This is how system knows where daytab[0] [j] ends and daytab[1] [0] begins. The expression *((char *)daytab + 13*i+j) is same as daytab[i] [j], but cast daytab so doesn't jump 13 bytes at a time.

```
void func(char daytab[2] [13])  { . . . }  is tolerated
void func(char daytab[ ] [13])  { . . . }   is OK
void func(char (*daytab) [13])  { . . . }  is also OK
```

IN GENERAL, array handling is poor in C; FORTRAN much better. Efficiency and object-oriented implications. Reason for C problem? NXN array can't be handled by function: Need to specify second subscript in call.

As it is pointed out in Section 5.9, the declarations

```
int a[10] [20]
int *b[10]
```

differ in that, although a[3] [4] and b[3] [4] are both legal, the declaration of a actually sets aside 200 elements in memory, whereas the declaration of b only sets aside 10 locations for pointers.

Then b[3] is a pointer to an int, and b[3] [4] is the same as (b[3]) [4] , the fourth integer on from the integer pointed to by b[3].

Daily Course Notes for CS240
Computer Architecture 1
Patrick E. O'Neil

Class 19. Administer Quiz 2

Starting Chapter 7. We will be defining the standard library for ANSI C, available on any UNIX system, portable to other OS's (NT).

You will be responsible for having Glass & Ables (call it Glass) UNIX 2nd Edition in class; questions from Glass will be on Quizzes and Exams.

Library function specifications are in K&R Appendix B,. You are responsible for all specifications in Appendix B mentioned in class.

Note: all header files for libraries (stdio.h, string.h, ctype.h, etc.) are online in /usr/include. Now cd to this directory and look at C contents.

Here's an important command you should know: grep. Find it in the Glass & Ables UNIX manual (look up in index).

NOTE: I often call Glass & Ables simply **"Glass"**.

%grep -hilnvw *pattern* {fileName}*   (See pgno (220,84) in Glass: a page spec like this represents (Ed2.pgno,Ed3.pgno), for 2nd and 3rd Editions.)

The grep command will search for a pattern in a list of files. It's like the program we wrote to illustrate command line argument use. (The example for that was: find -xn *pattern*. But use -v instead of -x in grep.)

Thus grep looks for and prints out lines in a file or list of files that match a *pattern*. Each file name, and its appropriate line, is named in output unless -h option is given, in which case file name is not listed in output.

A list of file names, delimited by spaces, is what is meant by the argument {fileName}*   We could write:

%grep US studentfile staffile facultyfile (3 files to search, pattern: US)

In the command: grep printf cs240/hw5/*.c, the *.c gets expanded by the shell to a list of files that end in .c, and grep looks at them one by one.

Options: -n means give line numbers, -i means ignore case ("And" matches "and"), -l displays only filenames that contain pattern, -v displays lines that DON'T match pattern, -w means only matching for whole words count.

Use grep to search for function names in header files.
%grep isupper /usr/include/* (see 4 different lines in ctype.h)

Note, if no filenames are specified, grep will search stdin. Common idea in UNIX commands, useful for "pipes", as explained below.

We can use "metacharacters" in "Regular Expressions" in a grep pattern to describe a family of strings. See pg (606,665) and following in Glass. The Regular Expression Metacharacters are NOT the same as Shell Metacharacters listed on pg (81,151), though there are some chars in common.

Now assume that instead of "US" on lines of studentfile, etc., we had "U.S.". It turns out to be difficult to represent this on a command line. Writing

%grep U.S. studentfile staffile facultyfile

doesn't work, because the metacharacter period, ".", is interpreted by grep regular expressions to stand for any single character (FIND IT). So if we really want to represent periods in a regular expression, we can "quote" each period with a "\" sign.

But the "\" sign itself will be interpreted by the shell so only the "." will be passed, so "." will continue to be interpreted by grep as "any character".
To get the "\" sign seen by grep, you would need to write:

%grep U\\.S\\. studentfile staffile facultyfile (echo U\\.S\\., see U\.S\.)

As you know, the sequence "\\" is interpreted by the shell as REALLY the character "\"; don't interpret it as a shell quote, so "\." gets through to grep in this case.

See some of the Regular Expressions that work on pg (607,666). But you need to VERY defensive getting grep regular expressions passed through the shell. YOU SHOULD TEST IDEAS. In a fresh directory, create one file with "U.S." and another with "U_S_", and a third with "US" and try:

%grep U.S. *   (* stands for all files in current directory to the shell.)
   This will return lines with "U.S." or "U_S_: or indeed: "U Shaped".
%grep U\.S\. *   (This will return the same as above, grep U.S. *)

%grep U\\.S\\. * (This should work to retrieve "U.S." lines only.)

If we want grep to return either US or U.S. or "United States". Note on pg (606,665) "*" following a character denotes zero or more occurrences of that character. (Thus using it will also give us "U..S...." or "Ugly & Stupid".)

The following doesn't work to give us 'U.S.' and 'US':

%grep U.*S.* * (The * at the end still stands for all files.)

since "." is interpreted by grep as any character, so we won't see"US". Try:

%grep U\\.*S\\.* *

But this doesn't work because * after \\. is interpreted by the shell. Try:

%grep U\\.*S\\.* *

This doesn't work because grep sees "U\.*S\.*" (the \\ before * eaten by the shell and leaving \) and grep will now be looking for the literal characters "U.*S.*" But this works:

%grep U\\.*S\\.* *

Because grep sees "U\.*S\.*". See what I mean about being "defensive" in using regular expressions?

You should read the Glass pages referenced to understand why this works. You can type "man grep" when you're online and forget some feature.

Chapter 7. Now start going through Chapter 7 of K&R. Follow along. **Also, look for MORE DETAILS of everything we cover in Appendix B!**

You should now learn and use <u>**EVERYTHING YOU CAN**</u> in the C Library Function coverage of Appendix B! You'll be expected to know most of it.

Section 7.1 Standard Input (stdin) and Standard Output (stdout.) Already know a lot of this. Standard I/O redirection:

%prog <infile >outfile

Idea of a pipe is new: prog1 | prog2. This will execute prog1 and prog2, send stdout from prog1 into stdin for prog2. For example:

%grep S *|grep U

Will perform "grep S *" on all files, pass lines through the interface as stdin, then grep U will find all lines with both U and S in them somewhere!

Note, specification in Glass, pg. 29: more -f [+lineNumber] {fileName}*

The option -f means don't fold long lines (useful if piping stdout to something else that deals with lines, e.g., wc command); can start at given line number. If no file name specified, will use stdin. See how this is useful?

