C read/write functions

FILES:

1. `fscanf`
    - **Description**: Reads formatted data from a file.
    - **Notes**:
        - Data is read up until a whitespace character is encountered, which is not stored in the buffer (similar to `scanf`).
        - Whitespace characters are discarded (per the above note), but are considered “read” so the file pointer is advanced as if they were read.
        - The string is always terminated with a null character.
    - **File type**: A structured file (text data) with data separated by spaces, tabs, or newlines.
    - **Formal syntax**: `int fscanf(FILE *stream, const char *format, ...);`
    - **Simple syntax**: `fscanf(file, format, adresses of variables)`
    - **Returns**: Number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion.
    - **Example**:
      Given that the following text file named `file.txt` exists:

      ```
      1 2 3 4 5
      ```

      ```c
      int a, b, c;
      int i, j;
      FILE *file = fopen("file.txt", "r");
      i = fscanf(file, "%d %d %d", &a, &b, &c);
      printf("%d %d %d %d\n", a, b, c, i); // 1 2 3 3
      j = fscanf(file, "%d %d %d", &a, &b, &c);
      printf("%d %d %d %d\n", a, b, c,
              j); // 4 5 3 2 (c is not overwritten; only 2
              values are read)
      fclose(file);
      ```

- **Summary**:
  - Argument order: file, format, adresses of variables
  - Returns: number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion. Example uses of this return value are:

```
int i = fscanf(file, "%d %d %d", &a, &b, &c);
if (i == 3) {
  // success
} else if (i == 0) {
  // did not match any items - total type mismatch
} else if (i == EOF) {
  // error - read failure, or end of file reached
        before we could convert anything
} else {
  // partial success - matched some items, partial
        type mismatch
}
```

2. **`fprintf`**
   - **Description**: Writes formatted data to a file.
   - **Notes**:
     - The newline character is not added to the string (similar to `printf`).
   - **File type**: A structured file (text data) with data separated by spaces, tabs, or newlines.
   - **Formal syntax**: `int fprintf(FILE *stream, const char *format, ...);`
   - **Simple syntax**: `fprintf(file, format, values)`
   - **Returns**: Number of characters written, or a negative value if an output error occurs.
   - **Example**:

```
int a = 1, b = 2, c = 3, i;
FILE *file = fopen("file.txt", "w");
i = fprintf(file, "%d %d %d", a, b, c);
printf("%d\n", i); // 5 (3 digits + 2 spaces)
fclose(file);
```

- **Summary**:
  - Argument order: file, format, values
  - Returns: number of characters written, or a negative value if an output error occurs.

3. **`fgets`**
   - **Description**: Reads a line from a file.
   - **Notes**:
     - The newline character, if found, is stored in the buffer (**unlike `gets`**).
     - The string is always terminated with a null character.
   - **File type**: A structured file (text data) with data separated by spaces, tabs, or newlines.
   - **Formal syntax**: `char *fgets(char *str, int n, FILE *stream);`
   - **Simple syntax**: `fgets(buffer, max_length, file)`
   - **Returns**: `str` if successful, `NULL` if end of file or error occurs before any characters are read.
   - **Example**:
     Given that the following text file named `file.txt` exists (notice there is no newline at the end):

     ```
     1 2 3 4 5
     ```

     ```c
     char buffer[100];
     FILE *file = fopen("file.txt", "r");
     fgets(buffer, 100, file);
     printf("%s", buffer); // 1 2 3 4 5 (no newline since the
             file doesn't have one. If the file had a
             newline, it would be printed. This is different
             from gets.)
     printf("\n"); // optional newline for readability
     fclose(file);
     ```

   - **Summary**:
     - Argument order: buffer, max_length, file
     - Returns: `str` if successful, `NULL` if end of file or error occurs.

