Lecture #1: Introduction

Reading: Holmes, Chapters 1–3, 6–8

Handouts: Course Information; Syllabus; All About Homework; Program Standards; CSIF Accounts; Reading USENET News with *tin*; Sample Basic C Programs; Pascal and C

Homework: Assignment #1, due Tuesday, October 7, at 6PM

---

1. Greetings and felicitations
   - Goals: learn about software development; as will study this in a UNIX environment, will use that operating system and C (but it generalizes easily)
   - Go over academic integrity; emphasize that if you show programs to someone who hasn't done them, you cheated
   - Go over homework policy; emphasize no lates and redo policy!
2. What are the UNIX System and our campus setup?
   - Single computer: UNIX is operating system plus user environment
   - Networks link computers; campus network, Internet
   - Instructional computers are workstations running NFS
3. How to log in
   - Login name, password to xsession; get xterm
   - What login name, password are
   - Discuss good and bad passwords here, why you should change it as quickly as possible
   - Discuss department policy here
4. How to send email
   - Important addresses: *cs40*, *cs40t*, *cs40r*, *support*
   - Show the command (`mail` *`user`*)
5. How to compile a C program
   - Use the DECs!!!
   - Type program into text file using editor (*vi*, *emacs*, *pico*); file name must end in .c
   - Compile: gcc –ansi –Wall *file*.c; output in a.out
   - Show how to change output file name
6. First C Program: fahr1.c
   - Show how to compile it (UNIX commands)
   - What happens: preprocessor, compiler proper, assembler, linking loader
7. fahr1.c
   - program (main, return or exit for main)
   - declarations (int, register int)
   - body (while, functions; printf)
8. Basic operators

- arithmetic: +, -, *, /, %
- % defined so that n % p = r implies n = a * p + r for some a
- 5 % -2 can be 1 (5 = -2 * 3 + 1) or -1 (5 = -2 * 2 + -1)

9. fahr2.c
   - macros
   - include <stdio.h>
10. cat1.c
    - EOF, getchar; show why c **must** be int, not char (c is unsigned char)
11. cat2.c
    - do ... while loop
12. wc1.c
    - void
13. dands1.c
    - arrays (0-based, not 1-based)
    - ++, logical operator evaluation (lazy evaluation)
14. dands2.c
    - switch
15. pow.c
    - function prototype & declaration

Course Information

Instructor

Matt Bishop

office	3059 Engineering Unit II	*email*	bishop@cs.ucdavis.edu
telephone	(916) 752-8060	*URL*	http://seclab.cs.ucdavis.edu/~bishop

Lectures

Sections A01 and A02: TuTh 9:00–10:20AM in Art Building, Room 217
Sections B01 and B02: TuTh 12:10–1:30PM in Temporary Classroom 3

Course Outline

Elements of program style, documentation, efficiency. Methods for debugging and verification. Application of dynamic data structures. Introduction to the programming language C.

Course Goals

1. Become a good C programmer
2. Develop strong debugging skills (especially in the UNIX environment)
3. Learn how to analyze a program for potential portability problems or inefficiencies
4. Learn about pointers and recursion
5. Learn the basics of file manipulation
6. Understand the essentials of developing and organizing large (multi-file) programs
7. Learn the basics about the UNIX file system and processes

This course is a prerequisite for several upper division computer science classes; hence, we use the C programming language and the UNIX environment to demonstrate the ideas and principles of software development, because that will prepare you for later courses in the department. But remember these principles are universal, and would apply even if we used BASIC and the CP/M environment!

Discussion Sections

Section A01: M 5:10–6:00PM in Banier Hall, Room 1132
Section A02: W 5:10–6:00PM in Banier Hall, 1132
Section B01: M 4:10–5:00PM in Banier Hall, 1132
Section B02: F 4:10–5:00PM in Hoagland Hall, 108

These are ***required***; they present new material as well as review material presented in class. Material presented in these sections is important and ***will be*** covered on the exams. The same material will be covered in these sections, so it does not matter which one you go to.

