Gorgo C for C++ Programmers

April 11, 2026

10. Standard I/O

C's `<stdio.h>` library is your replacement for C++ `iostream`. It provides `printf` and `scanf` for formatted output and input, file operations with `fopen` and `fclose`, and binary I/O with `fread` and `fwrite`. Everything flows through the `FILE *` type — an opaque pointer to a structure that tracks the state of an I/O stream.

`scanf` for Input

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

You have already seen `printf` for output. For input, C uses `scanf`, which reads formatted data from standard input:

```
#include <stdio.h>

int main(void) {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);
    printf("You entered: %d\n", year);
    return 0;
}
```

Notice the `&` before `year`. Since C is pass by value, `scanf` needs the *address* of the variable so it can store the result there. Forgetting the `&` is a classic bug — the program compiles but crashes or produces garbage at runtime.

`scanf` uses similar format specifiers to `printf`, but not identical ones. Notably, `scanf` uses `%lf` for `double` while `printf` uses `%f`:

```c
char name[50];
double gpa;
scanf("%s %lf", name, &gpa);
```

Note that `name` does not need `&` because an array name already points to the bytes we want to read into. But `gpa` does need a `&`, because `scanf` needs to know where `gpa` is stored to fill it in.

Trap: `scanf("%s", ...)` reads a single word (stopping at whitespace). It also has no bounds checking — it will happily overflow your buffer. Use a width specifier like `%49s` to limit input to 49 characters (plus `'\0'`).

Scan Sets

`scanf` supports **scan set specifiers** with `%[...]`, which let you define exactly which characters to accept. The scan set reads characters as long as they are in the set, and stops at the first character that is not:

```c
char vowels[20];
// reads only vowels, stops at first non-vowel
scanf("%19[aeiouAEIOU]", vowels);
```

A caret `^` at the start of the set **negates** it — read everything *except* the listed characters. This gives you a way to read an entire line with `scanf`, since `%[^\n]` reads everything up to (but not including) the newline:

```c
char line[80];
scanf("%79[^\n]", line);   // reads a full line (up to 79 chars)
printf("You said: %s\n", line);
```

Always include a width limit to prevent buffer overflow, just like with `%s`.

Scan sets work with `sscanf` too. Here is an example that parses a structured string:

```c
char buf[] = "Track 03: 99 Luftballons";
int track;
char title[50];
sscanf(buf, "Track %d: %49[^\n]", &track, title);
// track is 3, title is "99 Luftballons"
```

Tip: `%[^\n]` is the `scanf` way to read a line, but `fgets` is generally safer and simpler for line-oriented input. Use scan sets when you need to parse structured input where only certain characters are valid.

The `%m` Modifier (POSIX)

With `%s` and `%[...]`, you must always provide a buffer that is large enough. The POSIX `%m` modifier (called the **assignment-allocation** modifier) tells `scanf` to `malloc` the buffer for you. Instead of passing a `char[]`, you pass a `char **` and `scanf` allocates exactly enough memory:

```c
char *line = NULL;
scanf("%m[^\n]", &line);       // scanf mallocs the buffer
printf("You said: %s\n", line);
free(line);                     // you must free it
```

Notice `&line` — `scanf` needs a pointer to your `char *` so it can fill it in with the address of the newly allocated buffer. This eliminates buffer overflow risk entirely, since the buffer is always the right size.

`%ms` works the same way for single words:

```c
char *word = NULL;
scanf("%ms", &word);   // reads one word, malloc'd to fit
free(word);
```

Tip: `%m` is a POSIX extension (available on Linux, macOS, and most Unix systems) and is not part of the C standard. It will not work with MSVC on Windows. When portability is not a concern, `%m` is an excellent way to avoid buffer sizing headaches.

`stdin`, `stdout`, and `stderr`

When your C program starts, three streams (of type `FILE *`) are already open:

Stream	Purpose	C++ equivalent
`stdin`	Standard input (keyboard)	`std::cin`
`stdout`	Standard output (screen)	`std::cout`
`stderr`	Standard error (screen)	`std::cerr`

`printf(...)` is actually shorthand for `fprintf(stdout, ...)`. You can write to `stderr` for error messages:

```c
fprintf(stderr, "Error: file not found\n");
```

Error messages sent to `stderr` are not affected by output redirection (`./program > output.txt` only redirects `stdout`), so error messages still appear on the screen. `./program 2> err.txt` will redirect errors to `err.txt` and `stdout` will appear on the screen.

