Gorgo C for C++ Programmers

April 11, 2026

2. Variables

In C++, you have `auto` to let the compiler figure out types, `std::string` to handle text, and classes to bundle data with behavior. In C, none of that exists. Every type is explicit, strings are raw character arrays, and if you want to group data together, you use a `struct` with no member functions. This chapter covers how C handles variables — from basic types and arrays to pointers, `const`, and structures.

Basic Types

C provides a small set of built-in types. There are no classes, no `std::string`, and no `bool` keyword (without a header). Here are the types you will use most:

Type	Typical Size	Description
`char`	1 byte	A single character (or small integer)
`short`	2 bytes	Short integer
`int`	4 bytes	Standard integer
`long`	4 or 8 bytes	Long integer (platform-dependent)
`long long`	8 bytes	At least 64-bit integer
`float`	4 bytes	Single-precision floating point
`double`	8 bytes	Double-precision floating point
`_Bool`	1 byte	Boolean (C99); use `bool` via `<stdbool.h>`

Each integer type has an `unsigned` variant that stores only non-negative values, giving you twice the positive range. For example, a `signed char` holds -128 to 127, while an `unsigned char` holds 0 to 255:

```c
unsigned char brightness = 255;
unsigned int count = 4000000000U;
```

Tip: C99 added `<stdbool.h>`, which defines `bool`, `true`, and `false`. Without it, you must use `_Bool` for the type and integer values `0` and `1`. Most modern C code includes `<stdbool.h>` and uses `bool` just like C++.

```c
#include <stdbool.h>

bool done = false;
if (!done) {
    /* keep going */
}
```

Variables as Named Memory

A variable declaration does two things: it allocates a region of memory large enough to hold the declared type, and it gives that region a name. The amount of memory allocated depends on the type:

```c
#include <stdio.h>

int main(void) {
    char letter = 'J';
    int year = 1981;
    double rating = 9.5;

    printf("char:   %zu bytes\n", sizeof(letter));   // 1
    printf("int:    %zu bytes\n", sizeof(year));      // 4
    printf("double: %zu bytes\n", sizeof(rating));    // 8

    return 0;
}
```

The `sizeof` operator returns the size in bytes of a type or variable. It evaluates at compile time, so there is no runtime cost. You can use it with a type name or with a variable:

```c
printf("int is %zu bytes\n", sizeof(int));
printf("year is %zu bytes\n", sizeof(year));
```

Tip: `sizeof` returns a value of type `size_t`. Always use `%zu` to print it. Using `%d` is technically undefined behavior, even though it often appears to work.

Pointer Declarations

A pointer variable holds the address of another variable. You declare a pointer by placing `*` after the base type:

```c
int score = 100;
int *p = &score;   /* p holds the address of score */
```

The type before the `*` tells the compiler what kind of data lives at that address. An `int *` points to an `int`, a `char *` points to a `char`, and so on.

We will not go deeper into pointers here. The Pointers chapter covers dereferencing, pointer arithmetic, pointers to pointers, and how pointers interact with arrays and structures in detail.

Arrays

An array is a fixed-size sequence of elements of the same type, declared with `[]`:

```c
int scores[5];                          /* 5 uninitialized ints */
int primes[5] = {2, 3, 5, 7, 11};      /* initialized */
/* compiler counts: 5 chars (including '\0') */
char greeting[] = "Hola";
```

When you provide an initializer, the compiler can determine the size for you, so you can leave the brackets empty.

The "Value" of an Array Name

In most expressions, the name of an array evaluates to the address of its first element. This is called **decay** — the array "decays" into a pointer:

```c
int primes[5] = {2, 3, 5, 7, 11};
int *p = primes;    /* p points to primes[0]; no & needed */
```

This is why you can pass an array to a function that expects a pointer. The Pointers chapter will explore this relationship thoroughly.

Wut: The `sizeof` operator is one of the few contexts where an array does *not* decay to a pointer. `sizeof(primes)` gives the total size of the array, not the size of a pointer. If `int` is 4 bytes and an address is 8 bytes (common sizes on modern hardware), `sizeof(primes)` is 20 and `sizeof(p)` is 8.

Initialization

You can partially initialize an array — remaining elements are set to zero:

```c
int totals[10] = {1, 2, 3};   /* totals[3] through totals[9] are 0 */
int zeros[100] = {0};          /* all 100 elements are 0 */
```

Multidimensional Arrays

C supports multidimensional arrays. A two-dimensional array is really an array of arrays, stored in **row-major** order — all elements of row 0 come first, then all elements of row 1, and so on:

```c
int grid[3][4] = {
    {1,  2,  3,  4},
    {5,  6,  7,  8},
    {9, 10, 11, 12}
};
```

In memory, this is stored as: `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12` — twelve consecutive integers. The expression `grid[r][c]` accesses row `r`, column `c`.

