Redirecting I/O connections

- Typically these are connected to the terminal so the user can see what the program prints, and type stuff into the program.
- The user can change (redirect) the standard I/O connections for a program before it runs, so that they are not connected to the terminal. Instead, they can be connected to a file, or even another program.
- Redirecting the standard I/O connections is handled by the shell, the UNIX program that gives you the prompt at which you type commands.

Redirecting I/O connections...

- Different shells have different syntax for redirection.
- The C-shell (csh) uses the following:
  
  `$ a.out > out`

  This command causes the shell to run the program a.out, but to connect the program’s stdout to the file out. Everything the program prints to stdout is written to the file. If the file doesn’t exist, it is created, and if it does exist, its contents are overwritten. stdin and stderr remain connected to the terminal.

UNIX I/O

- Every UNIX process (running program) has three standard I/O connections to the outside world:
  
  **stdin**: Standard in. When the user types stuff to the terminal, it appears on the stdin I/O connection.

  **stdout**: Standard out. When the program prints something to the stdout connection it typically shows up on the terminal.

  **stderr**: Standard error. The program prints its error messages to stderr, which also typically shows up on the terminal.
Redirecting I/O connections...

- Both stdout and stderr are written to the file out.
  
  $ a.out >& out

- stdin is connected to the file in. Instead of reading what the user types, the program reads it from the file in. It is an error if in doesn’t exist.
  
  $ a.out < in

- stdin, stdout, and stderr are all redirected. stdin is connected to the file in, stdout and stderr to the file out.
  
  $ a.out < in >& out

Redirecting I/O connections...

- Redirection can also be done to another program. This creates a pipeline, because the stdout of one program is connected to the stdin of another:

  $ grep Heap heap.c | wc

The stdout of the grep program is connected to the stdin of the wc program. This pipeline tells you how many times the string "Heap" appears in heap.c.
- A percent sign ‘%’ indicates the start of a conversion specification. A conversion specification causes `printf` to convert its next parameter to an ASCII string and print it:
  1. `%d` – signed decimal integer
  2. `%u` – unsigned decimal integer
  3. `%x` – hexadecimal integer (characters in lower-case)
  4. `%X` – hexadecimal integer (characters in upper-case)
  5. `%s` – null-terminated string
  6. `%f` – floating-point number
  7. `%p` – pointer
  8. `%%` – percent sign (doesn’t consume parameter)

- Conversion specifications can be very complicated, and in general, have the following elements after the ‘%’, in order:
  - Flags modifier. Used to convert longs and shorts.
  - Flags. Precision (period followed by an optional decimal).
  - Size. Number of digits to the right of the decimal point, or number of characters from string.

- `printf` has a variable number of parameters, as specified by the format string. It returns the number of characters printed.

- The converted value is left-justified within the field.

- Positive numbers are preceded by ‘+’.

- Numbers are padded to the left with ‘0’ instead of space.

- A space is put to the left of numbers without signs.

- Alternate form of conversion.

- Minimum field width (a decimal number). Causes output to be padded to this many characters.
Always check the return value from `scanf` to verify that the input was converted properly.

Example:
```c
char name[21];
int value;
n = scanf(" %20s", name);
n = scanf(" %d", &value);
```

`fprintf`, `fscanf`

```c
int fprintf(FILE *stream, char *fmt, ...);
int fscanf(FILE *stream, char *fmt, ...);
```

`fprintf` and `fscanf` work like `printf` and `scanf`, except you can specify which I/O connection (file) to use. What I've been calling an "I/O connection" is represented in UNIX by a pointer to a structure of type `FILE`. This is often called a stream. `stdin`, `stdout`, and `stderr` are pre-defined streams.

```c
fprintf(stderr, "Something went wrong!\n");
fscanf(stdin, " %d", &value);
```

`int scanf(char *fmt, ...)`

- Reads characters from `stdin`, formats them according to the conversion specifiers in the format string, and stores them in the pointers passed as parameters. Returns the number of conversions performed.
- The `fmt` string has the same format as for `printf`, with a few exceptions:
  1. Whitespace matches any amount of whitespace in the input, including none.
  2. `[]` matches a sequence of characters from those between the brackets.
Misc.

- There are many I/O routines in the C library. Refer to the man pages or a reference manual. Here are a few examples:
  
  `fread`, `fwrite` - read and write arrays of data
  `fgetc`, `getc` - reads the next character from a stream
  `fgets`, `gets` - reads a string from a stream (up to newline)
  `fputc`, `putc` - writes a character to a stream
  `fputs`, `puts` - writes a string to a stream

fopen, fclose

- `fopen` and `fclose` are used to open and close new I/O connections. `name` is the name of the file to open, `mode` is how to open it: "r"=read-only, "w"=write-only, "rw"=read/write.

```c
FILE *file;
file = fopen("/tmp/foo", "rw");
(void) fprintf(file, "Hello World\n");
(void) fclose(file);
```

errno

- The global variable `errno` is used by many C library routines to return error codes. The standard error codes are defined in `/usr/include/errno.h`
- The easiest way to deal with `errno` is to use the `perror` function to print a reasonable message to `stderr`

```c
stream = fopen("foo", "r");
if (stream == NULL) {
    perror("Unable to open file");
}
```

sprintf, sscanf

- `fprintf` and `scanf` work exactly like `printf` and `scanf`, except they read write to another string:

```c
int sprintf(char *string, char *fmt, ...);
int sscanf(char *string, char *fmt, ...);

char buffer[100];
sprintf(buffer, "Value = %d", value);
sscanf(buffer, "Value = %d", &value);
```
• If `fopen` fails because "foo" doesn't exist, the output might be:
  
  `Unable to open file: file not found`

• The function `strerror` returns the error message associated with an error code.