LECTURE 24

VARIABLE-LENGTH ARGUMENT LISTS (K&R § 7.3)

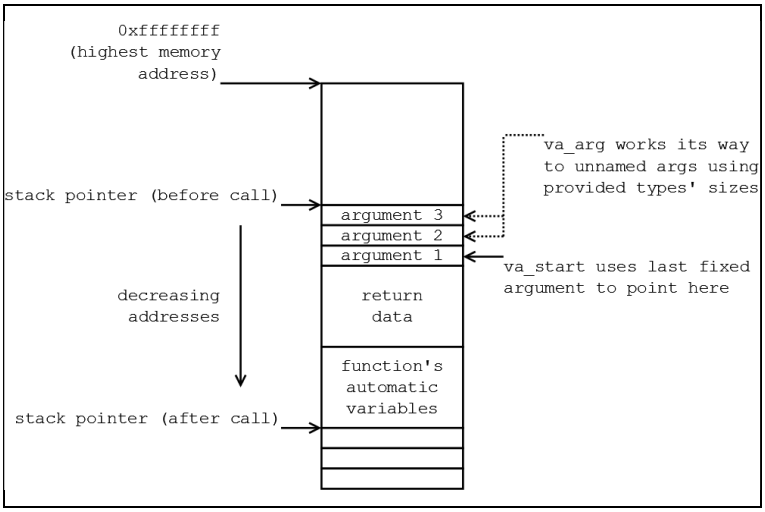
Both printf and scanf have an argument (the format string) that defines the number and type of the remaining arguments in the list:

```
printf("Age: %d\n Height: %d\n", 25, 175)
```

"Age: %d\n Height: %d\n"	25	175
argument 1	argument 2	argument 3

C does not support multiple declarations of the same function with different formal parameters (i.e., you cannot overload functions in C).

To achieve this functionality in C:	
printf is declared as follows: `int printf(char *fmt, ...);`	
The elipsis (...) means that there can be additional arguments of unknown number and type.	
The following are defined in the stdarg.h header and provide the means to step through the unnamed arguments:	
va_list ap;	a va_list is a pointer that will point at each unnamed argument in order.
va_start(ap, fmt)	va_start is a macro that will initialize ap to point at the first unnamed argument there must be at least one named arg (va_start uses this to find where to point ap)
va_arg(ap, int)	each call to va_arg returns one argument and steps ap to the next you must provide the type of the argument so va_arg knows what to return and how far to step in memory to get to the next argument
va_end(ap);	does whatever cleanup is necessary must be called before function returns
Note:	This requires `#include <stdarg.h>`



Example	
`void foo (int n, ...)`	note ellipsis ...
`va_list ap;`	variable name ap
`va_start(ap, n);`	n is last named arg
now ap points just before first unnamed arg	

`ival = va_arg(ap, int);`
`fval = va_arg(ap, float);`
`sval = va_arg(ap, char *);`
	each call to va_arg( ) advances pointer ap by one argument and returns value by type.

`va_end(ap);` `}`	function must clean up before returning:

FILE ACCESS (K&R § 7.5)

We have used files with our programs so far by redirecting input and output using > and < at from the shell.

C provides the library function fopen to open a file so that it may be read or written.

<stdio.h> contains the definition of a file structure using the name FILE. We use this structure to declare a file pointer (e.g., FILE *fp;).

fopen returns a file pointer that we'll use to read or write.

fopen returns NULL if it fails to open a file.

fopen is called with two arguments:

```
FILE *fopen(const char *name, const char* mode);
```

char *name	a string containing the name of the file
char *mode	a string indication the intended use of the file: "r" read "w" write "a" append "r+" open file for update (i.e., read/write) "w+" create text file for update (discards previous contents, if any) "a+" append; open or create text file for update, writing at end see pg.242 for more detail

```
FILE *fp;
fp = fopen("file.txt", "r");
```

"r" mode means we are using the file like stdin

To read a character from a file, can use getc:

```
int getc(FILE *stream);
```

Open a file in "r" mode, as above, then:

```
char c;
c = getc(fp);
```

c now contains a character read from the file (or EOF if getc fails).

```
FILE *fp;
fp = fopen("file.txt", "w");
// or
fp = fopen("file.txt", "a");
```

"w" or "a" mode means we are using the file like stdout

To put a character to a file, can use putc:

```
int putc(int c, FILE *stream);
```

Open a file in "w" or "a" mode, as above, then:

```
char c = 'x'; // this is the int argument (ASCII code)
status = putc(c, fp);
```

returns the character written as an unsigned char cast to an int (or EOF if putc fails).