Texts

- Paul W. Abrahams and Bruce R. Larson, *UNIX® for the Impatient*, Second Edition, Addison-Wesley, Reading, MA ©1997; ISBN 0-201-82376-4, ISBN 0-201-41979-3 (with CD-ROM)
- Barry Holmes, *Through C to C++*, Jones and Bartlett Publishers, Sudbury, MA ©1997; ISBN 0-7637-0319-2.

Course Handouts

Handouts and programs will also be in the class directory ~cs40, or in subdirectories; see the *README* file in each directory for more information. You can also access the handouts via the World Wide Web; the URL is http://wwwcsif.cs.ucdavis.edu/~cs40.

Class Newsgroup

Information about this class, homework assignments, office hours, and so forth, will be posted to the newsgroup *ucd.class.ecs40* newsgroup. Read this newsgroup daily, especially near the time assignments are due. **You are responsible for everything posted to this newsgroup.** This newsgroup is not for discussion about the class, but information from the instructor and teaching assistants to you.

Please post things about the class in the discussion newsgroup *ucd.class.ecs40.d*. Discussing something in this group is perfectly fair!

Exams

Midterm Examination: Thursday, October 30, in class

Final Examination: Sections A01 and A02: Friday, December 13, 4:00PM–6:00PM
Sections B01 and B02: Saturday, December 14, 10:30AM–12:30PM

These exams are open book/open notes. If you miss an exam for medical reasons (you ***must*** document this), you may be allowed or required to take a make-up exam, or the other parts of the course will be counted proportionally more (the choice is the instructor's). In particular, forgetting the time or place of an exam is ***not*** an excuse for missing it!

Lab Exam: take-home

This exam will require you to demonstrate that you know how to use *gdb*, among other things. It will be graded pass/fail.

Grading

Programs	50%	Midterm exam	20%
Lab exam	5%	Final exam	25%

Academic Integrity

Please see the *Code of Academic* Conduct on pp. 148–149 of the *Fall 1997 Class Schedule & Room* Directory. In particular, for this course, you may discuss general approaches with your classmates, teaching assistant(s), and/or the instructor. However, the actual expression of the answer, or the coding of the program and its attendant documentation, must be your own. Unless ***explicitly*** stated otherwise ***in the assignment***, collaboration is considered cheating and will be dealt with accordingly. Further:

- We consider it cheating to show your written work (or programming assignment) to a classmate, with one exception. If you need help debugging a *completed* version of your program, you may ask a classmate who also has completed a version of the program. You **must** write the program before you ask for help debugging it. You can **only** ask those who have finished a version of their program. The theory here is that since each of you has a version, you'll be unlikely to discard that and start over, so any (conscious or subconscious) copying is unlikely to occur.
- We consider it cheating to copy another's written work (or programming assignment).

A good analogy between appropriate discussion and inappropriate collaboration is the following: you and a fellow student work for competing software companies developing different products to meet a given specification. You and your competitor might choose to discuss product specifications and general techniques employed in your products, but you certainly would not discuss or exchange proprietary information revealing details of your products. Ask the instructor or a teaching assistant for clarification ***beforehand*** if the above rules are not clear.

Syllabus

lecture	*date*	*topic*
1.	Thu, Sep. 25	Getting Started; Writing and Compiling C
2.	Tue, Sep. 30	C Program Basics, Operators
3.	Thu, Oct. 2	Operators, Control Flow
	Discussion	The UNIX File System
4.	Tue, Oct. 7	Pointers, Command-Line Interface
5.	Thu, Oct. 9	Pointers; Storage, Structures, Casting
	Discussion	UNIX Processes and Shells
6.	Tue, Oct. 14	Functions and Scope; Program Stack
7.	Thu, Oct. 16	Recursion, Program Stack
	Discussion	Internet, World Wide Web
8.	Tue, Oct. 21	Preprocessor
9.	Thu, Oct. 23	Standard I/O Library
	Discussion	Other Libraries
10.	Tue, Oct. 28	UNIX/process interface, system calls
11.	Thu, Oct. 30	Midterm
	Discussion	Review for Midterm
12.	Tue, Nov. 4	C compiler details, make
13.	Thu, Nov. 6	Debugging, gdb
	Discussion	Introduction to C++
14.	Tue, Nov. 11	Modularity and Interfaces; Portability
15.	Thu, Nov. 13	Robust Programming
	Discussion	Data Abstraction in C++
16.	Tue, Nov. 18	Top-Down Design
17.	Thu, Nov. 20	Bottom-Up Design, Toolkit Approach
	Discussion	Overloading in C++
18.	Tue, Nov. 25	Rapid Prototyping, Shell Programming
	Thu, Nov. 27	***Thanksgiving Holiday***
	Discussion	***no section meetings***
19.	Tue, Dec. 2	Profiling
20.	Thu, Dec. 4	Conclusions and review
	Discussion	Review for final

