UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Spring 1998

General Guidelines for Programming Projects

The programming projects constitute the most important, time-consuming, strenuous, frustrating, and fun part of any programming course. Unfortunately, they are also the most time-consuming and difficult coursework to grade. The criteria for a program being good are not easily defined.

Good programs must, of course, work. Even such an obvious statement is not as simple as it seems. For example, a program that works most of the time can be extremely useful, even though there are some inputs for which it fails. In such cases, it is naturally desirable that the program detect its failure and avoid disastrous actions, and that the documentation of the program indicate its limitations. A program may work correctly for all legal inputs, but fail on illegal inputs. True, that's not the program's fault, but a program designed under the assumption that input will be perfect reflects rather badly on its implementor's grasp of reality.

Furthermore, our program descriptions (as in real life) tend to be incomplete or ambiguous. There are some details of a program that are either unimportant to pin down, or that require some design to get right. For example, the handling of erroneous or pathological input is generally not part of a project description. It is *assumed* that the programmer will "do something reasonable." So it will be in this course; you will be expected to recognize places where you must fill in details of the design, to fill them in some reasonable fashion, and to document your decisions, where appropriate.

Beyond mere functional correctness, good programming demands some attention to the process of programming itself and to stylistic qualities of the program produced. Programs tend to last longer than we originally intend. As a result, others (or even we, years later) must be able to read, understand, and modify them. Readability in programs, as in books, requires clear and logical organization and clear, straightforward expression. Errors in well-organized programs are easy to find, since it tends to be clear where the error must be.

Finally, the current state of the art does not allow us to forgo testing our programs. While testing, in Dijkstra's oft-quoted phrase, "can show the presence of bugs, but not their absence," it is the usual method of increasing our confidence that our programs, though probably not perfect, are fit for their intended purposes. Too often in programming courses, we spend much too little time on this subject. I venture to guess that most of you have been content to run a few examples through your programs—say, the examples given in the problem description—and then turn them in as quickly as possible, lest continued testing uncover another ten hours' work. In real life, something more systematic and thorough is called for.

Programming style often becomes a "religious" issue amongst practitioners. These particular religious beliefs, however, are not among those protected by the Constitution, and I will accordingly feel free to lay down some law in the rest of this document.

1 Design

In CS 61B, you will not be called upon to do real program specification from the bottom up, but there will always be minor gaps and ambiguities in the project descriptions, and you will be expected to resolve them appropriately. Where needed, you will also be expected to document what you've done (see section 4).

It is difficult to say anything general about what "appropriate resolution" is, but a few points are clear. It is appropriate that programs detect erroneous input; make some attempt at isolating and reporting the problem; and terminate or recover gracefully (i.e., by some means that, unlike a segmentation fault or a Java stack dump, gives the impression of being planned). It is appropriate that program inputs and outputs be non-cryptic. It is appropriate that programs fail gracefully when they run into implementation limits. For example, an appropriate response to running out of memory might be to print a message to that effect, remove temporary files, and terminate cleanly. If your program fails to check for this situation, you will probably end up with an abrupt and enigmatic exception or (in C and C++) segmentation fault.

2 Modules, Headers, Comments

I am not a big fan of *internal* comments, such as this sort of thing.

```
if (qr > z + 1) {             // Check for end-of-buffer
    puts(buf);                // ... and start new line if needed.
    qr = 0;
}
```

I would much rather see subprograms (procedures or functions) chosen and named to make the source code self-documenting:

```
if (endOfBuffer())
    beginNewLine();
```

Each subprogram, in turn, can have a single comment at the beginning (either just before the subprogram, or just before the opening '{' of the body) that explains what it does. Indeed, a sufficiently small procedure with a self-documenting name may need no comment (for example, `endOfBuffer` may well not need a comment, but `beginNewLine` probably does). The test of ideal commenting on subprograms is that a programmer browsing through a particular piece of code can tell what it does by consulting only the *comments* and headers of the subprograms it calls and the comments and declarations of the variables it references.