Iterating through a 2D array:

```c
#include <stdio.h>

int main(void) {
    int grid[3][4] = {
        {1,  2,  3,  4},
        {5,  6,  7,  8},
        {9, 10, 11, 12}
    };
```

```
    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 4; c++) {
            printf("%3d", grid[r][c]);
        }
        printf("\n");
    }
    return 0;
}
```

Trap: When passing a multidimensional array to a function, you must specify all dimensions except the first. The compiler needs the column count to calculate offsets:
`void print_grid(int grid[][4], int rows);  /* OK — column size specified */`
You cannot write `int grid[][]` — the compiler would not know how wide each row is.

const

The `const` qualifier marks a variable as read-only. Any attempt to modify it is a compile-time error:

```
const int MAX_TRACKS = 12;
const double PI = 3.14159265358979;
```

const with Pointers

Things get interesting when `const` meets pointers. There are three combinations, and the **read right-to-left** rule helps you decode them:

Pointer to const data — you cannot modify the data through this pointer, but you can change where the pointer points:

```
const int *p = &x;
*p = 10;      /* ERROR: cannot modify data through p */
p = &y;       /* OK: p itself can change */
```

Read right-to-left: `p` is a pointer (`*`) to `int` that is `const`. The data is const, not the pointer.

const pointer to data — the pointer itself cannot change, but you can modify the data it points to:

```
int *const p = &x;
*p = 10;      /* OK: data can be modified */
p = &y;       /* ERROR: p itself is const */
```

Read right-to-left: `p` is a `const` pointer (`*`) to `int`. The pointer is const, not the data.

const pointer to const data — neither the pointer nor the data can change:

```
const int *const p = &x;
*p = 10;      /* ERROR */
p = &y;       /* ERROR */
```

Tip: The most common form is `const int *p` — a pointer through which you promise not to modify the data. You will see this constantly in function parameters, like `const char *msg`, where the function reads the data but does not change it.

Structures

A `struct` groups related variables together under one name. If you come from C++, think of a struct as a class with only public data members — no member functions, no constructors, no destructors, and no access specifiers:

```
struct song {
    char title[40];
    int year;
};
```

Declaring and Initializing

To declare a variable of a struct type, you write `struct` followed by the tag name:

```
struct song track;
track.year = 1981;
```

You can also initialize at declaration time:

```
struct song track = {"I Love Rock 'n' Roll", 1981};
```

Accessing Members

Use the `.` operator to access members:

```
#include <stdio.h>

int main(void) {
    struct song {
        char title[40];
        int year;
    };

    struct song track = {"I Love Rock 'n' Roll", 1981};

    printf("Title: %s\n", track.title);   // I Love Rock 'n' Roll
    printf("Year: %d\n", track.year);     // 1981

    return 0;
}
```

Assignment Copies

Assigning one struct to another copies the entire contents — every byte:

```
struct song original = {"I Love Rock 'n' Roll", 1981};
struct song copy = original;

printf("%s (%d)\n", copy.title, copy.year);   // I Love Rock 'n' Roll (1981)
```

This is a **shallow copy**. If the struct contained a pointer, both copies would point to the same memory. For structs that contain only arrays and plain values (like `struct song` above), the copy is complete and independent.

Wut: Unlike C++, there is no copy constructor or `operator=` to customize what happens during assignment. C copies the raw bytes, period. If your struct contains a pointer to dynamically allocated memory, the copy will share that memory, leading to double-free bugs if you are not careful.

No Member Functions

In C++, you might write:

```
class Song {
public:
    void print() { std::cout << title << " (" << year << ")\n"; }
    // ...
};
```

In C, structs cannot have member functions. Instead, you write standalone functions that take a pointer to the struct:

```
void song_print(const struct song *s) {
    printf("%s (%d)\n", s->title, s->year);
}
```

The `->` operator accesses a member through a pointer — it is shorthand for `(*s).title`. We will cover this in detail in the Pointers chapter.

Tip: A common C pattern is to prefix functions with the struct name they operate on: `song_print`, `song_init`, `song_compare`. This gives you something like namespaced methods. Function pointers (covered in a later chapter) can even be stored in structs to simulate virtual functions.

Designated Initializers

C99 introduced **designated initializers**, which let you initialize struct members by name instead of relying on declaration order:

```
struct song track = {.year = 1985, .title = "I Can't Drive 55"};
```

This is clearer than positional initialization because the reader does not need to remember the order of members. It also lets you skip members you want left at zero:

```
struct album {
    char title[40];
    int year;
    int tracks;
};

struct album a = {.title = "Hysteria", .year = 1987};
// a.tracks is initialized to 0
```

Any member you do not mention is initialized to zero (or null for pointers).

Tip: Prefer designated initializers over positional initialization. They make the code self-documenting and protect you from bugs when someone reorders the struct members later.