When we fopen a file in "w" or "a" mode:	
If the file does not already exist:	it will be created
If the file already exists:	"w" mode will throw away the old contents, overwriting the file
	"a" mode will append new contents to the end of the existing file

When you're done with the file, call fclose:

```
int fclose(FILE *stream);
```

For our example above:

```
status = fclose(fp);
```

fclose returns:

zero for success or EOF if an error occurs.

Every file open requires resources and there is a limit on the number of files open at any one time:
-close each fp when done using it
-close all files before program termination

the FILE structure has a buffer for disk data in memory

When putc returns data may not get written to file;
THE DATA IS NOT SAFELY ON DISK YET!

fclose() and fflush() flush buffer to disk.

FPRINTF / FSCANF

We've now seen basic formatted input / output:

output	int printf(const char *format, ...) send formatted output to stdout.
input	int scanf(const char *format, ...) read formatted input from stdin.

Formatted input / output on strings:

output	int sprintf(char *str, const char *format, ...) send formatted output to a string
input	int sscanf(const char *str,const char *format, ...) read formatted input from a string

We also have formatted input from streams:

output	int fprintf(FILE *stream, const char *format, ...) stream is a pointer to the FILE object that identifies the stream. format is the usual formatting string.
input	int fscanf(FILE *stream, const char *format, ...) [as above]

ERROR HANDLING (K&R § 7.6)

When a C program is started, the operating system opens three files and provides file pointers (FILE *) to them:
1) stdin
2) stdout
3) stderr

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

This is useful when working with files because we could have various errors, for example:
-trying to open a file that doesn't exist
-reading or writing without appropriate permission

How do we send our error messages to stderr?

Example:
(a program that operates on several files, e.g., cat)

```
char *prog = argv[0];  /* pick up command name */
if ((fp = fopen(*++argv, "r")) == NULL) {
  fprintf(stderr, "%s: can't open %s\n", prog, *argv);
  exit(1);
}
```

prog	pointer to char array containing the name used to invoke this program, e.g., cat
*argv	pointer to char array containing the file name that couldn't be opened (i.e., one of the elements of argv[])
exit	The exit(int) function (arg: 0-255) terminates program execution and returns argument to invoking process. 0 usually means your program executed successfully, nonzero values are used as error codes (usually positive numbers). exit will terminate execution as if we executed a return from main(). It can be called from anywhere in the program.

Examples of how to use exit() values:

UNIX conditional sequence &&
```
% gcc myprog.c && a.out
```
Second program executes only if first returns = 0

UNIX conditional sequence ||
```
% gcc myprog.c || echo compilation failed
```
Second program executes only if first returns != 0

To handle errors, use:
```
#include <errno.h>
```

ferror()	tells us the last error that occurred for a stream.

```
if (ferror(stdout)) {
  fprintf(stderr, "%s: error writing stdout\n", prog);
  exit(2);
}
```

If not exiting on error, to avoid retaining a stale error value from a previous failed operation use:
```
void clearerr(FILE *stream);
```
E.g.,
```
clearerr(stdout);
```

errno.h contains a macro errno that can be tested;

0	non-zero
no problem	problem

We can use the function perror to write out a standard error message associated with errno, but we have to test for this error right after it occurs to get the correct error:
```
if(errno != 0) {
  perror("Error at myprog: exiting.");
  exit(2);
}
```

perror will print out a standard error message corresponding to the integer in errno, as if executing:
```
fprintf(stderr, "%s: %s\n", s, "error message");
```
(where s is the string called with perror above)

stdio.h provides the ferror function:
```
int ferror(FILE *stream)
```

ferror returns a nonzero value if the specified stream's error indicator was set.

If not exiting on error, to avoid retaining a stale error value from a previous failed operation use:
```
void clearerr(FILE *stream);
```

```
if (ferror(stdout)) {
  fprintf(stderr, "%s: error writing stdout\n", prog);
  exit(2);
}
clearerr(stdout);
```

if we include the errno.h header, we can use errno.

errno	a macro that can be tested: it is 0 if there is no problem nonzero otherwise

the function perror (in stdio.h), prints a descriptive error message to stderr:
```
void perror(const char *str)
```

prints to stderr str followed by standard error message.

```
if(errno != 0) {
  perror("Error at myprog: exiting.");
  exit(2);
}
```

what happens is as if we wrote:
```
fprintf(stderr, "%s: %s\n", s, "error message");
```
where s is "Error at myprog: exiting."
and "error message" is the error message associated with errno.