LECTURE 23

GREP (Glass, pg 109)

This is a command line utility for filtering the contents of a file. It finds matches for a specified string in identified filenames.

If no filename is provided, grep searches stdin.

grep options	
-i	ignore case ("TEXT" same as "text")
-n	adds line numbers to display
-v	gives only lines that don't match
-w	only matches complete words

Example:
Say we have a file "numbers.txt" with the following text:

```
line one
line two
line three
line four
```

We can use grep to find any lines with our search word:

`$ grep two numbers.txt`	`line two`

`$ grep line numbers.txt`	`line one`
	`line two`
	`line three`
	`line four`

`$ grep -n three numbers.txt`	`3:line three`

STANDARD LIBRARY FUNCTIONS

K&R Appendix B (page 241) documents C's standard library functions.

You can fine this documentation online at:
http://www.digitalmars.com/rtl/stdio.html

On UNIX, you can use grep to find them in the .h files, e.g.:

```
$ grep strcmp /usr/include/*
```

(usr/include is a standard location for header files on most UNIX systems)

You can also use the manual command, e.g.:

```
$ man strcmp
```

UNIX "Pipes" (Glass, pg 175)

We've used redirection often:

```
$ command < filename1 > filename2
```

(run command with filename1 as input, and send the output to filename2)

The shell allows you to use the standard output of one process as the standard input of another using the pipe character: | (shift backslash).

```
$ command1 | command2
```

(the output of command1 flows directly into the input to command2)

Example:
(wc is the unix word count utility, its -w option prints the wordcount)
If we are in a directory containing four files, the following pipe command will print 4:

```
$ ls | wc -w
```

STANDARD INPUT / OUTPUT (K&R § 7.1)

A text stream consists of a sequence of lines.	Each line ends with a newline character.

The most basic input mechanism is reading a single character from standard input. We have the getchar function for this:

```
char getchar(void)
```

To send a single character to stdout, we use putchar:

```
void putchar(char)
```

We can adjust what input stream is treated as stdin by using redirection or pipes:

```
$ ./tail <tail.in | more
$ cat filename1 | ./tail >filename2
```

FORMATTED OUTPUT (K&R § 7.2)

We have a lot of experience using the output function printf:

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

printf converts, formats, and prints its arguments on the standard output as directed by the format argument.

printf has a variable length argument list (the number of arguments after the format string depends on the number of % conversions it contains).

printf returns the number of characters it printed (we have not used this so far).

Each conversion specification may contain certain characters between the % and the conversion character.

- (minus sign)	left adjust printing of argument
m (integer m)	minimum field width
. (dot)	separates min field & precision
p (integer p)	for string: max chars for number: min digits
h or l (letters)	h for short int l for long int

Must be in the following order (with %d as an example):

```
%[-][m][.][p][h|l]d
```

There can be no spaces inside the conversion specification.

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);
```

We can print a string literal as the format string with no %s:

```
printf("hello, world!\n");
```

The following is allowed but NOT RECOMMENDED:

```
char s[] = "hello, world!\n";
printf(s);
```

Why? Because if string s has a % character in it, printf will look for another argument after s (which is the format string).

For this reason, it's SAFER to use:

```
printf("%s", s);
```

string precision w/ printf on "hello, world" (colons used to show leading/trailing space)		
%s	:hello, world:	print the full string
%10s	:hello, world:	print the string, use at least width of 10
%.10s	:hello, wor:	print up to 10 characters of the string
%-10s	:hello, world:	print the string left adjusted, use at least width of 10,
%.15s	:hello, world:	print up to 15 characters of the string
%-15s	:hello, world :	print the string left adjusted, use at least width of 15
%15.10s	: hello, wor:	print up to 10 characters of the string, use at least width of 15
%-15.10s	:hello, wor :	print up to 10 characters of the string left adjusted, use at least width of 15

sprintf (Appendix B, pg 245)	

Function sprintf is similar to printf, but it writes to a specified string and adds a trailing '\0':

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

(the result of applying the format and arguments will be stored at the location pointed to by string)

Note:	int return value does not include trailing '\0'

Use sprintf() to print int into a string using %d or %x.

FORMATTED INPUT (K&R § 7.4)

scanf is the opposite of printf; it reads variables from stdin using a conversion format string.

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

scanf's return value is the number of successfully scanned tokens.

scanf fails when it can't parse any value brought in from stdin according to the specified format.

scanf must be called with a pointer to each variable so that values can be set by scanf.

Example:

```
int age;
int weight;
int count;
while(some condition) {
  printf("Input your last name, age, and weight:\n");
  // assume lname is a char array w/ enough space
  count = scanf("%s %d %d", lname, &age, &weight);
```

scanf allows you to read in an int or a double as a number (as opposed to reading a string and then converting it yourself).

scanf reads a character string, but it's able to do its own conversion to int or double.

There is a problem with scanf, however:
it ignores '\n' characters.

This can get confusing because it's possible for our input to get out of sync if the user makes a mistake.

Example:

```
Input your last name, age, and weight:
$ Smith 23
(user presses return and get no response)
$ Smith 23 185
(user tries again, remembers to input weight this time)
```

scanf sees:
Smith 23 Smith
It assumes the second Smith is a bad weight value and returns 2 as the number of successfully scanned tokens.
The re-entered 23 will be seen as last name and 185 will be seen as age in next prompt/scanf loop.

How do we get around this?

Use scanf for programs needing only one input item.

For more complicated input, we can read a line into an array and use sscanf() to parse the arguments in the line.

sscanf works like scanf, but it reads from a specified string with a trailing '\0':

```
int sscanf(char *string, char *format, &arg1, &arg2, ...)
```

Recall how we wrote function atoi, axtoi to convert a decimal or hex character string s to an integer i?

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

Note:	Note: with both scanf and sscanf, if you put specific characters in the format string, the functions must see exactly those specific characters in the user input.

```
count = sscanf(s, "%d/%d/%d", &month, &day, &year);
```

The input must have the slashes, e.g.:

```
1/1/16
```

If the input is not formatted like this, the count value returned by sscanf is less than 3.