`fprintf` and `fscanf`

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

`fprintf` and `fscanf` are the file versions of `printf` and `scanf`. They take a `FILE *` as the first argument:

```c
fprintf(stdout, "Hello\n");           // same as printf("Hello\n")
fprintf(stderr, "Something broke\n"); // write to stderr
```

More usefully, you can use them with files you have opened yourself. Here is `fscanf` reading from a file:

```c
FILE *f = fopen("scores.txt", "r");
if (f != NULL) {
    char name[50];
    int score;
    while (fscanf(f, "%49s %d", name, &score) == 2) {
        printf("%s scored %d\n", name, score);
    }
    fclose(f);
}
```

`fscanf` returns the number of items successfully read, so checking the return value tells you whether the read succeeded.

Opening and Closing Files

```c
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
```

To read or write a file, you open it with `fopen` and close it with `fclose`:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("log.txt", "w");
    if (f == NULL) {
        fprintf(stderr, "Cannot open file\n");
        return 1;
    }

    fprintf(f, "Under Pressure\n");
    fprintf(f, "Year: %d\n", 1981);

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

The second argument to `fopen` is the **mode string**:

Mode	Meaning
`"r"`	Read (file must exist)
`"w"`	Write (creates or truncates)
`"a"`	Append (creates or appends)
`"r+"`	Read and write (file must exist)
`"w+"`	Read and write (creates or truncates)
`"a+"`	Read and append

To open a file in **binary mode**, add `b` to the mode string: `"rb"`, `"wb"`, `"ab"`, etc. On Unix systems, binary and text modes behave identically. On Windows, text mode translates `\r\n` to `\n` on input and vice versa on output — binary mode does not.

`sprintf` and `sscanf`

`sprintf` writes formatted output into a string buffer instead of a stream. `sscanf` reads formatted input from a string:

```c
char buf[100];
sprintf(buf, "Track %02d: %s", 3, "99 Luftballons");
// buf is now "Track 03: 99 Luftballons"

int track;
char title[50];
sscanf(buf, "Track %d: %49[^\n]", &track, title);
// track is 3, title is "99 Luftballons"
```

> **Trap:** `sprintf` has the same buffer overflow risk as `strcpy` — it does not check the size of the destination buffer. Use `snprintf` for safety:
>
> ```c
> int snprintf(char *str, size_t size, const char *format, ...);
> ```
>
> ```c
> snprintf(buf, sizeof(buf), "Track %02d: %s", 3, "99 Luftballons");
> ```
>
> `snprintf` guarantees it will not write more than `sizeof(buf)` bytes, including the null terminator.

`asprintf` (POSIX) goes one step further — it `malloc`s a buffer that is exactly the right size, so you never have to guess:

```
int asprintf(char **strp, const char *format, ...);
```

```
char *msg;
asprintf(&msg, "Track %02d: %s", 3, "Mis Ojos Lloran Por Ti");
printf("%s\n", msg);   // "Track 03: Mis Ojos Lloran Por Ti"
free(msg);              // you must free it
```

Like `%m` in `scanf`, you pass a pointer to a `char *` and `asprintf` fills it in with the address of the newly allocated string. It returns the number of characters written, or -1 on failure.

Tip: `asprintf` is a POSIX/GNU extension, not part of the C standard. It is available on Linux, macOS, and most Unix systems but not MSVC. When it is available, it is the safest and most convenient way to build formatted strings — no buffer sizing, no truncation, no overflow.

Binary I/O: `fread` and `fwrite`

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```

For reading and writing raw binary data (not text), use `fread` and `fwrite`:

```
#include <stdio.h>

int main(void) {
    int nums[] = {10, 20, 30, 40, 50};

    // Write binary data
    FILE *f = fopen("data.bin", "wb");
    fwrite(nums, sizeof(int), 5, f);
    fclose(f);

    // Read it back
    int result[5];
    f = fopen("data.bin", "rb");
    fread(result, sizeof(int), 5, f);
    fclose(f);

    for (int i = 0; i < 5; i++) {
        printf("%d ", result[i]);   // 10 20 30 40 50
    }
    printf("\n");
    return 0;
}
```

Both functions take four arguments: a pointer to the data, the size of each element, the number of elements, and the file stream. `fwrite` returns the number of elements successfully written; `fread` returns the number of elements successfully read.

