CS107 Handout 12
Spring 2002 April 8, 2002

Achieving Readability

This wonderful handout was lifted from Nick Parlante's CS107 handout stream.

A program is undoubtedly read many more times that it is written. A program must strive to be readable, and not just to the programmer who wrote it. Any program expresses an algorithm to the computer. A program is clear or "readable" if it also does a good job of communicating that algorithm to a human.

Readability is vital for projects involving more than one person. It's also important when the program is of sufficient size that you come across pieces of code you wrote, but which you don't remember. It's a traumatic experience the first time it happens. A bug is essentially a piece of code which does not say what the programmer intended. So readable code is easier to debug since the discrepancy between the intention and the code is easier to spot.

Documentation

Given that most programming languages are 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 which is difficult or impossible to get from reading the code. Some examples of information worth documenting:

- 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? Were 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 future. How might one might make modifications or extensions later?
- And much more... (This list is by no means exhaustive)

In a tale told by my office-mate, 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 which 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 which 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.

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.

Overview Comments

Every program, module, or class should begin with an overview comment. The overview is the single most important comment in a program. It's the first thing that anyone reading your code will read. 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 which the program depends on. Essentially, the overview contains all the information which is not specific or low-level enough to be in a function or method comment, but which is helpful for understanding the module as a whole.

In the latter paragraphs of the overview, you might include the engineering rational for the algorithm chosen, or discuss alternate approaches which might be better. The overview can also introduce the programmer's opinions or suggestions. It's often interesting to see the programmer's feelings on which parts of the program were the hardest or most interesting, or which parts most need to be improved.

For coursework, the overview should also include uninteresting but vital information like: your name, what class the program is for, your section, and when the program is being handed in. In commercial code, the overview will also list, most recent first, all the revisions made to the code with author and date.

Choosing Good Identifiers

The first step in documenting code is choosing meaningful names for things. This is potentially the last step, since code with good identifiers will need little additional commenting. For variables, types, and record field 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.

Common Idioms for Variables

There are a couple variable naming idioms that are so common among programmers, that even short names are meaningful, since they have been seen so often.

`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.
`head, current, tail`	Pointers used to iterate over lists.

Nouns for Variables and Types

The uses of the above are so common, that I don't mind their lack of content. However, in all other cases, I prefer identifiers that mean something. Avoid content-less, terse, or cryptically abbreviated variable names. Names like "`a`", "`temp`", "`nh`" may be quick to type, but they're awful to read. Choose clear, descriptive labels: "`average`", "`height`", or "`numHospitals`". If a variable contains a list of floats which represent the heights of all the students, don't call it `list`, and don't call it `floats`, call it `heights`. Plurals are good for variables which contain many things. This applies to names for structure members as well as variables.

Don't reiterate the data structure being used. e.g. `list`, `table`, `array`.

Don't reiterate the types involved if you know more specifically what the value is. e.g. `number`, `string`, `floatValue`, anything containing the word `value` or `temp`.

Do say what value is being stored. Use the most specific noun which is still accurate. e.g. `height`, `pixelCount`, `names`. If you have a collection of floating point numbers, but you don't know what they represent, then something less specific like `floats` is ok.

Defining Constants and Macros

Avoid embedding magic numbers and string literals 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.

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. You also may want to choose a capitalization scheme that identifies macros as such (some programmers use all upper case e.g. `BOARD_WIDTH`).

Verbs for Function Names

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`, `AtEndOfLine`. Again, you may want to choose a capitalization scheme that helps make your function names readable and consistent (perhaps capitalizing each new word or separating words with underbars).

Comments for Functions

The comments for a function or method need to address two different things:

1) What does it do?	(Abstraction comments)
2) How does it do it?	(Implementation comments)

The abstraction is of interest for someone who wants to use it. The implementation is of interest to someone trying to modify or debug it. A function with a well-chosen name and well-named parameters may not need any abstraction documentation. A routine where the implementation is very simple may not need any implementation documentation.