These comments on subprograms should not generally mention just how a subprogram does what it does. As the previous discussion suggests, the body of the subprogram should suffice for that. When the subprogram gets too complicated for this to be feasible, it's probably time to break it up. Occasionally, the algorithm employed will be sufficiently subtle that you'll want to say something about it. In this case, use a separate comment (possibly at the beginning of the body of the subprogram) to describe the algorithm.

All this assumes that you choose reasonable things for subprograms to do. A beginning programmer tends to bundle together the strangest sorts of activities into a single subprogram. If you had to write a comment describing one of these, it might read like this.

```
/** Outputs characters pos[k] through pos[k+1] of S; increments
 *  count[n] by pos[k+1]-pos[k]; reads the next line into
 *  S, starting at pos[m], unless stopRead is true; and sets
 *  stopRead to false. */
int doIt(int n, char[] S)
```

This can't be described by a simple crisp phrase, because it doesn't correspond to a single, easily described action. It was not a good idea to bundle this particular set of actions together; it is symptomatic of a disorganized program. Of course, programmers that do not bother to comment their subprograms never have to face up to this chaos, which no doubt explains much.

A maxim due to Tom Duff states, "Whenever possible, steal code." This is good advice (although we do insist that such thefts be properly credited), but it requires a certain kind of programming to work. In particular, subprograms are more easily re-used in other, unrelated programs if they are written to fulfill a single, clear purpose.

I recommend writing any comments *before* writing the code of a subprogram. If you have to change the subprogram later, be sure the comments are changed first[1]. This approach has the advantage that you actually get to think a little before starting to write a subprogram, and you aren't stuck at the last minute having to "add the comments."

Related groups of declarations of subprograms, global variables, and (in C or C++) macros and types should be gathered together, either in sections of a file (separated by the formfeed character, entered in Emacs as `C-q C-l`), or in separate files. Just to have a name, we'll call these sections or files *modules*. Any type, macro, variable, or subprogram defined within a module that is used outside the module is said to be *exported* from that module.

In C and C++, any module big enough to have its own file should have a header (`.h`) file (or a section of a header file) that defines all the exported items of the module, and will be included (via the `#include` directive) in any other source file that uses the exported items. The header file should contain a complete prototype for any exported subprogram, including a comment, as in this example:

```
/** Return the  next record from standard input, advancing the input.
 *  Illegal if endOfData() is true. */
extern DataRecord nextRecord(void);
```

[1]One otherwise-useful book about programming quotes some guy in Iowa as saying, "Don't get suckered in by the comments—they can be misleading. Debug only the code." Wrong. Debug *both* the code *and* the comments.

Repeat the comment in the implementation of the function *and keep the two comments consistent.* I know: this differs from advice other people give about how you shouldn't do that because the comments get out of sync. However, when I am using a function that is exported from elsewhere, I shouldn't be looking at the code. At the same time, when I am trying to debug the code, it's useful to have the comment in front of me.

It is useful to put a long comment at the beginning of a module, giving a summary of the module, and describing its data structures, if any. Global variables (or groups of them) should carry comments describing the interpretation of their contents and their relationships, as in the following examples.

```
/** Number of bins. */
int NB;

/* The data in bin i, 0<=i<NB, comprises elements
/* binStart[i] through binStart[i+1]-1 of the array data. */

int binStart[MAXBINS+1];
int data[MAXDATA];
```

3 Describing a Function or Macro

The idea in commenting a function or macro is that, with the comment, it should be unnecessary to read the body to figure out how to use the function or macro, or what it does. The information needed includes the types and necessary properties of all arguments, assumptions made about global variables, the results produced, and all side-effects on global variables (a file is a kind of global variable). For a function, the number and types of most arguments are supplied by the header, and I see no reason to repeat that information in an explanatory comment. There are numerous ways of organizing the rest of the information.

Many projects adopt a stylized format, as in the example in Figure 1. The skeleton of the comment should be kept in a file, so that you don't have to re-type it each time, but can simply insert it and fill in blanks. Such formats sometimes include extra clerical information such as the functions that call or are called by the one being described. I am not in favor of including such information, unless it is supplied automatically; it too easily gets out of date, and is not needed for understanding the function.