All About Homework

The homework will consist primarily of programming questions, but sometimes will include questions which require explanation, analysis, or a demonstration of how to do something. This handout describes some general thoughts and techniques for doing homework, as well as what is required, how to submit it, how late homeworks are handled, and other administrative matters.

What We Look For In Homework

When we grade your homework, we look for simplicity, clarity, elegance, and documentation. The requirements for documentation, commenting, and coding are described later in this handout. Unless ***explicitly*** stated otherwise, *every* program you turn in must meet those standards and have the attendant documentation.

When you are asked to analyze something, or explain something, please be complete, and ***show your work*** (including any commands you give, and their output, to show how you did the problem); if you don't, your answer must be exactly the correct one or you get no points. Think your answer through and do a rough draft before you begin to write or type. Write clearly and cogently. If the question asks for an opinion, state your opinion clearly, and don't ramble. Answers which start, "My opinion is yes …" and conclude with " … so on the other hand I guess it could be no equally well" won't get much credit.

Doing Homework

We must emphasize the importance of taking time to design your program, or outline and draft your answer, thoroughly. More programming problems arise from improper design than anything else, and the few hours you spend on design will be amply repaid by shorter coding and debugging phases. So do think the design and interfaces through, and — as always — try to find the simplest way to do the assignment (within the limits given in the assignment, of course)!

Do not leave assignments for the last minute. The assignments will require significant design time before you start programming and debugging. When we decide on the due dates, we assume you will spend significant amounts of time on design as well as coding and debugging. If you choose not to do this, you will have difficulty finishing the assignments on time.

We do not mind being asked for help; indeed, we welcome it because it helps us know what the students are finding difficult or confusing, and sometimes a few words about the problem in class will clarify the assignment immensely. We **do** mind being asked for help before you have tried to think the problem through; the classic objectionable question (this really happened) occurred on a homework assignment in which the class was given a buggy program. The assignment said the program did not work, and the homework was to debug it and make it work. Within 10 minutes of the end of the class during which the assignment was given out, we got a request for help: "The program doesn't run. What do I do now?"

So, before asking for help, please be sure that you have:

1. spent a significant amount of time on the design of your solution;
2. used a debugger if the problem is a programming bug;
3. read all relevant handouts, sections of the textbook, and news articles (because your question may be answered there); and
4. tried everything you could think of to solve the problem.

When you come to us, or send us a note, asking for help, please bring whatever you have done to solve the problem, because the first question we will ask you is "What have you tried to solve the problem?" This isn't because we think you're wasting our time; it's because understanding how you have tried to solve the problem will help us figure out exactly what the problem is and what we can do to help you solve it. Remember, we will do everything we can to avoid solving the problem for you; when we give you help, our goal is to help *you* solve the problem yourself.

What You Are To Turn In

You must subqmit your homework electronically. Suppose your program for hw3 is in the files *getnum.c* and *printnum.c*. To turn it in, say

```
handin cs40r hw3 getnum.c printnum.c
```

This program will submit your files to the ECS 40 grader. Once you submit your homework, you cannot alter it in any way; should you wish to change your answer, make the change and resubmit the homework. We'll take the date and time of the last file you submit as the date and time of submission of your homework, and base lateness on that.

A manual page for the handin program is at the end of this section.

Some assignments will ask you to show the output on your screen; programming assignments require that you submit a sample run (using either your data or, if we give you some, our data). To do this, use the *script*(1) program to save the output into a file. For example, suppose the program you had to write is called *randx* and generates random numbers between 30 and 60. The number of numbers it produces is given by a command line argument. Here's how you would generate the file for the output (you type what is in **boldface**):

```
dec32 % script hw1–4–run
typescript started, file is hw1–4–run
dec32 % randx 5
34
49
58
43
38
dec32 % exit
dec32 % dec32 %
```

Then the file *hw1-4-run* contains the command, and output, given above. Submit it as above.