Abstraction Comments for Functions

Usually the comments in an interface (.h) file are abstraction comments: you are telling the client how to use the functions. Abstraction documentation is like an owner's manual for the function— what the function does and what it can and cannot tolerate as input. An easy way to come up with a good abstraction comment is to look at the parameters of the routine, and then explain what the routine does to them. You should describe any special cases or error conditions the function handles (e.g. "...will abort if divisor is 0", or "...returns the constant NOT_FOUND if the word doesn't exist") It is not necessary or appropriate to go into the gory details of how the function is implemented.

Implementation Comments for Functions

Specifics on the inner workings of a function (algorithm choice, calculations, data structures, etc.) should be included in the implementation comments in the corresponding .c file, where your audience is a potential implementor who might extend or re-write the function. Implementation commenting is part is the traditional programmer-to-programmer documentation which describes how the code implements the abstraction. Some programmers make a point of not letting any implementation-oriented comments make their way into the .h files where they might be seen by a client.

The following examples are meant to illustrate function documentation. They are not necessarily a representation of how much you should comment every function. The amount of commenting a function requires is related to its complexity and importance. Some programs break down so nicely that no one function is very complex, and the names of the functions document most of what's going on. Other programs have a fundamental complexity which emerges in a few key functions. These functions deserve a lot more commenting. Some people like to label the "abstraction" and "implementation" documentation and write them as separate paragraphs. Some don't use the labels and document the routine in a single paragraph which addresses both. Some people prefer to defer more of the implementation discussion to inline comments.

Please use whatever style you are most comfortable with. Any reasonable, consistent approach is acceptable.

```
void SortedListInsert(List list, Element elem);
```

No abstraction comment is really required— what the function does is apparent from its name, assuming the function does not make any hidden assumptions about the list parameter beyond those given in the program header about the use of the type `List`.

```
   /*
    * Implementation:  This function recurs down LIST to find the right
    * place to insert ELEM into increasing order.  When the correct spot is
    * found, a new LISTELEMENT is allocated, initialized, and inserted.  The
    * recursion has the general flow:
    *     1) base: if LIST is empty, then insert at head
    *     2) check: if ELEM is less or equal to the first list element, then
    *        insert at head
    *     3) recur: otherwise recur on the rest of the list
    * A while loop could be used for a little more efficiency, but it's messier.
    */
```

The comment is a general description of the flow and purpose of the body of code. The comment takes advantage of the natural breakdown provided by the three cases of the recursion. It doesn't get into the detail of individual lines— instead it outlines the general flow. Almost all routines have some sort of natural decomposition into "cases" or "phases" which can be a good starting point for the documentation. This might be a bit much documentation for a program where you would expect the reader to know how to do something as common as a recursive linked list insertion.

```
   Relationship FindFirstRelationship(FamilyTree tree, Person male, Person female);

   /*
    * Abstraction:  This function finds the first relationship between MALE and
    * FEMALE who are assumed to be distinct PERSONs, both of whom are in TREE
    * somewhere.  The "first relationship" is defined to be the relationship
    * defined by the common ancestor lowest in the tree.  The "lowest" part of
    * the restriction is important in the case that MALE and FEMALE are related
    * in several different ways.
    *
    * Implementation: This function first uses the function SETOFANCESTORS to
    * return all ancestors of MALE and EVE.  The Set Module function INTERSET
    * is used to find the set of common ancestors.  A single pass of this locates
    * the person who is lowest in the tree.  This is easy since each PERSON rec
    * contains that person's level in the tree.  The AncestorToRelationship fn
    * is then used to compute the RELATIONSHIP.  An alternate approach might be
    * to compute and compare MALE and FEMALE 's ancestors breadth first, one
    * level at a time, so as to find the lowest common ancestor more quickly and
    * potentially without having to look at most of the tree.  The disadvantage
    * is that the cost at each level is much higher even if the number of levels
    * seen is less.  Such an approach might make sense if MALE and FEMALE were
    * going to be closely related most of time, or if the tree were very large.
    */
```

The abstraction comment makes the assumptions about the input explicit, and gives a good definition of the output. The implementation comment outlines the three logical steps of the code. Each step may require all sorts of messy pointer manipulation and special case testing which the comment does not get into. The comment sticks to the point of what is going on at each step. It's fine to refer to other routines in the program in the explanation. The digression into the alternate approach is reasonable because in the case of this program, this function is the hardest, most important part of the whole thing. From an engineering standpoint it is unclear which algorithm is better. The comment shares what the programmer has thought of so far, which is sure to be useful for the next programmer who has to come through and improve/repair the code. Such a discussion of alternate approaches might reasonably occur in the program header instead.