Figure 2 illustrates a less-stylized, more compact style. Here, the information is formatted in paragraph form. This allows a freer style, but it lacks the convenient reminder supplied by the titles of the stylized format of what information you must supply.

Java has a special kind of comment, known as a *documentation comment,* or simply *doc comments.* Actually, I've been using them all along, since they happen to be a subset of standard C and C++ comments. A doc comment is a comment that begins with `/**`. The `javadoc` tool converts the doc comments in a file into a set of HTML files, suitably cross-indexed for Web browsing. Certain tokens in a doc comment have special meaning to `javadoc`, enabling a stylized commenting style such as those illustrated previously. Figure 3 gives an example. See also Chapter 11 of *The Java Programming Language,* by Arnold and Gosling, (second edition).

```
/**
 * ----------------------------------------------------------------------
 *
 * -- schedProcess(F, t)
 *
 * Requires:
 *       t > 0, process queue not full.
 *
 * Returns:
 *       A handle allowing later adjustments to F's schedule.
 *
 * Side effects:
 *       F is scheduled to be executed in t time units.
 *
 * ----------------------------------------------------------------------
 */

SchedHandle schedProcess(int (*F)(void), TimeType t)
{
   ...
}
```

Figure 1: Example of a stylized comment for a function in C or C++.

```
/** DEBUG(L, S) executes statement S if the debugging level is at least
 *  L (an integer).  */

#define DEBUG(L, S) if ((L) > DEBUGGING_LEVEL) { S; }


/** Assumes t > 0 and the process queue is not full.  Causes F to
 *  be scheduled in t time units, and returns a handle allowing later
 *  adjustments to this schedule. */
SchedHandle schedProcess(int (*F)(void), TimeType t)
{
   ...
}
```

Figure 2: Examples of a compact, paragraph-style comment for functions and macros.

```
/**
 *  Schedule a process to run in a given number of time units.
 *  Requires that the scheduled time be positive and that the process
 *  queue not be full.
 *  @param F  The process to be scheduled.
 *  @param t  The number of time units in the future at which to
 *            which to schedule F.
 *  @return   A handle allowing later adjustments to F's schedule.
 *  @exception IllegalArgumentException
 *            Argument t is negative.
 *  @exception IllegalAccessException
 *            Process queue is full.
 */
SchedHandle schedProcess(Runnable F, long t) ...
```

Figure 3: An example of a documentation comment in Java.

4 Documentation

A program has at least two audiences: users and implementors. These correspond to two types of documentation: external and internal. The comments we've been describing are part of the *internal* documentation of a program, which is concerned with explaining how a program does what it does. The user generally has no interest in such things, and wants to know the *externals* of the program—what it's for, how to use it, what its limitations are. We will expect both kinds for each of your projects.

It is a very good idea to write the external documentation first, updating it as soon as you find that you have to change the effects of your program. We won't worry about elegance and formatting; a simple and thorough users' manual will suffice. You may also use a UNIX `man` page format. We won't go into great detail as to how that is done. The easiest way is to steal a `man` page from one of our demonstration programs and substitute your own documentation.

Besides the comments, internal documentation may require some sort of a logic manual to provide a roadmap to the entire program. This may either be a large comment in the major source file, or a separate document. For examples, see the directory `$master/demo`.

If you are really ambitious and have more time than this course will allow you, you can produce a texinfo document that can be turned into a paper manual or used by the Emacs documentation browser (`C-h i`). This can be used both for internal or external documentation. The texinfo format is documented in the documentation browser.

