
Formatted Input

- `int scanf (char *format, arg1, arg2, ...)`
 - reads characters from the standard input
 - interprets them according to the specification in format
 - stores the results through the remaining arguments, each of which must be a pointer
 - stops when it exhausts its format string, or when some input fails to match the control specification
 - returns the number of successfully matched and assigned input items; can be used to decide how many items were found.
- `int sscanf (char *string, char *format, arg1, arg2, ...)`
 - scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.
 - blanks or tabs are not ignored
- Show rudimentary.c, ask what the output of the program?

```
/**
 * rudimentary.c
 */

#include <stdio.h>

int main () {
    double sum, v;

    sum = 0;
    // compile time warning and run time segment fault; should be scanf("%lf", &v)
    while (scanf("%lf", v) == 1)
        printf("\t%.2f\n", sum +=v);
    return 0;
}
```

File Access

- **fopen:**
 - Before it can be read or written, a file has to be opened
 - fopen takes an external name like x.c does some housekeeping and negotiation with the operating system, and returns a pointer to be used in subsequent reads or writes of the file.
 - **FILE *fopen(char *name, char *mode)**
 - The pointer called the file pointer
 - Points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred.
 - Users don't need to know the details, since the definitions obtained from <stdio.h> include a structure declaration called FILE.
 - The only declaration needed for a file pointer is exemplified by **FILE *fp;**
 - Notice that **FILE is a type name**, like `int`, not a structure tag; it is defined with a typedef.

- The call to fopen is **fp = fopen(name, mode)**
- **mode**
 - a **character** string indicates how one intends to use the file
 - **read ("r")**, **write ("w")**, and **append ("a")**
 - some systems distinguish between text and binary files; for the latter, a "b" must be appended to the mode string. E.g. "rb" means reading binary file.
 - If a file that does not exist is opened for writing or appending, it is created if possible.
 - Opening an existing file for **writing causes the old contents to be discarded**, while opening for **appending preserves them**.
 - Trying to **read a file that does not exist is an error**. Trying to read a file when you don't have permission is also an error. If there is any error, open will return NULL.
- **int fclose(FILE *fp)**
 - **Free the file pointer for another file**
 - Since most operating systems have some limit on the number of files that a program may have open simultaneously, it is a good idea to free the file pointers when they are no longer needed.
- For **formatted input or output of files**, the functions **fscanf** and **fprintf** may be used.
 - Identical to scanf and printf, except that the first argument is a file pointer


```
int fscanf(FILE *fp, char *format, ...)
```

```
int fprintf(FILE *fp, char *format, ...)
```
 - Show formatIO.c and input.txt, run ./formatIO input.txt output.txt, and show the output.txt.
 - Show formatIO2.c, run ./formatIO2 with the same input.txt, and show the output.txt. Ask why the outputs are different. [%d vs %c]

```
/**
 * formatted file I/O
 * formatIO.c
 */

#include <stdio.h>

int main (int argc, char* argv[]) {
    FILE *fp1, *fp2;
    int value;

    fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "w");
    while (fscanf(fp1, "%d", &value) == 1) {
        fprintf(fp2, "%d", value);
    }

    fclose(fp1);
    fclose(fp2);

    return 0;
}
```

input.txt

```
1
10
100
1000
10000
```

```
$ ./formatIO input.txt output.txt
$ cat output.txt
110100100010000
```

```

/**
 * formatted file I/O
 * formatIO2.c
 */

#include <stdio.h>

int main (int argc, char* argv[]) {
    FILE *fp1, *fp2;
    unsigned char value[20];

    fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "w");
    while (fscanf(fp1, "%c", value) == 1) {
        fprintf(fp2, "%s", value);
    }

    fclose(fp1);
    fclose(fp2);

    return 0;
}

```

input.txt

```

1
10
100
1000
10000

```

```

$ ./formatIO input.txt output.txt
$ cat output.txt
1
10
100
1000
10000

```

* Block IO: Read and write as a stream (binary files)

```

size_t fread(void *array, size_t size, size_t count, FILE *fp);
size_t fwrite(void *array, size_t size, size_t count, FILE *fp);

```

- array: a block of memory with the minimum size of size*count bytes
- size: the size in bytes of each element to be read
- count: number of elements
- return: the total number of elements successfully read
- This is useful if we want to store the values just as they are stored in a program - i.e. using the same bits for the 64-bit double rather than putting it into decimal format and choosing the number of decimal places
- Show fishIO.c, which mixes formatted text and binary data.

- Unformatted file I/O (i.e. just reading in one character a time, regardless of what it is)
 - `int getc(FILE *fp)`
 - return the next character from a file referred to by `fp`
 - it returns EOF for end of file or error
 - `int putc(int c, FILE *fp)`
 - writes the character `c` to the file `fp` and return the character written, or EOF if an error occurs
- Fun facts
 - When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`.
 - `getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follow:


```
#define getchar() getc(stdin)
#define putchar(c) put((c), stdout)
```
 - The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are **constants**, however, not variables, so it is **not possible to assign to them**.
- Example:
 - Implement a C program to realize the command “cat”
 - “cat” can be used in two ways:
 - `$ cat file1 file2`: print the contents of `file1` and `file2` to the terminal
 - `$ cat`: print the standard input contents to the standard output

```
[mbp-190250:code yingli$ cat test1.txt
A
B
C
D
[mbp-190250:code yingli$ cat test1.txt test2.txt
A
B
C
D
1
2
3
4
[mbp-190250:code yingli$ cat
HELLO
HELLO
WORLD
WORLD
[mbp-190250:code yingli$
```

- To implement it, we can either use a conditional statement to choose between `getChar` and `getC` to read the contents from files or standard input or use the file pointers (use regular file pointers for input files and use `stdin` to read from standard input)
- The following program gives an example to use file pointers coupled with `getc` and `putc` to implement “cat”

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        FILE *fp = fopen( argv[i], "r" );
        char c;
        while ((c = fgetc( fp)) != EOF) {
            putchar( c );
        }
    }
}
```

In class we also went over examples of how to intermingle formatted text with binary and an example that we need to make it possible for scanf to be able to detect the end of a token (we can't just print two floats back-to-back - we need something like a space to separate them).