If you want to be sure your submission made it, execute

```
handin ecs40r hw3
```

and you'll get a list of what you have submitted so far:

```
The following input files have been received:
Thu Mar 17 14:50:49 1994        1599 bytes     getnum.c
Thu Mar 17 14:50:49 1994        3412 bytes     printnum.c
```

We expect any file you submit to be readable *as is* on the instructional DECStations. This means that you are free to type in your answers on your personal computer (or anywhere else), but we'll use *more*, *cat*, *lpr*, or one of the UNIX text editors to read the file. So be sure to look at what you're going to turn in on the instructional DECs ***before*** you turn it in!

We also expect any program you submit to compile (using the *gcc*(1) compiler with the **–ansi** and **–Wall** flags, unless stated otherwise) and execute on the instructional DECStations without change. (Don't forget to include those header files, programs, and other parts when we give them to you!) This means that you are free to develop your program on your personal computer (or anywhere else), but we'll use your Makefile to compile whatever you submit. So be sure to test what you're going to turn in on the instructional DECs ***before*** you turn it in!

Late Homework

If an assignment is not turned in by the date and time specified on the assignment, it is late. We will not accept late assignments, because we will make a sample solution available after the assignment is due. Exceptions will be granted only for a very good reason, and will be handled on a case-by-case basis; if an extension is given, the assignment may be changed (possibly substantially).

To see if a homework is late, we check the submission times on the electronically submitted homework. The *latest* date is the one that we use.

How We Grade It

Here's a rough weighting of the various factors that go into the grade of each program:

Correctness	70%
Documentation, commenting, ease of reading	20%
Clean, readable output	10%

If your program does not compile, the ***most*** you can get is 30%.

There is one catch to the grade. If your program does not work correctly, you will be given at least one example of input that produces an error. You must fix the program, so it does work correctly, within one week of being notified. If you do not, your score on the assignment will become 0%. When you resubmit the fixed assignment, submit the entire fixed assignment to *hwn*-fix; for example, if the above had a bug in *getnum.c*, you would fix the bug and then do:

```
handin cs40r hw3-fix getnum.c printnum.c
```

Grade Appeals

If you wish to appeal a grade you are given, first write out *all* your objections to the grade and request a regrade. Then give (or email) it to your TA; he or she will pass it to the grader. The grader will regrade it, and report your new score (which can go down as well as up). Should you wish to challenge the grade further, write out your objections to the new score, and give your TA a copy of the original objections, the grader's response, and your new objections, as well as what you turned in. The TA will then regrade the assignment; again, your score can go up or down. If you still wish to challenge the TA's grade, again write out your objections, and come see me. **Be aware that failing to follow the requirements is *not* grounds for changing a grade; in particular, if your program does not compile on the instructional DECStations, you'll get no sympathy from us at all.**

All grade appeals must be made within two class periods of when the returned assignments were made available. After that, we won't look at it.

NAME

handin – file submission program

SYNOPSIS

handin *touser* [*subdirectory* [*files* ...]]

DESCRIPTION

handin provides a secure means of submitting files to another user, recounting what has already been submitted, and listing what subdirectories exist for containing submissions.

USAGE

Submitting files

With *touser*, *subdirectory* and *files* all specified, each file is copied to ~*touser*/handin/*subdirectory*/fromuser, named with the original file's *basename*(1), and made owned by *touser*. The directory *fromuser* is made if it doesn't already exist and is named after the invoking user. Each file specified should have a *basename*(1) unique among any files already submitted by that user to subdirectory, unless overwriting is desired.

Recounting submissions

Without files specified, information on previous submissions by the user to the specified subdirectory is shown.

Listing existing subdirectories

Run with only *touser* specified, *handin* just lists the existing subdirectories (regardless of accessability).