4. **`fputs`**
    - **Description**: Writes a string to a file.
    - **Notes**:
        - The newline character is not added to the string (**unlike `puts`**).
    - **File type**: A structured file (text data) with data separated by spaces, tabs, or newlines.
    - **Formal syntax**: `int fputs(const char *str, FILE *stream);`
    - **Simple syntax**: `fputs(string, file)`
    - **Returns**: Non-negative value if successful, `EOF` if error occurs.
    - **Example**:

```c
FILE *file = fopen("file.txt", "w");
fputs("Hello, world!", file);
fclose(file);
```

    - **Summary**:
        - Argument order: string, file
        - Returns: Non-negative value if successful, `EOF` if error occurs.

5. **`fread`**
    - **Description**: Reads data from a file.
    - **File type**: A binary file (unstructured/non-text data).
    - **Formal syntax**: `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
    - **Simple syntax**: `fread(buffer, size, count, file)`
    - **Returns**: Number of items read. Can be less than `count` if end of file is reached or an error occurs.
    - **Example**:
      Given that the following binary file named `file.bin` exists (represented as hex):

```
01 02 03 04 05
```

```c
char buffer[100];
FILE *file = fopen("file.bin", "rb");
```

```
fread(buffer, sizeof(char), 5, file);
printf("%d %d %d %d %d\n", buffer[0], buffer[1],
        buffer[2], buffer[3], buffer[4]); // 1 2 3 4 5
fclose(file);
```

- **Summary**:
  - Argument order: buffer, size, count, file
  - Returns: Number of items read.

6. **`fwrite`**
   - **Description**: Writes data to a file.
   - **File type**: A binary file (unstructured/non-text data).
   - **Formal syntax**: `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
   - **Simple syntax**: `fwrite(buffer, size, count, file)`
   - **Returns**: Number of items written. Can be less than `count` if an error occurs.
   - **Example**:

```
char buffer[100] = {1, 2, 3, 4, 5};
FILE *file = fopen("file.bin", "wb");
fwrite(buffer, sizeof(char), 5, file);
fclose(file);
```

   - **Summary**:
     - Argument order: buffer, size, count, file
     - Returns: Number of items written.

STDIN/STDOUT:

1. **`scanf`**
   - **Description**: Reads formatted data from standard input.
   - **Notes**:
     - Data is read up until a whitespace character is encountered, which is not stored in the buffer.
     - Whitespace characters are discarded (per the above note), but are considered “read” so the file pointer is advanced as if they were read.
     - The string is always terminated with a null character.

- **Formal syntax**: `int scanf(const char *format, ...);`
- **Simple syntax**: `scanf(format, adresses of variables)`
- **Returns**: Number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion.
- **Example**:

```
int a, b, c;
int i, j;
i = scanf("%d %d %d", &a, &b, &c);
printf("%d %d %d %d\n", a, b, c, i); // 1 2 3 3
j = scanf("%d %d %d", &a, &b, &c); // we entered 4 5
        <enter> 6
printf("%d %d %d %d\n", a, b, c, j); // 4 5 6 3
```

- **Summary**:
  - Argument order: format, adresses of variables
  - Returns: number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion. Example uses of this return value are:

```
int i = scanf("%d %d %d", &a, &b, &c);
if (i == 3) {
  // success
} else if (i == 0) {
  // did not match any items - total type mismatch
} else if (i == EOF) {
  // error - read failure, or end of file reached
        before we could convert anything
} else {
  // partial success - matched some items, partial
        type mismatch
}
```

2. ## `sscanf`

- **Description**: Reads formatted data from a string.
- **Formal syntax**: `int sscanf(const char *str, const char *format, ...);`
- **Simple syntax**: `sscanf(string, format, adresses of variables)`

- **Returns**: Number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion.
- **Example**:

```
int a, b, c;
int i, j;
i = sscanf("1 2 3", "%d %d %d", &a, &b, &c);
printf("%d %d %d %d\n", a, b, c, i); // 1 2 3 3
j = sscanf("4 5 c", "%d %d %d", &a, &b, &c);
printf("%d %d %d %d\n", a, b, c,
        j); // 4 5 3 2 (c is not overwritten; only 2
        values are read)
```

