Gorgo C for C++ Programmers

April 11, 2026

3. Strings

In C++, you use `std::string` and barely think about what is happening under the hood. In C, there is no string type at all. A "string" in C is just an array of `char` that ends with a null character `'\0'`. Every string function in C depends on finding that null terminator to know where the string ends.

Declaring C Strings

There are several ways to create a string in C:

```
// compiler sizes it: 12 bytes (11 + '\0')
char greeting[] = "Hola, mundo";
char band[20] = "Depeche Mode";         // 20-byte buffer, only 13 used
char empty[10] = "";                     // 10 bytes, first byte is '\0'
```

When you write `"Hola, mundo"`, the compiler automatically adds a `'\0'` at the end. The array `greeting` is 12 bytes: 11 visible characters plus the null terminator.

Trap: Always remember the null terminator when sizing your buffers. The string `"hello"` needs 6 bytes, not 5. Off-by-one errors with null terminators are one of the most common bugs in C.

String Literals vs. Arrays

There is an important difference between these two declarations:

```
char arr[] = "I Can't Drive 55";    /* modifiable copy on the stack */
const char *ptr = "I Can't Drive 55"; /* pointer to read-only memory */
```

With `char arr[]`, the compiler creates a local array and copies the string into it. You can modify the contents freely.

With a string literal like `"I Can't Drive 55"`, the compiler stores the text in read-only memory. A pointer to a literal should be declared `const char *` because modifying the literal is undefined behavior:

```
arr[0] = 'i';   /* fine — arr is a modifiable array */
ptr[0] = 'i';   /* COMPILE ERROR — ptr is const char * */
```

If you forget the `const`, the compiler will not prevent the mistake:

```
char *bad = "I Can't Drive 55";  /* compiles, but dangerous */
bad[0] = 'i';                    /* UNDEFINED BEHAVIOR — likely a crash */
```

Trap: Modifying a string literal is one of the most common mistakes C++ programmers make when moving to C. The compiler may not warn you (unless you use `-Wwrite-strings`), and the program might even appear to work — until it crashes on a different platform or optimization level. Always use `const char *` when pointing to a string literal.

String Functions

C provides a library of string manipulation functions in `<string.h>`. These are the ones you will use most:

`strlen` — Get the Length

```
size_t strlen(const char *s);
```

Returns the number of characters before the null terminator:

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

int main(void) {
    char song[] = "Take On Me";
    printf("'%s' has %zu characters\n", song, strlen(song));  // 10
    return 0;
}
```

Note that `strlen` does not count the `'\0'`. The array holding `"Take On Me"` is 11 bytes, but `strlen` returns 10.

`strcpy` / `strncpy` — Copy a String

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

`strcpy` copies the source string (including the null terminator) into the destination buffer. You must make sure the destination is large enough:

```
char dest[20];
strcpy(dest, "Tainted Love");  // copies 13 bytes (12 chars + '\0')
```

`strncpy` is the safer variant — it copies at most `n` characters:

```
char dest[10];
strncpy(dest, "Enjoy the Silence", 9);
dest[9] = '\0';  // strncpy does NOT guarantee null termination!
```

Wut: `strncpy` does not null-terminate the destination if the source is longer than `n`. Always set the last byte yourself: `dest[sizeof(dest) - 1] = '\0';`

`strcmp` — Compare Strings

```
int strcmp(const char *s1, const char *s2);
```

In C++, you can compare strings with `==`. In C, you cannot — using `==` on two `char` arrays compares the *addresses*, not the contents. Use `strcmp` instead:

```
char a[] = "Rush";
char b[] = "Rush";

if (a == b) {
    // This compares ADDRESSES, not content — almost always false
}

if (strcmp(a, b) == 0) {
    // This compares the actual characters — correct!
    printf("Same band!\n");
}
```

`strcmp` returns 0 if the strings are equal, a negative value if the first is lexicographically less, and a positive value if it is greater.

Trap: Yes, `strcmp` returns 0 for equal strings. It is a common source of confusion. Think of it as returning the "difference" between the strings — zero means no difference.

`strncmp` — Compare a Prefix

