CS106X Handout #9
Spring 99-00 April 3, 2000

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 in the course! Your section leader's feedback on your code will be invaluable in helping you learn how to design and organize code in a clear, readable manner that works for you. As with any complex activity, there is no one "right" way that can easily described, 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 we're working towards.

Commenting

The motivation for commenting comes from the fact that a program is read many more times that it is written. A program must strive to 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 data is 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 labelled "clever") paths of the code.
- Planning for future. How might one might make modifications or extensions later?
- And much more... (This list is by no means exhaustive)

At the top of each file, it is a good convention to begin with an overview comment for the program, interface, or implementation contained in the file. The overview is the single most important comment in a program. It's the first thing that anyone reading your code will see. The overview comment explains, in general terms, what strategy the program uses to produce its output. The program header should lay out a roadmap of how the algorithm works— pointing out the important routines and discussing the data structures. The overview should mention the role of any other files or modules that the program depends on. Essentially, the overview contains all the information which is not specific or low-level enough to be in a function comment, but which is helpful for understanding the program as a whole.

Each noteworthy function is often preceded by a comment that contains the function's purpose, a description of the parameters, and details of the function's return value, if any. You might also include information about the expected state ("this function expects the pen to be positioned.."). It's a good idea to mention if the function relies directly on any #define-d 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 not 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 .c file, where your audience is a potential implementor who might extend or re-write the function.

Some programmers like to comment first, before writing any code, as it helps solidify for themselves 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 Pascal program that would go through a Pascal 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...

The best commenting comes from giving types, variables, functions, etc. meaningful names to begin with so the code where they appear doesn't need comments. Add in a few comments where things still need to be explained and you're done. This is far preferable to a large number of low-content comments.

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, the comments

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

i = i + 1;                       /* add 1 to i */

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

num = num + 3 - (num % 3);  /* add 3 & subtract num mod 3 */
```

do not give any additional information that is not apparent in the code. Save your breath for important higher-level comments! Only illuminate low-level 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 from books, handouts 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 taken from Eric Roberts text,
 * _The Art and Science of C_, p. 200.
 */
```

or

```
/* I received help designing this data structure, in
 * particular, the idea for storing ships in alphabetical order,
 * from Andrew Toy, a course helper, on Tuesday, Apr. 11, 1999.
 */
```

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 #define-d constants should make it readily apparent how the constant will be used. **`MaxNumber`** is a rather vague name: maximum number of what? **`MaxNumberOfStudents`** is better, because it gives more 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`**. In general, I prefer identifiers that mean something. If a variable contains a list of doules that 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, length`	Integer number of elements in some sort of aggregation
`x, y`	Cartesian coordinates. May be integer or real.

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 about a property of an object should be named accordingly: e.g. **`IsPrime()`**, **`StringLength()`**, 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.

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 <u>be consistent</u>!

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, #define constants will be capitalized, and so on. This allows a 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:

```
if (flag == TRUE)         is better written as          if (flag)
```

```
if (matches > 0)          is better written as          found = (matches > 0);
    found = TRUE;
else
    found = FALSE;
```

```
if (hadError)             is better written as           return hadError;
   return TRUE;
else
   return FALSE;
```

Constants

Avoid embedding magic numbers and string constants into the body of your code. Instead you should #define a symbolic name to represent 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.

#define-d constants should be independent; that is, you should only need to change one #define to change something about a program. For example,

```
#define RectWidth 3
#define RectHeight 2          /* WARNING: problem */
#define RectPerimeter 10
```

is not so hot, because if you wanted to change **`RectWidth`** or **`RectHeight`**, you would also have to remember to change **`RectPerimeter`**. A better way is:

```
#define RectWidth 3
#define RectHeight 2
#define RectPerimeter (2*RectWidth + 2*RectHeight)
```

Decomposition

Decomposition does not mean taking a completed program and then breaking up large functions into smaller ones merely 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 are aiming to <u>decompose problems, not programs</u>!

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—sometimes the implementation of two or more functions can be sensibly unified into one general function, resulting in less code to develop, comment, maintain, and debug. Avoid repeated code. Even a handful of lines repeated is worth breaking out into a helper function called in both situations.

Decomposition is important and deep enough to deserve its own handout, so refer to our decomposition handout for more lots of good advice, courtesy of my next-door neighbor.

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!

When cryptography is outlawed, zbcq zetcxnf nrcc jxym lsryxcq.