- **Summary**:
  - Argument order: string, format, adresses of variables
  - Returns: number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion. Example uses of this return value are:

```
int i = sscanf("1 2 3", "%d %d %d", &a, &b, &c);
if (i == 3) {
  // success
} else if (i == 0) {
  // did not match any items - total type mismatch
} else if (i == EOF) {
  // error - read failure, or end of file reached
        before we could convert anything
} else {
  // partial success - matched some items, partial
        type mismatch
}
```

3. **`printf`**
   - **Description**: Writes formatted data to standard output.
   - **Formal syntax**: `int printf(const char *format, ...);`
   - **Simple syntax**: `printf(format, values)`
   - **Returns**: Number of characters written, or a negative value if an output error occurs.
   - **Example**:

```c
int a = 1, b = 2, c = 3, i;
i = printf("%d %d %d", a, b, c); // 1 2 3
printf("%d\n", i); // 5 (3 digits + 2 spaces)
```

- **Summary**:
  - Argument order: format, values
  - Returns: number of characters written, or a negative value if an output error occurs.

4. ## `gets`

- **Description**: Reads a line from standard input.
- **Notes**:
  - The newline character is not stored in the buffer(**unlike `fgets`**).
  - The string is always terminated with a null character.
- **Formal syntax**: `char *gets(char *str);`
- **Simple syntax**: `gets(buffer)`
- **Returns**: `str` if successful, `NULL` if end of file or error occurs before any characters are read.
- **Example**:

```c
char buffer[100];
gets(buffer);
printf("%s", buffer); // 1 2 3 4 5 (no newline)
```

- **Summary**:
  - Argument order: buffer
  - Returns: `str` if successful, `NULL` if end of file or error occurs.

5. ## `puts`

- **Description**: Writes a string to standard output.
- **Notes**:
  - The newline character is added to the string (**unlike `fputs`**).
- **Formal syntax**: `int puts(const char *str);`
- **Simple syntax**: `puts(string)`
- **Returns**: Non-negative value if successful, `EOF` if error occurs.

- **Example**:

```
puts("Hello, world!"); // Hello, world! (newline)
```

- **Summary**:
  - Argument order: string
  - Returns: Non-negative value if successful, `EOF` if error occurs.

STRING FUNCTIONS:

1. **`strncpy`**
   - **Description**: Copies a string to another string, up to a maximum length.
   - **Notes**:
     - If the length of `src` is greater than or equal to `n`, the string will not be null-terminated.
     - If the length of `src` is less than `n`, the remainder of `dest` will be filled with null characters.
   - **Formal syntax**:
     `char *strncpy(char *dest, const char *src, size_t n);`
   - **Simple syntax**: `strncpy(destination, source, max_length)`
   - **Returns**: `dest`
   - **Example**:

```
char a[100] = "Hello, world!";
char b[100];
strncpy(b, a, 5);
printf("%s\n", b); // Hello
```

   - **Summary**:
     - Argument order: destination, source, max_length
     - Returns: `dest`
   - **Common cases**:
     - Copying a range of character from string `a` to string `b`. Note that the range is inclusive, so `start` and `end` are both copied.

```
strncpy(b, a + start, end - start + 1);
```

2. **`strcpy`**
    - **Description**: Copies a string to another string.
    - **Notes**:
        - The resulting string is null-terminated, unless the `dest` string is not large enough to hold the result, in which case the behavior is undefined.
    - **Formal syntax**: `char *strcpy(char *dest, const char *src);`
    - **Simple syntax**: `strcpy(destination, source)`
    - **Returns**: `dest`
    - **Example**:

```c
char a[100] = "Hello, world!";
char b[100];
strcpy(b, a);
printf("%s\n", b); // Hello, world!
```

    - **Summary**:
        - Argument order: destination, source
        - Returns: `dest`