For example: ls -lt|more   This gives a long listing of files in the current directory, most recent first (-t option), & pipes through "more"; more uses stdin, so most recent files won't go off screen. Also: finger name|more

7.2. Formatted output: Printf. Formats and prints out internal values.

    int printf(char *format, arg1, arg2, . . .);

Note printf has a VARIABLE LENGTH ARGUMENT LIST (as many as there are % conversions in the format string). We will learn how to do this shortly.

The return from printf is the number of characters printed (haven't used this up to now, but useful if there is error or some limit truncation).

Between the % and the conversion character, there are a number of other characters which may exist. In the order they must be placed, they are:
   -   (minus sign) left adjust printing of argument
  m (number m)   minimum field width
  . (dot)   separates min field width from precision
  p (integer p)   precision: max chars for string, min digits for int
  h or l (letter h or l)   h for short int, l for long int

 (ORDER of options for %d is: %[-][m][.][p][h|l]d, Note: no embedded spaces!)

E.g., figure out what these would do: %10d, %-10d, %hd, %e, %10ld, %10.p

Try spending 10 minutes with a program, using different formats.
Learn these with examples of string precision given on pg. 154, bottom.

Also, to print at most max characters from string s (max is int type var or const), use * after % and include the int max as an argument before s:

    printf("%.*s", max, s);

Note, it is possible to print out a character string as a format string with no % variables: what we do with printf("hello, world!\n"); could write:

    char s[ ] = "hello, world";
    printf(s);

But if s might get a % character in it, this is unsafe, since format will then require another argument after s. Better to write out s[ ] as:

    printf("%s", s);

Finally, the function sprintf will work same as printf, but write to string.

    int sprintf(char *string, char *format, arg1, arg2, . . .);

See sprintf in Appendix B, pg 245 K&R. Note how useful sprintf is!

Recall how we wrote itoa() and itox() functions. No functions like this in C library! Instead use sprintf() to print int into a string, using %d or %x.

Might now want to strcpy the string into a malloc'd area you create. You can start reading ahead now, using any C library function. See Appendix B.3 for strcpy (or strncpy), and B5 for malloc.

7.3 Variable-length argument lists, E.g.: printf( ). Not available in JAVA, but JAVA has overloading.

    void minprintf(char *fmt, . . .);   /* three dots in a row: var arg list   */

Here is how this works. Recall that when a function call is called, a new stack frame is created for the function execution.

The stack frame holds a memory location to return to from this func, the arguments of the function, and local variables of the function. **Leave Up!**

Return location
Argument 1
Argument 2
. . .
Argument k
Local variable 1
Local variable 2
. . .
Local variable n

This stack frame representation is not exact: it will be covered in cs341. The va_ package we cover here hides the actual representation.

The function can pick off one argument after another, and in the case of variable length argument lists, the k is not set in advance.

In a function with a var-arg list, we start by declaring a variable (say, ap for arg pointer), of type va_list, to refer to the arguments in the list.

```
void func(int n, . . .)      /* Note the "ellipsis", i.e.  (. . .)                    */
{
    va_list ap;              /* ap is just a variable name; could have any name      */
```

Now use macros va_start, va_arg, and va_end to retrieve argument values. These macros exist in stdarg.h library: B7, pg 254 of K&R. Look at it.

Start by initializing ap, using va_start:

```
    va_start(ap, n);
```

The variable "n" named in va_start is the last named argument before the ellipsis (. . .) in the var-arg list function, function. There must be at least one such argument: use it to tell how many arguments in list.

E.g., in printf( ), first argument gives string with % conversions for all later arguments.

Now ap points just BEFORE first unnamed arg. Each call to va_arg will advance ap one argument, return value; va_arg must name type of arg:

```
ival = va_arg(ap, int);        /* use this where int argument       */
dval = va_arg(ap, double);          /* use where double argument        */
sval = va_arg(ap, char *);          /* use where string argument (ptr)  */
```

But think about how you are going to KNOW the type of each arg!!! This is why format string is normally passed for printf( ): % conversions tell you.

Of course, you could assume that all the arguments for a specific var-arg function are of the same type, say string.

But you MUST know HOW MANY arguments. If overestimate, parsing past Argument k in stack frame, get garbage. To terminate, write: va_end(ap);

Example minprintf, page 156 (PUT ON BOARD). Will have homework on this.

7.4 Formatted input: scanf -- pg 157. This is the opposite of printf. Reads in variables from stdin using conversion format string. See pg. 246 (and prior pg 245 which explains everything).

```
   int scanf(char *format, . . .);
```

The value returned from scanf( ) is the number of successfully scanned tokens: not successful if can't parse the value brought in from stdin.

In calling scanf, call with any number of arguments. but must call with POINTER to variable so that the variable values can be set by scanf!

```
   int age, weight;
   char lname[100];

   while(some condition) {
      printf("Input your last name, age, and weight, separated by spaces);
      cnt = scanf("%s %d %d", lname, &age, &weight);
      . . .
   }
```

Note: name is an array, and is already like a pointer to the char string.

Scanf is useful to allow you to read in int or double value AS A NUMBER, instead of a character string, where you have to do your own conversion.

(Of course, scanf() will always see a character sequence in stdin: just does its own conversion to int or double.)

However, scanf is FLAWED, because it ignores '\n' characters. Can get very confusing if user puts too few arguments on some line.

(**Prompt**) Input your last name, age, and weight, separated by spaces:

(**User input**) Clinton 52

(<u>No response after carriage return. User tries again, remembers to include weight this time</u>.)

(**User input**) Clinton 52 200

(scanf will see: Clinton 52 Clinton, since '\n' character from user carriage return is seen as white space separator; so thinks weight is weird value, and will return 2 as the number of successfully scanned tokens.)

Worst part is we're out of synch, since now 52 will be seen as last name in next prompt loop. Can't code defensively with scanf( ): can't count number of tokens parsed ON A LINE — scanf doesn't care about input lines.

The best approach is to read a line (use fgets( )) into an array s[ ], and use "sscanf( )" function to pick apart the arguments in the line just input. This also allows you to try to interpret things in more than one way.

Know that sscanf works on a string if successfully scans all tokens in the string; there's a homework problem on this.

But instead of getline( ) to bring in a user line, use library function fgets(), K&R pg. 247. Cover a bit later, Section 7.7. Prefer fgets( ) to gets() since they behave differently, and we must use fgets() for files.

Note in in both scanf and sscanf, if you put characters (space, :, -, /) in the format string, we MUST see exactly those special characters in user input.

```
   cnt = sscanf(s, "%d/%d/%d", &month, &day, &year);        /* s has string     */
```

will expect input like: 07/23/96. If not, cnt value returned by sscanf will be less than 3.

RECALL how we wrote function atoi, axtoi to convert character string s to integer i. There is a function atoi in C library, but no axtoi. <u>Question</u>: How would we do this?

Use sscanf(s, "%d", &i) for atoi, or sscanf(s, "%x", &i) for axtoi.

