09 - Lecture - I/O

Reading
  - Chapter 7

Standard I/O

The C standard library automatically provides every running program
with 3 I/O channels:

```
    stdin (standard input):
      - incoming character stream, normally from keyboard

    stdout (standard output):
      - outgoing character stream, normally to terminal screen
      - buffered until newline comes or buffer is filled

    stderr (standard error):
      - outgoing character stream, normally to terminal screen
      - unbuffered
```

<stdio.h> contains prototypes for standard I/O functions such as
printf(), scanf(), and many many more (see K&R2, appendix B)

Redirection

You can have stdin come from a file instead of the keyboard:

```
        ./isort < file_containing_a_number
```

And have stdout go to a file instead of the screen:

```
        ./isort > my_sorting_result
```

Use "2>" to redirect stderr:

```
        ./myprogram 2> myerrors
```

Use ">>" to append to an existing file:

```
        ./myprogram  >> myoutput_of_multiple_runs

        ./myprogram 2>> myerrors_of_multiple_runs
```

Use "2>&1" to have stderr go to the same place that stdout is going to:

```
        ./myprogram > my_output_and_errors 2>&1
```

  - But the following is not the same thing:

```
        # this does NOT work
        ./myprogram 2>&1 > my_output_and_errors
```

  - So here is how you have your valgrind output appended to your
    README.txt:

```
        valgrind --leak-check=yes ./myprogram >> README.txt 2>&1
```

Pipelines

A pipe connects stdout of one program to stdin of another program:

```
        prog1 | prog2 | prog3
```

You can throw redirections in there too:

```
        prog1 < input_file | prog2 | prog3 > output_file
```

    Or equivalently:

```
        cat input_file | prog1 | prog2 | prog3 > output_file
```

You can include prog1’s stderr in the flow like this:

```
        cat input_file | prog1 2>&1 | prog2 | prog3 > output_file
```

[A little demo of manipulating isort output.]

Formatted I/O

int scanf(const char *format, ...)

```
        int n;
        double x;
        char a[100];

        scanf("%d", &n);

        scanf("%lf", &x);

        scanf("%s", a);  // Never do this; it’s UNSAFE!

        scanf("%99s", a);  // Now it’s safe, but ugly.
```

  - Most conversions skip leading white space ("%c" is an exception).

  - Type "man 3 scanf" for details.

  - Error in K&R2, p157: it says that white spaces in the format
    string are ignored, which is not true.

int printf(const char *format, ...)

  - Common conversion specifiers:

```
        %d: int
        %u: unsigned int
        %ld: long
        %lu: unsigned long

        %s: string (char *)

        %f: double
        %g: double (trailing zeros not printed)

        %p: memory address (void *)
```

- See K&R2, p153-155 for more; type "man 3 printf" for even more.

scanf and printf take variable number of arguments. Such functions
are called variadic functions, and their argument list ends with "..."

- See K&R2, section 7.3 for how to write such a function.

```
int sscanf(const char *input_string, const char *format, ...)
```

- read from input_string rather than stdin

```
int sprintf(char *output_buffer, const char *format, ...)
```

- write to output_buffer rather than stdout
- output_buffer better be pointing to a large enough memory

```
int snprintf(char *output_buffer, size_t size, const char *format, ...)
```

- safer version of sprintf

File I/O

EOF

- A special value indicating end-of-file, usually #defined to be -1.

- getchar(), and other similar functions, returns an "int" rather
  than an "unsigned char", so that it can return EOF.

```
FILE *fopen(const char *filename, char char *mode)
```

- Opens a file, and returns a FILE* that you can pass to other
  file-related functions to tell them on which file they should
  operate. FILE* is an example of an "opaque handle".

- mode:

```
    "r" open for reading (file must already exist)
    "w" open for writing (will trash existing file)
    "a" open for appending (writes will always go to the end of file)

    "r+" open for reading & writing (file must already exist)
    "w+" open for reading & writing (will trash existing file)
    "a+" open for reading & appending (writes will go to end of file)
```

- returns NULL if file could not be opened for some reason

```
char *fgets(char *buffer, int size, FILE *file)
```

- reads at most size-1 characters into buffer, stopping if newline is read (the newline is included in the characters read), and terminating the buffer with '\0'

- returns NULL on EOF or error (you can call ferror() afterwards to find out if there was an error).

- Never use the similar gets() function; it's UNSAFE!

```
int fputs(const char *str, FILE *file)
```

- writes str to file.

- returns EOF on error.

```
int fscanf(FILE *file, const char *format, ...)
int fprintf(FILE *file, const char *format, ...)
```

- file I/O version of scanf & printf

```
fclose(FILE *file)
```

- closes the file

```
stdin, stdout, stderr
```

- These are all FILE* objects that have been pre-opened for you.

- printf(...) is the same as fprintf(stdout, ...)

example:

```
    /*
     * ncat <file_name>
     *
     *   - reads a file line-by-line,
     *     printing them out with line numbers
     */

    #include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char **argv)
    {
        if (argc != 2) {
            fprintf(stderr, "%s\n", "usage: ncat <file_name>");
            exit(1);
        }

        char *filename = argv[1];
        FILE *fp = fopen(filename, "r");
        if (fp == NULL) {
            perror(filename);
            exit(1);
        }

        char buf[100];
        int lineno = 1;
        while (fgets(buf, sizeof(buf), fp) != NULL) {
            printf("%4d ", lineno++);
            if (fputs(buf, stdout) == EOF) {
                perror("can’t write to stdout");
                exit(1);
            }
        }

        if (ferror(fp)) {
            perror(filename);
            exit(1);
        }

        fclose(fp);
        return 0;
    }
```

- Can you find a bug in this program? Try fixing it.

Buffering on standard I/O

Three types of buffering:

```
  (1) Unbuffered     - stderr
  (2) Line-buffered  - stdout when it’s connected to terminal screen
  (3) Block-buffered - all other files
```

fflush(fp) manually flushes the buffer for fp (which is a FILE*).

setbuf(fp, NULL) turns off buffering for fp.

Using standard I/O for binary files

Add ’b’ to mode parameter in fopen():

```
    FILE *fp = fopen(filename, "rb");
    FILE *fp = fopen(filename, "wb");
    ...
```

- In UNIX, there is no distinction between text and binary files, so ’b’ has no effect.

- In Windows, ’b’ suppresses newline translation that it normally performs for text files:

    - when reading, turn "\r\n" into "\n"
    - when writing, turn "\n" into "\r\n"

```
int fseek(FILE *file, long offset, int whence)
```

  - Sets the file position for next read or write. The new position,
    measured in bytes, is obtained by adding offset bytes to the
    position specified by whence. If whence is set to SEEK_SET,
    SEEK_CUR, or SEEK_END, the offset is relative to the start of the
    file, the current position indicator, or end-of-file,
    respectively.

  - returns 0 on success, non-zero on error

```
size_t fread(void *p, size_t size, size_t n, FILE *file)
```

  - reads n objects, each size bytes long, from file into the memory
    location pointed to by p.

  - returns the number of objects successfully read, which may be less
    than the requested number n, in which case feof() and ferror() can
    be used to determine status.

```
size_t fwrite(const void *p, size_t size, size_t n, FILE *file)
```

  - writes n objects, each size bytes long, from the memory location
    pointed to by p out to file.

  - returns the number of objects successfully written, which will be
    less than n when there is an error.