Designated initializers also work with arrays. You can initialize specific elements by index:

```c
int flags[8] = {[0] = 1, [4] = 1, [7] = 1};
// flags is {1, 0, 0, 0, 1, 0, 0, 1}
```

Wut: C++ did not support designated initializers until C++20, and even then it requires the initializers to appear in declaration order. C has no such restriction — you can list members in any order.

typedef

The `typedef` keyword creates an alias for an existing type. It does not create a new type — it just gives you a shorter or more descriptive name:

```c
typedef unsigned long ulong;
typedef unsigned char byte;

ulong population = 4000000000UL;
byte channel = 83;
```

To remember the syntax, notice that without the `typedef` above, `byte` would have been a variable named `byte` of type `unsigned char`. With the `typedef`, instead of declaring `byte` to be a variable, we are declaring it to be the name of a new type alias.

One of the most common uses of `typedef` is to simplify struct declarations. Without `typedef`, you must write `struct` every time you use the type:

```c
struct point {
    double x;
    double y;
};

struct point origin;   /* must say "struct point" every time */
```

With `typedef`, you can drop the `struct` keyword:

```c
typedef struct {
    double x;
    double y;
} Point;

Point origin = {0.0, 0.0};   /* much cleaner */
```

Tip: In C++, you can use a struct name directly as a type. In C, you cannot — you must either write `struct name` every time or use `typedef` to create an alias. Most C codebases use `typedef` for any struct that appears frequently.

Try It: Variables Starter

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

struct song {
    char title[40];
    int year;
};

int main(void) {
```

```c
    /* Basic types and sizeof */
    char initial = 'J';
    int year = 1981;
    double rating = 9.5;
    bool classic = true;

    printf("=== Sizes ===\n");
    printf("char:   %zu byte\n", sizeof(initial));
    printf("int:    %zu bytes\n", sizeof(year));
    printf("double: %zu bytes\n", sizeof(rating));
    printf("bool:   %zu byte\n", sizeof(classic));

    /* Arrays */
    int scores[] = {95, 87, 92, 78, 100};
    int n = sizeof(scores) / sizeof(scores[0]);
    printf("\n=== Array ===\n");
    for (int i = 0; i < n; i++) {
        printf("scores[%d] = %d\n", i, scores[i]);
    }
    printf("Total array size: %zu bytes (%d elements)\n",
           sizeof(scores), n);

    /* const */
    const int MAX = 100;
    printf("\nMAX = %d\n", MAX);

    /* Struct */
    struct song track = {"I Love Rock 'n' Roll", 1981};
    printf("\n=== Struct ===\n");
    printf("Title: %s\n", track.title);
    printf("Year:  %d\n", track.year);
    printf("Size:  %zu bytes\n", sizeof(track));

    /* Struct copy */
    struct song backup = track;
    printf("Copy:  %s (%d)\n", backup.title, backup.year);

    /* 2D array */
    int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printf("\n=== 2D Array ===\n");
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 3; c++) {
            printf("%3d", grid[r][c]);
        }
        printf("\n");
    }

    return 0;
}
```

Key Points

- C has no `auto`, no `std::string`, and no classes. Every type is spelled out explicitly.
- A variable declaration allocates memory of the type's size and gives it a name. Use `sizeof` to see how

many bytes each type occupies.
- `typedef` creates type aliases — especially useful for structs so you do not have to write `struct` everywhere.
- Arrays are fixed-size, and the array name decays to a pointer to the first element in most contexts.
- Multidimensional arrays are stored in row-major order. When passing them to functions, all dimensions except the first must be specified.
- `const` marks a variable as read-only. With pointers, read right-to-left to determine what is const — the data, the pointer, or both.
- Structs group data together but have no member functions, constructors, or access specifiers. Assignment copies the raw bytes.

Exercises

1. **Think about it:** In C++, `auto x = 42;` lets the compiler deduce the type. C has no `auto` type deduction. What advantage does requiring explicit types give to someone reading unfamiliar C code?

2. **What does this print?**

```c
int a[] = {10, 20, 30, 40};
printf("%zu %zu\n", sizeof(a), sizeof(a[0]));
```

3. **Calculation:** Given the following declarations on a system where `int` is 4 bytes, what is `sizeof(grid)`?

```c
int grid[3][5];
```

4. **Where is the bug?**

```c
const int MAX = 100;
int *p = &MAX;
*p = 200;
printf("MAX = %d\n", MAX);
```

5. **What does this print?**

```c
struct point { int x; int y; };
struct point a = {3, 7};
struct point b = a;
b.x = 99;
printf("%d %d\n", a.x, b.x);
```

6. **Think about it:** In C, assigning one struct to another copies the raw bytes. What problem could this cause if the struct contains a `char *` member that points to dynamically allocated memory (via `malloc`)? How is this different from what C++ does by default?

7. **Write a program** that declares a `struct student` with fields `name` (a `char` array), `id` (an `int`), and `gpa` (a `double`). Create an array of 3 students, initialize them with values, and print each student's information using a loop. Use `%s`, `%d`, and `%.2f` in your `printf`.