RULE: Use scanf only for programs needing only ONE input item, usually "quick and dirty" programs with no input checking.

HOMEWORK on this!

Class 20.

7.5 File Access.

We have had practice reading from stdin and writing to stdout. We can redirect stdin from a file and stdout to a file at the command level.

In this Section, we learn to get program control over reading and writing <u>named</u> files. An example of an application of this is the command "cat".

   cat fname1 fname2

This reads from file fname1, then from file fname2, and puts all characters it reads to standard output. Can catenate two files to a third, thus:

   cat fname1 fname2 >fname3

Any number of files can be catenated; the Glass UNIX syntax is:

   cat -n {FileName}*

The option -n gives line numbers to the output. What do you think happens if no files are named? (Yes. Read from stdin.)

Dealing with named files is surprisingly similar to dealing with stdin and stdout. Start by declaring a special named object, a "file pointer":

   FILE *fp; (See Appendix B1.1, pg. 242 -- return from fopen)

The <stdio.h> header contains a structure definition with typedef name FILE, which contains component variables (buffer, etc.) used in file I/O.

You don't need to know the details of structs to use simple file I/O. Just use primitive functions, such as fopen():

fp = fopen(name, mode)
(Functional prototype: FILE * fopen(const char *name, const char *mode);)

<u>Here fp is the return value, set to NULL if fopen fails</u>! Now fopen is asked to open a named file (character string "name") in a particular "use mode".

Legal mode values include "r" for read, "w" for write, and "a" for append.

These modes cause the open file to have different behaviors. We can make calls to get a char out of an "r" file with getc(): (Appendix B1.4, pg 247)

   c = getc(fp);   /* like getchar(): an "r" mode file acts like stdin   */
   (Functional prototype: int getc(FILE *stream); <u>Return EOF if fails</u>.)
   NOTE: Sometimes Failure gives EOF return, sometimes NULL; LOOK
         IT UP and GET IT RIGHT!

   NOTE: use getc( ) & putc( ) instead of fgetc & fputc - probably faster.

   status = putc(c, fp);   /* like putchar: "w" or "a" mode files like stdout */
   (Func. prototype: int putc(int c, FILE * stream); <u>Return EOF if error</u>.)

When we fopen a file in "w" or "a" mode, if the file does not already exist, it will be created (as the vi editor creates a file it has never heard of).

If the file does already exist, then "w" mode fopen will destroy the old contents (like the command mv) and "a" mode will append new material to the end of the existing file (like the "save" command in mail).

More modes are given in Appendix B, pg. 242. (Will come to update later.)

When you have finished reading from a file or writing to a file, you should call fclose to close the file.

   status = fclose(fp);
   (Functional prototype: int fclose(FILE *stream);)

<u>The function fclose( ) returns EOF if any error occurs, and zero otherwise.</u>

Example follows of programs opening named file.

```
Script started on Sun Nov 16 11:03:54 2003
blade64(1)% cat writefile.c
/* Simple test program to write a text file */
#include <stdio.h>
char outfile[] = "test.dat";

int main()
{
    FILE *fp;
    int i;

    if ((fp = fopen(outfile, "w")) == NULL) {
        fprintf(stderr, "Can't open file %s\n", outfile);
    } else {
        for (i = 0; i < 8; i++)
            fprintf(fp, "%d\n", i);
        fclose(fp);
    }
    return 0;
}
blade64(2)% cat readfile.c
/* Simple test program to read a text file of numbers and sum them */
#include <stdio.h>
#define MAXBUF 100
char infile[] = "test.dat";

int main()
{
    FILE *fp;
    int x, sum = 0;
    char buffer[MAXBUF];

    if ((fp = fopen(infile, "r")) == NULL) {
        fprintf(stderr, "Can't open file %s\n", infile);
    } else {
        /* loop through lines of the file */
        while (fgets(buffer, MAXBUF, fp)) {
            /* use sscanf to convert from text to binary number */
            if (sscanf(buffer, "%d\n", &x) != 1) {
                fprintf(stderr, "bad line: %s, continuing\n", buffer);
                continue;
            }
            sum += x;
        }
        printf("Sum = %d\n", sum);
        fclose(fp);
    }
    return 0;
}
blade64(3)% gcc -Wall writefile.c -o writefile
blade64(4)% gcc -Wall readfile.c -o readfile
blade64(5)% writefile
blade64(6)% cat test.dat
0
1
2
3
4
5
6
7
blade64(7)% readfile
Sum = 28
```

Reading from stdin or writing to stdout, you sit at a character in a virtual file, called a "stream", and move only to the right to the next character.

But as we will see, in named files it is possible to "go to the left" to read characters over again (using the function fseek(), App. B1.6, pg 248).

When a C program is started, the operating system opens three files and provides file pointers (FILE *) to them: stdin, stdout, and stderr. We can now define our old friends getchar and putchar as macros:

```
#define  getchar( )  getc(stdin)
#define  putchar(c)  putc((c), stdout)  <-- see why (c) is in parens?
```

Other file oriented analogs to input and output functions we've known are:

```
int fscanf(FILE *fp, char *format, . . .);   /* mode of fp must be "r"     */
int fprintf(FILE *fp, char *format, . . .);   /* mode of fp is "w" or "a"     */
```

We always use an "f" version of primitives, fgets rather than gets. (But use getc and putc) (Note, we would still use fgets to bring in a line and sscanf in preference to fscanf.) OK, Now here's the cat program.

```c
#include <stdio.h>     (LEAVE UP)
/* program to be compiled as "cat" executable (gcc cat.c -o cat) pg. 163 */
int main(int argc, char *argv[ ])
{
   FILE *fp;
   void filecopy(FILE *, FILE *); /* funtional prototype                    */

   if (argc == 1)                 /* no args: copy standard input          */
      filecopy(stdin, stdout);         /* from on left, to on right is common     */
   else
      while (--argc > 0)
        if ((fp = fopen(*++argv, "r")) == NULL)    {
           printf("cat: can't open %s\n", *argv);
           return 1;
        } else                    {
           filecopy(fp, stdout);  /* copy this file to stdout                 */
           fclose(fp);
        }                                 /* loop through all files named        */
   return 0;
}
```

```
/*filecopy: copy file ifp to ofp                                */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Every file open requires resources, and there is a limit on files open at once; good idea to close fp when done (all close at program termination).

FILE structure has buffer for disk data in memory; when putc, may not get written out to file. THUS THE DATA IS NOT SAFELY ON DISK.

This is important to a database, say. Functions fclose() (and fflush()) will flush buffer to disk file.

Look at Appendix B, pg. 241. Appendix B describes the standard C library. B1 contains <stdio.h> stuff.

ALWAYS LOOK AT descriptions in Appendix B, because they often have more information than descriptions in the text itself.

Class 21. Bring glass unix for lecture a week from today!!!

7.6 Error Handling

Trying to fopen a file that does not exist is an error, and there are other errors as well: reading or writing a file without appropriate permission.

With the "cat" program just covered (pg 85, **put up again**), an error executing fopen will write something to stdout; maybe this was redirected to a file.

  %cat fname1 fname2 >fname3

But recall there are <u>three</u> streams opened by the operating system when a program begins execution, stdin, stdout, and stderr.

And stderr usually goes to the screen even if stdout is redirected to a file

  prog . . . >outfile   (redirect stdout to outfile; destroy old outfile)
  prog . . . >&outfile   (redirect stdout <u>and stderr</u> to outfile, destroy old)
                      (The >& redirection works only in the C Shell)
  prog . . . >>outfile   (redirect stdout to outfile; append on end)

Note it is reasonably common to use both > and >& in a single command:

  prog . . . >outfile1 >&outfile2

Then you can search for an error in outfile2, but in any case outfile1 has no error msgs.

How do we rewrite the "cat" program so write error msgs to stderr? Done on pg 163 (<u>open to</u>) of K&R (below). Rewrite fopen() in loop of that program as:

```
        if ((fp = fopen(*++argv, "r")) == NULL     {
            fprintf(stderr, "%s: can't open %s\n", prog, *argv);
            exit(1);
        }
```
Above, the error msg goes to stderr; the variable prog printed out under %s is a char array containing the name of this compiled program, initialized:

  char *prog = argv[0]; /* name invoked for this program   */

The exit(int) function (arg: 0-255) terminates program execution and returns argument to invoking process (debugger, shell program, fork parent)

Of course, a "return value" from a main program would do this as well, but exit() will terminate execution as if we executed a return from main(), and can be called from any nested function! Use only for CRUCIAL errors!

A zero returned by a program means no error; non-zero values mean exceptional condition. You can set up conventions as to what values mean, but it's best to keep the values positive. (Why?)

At the end of the main program (new cat) on pg. 163, have new statement:

```
if (ferror(stdout))     {
    fprintf(stderr, "%s: error writing stdout\n", prog);
    exit(2)
}
```
The function ferror returns non-zero if an error occurred on the stream fp:

```
int ferror(FILE *fp);          /* See B1.7, pg. 248                        */
```

But this doesn't give the actual error number to allow us to tell the user or programmer what problem has occurred. Need Error handling functions.

Error handling functions covered in Appendix B, Section B1.7. (See pg 248) These are errors that can arise from any library calls, not just I/O.

Note there is a problem with program on pg. 163. To handle errors, we should #include <errno.h> (See B1.7, pg. 248). The function ferror() tells us if there is an error indication for a stream (the last one that occurred).

More generally, errno.h contains a macro expression "errno" which can be tested as an int; it is zero if there is no problem and non-zero otherwise.

(Text in B1.7 says errno "may" contain an error number; it will contain one if there has been an I/O or library call error—any such error, unless the error is so serious it has corrupted the error structs.) E.g.:

```
errno = 0; x = sqrt(y);
if (errno)
    printf("sqrt failed, code %d\n', errno);
```

We can use the function perror to write out the error msg associated with errno, but we must test for error right after it occurs to get the right one.

Note too that the most recent error that has occurred ON A STREAM may not be the most recent error that occurred ON THE SYSTEM.

Since perror will print out last error on the system, this might be an error that occurred on a different file!

So do test after each system call or library function - fopen, etc.:

```
if (errno != 0)   {
    perror("Error at myprog: exiting.");
    exit(2);        /* exit from internal function, like "return 2" in main */
}
```

after each system call. perror will print out an errormsg corresponding to integer in errno, as if by: fprintf(stderr, "%s: %s\n", s, "*error message*");

7.7 Line input and output. The standard C library equivalents to getline and putline: fgets and fputs. only slightly different from getline().

```
char *fgets(char *line, int maxline, FILE *fp); (like getline from file)
      (Find this on pg. 164 and in B1.4, pg. 247.)
```

Reads the next input line (including '\n' at the end) from file fp into the char array line; at most maxline-1 chars will be read, then there will be a terminal '\0' added. Returns ptr to line or NULL (means EOF).

For output, the function fputs writes out line to fp. Usually printed out if end with '\n'.

```
int fputs(char *line, FILE *fp);
```

It returns EOF if error occurs (disk fills up?), and zero otherwise. Recall can use perror to print out exact error cause (to stderr, not user screen).

Don't use gets() (stdin) and puts() (stdout). They are confusing in inclusion of newline char. (No maxline in gets.) Always use fgets and fputs.

K&R shows (pg 165) how fgets could be written in terms of getc in the standard library that was present on their system. Very simple.

```c
/* fgets: get at most n chars from iop into char array s                */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n >0 && (c = getc(iop)) != EOF))
        if ((*cs++ =c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;  /* error return if nothing new */
}
```

Class 22.

7.8. Miscellaneous functions (pg 166).
7.8.1 String operations. Talk through. Note strncpy variant, etc. Anything important missing? See Appendix B3, pg 249-250. Several of interest.

One valuable one: char * strstr(cs, ct). Find first example of char string ct in cs. (Look for string ct "01/05/97" in cs named "movie_times".)

Note you can look for ALL occurrences of ct in cs: After you find a match (ptr to char), advance pointer cs to that ptr + 1.

Function char *strtok(s, ct), pg 250, very commonly used indeed. Does the same thing I suggested for strstr(), but does it automatically. E.g.:

```
  char *tok[30]                        /* handle up to 30 tokens            */
  char s[ ] = " , Clinton, 50, 300.25";     /* normally would use fgets    */
  char ct[ ] = " ,";        /* space and comma are two delimiters      */
  int count = 1;