EXAMPLES

The following examples illustrate the use as a homework submission facility to the pseudo-user ``cs101'' created for this purpose:

```
example1% handin cs101
Existing subdirectories (comments in parentheses):
Asn1    (Due Mar 18)
Asn2    (Due Mar 25)
example2% handin cs101 Asn1 part1 part2
Submitting part1... ok
Submitting part2... ok
example3% handin cs101 Asn1
The following input files have been received:
Thu Mar 17 14:50:49 1994        1599 bytes      part1
Thu Mar 17 14:50:49 1994        3412 bytes      part2
```

SEE ALSO

rcvhandin(8)

DIAGNOSTICS

handin itself provides only a little of the diagnostic information that's given and returns the number of errors encountered as its exit status. Any other information comes from *rcvhandin*(8).

Skipping *file*: file non-existant or irregular

The named file didn't exist or was probably a directory. The user should check to make sure that the file they specified was indeed the file they intended to submit.

Skipping *file*: file not readable

The named file was not readable by the user.

Submitting *file*... failed [: *reason*]

The named file was not successfully submitted. If at all possible a reason is provided by *rcvhandin*(8).

Submitting *file*... ok

The named file was successfully submitted.

NOTES

handin is really just a front-end to the *rcvhandin*(8) program. The primary function of **handin** is to open the named *files* with the effective user ID of the invoking user and pass on their contents to the *rcvhandin*(8) program having the effective user ID of *touser*. This design provides a simple and portable means for implementing a file submission facility in even a non-homogeneous, network-file-system environment.

AUTHOR

Lou Langholtz, Department of Computer Science, University of Utah, 1994

Program Standards

Computer programs are meant for two audiences: the *computer* that compiles and runs it, and the *people* who must read, modify, or evaluate the program. Most programming shops have "inhouse standards" for programming style since a standard style tends to make programs more readable. Engineering and Computer Science 40 is no exception; its rules are listed in this section. Part of your homework grade reflects how well you meet these standards.

Code Indentation and Formatting must follow the standard C style, which is illustrated in the text. It's not the prettiest style but it is quite standard. Please indent eight spaces (a tab) for each level of indentation.

Capitalization rules in C are also fairly standardized, although a bit unpleasant. Every part of your code must be in lowercase, except for names of macros, defined types, and enumerated constants which are entirely UPPERCASE.

Comments should be used to make programs absolutely clear. Use them liberally but not so much that the meaning gets submerged.

You should **comment your program as you write it**. It is actually much faster than going back later, and comments are very useful when debugging. (If what the comment says the code does is different than what it really does, you found a bug.) Graders, teaching assistants, and instructors reserve the right to refuse to look at uncommented programs.

ECS 40 "house standards" call for four kinds of comments; we'll describe each in turn.

- Start-of-file comments
- Start-of-function comments
- Paragraph comments (also called box comments)
- Line comments

Start-of-file Comments

```
/*
 *
 * hashtest.c      Sam Bent          date unknown
 *
 * Test hash table routines. Loop endlessly, requesting keys from
 * stdin. If key doesn't exist, insert it in the table with value
 * n (where it's the n-th key added to table). If key does exist,
 * print its stored value of n.
 *
 * Modification History:
 *
 *     1 Jan 94   Matt Bishop; changed to conform to ECS 40 style
 *    29 Nov 90   Chip Elliott; changed to conform to CS 23 style
 *                and use ANSI C.
 *
 */
```

The comments at the beginning of a file, collectively called its "header," must contain your name in addition to other general information about the program and what it does. The date when the program was written and a history of major modifications are required.

Complex data structures used by a file should be illustrated with a figure, as shown below.

```
 * HASHTABLE       hashstruct
 *             +--------------------+     Allocated Block
 *    ht   ->  |      ht_ID         |       of HASHSLOTs
 *             +--------------------+    +------------------+
 *             |      ht_size       | +->|       ....       |
 *             +--------------------+ |  +------------------+
 *             |      ht_cnt        | |  |       ....       |
 *             +--------------------+ |  +------------------+
 *             |      ht_curslot    | |  |       ....       |
 *             +--------------------+ |  +------------------+
 *             |      ht_slots      |-+  |     HASHSLOT     |----+
 *             +--------------------+    +------------------+    |
 *                                               ....            |
 *                                               ....            |
 *             +-------------------------------------------------+
 *             |
 *             |         hashentry
 *             |   +-----------------+     +------------------+
 *             +-> |      h_key      |-->  |    key's text    |
 *                 +-----------------+     +------------------+
 *                 |      h_val      |
 *                 +-----------------+
```