Reading Lines: `fgets`

```
char *fgets(char *s, int size, FILE *stream);
```

`fgets` reads a line from a stream into a buffer. It stops when it has read `size - 1` characters, encounters a newline (which it includes in the buffer), or reaches end of file. It always null-terminates the result:

```
char line[80];
while (fgets(line, sizeof(line), f) != NULL) {
```

```
    printf("%s", line);   // line already includes '\n'
}
```

`fgets` returns `NULL` at end of file or on error, making it easy to use in a loop. It is generally safer than `scanf` for reading lines because it always respects the buffer size.

Buffering and `fflush`

`stdio` does not write directly to the output device on every call. Instead, it accumulates data in an internal buffer and writes it in larger chunks for efficiency. There are three buffering modes:

- **Full buffering:** Output is written when the buffer is full (default for files).
- **Line buffering:** Output is written when a `\n` is encountered (default for `stdout` when connected to a terminal).
- **Unbuffered:** Output is written immediately (default for `stderr`).

This means that `printf("Working...")` (no newline) will not appear on screen until the next newline when `stdout` goes to a terminal, and will not appear when redirected to a file until the buffer fills or the program exits. Use `fflush` to force the buffer to be written:

```
int fflush(FILE *stream);
```

```
printf("Working...");
fflush(stdout);   // force output to appear now
// ... long computation ...
printf(" done!\n");
```

Trap: When `stdout` is connected to a terminal, output is **line buffered** — a `\n` triggers a flush. When `stdout` is redirected to a file or pipe, it is **fully buffered** — output may not appear until the buffer fills up or the program exits. If you need output to appear immediately (e.g., progress indicators), call `fflush(stdout)` after printing. `stderr` is always unbuffered, which is why error messages appear immediately.

Try It: Standard I/O Starter

```
#include <stdio.h>
#include <string.h>

int main(void) {
    // sprintf: format into a string
    char buf[100];
    sprintf(buf, "Track %02d: %s", 7, "Hungry Like the Wolf");
    printf("sprintf: %s\n", buf);

    // snprintf: safe version with size limit
    char small[15];
    snprintf(small, sizeof(small), "Year: %d", 1984);
    printf("snprintf: %s\n", small);

    // sscanf: parse from a string
    int track;
    char title[50];
    sscanf(buf, "Track %d: %49[^\n]", &track, title);
    printf("sscanf: track=%d title='%s'\n", track, title);

    // fprintf to stderr
```

```c
    fprintf(stderr, "This goes to stderr\n");

    // fwrite/fread round-trip
    int nums[] = {10, 20, 30};
    FILE *f = fopen("/tmp/tryit_data.bin", "wb");
    fwrite(nums, sizeof(int), 3, f);
    fclose(f);

    int result[3];
    f = fopen("/tmp/tryit_data.bin", "rb");
    fread(result, sizeof(int), 3, f);
    fclose(f);

    printf("fread: %d %d %d\n", result[0], result[1], result[2]);

    return 0;
}
```

Key Points

- `printf` writes to `stdout`; `fprintf` writes to any `FILE *`.
- `scanf` needs the address (`&`) of each variable — arrays are the exception since they decay to pointers.
- `fopen` returns `NULL` on failure — always check before using the file pointer.
- Add `"b"` to the mode string for binary files. This matters on Windows.
- `fread` and `fwrite` transfer raw bytes — no format conversion.
- `stdout` is line buffered at a terminal and fully buffered when redirected. Use `fflush` when you need output immediately.

Exercises

1. **Think about it:** Why does `scanf` need the `&` operator for scalar variables but not for arrays?
2. **What does this print?**

   ```c
   char buf[50];
   sprintf(buf, "%s: %d", "Score", 100);
   printf("%zu\n", strlen(buf));
   ```

3. **Calculation:** If `buf` is declared as `char buf[20]` and you call `snprintf(buf, sizeof(buf), "Year: %d", 1984)`, how many characters (excluding the null terminator) are written to `buf`?
4. **Where is the bug?**

   ```c
   int x;
   scanf("%d", x);
   ```

5. **Where is the bug?**

   ```c
   FILE *f = fopen("noexist.txt", "r");
   fprintf(f, "Hello\n");
   fclose(f);
   ```

6. **Think about it:** You run `./program > output.txt` and your program contains both `printf` and `fprintf(stderr, ...)` calls. Which messages appear in `output.txt` and which appear on the screen? Why?
7. **Write a program** that opens a text file, writes five lines to it (your choice of content), closes it, reopens it for reading, reads and prints each line using `fgets`, then closes it again.