5 Indentation Conventions

One of the more contentious and, of course, least important stylistic issues is the indentation of source code. Code indented differently from one's standard practice is often extraordinarily annoying to read at first. To avoid having the staff go about in a state of perpetual extraordinary annoyance, I am going to require a set of conventions that is supported by the Emacs text editor, as set up for this course. These are illustrated by the skeletons in Figure 4. Before handing in any program file, first adjust its indentation by loading it into Emacs (with our standard class Emacs set-up) and indent the whole thing with the command sequence `C-h M-C-\` (mark-whole-buffer followed by indent-region). Be sure to use the indentation features of Emacs while you are entering or changing your program, too. Not only does it help you (and us) read it, but it also catches certain bracketing errors that would otherwise be rather obscure.

6 Testing

Any project you turn in should be well tested and there should be evidence of the testing in the directory turned in (see section 7). One can divide tests into two categories: "black box" testing, which tries to test that the program conforms to its users' manual and which uses no information about its internal structure, and "white box" testing, which in effect uses the internals manual and is designed to exercise each piece of the actual software. Both are important. I suggest that you consider writing some black box tests before writing the code, just so that you'll have them around. It is also important to be systematic in testing. If, for example, you are writing a program that reads and executes commands, write little tests designed to test each individual command or each critical piece of the program, not just huge, ad hoc conglomerations of random commands. Naturally, you must test large combinations of commands, too, in this case. The following quotation about writing test programs for a document compiler from an article by Donald Knuth conveys some of the right frame of mind[2]

> . . . I generally get best results by writing a test program that no sane user would ever think of writing. My test programs are intended to break the system, to push it to its extreme limits, to pile complication on complication, in ways that the system programmer never consciously anticipated. To prepare such test data, I get into the meanest, nastiest frame of mind that I can manage, and I write the nastiest code I can think of; then I turn around and embed that in even nastier constructions that are almost obscene. The resulting test program is so crazy that I could not possibly explain to anybody else what it is supposed to do. . ..

For large programs, you may want to do some extra, throw-away programming to provide *test scaffolding*—essentially a driver program that exercises just the part you are testing, and uses some quick-and-dirty way to get its own data.

The standard C header file `assert.h` provides a macro `assert`. When this file is included in a source file, and the file is compiled without the option `-DNDEBUG`, each call `assert(`C`)`, where C

[2] D. E. Knuth, "The Errors of TEX," *Software Practice & Experience*, 19 (7) (July, 1989), pp. 625–626.

```
                                /* TITLES GO HERE (with M-;) */

/* For general commentary about a section of code, use this format,
 * indented to the current margin. */

/**
 * Documentation comments describing a function or class may be formatted
 * like this. This same format works for variables.
 */

class Sample {

  /* Various general commentary about the internals of a class might
   * go here. */

  /** Comments on variables may go here */
  int x;

  /** Documentation comment for a function.  Let's just keep going and
   *  see what happens when we get to the next line of the input now
   that there
  */

  int f(double A[], double B[], double C[], int N)
    {
      if (N < 0) {
        fprintf(stderr, "Erroneous input to f: %d.\n", N);
        exit(1);
      } else if (N == 0)
        return 0;               // I really don't care if single
                                // statements have { }'s around them.
      else {
        int i;

        for (i = 0; i < N; i += 1) {
          ...
        }
      }

    StatementLabel:
      switch (...) {
      case 1:
        x = n;
        break;
      case 2: {
        int local_variable;
        ...
          }
      default:
        ...
                }

    }
}
```

Figure 4: Official indentation conventions (Spring 1998).

is any boolean expression, will terminate the program with an error message if C turns out to be false at that point. This is one useful way of inserting checks for internal consistency and has the advantages of providing internal documentation and of being easily turned off (just use the `-DNDEBUG` option). We will provide a similar facility for Java.

I have sometimes seen programs turned in with debugging output still present. Don't do this. On the other hand, one could go to the other extreme and turn in debugging output that was allegedly produced by some program, but turn in the source code with all debugging statements removed. Don't do this either. There are instead numerous useful techniques.

When you merely want to make sure that your program reaches certain points, or want to look at the values of a few variables, don't bother inserting statements into the code; it's slow and error-prone. Instead, use a debugger. If there are a large number of points you want to watch, most debuggers (including `gdb` and `gjdb`) allow you to read in command files that set breakpoints for taking any desired actions at any specified points in a program. In the case of `gdb`, these actions can even include calling debugging functions that you have defined for dumping large intermediate data structures (a very good idea, by the way).