Start-of-function Comments

```
/*
 * ht_enter
 *
 * Enter a key into a hashtable and return its HASHSLOT.
 * If necessary, expand the table to hold the new key.
 * If the key is already there, just return its HASHSLOT.
 *
 * Entered by:  ht   -- hash table
 *             *key  -- key to enter
 *
 * Exits with:  key in hashtable
 *                 if newly created, its val.p field is
 *                 set to NULL
 *              ht possibly expanded
 *              returns key's HASHSLOT
 *
 * Exceptions:  "parameter is not a hash table"  (fatal)
 *              "not enough memory to enter item into
 *                                  hash table"  (fatal)
 *
 */
HASHSLOT ht_enter(HASHTABLE ht, char *key)
{
```

Each function must have its own header. The reader should be able to determine what a function does and how to use it by reading only this header information. Things like global variables referenced or modified, assumptions about the input, or anything else that the reader should know before "lifting" a function and using it in another program should go here. So should the algorithm used, if it's not absolutely clear.

One thing that you may not be familiar with is input and output assertions. "Input assertions" is a fancy way of saying "I assume this about the input." For example, for a square root routine sqrt(x), the usual input assertion is "x > 0." Output assertions are similar — they describe what is true after the function has been called. It is a good habit to get into writing these assertions. It helps you specify exactly what the routine does, and this is precisely the information that someone needs to know in order to lift your routine and use it elsewhere.

You should also include a list of *Exceptions*, i.e., unusual/error cases and what the function does with them.

If you borrow code from another source (a fine idea, by the way) you should cite the author, book, and — if possible — page numbers. Be specific enough so another reader can find the reference.

Paragraph Comments

```
/*
 * update ht's HASHSTRUCT contents
 */
/* first, copy size and count */
ht->ht_size  = newht->ht_size;
ht->ht_cnt   = newht->ht_cnt;
/* release old hash slots, start using new ones */
free(ht->ht_slots);
ht->ht_slots = newht->ht_slots;
/* free newht's HASHTABLE */
free(newht);
```

When you write pseudo-code or describe how a subroutine accomplishes its task ,you usually break it down into a series of steps. (To insert an item into an array you find where it belongs, move everything after it down one place, and finally copy the item into the correct position.) These steps sometimes become subroutines with descriptive names, but often they become half a dozen lines of code. In this case, it helps to write a comment that explains the purpose of the next section of code. The first comment is an example of this kind of comment.

Sometimes a paragraph comment is completely boxed, like this:

```
/**********************************
 * update ht's HASHSTRUCT contents *
 **********************************/
```

These are also called boxed comments. You can use this style if you prefer.

Line Comments

```
main()
{
       HASHTABLE ht;          /* ht  = hashtable */
       HASHSLOT *hp;          /* hp  = ptr to hash table entry */
       int n=0;                      /* n   = key number (put into ht) */
       char key[100];         /* key = entry's key */


       /*
        * create a new hash table
        */
       ht = ht_create(0);

       /*
        * loop, reading input and acting accordingly
        */
       for(;;){
           /* prompt and get input */
           printf("Key: ");
           scanf("%s", key);

           /*
            * if "dump", show what's in the hash table
            */
```

```
                if (strcmp(key, "dump") == 0){
                    /* bingo -- dump the hash table */
                    ht_dump(ht);
                    continue;
                }

                /*
                 * enter the key into the hash table
                 * announcing if it is already there
                 */
                hp = ht_enter(ht, key);
                if (hp->h_data == NULL){
                    /* key is new -- say so, and store it in entry */
                    printf("%s is a new key, number %d. ", key, ++n);
                    hp->h_data = (caddr_t) n;
                }
                else{
                    /* key is new -- say so, and store it in entry */
                    printf("%s is key %d. ", key, (int) hp->h_data);
                }
            }
    }
```

In declarations. Explain the use of constants, types, and variables. Part of this is choosing descriptive names. Avoid single-letter names (except for *i*, *j*, *k*, and their ilk as loop variables when nothing more descriptive is useful). Most of the time you should supplement this with a more descriptive comment when the constant, type, or variable is declared. *Sum* is a fine variable name for a partial sum, but its declaration should state whether it is summing the grades for an exam, the cost of books for a given term, or whatever. Put these comments at the end of each declaration line, and align the beginning of the ordinary comments in a set of declarations, as shown. If the comment takes more than one line, align the second line too.