  tok[count-1] = strtok (s, ct);
  if (tok[count-1] == NULL)         /* would indicate no tokens in line    */
     . . .;                         /* take appropriate action                 */
  /* now calls for subsequent tokens                                        */
  while ((tok[count] = strtok(NULL, ct)) != NULL)
     count++;                       /* count is right when fall through     */
```

Following this, we can use sscanf to convert various arguments, e.g., second argument tok[1] under conversion %d.

The mem... functions are very much like the str... functions, except there is no null terminator. A good C library has very efficient mem... functions.

Note difference between memcpy and memmove (overlap objects check). Clearly the size n can be given by a sizeof(struct . . .) reference.

Section 7.8.2. Char Class testing. isalnum(). See App. B2, list pg 249. Less common but interesting: isxdigit(), iscntrl(), isspace(), ispunct().

These are usually faster than obvious tests you would write because they use tests that mask bits in the ascii code.

7.8.3 Ungetc. Had something like this in Chapter 4, ungetchar(). Only have guarantee can push one char back, but usually enough.

7.8.4 Command Execution. The function system(). See pg 253.

   int system(const char *s);

The string s has a system command (pwd, or date, or ls, or ls >fname, or could run a program: prog). Return depends on system. Learn from man.

A very common use is to run a program with parameters in another shell that will do some work I want.

   int a, b;

   char command[MAXCMD];
   sprintf(command, "prog %d %d > prog.out", a, b);
   system(command);

In program compiled as prog, get a and b values through argc and argv[ ].

The system call doesn't return string returned from the command, but by writing > prog.out, will create file with this output.

The calling function can then open this file, input the result and parse it (tokenize it).

7.8.5 Have talked about malloc()/free() before; see pg 252. On pg. 251, see atof, atoi, atol. For itoa, itox, use sprintf. Explain this.

7.8.6 Math functions. Need to have #include <math.h>, and use flag for gcc:
  gcc -lm source.c

7.8.7 Random functions: rand(), srand(); Look at pg. 252 where covered.

Also see bsearch() and qsort() on pg. 253. There is homework on this.

Here is the beginning of the man page on the C Library function qsort. (You should use the man command to get specifications of commands & fns?)
----
man qsort
Reformatting page. Please wait ... done
C Library Functions   qsort(3C)

NAME
    qsort - quick sort

SYNOPSIS
    #include <stdlib.h>

    void qsort(void *base, size_t nel, size_t width,
        int (*compar) (const void *, const void *));

DESCRIPTION
    The qsort() function is an implementation of the quick-sort
    algorithm. It sorts a table of data in place. The contents
    of the table are sorted in ascending order according to the
    user-supplied comparison function.

    The base argument points to the element at the base of the
    table.   The nel argument is the number of elements in the
    table. The width argument specifies the size of each ele-
    ment in bytes. The compar argument is the name of the com-
    parison function, which is called with two arguments that
    point to the elements being compared.

    The function must return an integer less than, equal to, or
    greater than zero to indicate if the first argument is to be
    considered less than, equal to, or greater than the second
    argument.

    The contents of the table are sorted in ascending order
    according to the user supplied comparison function.
EXAMPLES
    The following program sorts a simple array:

```
    static  int intcompare(int *i, int *j)
    {
         if (*i > *j)              OR:
              return (1);          return (*i - *j);
         if (*i < *j)
              return (-1);
         return (0);
    }
```

```
    main()
    {
        int a[10];
        int i;
    a[0] = 9; a[1] = 8; a[2] = 7; a[3] = 6; a[4] = 5;
        a[5] = 4; a[6] = 3; a[7] = 2; a[8] = 1; a[9] = 0;

        qsort(a, 10, sizeof(int), /* Need to cast intcompare */
    (int (*) (void *, void *)) intcompare);
        for (i=0; i<10; i++) printf(" %d",a[i]);
        printf("\n");
    }
```

ATTRIBUTES
    See attributes(5) for descriptions of the following attributes:

```
_____
|_ATTRIBUTE_TYPE___ATTRIBUTE_VALUE_  |
| MT-Level          | MT-Safe        |
|_____|_____|
```

SEE ALSO
    sort(1), bsearch(3C), lsearch(3C), string(3C), attributes(5)

NOTES
    The comparison function need not compare every byte, so
    arbitrary data may be contained in the elements in addition
    to the values being compared.

    The relative order in the output of two items that compare
    as equal is unpredictable.

Here's a more complex example of how to use qsort. Given an array emps[ ] of
struct empstr below, you want to sort them in order by lname.

```
  struct  empstr {
    char lname[12];
    char fname[12];
    int empno;
  } emps[100];
```

Let's say we fill in the first 5 entries of the emps array (subscripts 0 to 4) and just
want to sort the first five elements. The following will work.

qsort(emps, 5, sizeof(struct empstr),
    (int (*) (void *, void *)) cmplname); /* see also App. B.5 */

emps is a pointer (array name) to the base of the array (qsort doesn't recognize the pointer type, of course, but makes it a (void *) pointer.)
The number of elements to sort is 5. The size of an element is the sizeof expression given.

Need function cast or there will be a gcc warning: "passing arg4 of qsort from incompatible pointer type." The cmplname function we define is:

```
int cmplname(struct empstr *p1, struct empstr *p2)  {
  return(strcmp(p1->lname, p2->lname));
}
```

Note that the arguments must POINT TO the elements being compared. Then the function provided does the comparison based on these pointers and returns <0, =0, or >0 according to whether the first argument is <, =, or > the second argument.

Note that we could just as well have compared the elements on the basis of their empno, and then we would sort in order by empno.

To make everything cast properly in the call and do away with the warning, we would write instead when we call qsort:

