like to comment first, before writing any code, as it helps solidify for them what the program is going to do or how each function will be used. Others choose to comment at the end, now that all has been revealed. Some choose a combination of the two, commenting some at the beginning, some along the way, some at the end. You can decide what works best for you. But do watch that your final comments do match your final result. It's particularly unhelpful if the comment says one thing but the code does another thing. It's easy for such inconsistencies to creep in the course of developing and changing a function. Be careful to give your comments a once-over at the end to make sure they are still accurate to the final version of the program.

Overcommenting

In a tale told by my next-door-neighbor, he once took a class at an un-named east-bay university where the commenting seemed to be judged on bulk alone. In reaction, he wrote a program that would go through another program and add comments. For each function, it would add a large box of *'s surrounding a list of the parameters. Essentially the program was able to produce comments about things that could be obviously deduced from the code. The fluffy mounds of low-content comments generated by the program were eaten up by the unimaginative grader. Not so, here at Stanford...

Some of the best documentation comes from giving types, variables, functions, etc. meaningful names to begin and using straightforward and clear algorithms so the code speaks for itself. Certainly you will need comments where things get complex but don't bother writing a large number of low-content comments to explain self-evident code.

The audience for all commenting is a C++-literate programmer. Therefore you should not explain the workings of C++ or basic programming techniques. Useless overcommenting can actually decrease the readability of your code, by creating muck for a reader to wade through. For example, here are some useless comments**:**

```
int counter;                        // declare a counter variable

i++;                                // increment i

while (index < length)              // while index less than length

total = price + TAX - DISCOUNT;     // add sales tax and subtract discount
```

These comments do not give any additional information that is not apparent in the code. Save your words for important issues! Inline comments are best used to illuminate details of your implementation where the code is complex or unusual enough to warrant such explanation. A good rule of thumb is: *explain what the code accomplishes rather than repeat what the code says*. If what the code accomplishes is obvious, then don't bother.

Attributions

All code copied or derived from books, handouts, web pages, or other sources, and any assistance received from other students, section leaders, fairy godmothers, etc. must be cited. We consider this an important tenet of academic integrity. For example,

```
/* IsLeapYear is from Eric Roberts _Programming Abstractions in C_, p. 200.
 */
```

or

```
/* I received help designing this data structure, in
 * particular, the idea for storing ships in alphabetical order,
 * from Andrew Aymeloglu, a course helper in the Lair, on April 2.
 */
```

On choosing identifiers

The first step in documenting code is choosing meaningful names for things. For variables, types, and structure names the question is "What is it?" For functions, the question is "What does it do?" A well-named variable or function helps document all the code where it appears. By the way, there are approximately 230,000 words in the English Language — "temp" is only one of them, and not even a very meaningful one.

Names of constants should make it readily apparent how the constant will be used. **`Max`** is a rather vague name: maximum number of what? **`MaxScore`** gives more complete information about how the constant is used.

Avoid content-less, terse, or cryptically abbreviated variable names. Names like **`a`**, **`temp`**, or **`nh`** may be quick to type, but they're awful to read. Choose clear, descriptive labels: **`average`**, **`height`**, or **`numHospitals`**. Provide meaning where possible— if a list of doubles represent the heights of all students, don't call it **`list`**, and don't call it **`doubles`**, call it **`heights`**. There are a couple variable naming idioms that are so prevalent among programmers that they form their own exception class:

`i, j, k`	Integer loop counters
`n, len, count`	Number/count of elements
`x, y`	Cartesian coordinates

The uses of the above are so common that I don't mind their lack of content.

Function names should clearly describe their behavior. Functions which perform actions are best identified by verbs, e.g. **`FindSmallest()`** or **`DrawTriangle()`**. Predicate functions and functions which return information should be named accordingly: e.g. **`IsPrime()`**, **`NumEntries()`**, or **`AtEndOfLine()`**.

Formatting and capitalization

In the same way that you are attuned to the aesthetics of a paper, you should take care in the formatting and layout of your programs. The font should be large enough to be easily readable. Use white space to separate functions from one another. Properly indent the body of loops, if, and switch statements in order to emphasis the nested structure of the code. Most section leaders are pretty happy with code printed 2-up (e.g. two logical pages per physical sheet) in landscape orientation (as long as the font isn't too small), this saves a few trees and allows them to see more of the code at once than with traditional 1-up printing.

There are many different styles you could adopt for formatting your code (how many spaces to indent for nested constructs, whether opening curly braces are at the end of the line or on a line by themselves, etc.). Choose one that is comfortable for you and be consistent!

Likewise, for capitalization schemes, choose a strategy and stick with it. In class, we will typically capitalize each word in the name of a function, variables will be named beginning with lower case, constants will be capitalized, and so on. This allows the reader to more quickly determine which category a given identifier belongs to.

Booleans

Boolean expressions and variables seem to be prone to redundancy and awkwardness. Replace repetitive constructions with the more concise and direct alternatives. A few examples:

```
bool flag;
if (flag == true)
```

is better written as

```
if (flag)
```

```
if (count == 0)
    empty = true;
else
    empty = false;
```

is better written as

```
empty = (count == 0);
```

```
if (foundAnswer)
   return true;
else
   return false;
```

is better written as

```
return foundAnswer;
```

Constants

Avoid embedding magic numbers and string constants into the body of your code. Instead define a symbolic name for the value. This improves the readability of the code and provides for localized editing. You only need change the value in one place and all uses will refer to the newly updated value.