In code. Comments should also be placed above a line of code *if it will clarify what the code does*. These comments should explain how that particular line of code fits into the general scheme of things. The comment:

```
/* Assign item to ith position of list */
list[i] = item;
```

is worthless. It simply repeats what the code says. Assume that the reader can read the code, and explain the purpose of the statement. The comment:

```
/* Put item to be inserted into empty spot */
list[i] = item;
```

is much better. These comments are especially useful if the code is somewhat tricky or non-intuitive, and if an innocent-looking statement has consequences not obvious from the statement itself.

Align all these ordinary comments with the beginning of the line they explain. Otherwise they are messy and distracting.

Structured Programming

All the comments in the world will not make a 10-page monolithic block of spaghetti code intelligible. You should already know a little about breaking your programs into functions and subroutines. We will be discussing methods for decomposing problems in this class, and suggest that you use them. A common question is "how long should a function be?" Most functions end up being short. If a function is longer than a page, it is usually (but not always) better broken into separate functions.

We would like to also comment on the infamous **goto** statement. The uncontrolled use of **goto** can make programs totally unintelligible, and many of you have probably been convinced that a **goto** statement is programming's worst sin.

We consider **goto** statements justifiable in one case: "error bailout." In some cases, functions call other functions (often recursively) and it is possible to get 20 or 30 nested calls. If at this point an error is discovered, that error must be passed back up through the 20 or 30 calls. Each function must deal with the question “what should I do if this function I am about to call finds an error?” This is certainly doable and can sometimes lead to better error recovery, but functions must pass and test error flags everywhere. A **goto** back to the top level, if carefully used and properly commented, can greatly simplify life. We will discuss both approaches later in class.

Completeness and Correctness

Submitted assignments should be complete and correct. Listings must show the final version of the program; handwritten corrections to the program will not be accepted. If your program does not work perfectly by the time it is due, turn in your best effort with a clear explanation of what works and what does not. It is more valuable to demonstrate that you understand the merits and limits of your program than to hope the defects in a half-working program will go unnoticed.

CSIF Accounts

The staff has created instructional accounts for all students who are registered for Computer Science courses in Fall Quarter 1996 but who did not already have an account on the Computer Science instructional machines. Find the list of new accounts downstairs. The list is alphabetized by last name. Look down the list until you find your name. In the "Account Name" column is the name of the computer account that has been assigned to you. It's also called a "login name". (It does not apply to IT [Information Technology, essentially the campus computer center] accounts, to ACS [Academic Computing Services, a unit of the College of Engineering] accounts, or to ECE [Electrical and Computer Engineering] accounts. It applies only to your Computer Science account.) The Computer Science instructional machines are in the basement of the east wing of building Engineering II.

If your name does not appear on this list, either you already have an account on the Computer Science instructional machines, or your name was not included on the class lists that we got from the registrar. The latter may occur if you have added a Computer Science class, as opposed to pre-registering for one. In this case, your account will be added the day after we receive notification from the registrar that you have added the class. Your account name and login name will then be posted as described above.

An initial password has been assigned to your account: the last 8 numeric digits of your student id number. (The student id number has 9 numeric digits in it, so omit the first one, and omit the hyphens to form your initial password.) Because your student id number is something that other people can probably find out about you, a password based on your student id number is not very secure. That is, someone other than you could login to your account, including hackers. Since you are the only one who is authorized to use your account, it is ***imperative*** that you change your password as soon as possible. Use the *yppasswd*(1) command when logged in to the Computer Science instructional machines to do so. If you have not changed your password by 5pm on the Monday of the second week of classes, logins on your account will be disabled. To get logins on your account re-enabled, stop by room 047, whee the student assistants are located. You will need to bring your student id card.

Disk quotas have been installed. You may store no more than 10 Mbytes of data in your home directory.