No Useless Comments!

A useless comment is worse than no comment at all— it still takes time to read and distracts from the code without adding information. Remember that the audience for all commenting is a literate programmer. Therefore you should not explain the workings of the language or basic programming techniques. Useless overcommenting can actually <u>decrease</u> 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 the index is less than the length */
num = num + 3 - (num % 3);      /* add 3 to num and 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.

Inline Comments

Most of the rest of your comments will be "inline" comments. An inline comment explains the function of some nearby code. The golden rule for inline comments is: do not repeat what the code says. Code is a great vehicle for unambiguous, detail-oriented information. Comments should fill in the broader sort of information that code does not communicate.

If your identifiers are good, most lines will require no inline comments. An inline comment is appropriate if the code is complex enough that a comment could explain what is going on better than the code itself. Of the code snippet in the previous section, only the last is complex enough that its function is not completely obvious after a single reading. Complexity is probably the simplest reason a line might deserve a comment. A line may also deserve a comment if it's important, unintuitive, dangerous, or just interesting. Here's a more useful comment to replace the one from above:

```
num = num + 3 - (num % 3); /* increment num to the next multiple of 3 */
```

Another useful role for inline comments is to narrate the flow of a routine. An inline comment might explain the role of a piece of code in terms of the overall strategy of the routine. Inline comments can introduce a logical block in the code. Begin-End blocks and the beginnings of loops are good spots for this sort of comment. As above, it's most useful to describe what is accomplished by the code.

```
    /*
     * The following while loop locates the first vowel to occur
     * twice in succession in the array
     */
```

Another useful type of comment will assert what must be true at certain point.

```
   /*
    * The file pointer must now be at the left hand side of a parenthesized
    * expression.
    */
```

or

```
   /*
    * Because of the exit condition of the above loop, at least one of
    * the child fields must be NULL at this point.
    */
```

Such a condition is called an "invariant". Invariants are a useful sort of mental checkpoint to put in your code. You'll be less likely to get loop conditions, etc. wrong if you think about and put in invariants as you are writing. One way to put invariants in your code which help debugging *and* help documentation is to sprinkle your code with assert statements. Asserts are an excellent habit.

```
   assert(filePointer != NULL);
```

or

```
   assert((child1 != NULL) || (child2 != NULL));
```

Try not to allow inline comments to interfere visually with the code. Separate inline comments from the code with whitespace. Either set them off to the right, or put them on their own lines. In either case, it's visually helpful to align the left hand sides of the comments in a region. Alternately, some of the issues addressed in inline comments can be treated just as well in the implementation section of the function's comment. Whether you prefer inline comments or header implementation comments is a matter of personal choice.

Commenting Accuracy

Comments should correctly match the code; 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.

Attributions

All code copied from books, handouts or other sources, and any assistance received from other students, TAs, fairy godmothers, etc. must be cited. We consider this an important tenet of academic integrity, and as well it serves as useful information for the next person to come along and work with this code. For example,

```
/*
 * Predicate Function: IsLeapYear
 * -----------------------------
 * IsLeapYear is adapted from Eric Roberts text,
 * "The Art and Science of C", p. 200.
 */
```

or

```
/*
 * I received help designing the Battleship data structure, in particular,
 * the idea for storing the ships in alphabetical order, from TA Albert Lin
 */
```

Formatting, Capitalization, and White Space

One last little note. 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 be consistent!

Likewise, for capitalization schemes, choose a strategy and stick with it. I tend to capitalize each word in the name of a function, start variables with lower case, completely uppercase #define constants, and so on. This allows a reader to more quickly determine which category a given identifier belongs to.