```
  qsort((void *) emps, 5, sizeof(struct empstr),
      (int (*) (const void *, const void *))cmplname);
```

Now you should also know how to perform bsearch( ) (pg. 243)

Class 23.

Lecture on UNIX Shells (Created by E. O'Neil)

Ref: Glass, Chap. 3, Chap. 6; As before, there's an OLD and a NEW Page #s format: (Old,New), or (2ndEd.pgno,3rdEd.pgno). Also Chapter #s (Old,New).

Look at Glass, (77,147). There we see a nice picture of the relationship between the three most common shells. At UMB we use the C shell, or its descendent, tcsh, the "T shell"; tcsh should show up as a circle around the C shell, since it supports all the C shell actions and some more.

Note the "common core" of both shell variations. Many important features are there, and we'll cover them first, following Glass Chap. (3,4). Then we'll go on to the C shell, Chap. (6,7).

Fig. (3.2,4.2) shows breakdown of core features. We've covered redirection "prog>out", wildcards "ls *.c", pipes "ls -ltl|more", but not the others.

Processes

Basic to UNIX is the idea of a process. Each process contains a program in execution (it might be stopped, but it is living on the system, anyway).

Each process has own memory space with code, data, and a program stack.

Most commands we use daily are programs written in C and have "main(int argc, char *argv[])" to start, to access arguments on their own command line. Even if they aren't written in C, they are given such access.

Processes can give birth to other processes using the fork() system call (424-6,473-6). Then there is a parent and a child process.

Typically the parent keeps track of the child but not vice versa. A common thing for a parent to do is just wait around until the child finishes some work, and exits, pg (427, 476); also, exit() C library function, K&R pg 252.

Shells are just programs that provide the command interface you interact with; each shell runs in its own process for you as a user.

Typically have a shell in a process giving a command interface and sometimes a program running under it (e.g, a command), in its own process.

Shell operations (79,148) (We use the C Shell!)

The shell is a program that is basically an initialization and then a loop over user commands. The shell interprets a line of user input, does whatever that says, and waits for another user input line.

A shell terminates when a user types control-D at the beginning of a new line, or the shell command to exit, typically "exit". (If rlogin, use exit.)

Some shells disable the control-D option or allow you to customize this point. The "logout" command not only causes this shell to go away but in addition everything else in your UNIX session.

Non built-in shell commands: they are programs you run.

Consider the "ls" or "lpr" or "vi" commands, or "myprog" --these are all programs, some in system directories, and "myprog" in your own current directory. The shell simply runs the program in a child process, passing the arguments to it via argc/argv.

Built-in Shell Commands. pg (80,149-50)

echo and cd are built-in shell commands--instead of running a program, the shell program detects these commands in the input line from the user and does the right action itself.

Note that cd needs to be built-in, to change the current directory for the shell process (doing the action in a program run from the shell would only change it for that process. e.g. in vi, can type !cd subdir, but doesn't "last")

When you type "!" (pronounced "bang") before a command, the effect is to start a lower-level process for a temporary shell.

Shell variables.

Glass postpones variables until later, in C Shell (183-7, 258-63). We introduce now to use in examples.

A shell variable or local variable is a name with a value held just in the current shell. Follow this starting on (183,258).

% set x = 5
% set hwdir = ~poneil/cs240/hw4   --C shell: ~user expands to path for user

Once these are defined, we can access their values via $name, and this works in any shell.

% echo cking x: $x $hwdir   -- NOTE not $(x) as would be in makefile
cking x: 5 /home/poneil/cs240/hw4
% ls $hwdir
assignment …
% cd $hwdir

The full syntax and rules are given on pg. (184-185,). In particular, see $name[selector] syntax, or <u>Lists</u>. We will cover this later.

Note that makefiles have a <u>different</u> parser:

CC= gcc   later need to type:
$(CC) -g -o calcit -- WE DON'T NEED THE ()'s IN SHELL VARIABLE USE

<u>Environment variables</u>. (93-5,163-66)

An environment variable is a name with a value that gets communicated from the shell to programs running under the shell, including other shells.

Note for example that another shell is created when you type !ls in email!

There are lots of preexisting environment variables. See (93,164). E.g. TERM is type of your terminal (vt100). These are set in your .login file (executed when you login). Take a look at your .login file and understand what you have there. Responsible for.

To define an env var of your own while using the C shell:

% setenv y 10
% setenv printer lw_office

Their values are accessed just the same way as shell vars:

% echo $y $printer
10 lw_office
% lpr -P$printer *.c

Since they stick with you better, you might like to use env vars for "places to save info" while you work.

Metacharacters table (For the shell) (81,150)

Talked about these before when trying to grep the pattern "U.S."; these are SHELL metacharacters, not grep PATTERN metacharacters on (607,665).

All shells agree on a bunch of special marks and what they mean. You need to know them both for their utility and so you can avoid or sidestep them to get the right characters to arrive as arguments to the program you want to run. We'll discuss many of them.

Wildcards (83-4,153-4)

ls *.?   list filenames with anything before a dot, one char after dot.
lpr *.[ch] print files *.c and *.h: any chars, then dot, then either c or h.
lpr *.[a-z][a-z]   print files with lowercase double-char suffixes.

Pipes (84-6,154-6). Know this. Ignore mention here of awk, unless you need a way to print out just one or several columns for a file

Command substitution: `command` (86-7,156-7)

Enclose command in back-single-quotes, `command`, returns text from output into command line, as if you substituted text for `command`

A favorite use of this is to remember spots in the filesys where you are now, before you cd elsewhere, so you can get back later.

```
% pwd  <---- print working directory
/nfs/gnu/gcc2.11.5/src/gdb/arch/i386/new
(You, thinking . . . how can I remember this????  AHA!)
% setenv g `pwd`              Note: setenv is C-shell specific
% echo $g                     prove you have it saved in g
/nfs/gnu/gcc2.11.5/src/gdb/arch/i386/new
% cd ~/otherstuff
...later...
% cd $g                       get back to hard-to-type spot
...or use it without cd'ing there:
% more $g/README              display README of that dir
```

Scripts (91-2,161-2)

In theory, you should be able to put any command in a file and then execute it from there--this is true if you're careful enough.

Look at (91-2,162) for steps. We've added step 1a here, important for C shell users. It ensures that the shell used to interpret the script is the C shell, not the Bourne shell (often the default)

1. put commands in file fname, usually via an editor
1a. First line of file: #!/bin/csh <-- this starts C Shell even if not in one
                    (See (91-2,162) Example line 2)
2. % chmod +x fname   mark it executable
3. % fname   run it

Example: a "printdisc" command, to print discussion.doc on myprinter. Create a file called printdisc, with the following contents:

% cat printdisc
#!/bin/csh
lpr -Pmyprinter discussion.doc

% chmod +x printdisc   make it executable
% printdisc   use it to print discussion.doc

NOTE CAN LOOP IN SCRIPT! See (198,275) and following.

Background processes (89,159)

If you have work that requires no user interaction, you can spin off a subprocess that runs concurrently with your main shell, typically doing some long-winded processing like preparing a file for printing.

Another long-winded job is finding something by looking at all the filenames in a directory tree--this is a job for the find command (237,95):

% find . -name a.c -print > find.out

looks for file named "a.c" in all subdirs of ".", i.e, current dir. If you leave out the "-print", nothing at all is output. The "-print" specifies that the filename should be printed out. Here the output is redirected to find.out.

But this can take minutes. To free up the main shell from waiting for this to finish, make the find a background process by simply tacking on an "&":

% find . -name a.c -print > find.out&   -- Spin off background process

Now the shell is ready again for your commands. If you later need to kill this background process, use the ps command to find its pid and "kill pid". Use ps again to make sure it's gone.

Quoting (95,166)

As mentioned earlier, there are so many shell metacharacters they often interfere with what you want to do. But you're in charge -- you can tell the shell to ignore the special meaning when you need to.

Strangely enough, single quotes are "stronger" than double quotes in suppressing metachars. They both suppress wildcards (*.c expands to a.c b.c), but single quotes suppress $var substitution as well.

Example. To find all the C sources in all subdirs of the current dir, we could do this:

% find . -name *.c -print
would expand to
find . -name a.c b.c -print (assuming a.c and b.c in the current dir)

But *.c has expanded too soon, so find would be given 5 args. It would complain about syntax, to the great mystery of the user. This is the kind of problem that gives UNIX a bad name.

But here is what we want to do: suppress wildcard expansion by the shell:

% find . -name "*.c" -print   -- Quotes around *.c protect it from the shell
            -- Lets *.c through to find
Single quotes would work here too.

Basic Job Control

You should know how to use ps (97,167-8) (e.g., when use & to run background) and kill (101,171) (when get thrown off machine on home login and come back to find email read-only because email process still running).

On (98,168) the command *sleep* (in examples) is useful.

Example. You want to watch a long-running program, with pid 2345.
% ps -l lgrep 2345 (-l means long listing)

You write a little script in C-shell, to report on it every 5 secs (after pgm starts; *while* is described on (204,281)