If you have any questions, send mail to "support".

Computers

All programs and commands written for homework must work on the DEC, HP, and SGI systems **unless specifically stated otherwise in the assignment**. A goal of this class is for you to learn to write portable programs; also, the graders cannot know which system will be available to them when they grade your work. The computers are named according to the following schemes:

dec1, ..., dec62	Digital Equipment Corp. workstations running Ultrix
sgi1, ..., sgi23	Silicon Graphics, Inc. workstations running IRIX
hp2, ..., hp25	Hewlett-Packard workstations running HP/UX

You can log on to the instructional systems in two ways: directly into a **workstation** in the basement of Engineering Unit II, or through a network login program (*rlogin*(1) or *telnet*(1)) on the UC Davis network.

Workstation

To log in, type your account name and password on the login dialog box. Then click on OK. The workstation will (slowly) begin to open windows: two *DECterm* terminal windows, *xman*, and *dxclock*. Click on either terminal window to activate it, and start typing commands.

To open more DECterm windows, select **DECterm** from the (SM) system manager's **Applications** menu.

To end your session, select **Quit** from the (SM) system manager's **Session** menu.

Similar commands and actions work on the SGI's and the HP's.

Network Program

From any computer on the UC Davis network, use *rlogin* or *telnet* to connect to the instructional machines. This connection gives you one terminal window onto the UNIX system. You must first log in. Then be sure to set your terminal type to vt100 (or whatever is appropriate for your terminal) as shown below (you type what is in **bold**; the computer types what is in plain, and my comments are in *italics*):

```
ULTRIX V4.4 (Rev. 44) (dec32)

login: bishop
Password:                                   note my password is not echoed!
Last login: Mon Sep 20 16:11:56 from nob.cs.ucdavis.e
ULTRIX V4.3 (Rev. 69) System #1: Mon Aug 19 10:02:26 PDT 1996
UWS V4.4 (Rev. 11)


                Welcome to the CSIF!
You have mail.
dec32 1> setenv TERM vt100          be sure to do this first...
dec32 2> vi sort.c                  ...then precede with normal life
```

From workstations which do not run X Windows, you cannot use X-window tools. But you can access any of the UNIX machines from your dorm room. And all the really important tools (*vi*, *gcc*, etc.) work just the same on a plain terminal interface as they do through X Windows.

To open more terminal windows, use *rlogin* or *telnet* to create more connections to the UNIX machine. You'll have to log in for each new window.

To end your session, just log out.

The Special Keys

The UNIX operating system lets you configure which keys do what. That means, among other things, that no manual tells you which key does what — since it varies from site to site. So here's the scoop for the machines that you will be using.

The little caret, as in ^C, means "press the control key while typing C."

What	*At Workstation*	*Over the Network*
delete previous character	DEL	^H
delete line	^U	^U
kill a running program	^C	^C
end of file on standard input	^D	^D
escape key (useful for vi)	F11	ESC

A Sample Session

This extremely brief session shows how you might create a file, compile it, run the resulting executable file, print the source, and finally mail the program text to someone.

```
dec32 29> vi hi.c

       use vi to create/edit a new file hi.c

"hi.c" [New file] 10 lines, 135 characters
dec32 30> gcc -ansi hi.c     run the c compiler
ccom: Error: hi.c, line 9: syntax error
         printf("hi there!"\n);
      ----------------------^
volta 31> vi hi.c
```

```
        use vi to fix syntax errors

"hi.c" 10 lines, 135 characters
dec32 32> gcc hi.c                  run the c compiler again
dec32 33> a.out                     it compiled, so run the executable
hi there!
dec32 34> lpr hi.c                  print the source file
dec32 35> lpq               check the printer queue
hexus is ready and printing via dev
Rank   Owner      Job  Files                              Total Size
1st    bishop      49  hi.c                               135 bytes
dec32 36> mail myfriend             mail a note to your friend
Subject: hi.c written
                                    this is part of the letter
Here's my first C program!  It just says hello,
but it does compile and execute cleanly.

~r hi.c                             read message text from file hi.c
"hi.c" 10/135
.                                   this ends the letter
Cc:    just hit return
dec32 37>
```