```
int strncmp(const char *s1, const char *s2, size_t n);
```

`strncmp` works like `strcmp` but compares at most `n` characters. It is useful for checking whether a string starts with a particular prefix:

```
char cmd[] = "walk like an Egyptian";
if (strncmp(cmd, "walk", 4) == 0) {
    printf("Walking!\n");
}
```

Unlike `strncpy`, there are no gotchas — `strncmp` simply stops comparing after `n` characters or a null terminator, whichever comes first.

`strchr` / `strrchr` — Find a Character

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

`strchr` finds the first occurrence of a character. `strrchr` finds the last:

```
char lyric[] = "Don't You Forget About Me";
char *first_o = strchr(lyric, 'o');    // points to 'o' in "Don't"
char *last_o = strrchr(lyric, 'o');    // points to 'o' in "About"

if (first_o != NULL) {
    printf("First 'o' at position %td\n", first_o - lyric);
}
```

Both functions return a pointer to the found character, or `NULL` if the character is not in the string.

`strstr` — Find a Substring

```
char *strstr(const char *haystack, const char *needle);
```

Finds the first occurrence of a substring within a string:

```
char title[] = "Blade Runner 1982";
char *found = strstr(title, "Runner");
if (found != NULL) {
    printf("Found: %s\n", found);  // "Runner 1982"
}
```

Like `strchr`, it returns a pointer to the start of the match, or `NULL` if not found.

`strcat` / `strncat` — Concatenate Strings

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

`strcat` appends one string to the end of another:

```
char message[50] = "I'll be ";
strcat(message, "back");
printf("%s\n", message);  // "I'll be back"
```

This works fine as long as the destination buffer is large enough. But `strcat` does not check bounds — if you run out of space, it writes past the end of the array.

> **Trap:** `strcat` is one of the most dangerous functions in C. It has no way to know how large the destination buffer is, so it blindly appends bytes. If the combined strings exceed the buffer size, you get a **buffer overflow** — one of the most common security vulnerabilities in the history of software. Use `strncat` instead.

`strncat` takes a maximum number of characters to append as the last argument. Make sure to leave space for the null terminator, so if there are three bytes left in a buffer, the maximum number of characters to append would be 2.

```
char buf[20] = "Hello";
strncat(buf, ", World!", sizeof(buf) - strlen(buf) - 1);
```

`strdup` — Duplicate a String

```
char *strdup(const char *s);
```

`strdup` allocates new memory on the heap, copies the string into it, and returns a pointer to the copy. You are responsible for freeing the memory when you are done:

```
char *copy = strdup("Video Killed the Radio Star");
printf("%s\n", copy);
free(copy);  // you must free memory allocated by strdup
```

> **Tip:** `strdup` calls `malloc` internally. Every call to `strdup` must eventually be paired with a call to `free`. If you forget, you have a memory leak. Note that `strdup` is a POSIX function, not part of the C standard until C23. It is available on virtually every system you will use, but compiling with `-std=c99 -pedantic` or `-std=c11 -pedantic` will produce a warning.

`strtok` — Tokenize a String

```
char *strtok(char *str, const char *delim);
```

`strtok` splits a string into tokens separated by any character in a delimiter set. In C++, you might use `std::istringstream` with `>>` or `std::string::find` — in C, `strtok` is the standard approach:

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

int main(void) {
    char line[] = "Girls Just Want to Have Fun";

    char *tok = strtok(line, " ");
    while (tok != NULL) {
        printf("'%s'\n", tok);
        tok = strtok(NULL, " ");
    }
    return 0;
}
// Output:
// 'Girls'
// 'Just'
// 'Want'
// 'to'
// 'Have'
// 'Fun'
```

The first call passes the string to tokenize. Each subsequent call passes `NULL` to continue tokenizing the same string. `strtok` returns `NULL` when there are no more tokens.