```
#!/bin/csh
while (1)
  date >> watchfile
  ps -l|grep 2345 >> watchfile
  sleep 5
end
```

You can run this in the background and kill it when you're done.

Class 24.

Exam 2 soon. Homework 6 is due. Homework 7 due soon. Will cover only a bit of Chapter 8, but you should know parts of Appendix B we've mentioned AND ALSO Appendix A. You must be able to find things in these appendices.

Continuing C Shell Scripts from Glass
Example An improved print script. Earlier we had:

file printdisc:
```
#!/bin/csh
lpr -Pmyprinter discussion.doc
```

Now we would like to make a "print" command that can print any file, so that "print prog.c" does the command "lpr -Pmyprinter prog.c" and so on.   Here it is:

file print:
```
#!/bin/csh
lpr -Pmyprinter $1 (See (95,166) for meaning of $1 . . . $9)
```

The $1 stands for first argument, $2 for second, etc, so "print prog.c" makes $1 = "prog.c".

We can do better: $* stands for all args starting from the first, so we do:

```
#!/bin/csh
lpr -Pmyprinter $*
```

as the next version of print.   Then "print a.c b.c" prints both files.

You might worry that here we are using a metacharacter "*". Do we need to quote it to prevent it from being wildcard-expanded?

The answer is no, here we're talking to the shell, and we are seeing that the shell uses * for more than one kind of "all-of-these" indicators. It's only when we need to get a * *through* the shell to another program that we need to quote it.

Note: this kind of command is often made into an alias, instead of a script:

alias print lpr -Pmyprinter (See Aliases (191,267) & following.)

will do the job. Then "print a.c b.c" will print both files, etc., because the alias expands in place, replacing the "print" with "lpr -Pmyprinter" and leaving the a.c and b.c in the resulting command.

Typically this alias is put in .cshrc so that it is always available.

    E.g.   alias ls   ls -F

The .cshrc command file is run whan you start up after login; or, after an edit, you can say

% source .cshrc

to get the edits to "take" in your shell command environment.

But scripts can do bigger jobs than aliases, so let's return to them. Suppose you want to run a program for several different values of its args. The simplest way is to use an editor to write out all the ways:

#!/bin/csh   -- file runprog.csh: "runprog.csh" does 4 runs of myprog

myprog small 10 (two arguments are in argv[ ])
myprog small 20
myprog large 10
myprog large 20

Another way is to pass the args through to runprog.csh:

#!/bin/csh

-- now "runprog.csh small large 10 20" does the same 4 runs

myprog $1 $3
myprog $1 $4
myprog $2 $3
myprog $2 $4

Now you can tune the run as needed, for example doing

runprog.csh small large 15 30

Or you can use looping in C Shell (See (198,275)):

```csh
#!/bin/csh
foreach size (small large)        --using a list in a foreach
  foreach n (10 20)
    myprog $size $n
  end
end
```
This is clearly a good way to go if a larger number of cases are needed.

You can generate the 10, 20, 30, sequence using arithmetic in the C Shell— big advantage of the C shell over the Bourne shell: it can do arithmetic but the Bourne shell can't. See example on pg. (204,281) for how to do this.

Class 24. (Not on Exams: Relax!)

Chapter 8. UNIX System Interface: C Library functions & UNIX System calls

The C library functions are uniform across UNIX OS, written using UNIX system calls; UNIX system calls use NOTHING -- they are at the lowest level.

Can print descriptions using man online. There is no index to the names of the calls, but you can list all file names for functions in each category.

%cd /usr/man/man2   (directory of UNIX system calls files)
%cd /usr/man/man3   (directory of C library functions files)

Then the ls command will give you the names of all the files, which gives you the names of the functions that exist. See next pg for fork.

(Already seen early part of man page for C Library function qsort.)

Fork is a particularly basic "UNIX system call" by which your program creates another process that will run independently from you.

Idea: the shell is a command interpreter living in a process; when you run a program the shell performs a fork to run the program.

The parent process for your running program is the shell; when your program forks a process, it has you as parent (the shell as grandparent). mail runs its own process. When !ls, forks child process to ls; !cd doesn't STICK.

After the fork occurs, the child process seems to have EXACTLY the same program environment that created the new process (memory, data, etc.) See Glass pg. 424.

How you tell the different is the returned value pid_t is zero if you're the child and the pid of child if you're the parent. (SF idea: how do you know if you're the clone? Parents keep track of child processes, not vice-versa.)

cs240. Am examples man page for UNIX System Call: fork .====
terminus(3)% man fork (There's a new version in man now.)
Reformatting page. Please wait ... done

FORK(2V)   SYSTEM CALLS   FORK(2V)

NAME: fork - create a new process

SYNOPSIS: int fork()

SYSTEM V SYNOPSIS: pid_t fork()

DESCRIPTION
    fork() creates a new process. The new process (child process)
    is an exact copy of the calling process except for the following:

    + The child process has a unique process ID. The child process
      ID also does not match any active process group ID.

    + The child process has a different parent process ID (the pro-
      cess ID of the parent process).

    + The child process has its own copy of the parent's de-
      scriptors. These descriptors reference the same underlying
      objects, so that, for instance, file pointers in file objects
      are shared between the child and the parent, so that an
      lseek(2V) on a descriptor in the child process can affect
      a subsequent read(2V) or write(2V) by the parent. This de-
      scriptor copying is also used by the shell to establish
      standard input and output for newly created processes as well
      as to set up pipes.

    + The child process has its own copy of the parent's open direc-
      tory streams (see directory(3V)). Each open directory stream
      in the child process shares directory stream positioning with
      the corresponding directory stream of the parent.

    + All semadj values are cleared; see semop(2).

    + The child processes resource utilizations are set to 0; see
      getrlimit(2). The it_value and it_interval values for the
      ITIMER_REAL timer are reset to 0; see getitimer(2).

    + The child process's values of tms_utime(), tms_stime(),
      tms_cutime(), and tms_cstime() (see times(3V)) are set to zero.

    + File locks (see fcntl(2V)) previously set by the parent are not
      inherited by the child.

    + Pending alarms (see alarm(3V)) are cleared for the child process.

FORK(2V)   SYSTEM CALLS   FORK(2V)

   + The set of signals pending for the child process is
     cleared (see sigvec(2)).

RETURN VALUES
     On success, fork() returns 0 to the child process and returns
     the process ID of the child process to the parent process. On
     failure, fork() returns -1 to the parent process, sets errno to
     indicate the error, and no child process is created.

ERRORS
     fork() will fail and no child process will be created if one or more
     of the following are true:

     EAGAIN   The system-imposed limit on the total number
              of processes under execution would be
              exceeded. This limit is determined when the
              system is generated.

              The system-imposed limit on the total number
              of processes under execution by a single user
              would be exceeded. This limit is determined
              when the system is generated.

     ENOMEM   There is insufficient swap space for the new process.

SEE ALSO
     execve(2V), getitimer(2), getrlimit(2), lseek(2V), read(2V),
     semop(2), wait(2V), write(2V)

--EOF: /tmp/man.22130--

END OF MAN PAGE
==
The child will note a value of zero returned from the fork call and go do what the
child process is supposed to do, eventually exiting.

Note that the C library function system(char *s) (K&R pg. 167), which can run any
UNIX command, creates a new Shell program to run the command.

The name "system" for this function is a confusing one: it is NOT a system call. We refer to it as the system function in the C library.

K&R, in Chapter 8 (pg 171, midpage), mentions a header file, syscalls.h, that gives functional prototypes for system call functions.

Originally, there was NO SUCH HEADER (still unused): NO FUNCTIONAL PROTOTYPES needed for system calls; the standard is pre-ANSI C where the arguments for the functions are typed AFTER the function name.

E.g., pg 170, 8.2 would be
  int read(fd, buf, n)
  { int fd, n; char * buf;

The other thing to understand about a system call is that Operating System security is based on not allowing the caller to play any tricks.

When a program performs a system call, the processor performs a TRAP (flow of control stops, as though an error had occurred: divide by zero).

The trap handler in the "UNIX Kernel" looks at the processor stack to see which system call number was invoked; the system call then starts running in a privileged mode: e.g., it can modify memory locations that control your process.

It would be a bad thing to allow general program flow of control to run in this privileged mode, because it might be able to take over the processor, keep other processes from running, destroy the system disk, etc.

In Chapter 8 of K&R, some example system calls are covered that we will not have time to treat in detail, but here are some of the ideas.

NOTE that we DON'T want you to use these system calls for I/O. They're too low-level! We just want you to see how C Library Functions are implemented using System Calls. DON'T USE IN PROGRAMS ON EXAMS!

Open K&R to Section 8.1: File I/O (pg. 169-70). Note that file descriptors for system calls are simply integers (Not FILE *).

Each such file descriptor returned by a call to creat( ) or open( ) (pgs 172-173) is a subscript into an array of structs that contain info about I/O.

See Section 8.3, pg. 172 for how to open a file (PLACE ON BOARD):