Constants should be independent; that is, if changing one constant requires changing another, the two should be linked in such a way that that updating is done automatically. For example,

```
const int RectWidth = 3;
const int RectHeight = 2;
const int RectPerimeter = 10;          // WARNING: potential problem here!
```

is dicey, because if you change `RectWidth` or `RectHeight`, you also have to remember to change `RectPerimeter`. A better way is:

```
const int RectPerimeter = (2*RectWidth + 2*RectHeight);
```

Code unification and factoring

When first sketching out a passage of code, you may find steps that are repeated, if not exactly, sometimes very similarly. Take the time to consider how to rearrange the code to factor those common statements and unify them. This not only cleans up and simplifies the code, it means you have just that one passage to write, test, debug, update, and comment.

Typical opportunities are to be found within cases of a nested if-else or switch statement. Here is an example of factoring out common code from an if-else:

```
if (x == 1) {
    SetPenColor("Black");
    DrawCircle();
    WaitForMouseDown();
} else if (x == 2) {
    SetPenColor("Black");
    DrawSquare();
    WaitForMouseDown();
} else {
    SetPenColor("Black");
    DrawDiamond();
    WaitForMouseDown();
}
```

is better written as

```
SetPenColor("Black");
if (x == 1)
    DrawCircle();
else if (x == 2)
    DrawSquare();
else
    DrawDiamond();
WaitForMouseDown();
```

Another opportunity for factoring occurs around loops. Below the code on the left shows a "loop-and-a-half" construction where the lines of code outside the loop are repeated again within the loop. The statements outside the loop are said to "prime" the loop. Rearranging into a while(true) with a break as shown the on the right avoids that repetition.

```
cout << "Next? ";
x = GetInteger();
while (x != Sentinel) {
    sum += x;
    cout << "Next? ";
    x = GetInteger();
}
```

is better written as

```
while (true) {
   cout << "Next ? ";
   x = GetInteger();
   if (x == Sentinel) break;
   sum += x;
 }
```

You might feel repeating a few lines isn't a big deal, but those lines might someday need modification or grow into something more complex and having just one version to update and maintain is always preferred to managing duplicate code. You want to develop good habits now when the programs are still relatively simple, so that you have the right strategy in place to enable you to successfully write more complex ones in the future.

Decomposition

Decomposition does not mean taking a completed program and then breaking up large functions into smaller ones to appease your section leader. Decomposition is the most valuable tool you have for tackling complex problems. It is much easier to design, implement, and debug small functional units in isolation that to attempt to do so with a much larger chunk of code. Remember that writing a program first and decomposing after the fact is not only difficult, but prone to producing poor results. You should decompose the *problem*, and write the program from that already decomposed framework. In other words, you decompose problems, not programs!

The decomposition should be logical and readable. A reader shouldn't need to twist her head around to follow how the program works. Sensible breakdown into modular units and good naming conventions are essential. Functions should be short and to the point.

Strive to design function that are general enough for a variety of situations and achieve specifics through use of parameters. This will help you avoid redundant functions—if the implementation of two or more functions can be sensibly unified into one general function, the result is less code to develop, comment, maintain, and debug.

Avoid repeated code. The rule of thumb here is "once and only once." Don't copy and paste when you need similar code in more than one place, unify! Even a handful of repeated lines is worth

breaking out into a helper function called in both situations. Your code can become more readable as a result, too, when you give that helper a meaningful name.

Organization

The basic outline of C++ program goes something like this:

```
Overview comment
Include statements
Constant definitions
Type definitions (enums, structs, typedefs)
Function prototypes (see note below about need for prototypes)
Function definitions
```

Most of this order is dictated by the operation of the C++ compiler, which requires that program elements are declared/defined before use, so a program first includes necessary header files and sets up the definitions that are needed by the rest of the program. However once you're ready to code up your functions, you have control over the order you present them. There should be some logic to your order; this is especially important for longer programs with many functions where you want to help the reader navigate the flow of control.

One reasonable strategy is to list the functions in top-down order: start with **`main`**, then the functions **`main`** directly calls, followed by the functions called by those functions, and so on. The smallest building blocks that don't break down further will appear last in the file. An alternate strategy is the bottom-up: first listing the lowest level functions and working up to **`main`**, which appears last in the file.

Note that if you use a top-down listing you will have to give function prototypes in advance, since C++ requires that a function declaration or definition be seen before calls to that function. In the bottom-up version, the function definitions occur before the code that uses them, so no separate prototypes are necessary. I find program flow slightly easier to follow when the program is organized top-down, but bottom-up has the convenience of not maintaining separate prototypes. Either strategy is fine with us, but please be logical and consistent within one program.

Summary

Although this handout is not the final word on every style issue, hopefully it gives you a feel for the philosophy we are espousing. Interactive grading is your chance to receive one-on-one feedback from your section leader, ask questions, and learn about areas for improvement. Don't miss out on this opportunity!

"Real programmers don't document. If it was hard to write, it should be hard to understand." —Philington's Second Law