There are two important gotchas. First, `strtok` **modifies the original string** by replacing delimiter characters with `'\0'`. If you need the original string preserved, make a copy with `strdup` before tokenizing. Second, `strtok` stores its state in a hidden static variable, which means it is **not thread-safe** and you cannot tokenize two strings at the same time.

> **Tip:** Use `strtok_r` (POSIX) or `strtok_s` (C11 Annex K / Windows) instead of `strtok`. These reentrant versions take an extra `char **saveptr` parameter to store the tokenizer state, making them safe to use in multi-threaded programs and allowing nested tokenization:
>
> ```c
> char *strtok_r(char *str, const char *delim, char **saveptr);
> ```
>
> ```c
> char *saveptr;
> char *tok = strtok_r(line, " ", &saveptr);
> while (tok != NULL) {
>     printf("'%s'\n", tok);
>     tok = strtok_r(NULL, " ", &saveptr);
> }
> ```

The Dangers of `strcat`

Let's look at a concrete example of why `strcat` is dangerous:

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

int main(void) {
    char buf[12] = "Buenas ";    // 7 chars + '\0', 4 bytes remaining

    // "noches" is 6 chars — doesn't fit in 4 remaining bytes!
    strcat(buf, "noches");       // BUFFER OVERFLOW — undefined behavior

    printf("%s\n", buf);         // might print garbage, might crash
    return 0;
```

```
}
```

The buffer is 12 bytes, `"Buenas "` uses 8 (including the `'\0'`), and `"noches"` needs 7 more bytes (including its `'\0'`). That is 14 bytes total in a 12-byte buffer. The extra bytes overwrite whatever happens to be next to the buffer in memory, which can corrupt other variables, crash the program, or — worst of all — create a security exploit.

Character Classification: `<ctype.h>`

The `<ctype.h>` header provides functions for classifying and converting individual characters. These are great when processing strings character by character:

Function	Returns non-zero if
`isalpha(c)`	`c` is a letter (A–Z, a–z)
`isdigit(c)`	`c` is a digit (0–9)
`isalnum(c)`	`c` is a letter or digit
`isspace(c)`	`c` is whitespace (space, tab, newline, etc.)
`isupper(c)`	`c` is an uppercase letter
`islower(c)`	`c` is a lowercase letter

`toupper` and `tolower` convert a character's case:

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

int main(void) {
    char title[] = "hysteria";
    for (size_t i = 0; i < strlen(title); i++) {
        title[i] = toupper((unsigned char)title[i]);
    }
    printf("%s\n", title);  /* HYSTERIA */
    return 0;
}
```

Here is a handy function that checks whether a string is entirely digits:

```c
int all_digits(const char *s) {
    for (; *s; s++) {
        if (!isdigit((unsigned char)*s))
            return 0;
    }
    return 1;
}
```

Wut: The `<ctype.h>` functions take an `int`, not a `char`. On platforms where `char` is signed, a character with a value above 127 would be passed as a negative `int`, which is undefined behavior. Always cast to `unsigned char` first: `toupper((unsigned char)c)`.

A Preview: `sprintf` and `sscanf`

```c
int sprintf(char *str, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

C has two powerful functions for building and parsing strings that we will cover in detail in the Standard I/O chapter: `sprintf` writes formatted output into a string buffer (like `printf` but to a string), and `sscanf` reads formatted input from a string (like `scanf` but from a string). They are the C programmer's Swiss Army knife for string manipulation:

```
char result[50];
int year = 1985;
sprintf(result, "The year is %d. Que bueno!", year);
// result is now "The year is 1985. Que bueno!"
```

Just as `strncpy` is the safer sibling of `strcpy`, `snprintf` is the safer sibling of `sprintf`:

```
char buf[20];
snprintf(buf, sizeof(buf), "The year is %d", 2112);
/* buf is "The year is 2112" — safely truncated if it were longer */
```

`snprintf` guarantees it will not write more than the specified number of bytes, preventing buffer overflows. We will cover `sprintf`, `snprintf`, and `sscanf` in detail in the Standard I/O chapter.

Try It: String Starter

This program exercises several string functions so you can see them in action:

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

int main(void) {
    // strlen
    char song[] = "Sweet Dreams";
    printf("Song: '%s' (%zu chars)\n", song, strlen(song));

    // strcpy
    char copy[20];
    strcpy(copy, song);
    printf("Copy: '%s'\n", copy);

    // strcmp
    printf("Compare '%s' to '%s': %d\n", song, copy, strcmp(song, copy));
    printf("Compare 'A' to 'B': %d\n", strcmp("A", "B"));

    // strcat
    char greeting[30] = "Buenos ";
    strcat(greeting, "dias");
    printf("Greeting: '%s'\n", greeting);

    // strchr and strstr
    char *ch = strchr(song, 'D');
    if (ch) printf("Found 'D' at position %td\n", ch - song);

    char *sub = strstr(song, "Dreams");
    if (sub) printf("Found substring: '%s'\n", sub);

    // strdup
    char *dup = strdup("Never Gonna Give You Up");
    printf("Duplicate: '%s'\n", dup);
```

```
    free(dup);

    return 0;
}
```

Key Points

- C strings are `char` arrays terminated by `'\0'`. There is no `std::string`.
- Always account for the null terminator when sizing buffers.
- String literals like `"hello"` live in read-only memory. Use `char s[] = "..."` for a mutable copy and `const char *p = "..."` for a read-only pointer.
- Use `strcmp` to compare strings — the `==` operator compares addresses, not content. `strncmp` compares at most `n` characters — useful for prefix matching.
- `strcpy` and `strcat` do not check buffer bounds. Prefer `strncpy` and `strncat`.
- `strdup` allocates memory with `malloc` — you must `free` the result.
- `strtok` splits strings into tokens but modifies the original string and is not thread-safe. Use `strtok_r` or `strtok_s` instead.
- `<ctype.h>` provides character classification (`isalpha`, `isdigit`, etc.) and conversion (`toupper`, `tolower`). Cast `char` to `unsigned char` before passing.
- `snprintf` is the safe alternative to `sprintf` — it limits output to a maximum buffer size.
- `strlen` returns the number of characters *before* the `'\0'`, not the buffer size.

Exercises

1. **Think about it:** Why does `strcmp` return 0 for equal strings rather than 1? How does this relate to the function's actual purpose?
2. **What does this print?**

   ```
   char s[] = "Ghostbusters";
   printf("%zu %zu\n", strlen(s), sizeof(s));
   ```

3. **Calculation:** What is `sizeof(buf)` for `char buf[20] = "Hola";`?
4. **Where is the bug?**

   ```
   char buf[10] = "Livin'";
   strcat(buf, " on a Prayer");
   printf("%s\n", buf);
   ```

5. **Where is the bug?**

   ```
   char *a = "Hello";
   char *b = "Hello";
   if (a == b) {
       printf("Equal\n");
   } else {
       printf("Not equal\n");
   }
   ```

   (Hint: what does `==` actually compare here? Is the output guaranteed?)
6. **Write a program** that reads a string from the user, reverses it in place using pointer arithmetic, and prints the result.
7. **Where is the bug?**

   ```
   char *greeting = "We Got the Beat";
   greeting[0] = 'w';
   ```

```c
printf("%s\n", greeting);
```

8. **What does this print?**

```c
printf("%d\n", strncmp("Final Countdown", "Final Fantasy", 5));
```

9. **What does this print?**

```c
char c = '7';
printf("%d %d %d\n", isdigit((unsigned char)c) != 0, isalpha((unsigned char)c) != 0, isalnum((unsigne
```

10. **Write a program** that takes a string and counts how many uppercase letters, lowercase letters, digits, and other characters it contains. Use `<ctype.h>` functions.