3. **`strcmp`**
    - **Description**: Compares two strings.
    - **Formal syntax**: `int strcmp(const char *s1, const char *s2);`
    - **Simple syntax**: `strcmp(string1, string2)`
    - **Returns**: Negative value if `s1` is less than `s2`, 0 if `s1` is equal to `s2`, positive value if `s1` is greater than `s2`.
    - **Example**:

```c
printf("%d\n", strcmp("abc", "abc")); // 0
printf("%d\n", strcmp("abc", "abd")); // -1
printf("%d\n", strcmp("abd", "abc")); // 1
```

    - **Summary**:
        - Argument order: string1, string2

        - Returns: Negative value if `s1` is less than `s2`, 0 if `s1` is equal to `s2`, positive value if `s1` is greater than `s2`.
    - **Mnemonic**:
        - The return value of `strcmp`, when compared using </>/= can be thought of as comparing the two strings lexicographically, i.e.:

```
strcmp(s1, s2) < 0 // s1 < s2
strcmp(s1, s2) > 0 // s1 > s2
strcmp(s1, s2) == 0 // s1 == s2
```

4. ## `strcat`
    - **Description**: Concatenates two strings.
    - **Notes**:
        - `dest` must have enough space to hold the result.
        - `dest` must be a null-terminated string.
        - `src` must be a null-terminated string.
        - `dest` will be null-terminated after the concatenation.
        - `dest` will be modified.
    - **Formal syntax**: `char *strcat(char *dest, const char *src);`
    - **Simple syntax**: `strcat(destination, source)`
    - **Returns**: `dest`
    - **Example**:

```
char a[100] = "Hello, ";
char b[100] = "world!";
strcat(a, b);
printf("%s\n", a); // Hello, world!
```

    - **Summary**:
        - Argument order: destination, source
        - Returns: `dest`

5. ## `strlen`
    - **Description**: Returns the length of a string.
    - **Notes**:
        - The null character is not counted.

- **Formal syntax**: `size_t strlen(const char *s);`
- **Simple syntax**: `strlen(string)`
- **Returns**: Length of `s`.
- **Example**:

```
printf("%d\n", strlen("Hello, world!")); // 13
```

- **Summary**:
  - Argument order: string
  - Returns: Length of `s`.

Similarities and quirks:

- Read formatted data:
  - `scanf` - read from standard input, short for `fscanf(stdin, ...)`
  - `fscanf` - read from a file
  - `sscanf` - read from a string
- Read a line:
  - `gets` - read from standard input, quirk: does not store newline
  - `fgets` - read from a file, quirk: stores newline (mnemonic: “dumb” operation, does not assume the user wants to get rid of the newline, better than `gets`)
- Write formatted data:
  - `printf` - write to standard output, short for `fprintf(stdout, ...)`
  - `fprintf` - write to a file
- Write a string:
  - `puts` - write to standard output, quirk: adds newline
  - `fputs` - write to a file, quirk: does not add newline (mnemonic: “dumb” operation, does not assume the user wants to add a newline, better than `puts`)
- Read/write binary data:
  - `fread` - read from a file
  - `fwrite` - write to a file
- Copy a string:
  - `strcpy` - copy a string, quirk: does not check if the destination is large enough
  - `strncpy` - copy a string, quirk: does not null-terminate if the destination is not large enough
- Compare two strings:
  - `strcmp` - compare two strings
- Concatenate two strings:
  - `strcat` - concatenate two strings, quirk: does not check if the destination is large enough

- Get the length of a string:
  - `strlen` - get the length of a string, quirk: does not count the null character

Return values:

- `scanf/fscanf/sscanf` - number of input items successfully matched and assigned, or EOF if read failure occurs before the first conversion
- `printf/fprintf` - number of characters written, or a negative value if an output error occurs
- `fgets/fputs` - `str` if successful, `NULL` if end of file or error occurs
- `fread/fwrite` - number of items read/written
- `strncpy/strcpy/strcat` - `dest`
- `strcmp` - negative value if `s1` is less than `s2`, 0 if `s1` is equal to `s2`, positive value if `s1` is greater than `s2`
- `strlen` - length of `s`