If there are debugging output statements that you think ought to be a permanent part in your program (or whose results you think ought to be turned in), make sure they are optional. That is, make sure that your program can be compiled without the output (in the "released" version of the program) or with it, or provide a command-line option for turning the debugging information on (by default, of course, there should be no such output), or both[3]. In Java, these "command-line arguments" we've been talking about appear in the array of Strings that is passed to your main function. In C and C++, they appear as an array of `char*` values, together with a count.

Here is one technique for C and C++ programs. Wrap all debugging code in a macro called, e.g., `DEBUG`. In your program, you will have statements such as this.

```
DEBUG(3, dumpSomeStuff(...) );
```

The second argument is the statement to be executed. The first indicates the priority of this particular output. At the top of the file, or in a header, there will be a definition for `DEBUG`, such as this.

```
#ifndef NDEBUG
#define DEBUG(priority, stmt) \
    if (DEBUGGING_LEVEL <= priority) { stmt; }
extern int DEBUGGING_LEVEL;
#else
#define DEBUG(priority, stmt) /* DO NOTHING */ ;
#endif
```

Define the variable `DEBUGGING_LEVEL` in the main routine, and arrange for it to be set by an option to indicate selectively which debugging statements are to fire.

[3] I feel obliged to add a cautionary note from recent, well-publicized experience. The famous ARPANET virus created by Robert Morris used a debugging feature of the UNIX mail program as one way of breaking in. This feature should have been turned off in the released version, but wasn't.

In general, we will not want to see special debugging output from your programs, although tasteful use of techniques such as those above will count in your favor. We will, of course, be looking for test data. Here, there is an important consideration that applies also to special debugging output: errors in huge volumes of output are difficult to find. If your volumes of test data produce volumes of output, find some reliable way of validating that output. For example, if you have a program that is supposed to sort a file of names, and you have a large list of names as a test, write another program that checks the output to make sure it is in order (you will also have to figure out a way to make sure no names are dropped). The output of this checking program is the interesting part. Naturally, one has to be convinced that the checking program itself is correct. It is particularly good for it to be short, for example, and since speed is not important, it can be written using whatever language and algorithm allows the simplest program. If you are not familiar with the `diff` utility, find out about it, and see if you can think of ways it might be useful.

I will not insist on originality of test data. Feel free to trade it at will. Naturally, if we find that the entire class has relied on a single test suite, we will feel free to test everyone's program using cases that this public suite doesn't cover. We are nothing if not devious.

Bundle up your test suite into shell scripts and makefiles, so that you don't have to enter all the necessary commands by hand each time. Provide clear instructions in your turned-in directory for running the tests.

7 Mechanics and Grading

All source files, makefiles, documentation, test data, and sample test output for a project should go into a single directory in the home directory of one of the authors. There should be a makefile named `Makefile` in this directory written so that the command `gmake` compiles your program and `gmake test` runs your test cases. Because the project directory will be moved elsewhere, be very sure that your makefile does not use any special aliases or commands specific to you. Also, make sure that all file names referenced in the makefile will work when the directory is moved (ask your TA or a lab assistant if you don't understand this point). There must be a file named AUTHORS, containing the names and logins of all partners.

Grading will take into account the quality of your internal and external documentation, the quality of testing evidenced by the files you provide, the organization and general stylistic cleanliness of your program, and, of course, the correctness of your program.

Your program will not be graded if it does not compile successfully, if it is not properly indented[4], or will not execute a minimal test suite without blowing up. This test suite will be available while you are working on the program. If your program fails this minimal test, we will return it to you and allow you to make a few corrections (at the cost of several points). During grading, we will also use other test suites, of course.

Once you have assembled your directory, the comment

`submit` N

where N is the number of the project, will submit it.

[4] I know: that sounds fussy. Long experience has convinced me that I am not going to convince people to use indentation any other way.