```
   int fd;
   int open(char *name, int flags, int perms);      /* E.g. functional prototype    */
   fd = open(name, flags, perms);                    /* opens a file                 */
```

In arguments above, "name" is name of the file to open, "perms" is always zero if file exists, "flags" is an int, an OR of flag symbolic constants such as:

O_RDONLY   open for reading only
O_WRONLY   open for writing only
O_RDWR   open for both reading and writing

Now see pg. 176, contents of <stdio.h> (supports C Library functions fopen, fgets, fputs, fread, fwrite, getc, putc, etc.) Note enum_flags in middle of page. Not the same as O_ flags above, because those are for system calls.

However, probably O_RDONLY is 01 and O_WRONLY is 02 and O_RDWR is 03. System calls use the same _iobuf struct array elements to have file open.

You can cd to the UNIX system directory /usr/include/stdio.h for stdio.h; see the sub-directory /usr/include/sys/fcntl.h for the O_ flags.

Instead of open, might want to create a new file and open it for I/O:

```
   int creat(char *name, int perms);
   fd = creat(name, perms);
```

Note both open and creat return -1 (EOF) if failure to open. If a file already exists with this name, not an error: creat will discard prior contents.
Back in Section 8.2, pg 170, see system calls to read and write files. The examples given are confusing (neither calls nor functional prototypes).

#define BUFSIZE 1024

```
int read(int fd, char *buf, int n);                 /* functional prototypes        */
int write(int fd, char *buf, int n);

int x, y, fd, perms = 0;
char buff[BUFSIZ], fname[ ] = ". . ."; /* BUFSIZ == 1024                 */

fd = open(fname, O_RDONLY, perms);
x = read(fd, buff, BUFSIZE);
```

The read system call actually reads out of another buffer (the system buffer for I/O), and brings the required data into process local memory. The request MAY make the system perform a REAL disk I/O (expensive).

Databases use special "raw devices" in the open command to avoid system buffering: instead of fname, use /dev/DISK001 (a named disk device).

Then read( ) will actually read from disk (better have good sized buffer).

Now: How are C Library calls implemented in terms of System Calls?

Note that fopen in the C library uses these system calls: when you try to open a file and it doesn't exist, really invokes a system call, creat.

See Section 8.5, pg 176, the stdio.h contents (simplified). Note symbolic constants at the top, including NULL, EOF, BUFSIZ, and OPEN_MAX.

Of course stdio.h is included in a user program. What is the maximum number of files a user will be able to open? (Actually depends on the type of machine: defined with #if-#elif-#endif: See /usr/include/stdio.h.)

Read through more of stdio.h, especially the _iobuf struct, typedef name FILE. Recall in Library Function the file descriptor is of type FILE *.

Note declaration right after that struct definition says array of FILE type structs named "_iob" is extern (not part of header file naturally, but user can access it. It has OPEN_MAX entries).

And stdin, stdout, and stderr use the first three entries of these arrays.

Note the entry (of struct type _iobuf) contains information on fd (the array subscript) and lots of info on the buffer (1024 bytes) and pointer into the array and how many characters are left.

See how getc(p) is a macro (recall p is of type FILE *):
  (--(p)->cnt >= 0 ? (unsigned char) *(p)->ptr++ : _fillbuf(p))

getc( ) brings back one char from buffer if still has one, otherwise calls _fillbuf, which fills the area pointed to by *base from disk and returns the first character. See pg 178 for _fillbuf.

On page 177, fopen looks through the array _iob with a pointer fp (points to FILE) until fp->flag doesn't have the _READ or _WRITE flag turned on.

This means that an empty slot in _iobuf has been found.

Now if *mode == 'w', we call creat to create a new file. And so on.

On page 178, see function _fillbuf mentioned above to fill a buffer, that is pointed to by "base" in the FILE struct pointed to by argument fp.

Thus we see (halfway down), if no buffer exist yet:

   . . . fp->base = (char *) malloc(BUFSIZ) . . .

Then a bit later, the read command fills the buffer with data read from fp->fd, the UNIX file descriptor int.

This section tells you what the C library I/O functions ACTUALLY DO.

It's tremendously enticing for people who always want to get to the root of things, to understand the lowest level details of how C works.

In Section 8.6, we see an example of how system calls would be used to walk directories.