Gorgo Starting C++

April 11, 2026

Contents

2. Variables

In Chapter 1 you wrote programs that printed messages and read text from the user. But every value was either a literal typed directly into your source code or a string that was read and immediately used. You had no way to name a value, remember it, or reuse it later. Without that ability, you cannot track a score, accumulate a total, or compare two inputs. Variables solve this — they give a name to a piece of memory so your program can store, retrieve, and update data as it runs. C++ requires each variable to have a **type** so the compiler knows how much memory to allocate and what operations are valid. In this chapter you will learn about the basic types C++ offers, how to declare and use variables, how to group related data with structures, and how to protect values with `const`.

Basic Types

Every variable in C++ has a **type** that determines what kind of data it can hold and how much memory it uses. Here are the fundamental types you will work with:

Integer Types

Integers are whole numbers — no decimal point.

```
int score = 99;
short small_num = 42;
long big_num = 1000000L;
long long very_big = 9000000000LL;
```

The difference between these types is how much memory they use and how large a value they can hold. On most modern systems:

Type	Typical Size	Approximate Range
`short`	2 bytes	-32,768 to 32,767
`int`	4 bytes	-2.1 billion to 2.1 billion
`long`	4 or 8 bytes	at least as large as `int`
`long long`	8 bytes	-9.2 quintillion to 9.2 quintillion

By default, integer types are **signed**, meaning they can hold negative values. If you know a value will never be negative, you can use **unsigned** variants:

```
unsigned int positive_only = 42;
unsigned short flags = 255;
```

An `unsigned int` on a system with 4-byte ints can hold values from 0 to about 4.3 billion, because it does not need to reserve a bit for the sign.

Trap: Be careful mixing signed and unsigned values in comparisons. If you compare a negative `int` with an `unsigned int`, the negative value gets converted to a very large positive number, which is almost certainly not what you want.

Floating-Point Types

Floating-point types store numbers with decimal points.

```
float price = 9.99f;
double pi = 3.14159265358979;
```

`double` gives you more precision than `float` and is the default floating-point type in C++. Unless you have a specific reason to use `float` (like memory constraints), prefer `double`.

Type	Typical Size	Approximate Precision
`float`	4 bytes	~7 decimal digits
`double`	8 bytes	~15 decimal digits

Character Type

A `char` holds a single character and is typically 1 byte.

```
char grade = 'A';
char newline = '\n';
```

Single characters use single quotes. Double quotes are for strings, which are sequences of characters — we will cover those in the next chapter.

Under the hood, a `char` is just a small integer. Every character your computer knows about is assigned a number, and the character is stored in memory as that number. The standard mapping for the basic Latin alphabet, digits, and punctuation is called **ASCII** — the American Standard Code for Information Interchange. ASCII assigns the numbers 0 through 127 to a fixed set of characters: `'A'` is 65, `'a'` is 97, `'0'` is 48, and the space character is 32.

To the CPU, a `char` really is just a number. The CPU has no idea that 65 means the letter A; it sees an integer it can add, subtract, and compare like any other. The interpretation as a letter only happens when the value is sent somewhere that expects characters, like `std::cout`:

```
char letter = 65;
int  number = 65;
std::cout << letter << std::endl;   // prints A
std::cout << number << std::endl;   // prints 65
```

Both variables hold the value 65, but `letter` is a `char` and `number` is an `int`. `std::cout` looks at the type and chooses how to display the value: a `char` becomes a printed glyph, while an `int` becomes digits.

Because a `char` is a number, you can do arithmetic with it:

```
char letter = 'A';
letter = letter + 1;   // letter is now 'B' (66)
```

Adding 1 to `'A'` gives you the next code point, which is `'B'`. Subtracting `'A'` from another letter gives you its position in the alphabet (`'C' - 'A'` is 2). This trick is the basis for many simple text-processing algorithms.

Here is the full ASCII table. The first 32 entries (0 through 31) and entry 127 are **control characters** that do not have a printable glyph; they are listed by their conventional abbreviations (`NUL`, `LF`, `CR`, `ESC`, `DEL`, and so on). Entry 32 (`SP`) is the space character.

```
  0 NUL    16 DLE    32 SP     48 0     64 @     80 P     96 `    112 p
  1 SOH    17 DC1    33 !      49 1     65 A     81 Q     97 a    113 q
  2 STX    18 DC2    34 "      50 2     66 B     82 R     98 b    114 r
  3 ETX    19 DC3    35 #      51 3     67 C     83 S     99 c    115 s
  4 EOT    20 DC4    36 $      52 4     68 D     84 T    100 d    116 t
  5 ENQ    21 NAK    37 %      53 5     69 E     85 U    101 e    117 u
  6 ACK    22 SYN    38 &      54 6     70 F     86 V    102 f    118 v
  7 BEL    23 ETB    39 '      55 7     71 G     87 W    103 g    119 w
  8 BS     24 CAN    40 (      56 8     72 H     88 X    104 h    120 x
  9 HT     25 EM     41 )      57 9     73 I     89 Y    105 i    121 y
 10 LF     26 SUB    42 *      58 :     74 J     90 Z    106 j    122 z
```

```
11 VT     27 ESC    43 +      59 ;      75 K      91 [      107 k     123 {
12 FF     28 FS     44 ,      60 <      76 L      92 \      108 l     124 |
13 CR     29 GS     45 -      61 =      77 M      93 ]      109 m     125 }
14 SO     30 RS     46 .      62 >      78 N      94 ^      110 n     126 ~
15 SI     31 US     47 /      63 ?      79 O      95 _      111 o     127 DEL
```

Tip: ASCII fits in 7 bits, which gives it 128 values. A `char` is 8 bits, leaving room for 128 more values (128 through 255), but those slots are not part of standard ASCII. Different operating systems and locales fill them in different ways, which is one reason text from one system can look like garbage on another. The Numbers chapter has more on how bits, bytes, and ranges work.

Boolean Type

A `bool` holds either `true` or `false`.

```
bool is_alive = true;
bool game_over = false;
```

Booleans are used in conditions and comparisons. We will use them heavily in the Control Flow chapter.

Declaring Variables

When you declare a variable, you are asking the compiler to set aside memory for a value of the specified type and giving that memory a name.

```
int waterfalls = 3;
double speed = 88.0;
char initial = 'T';
```

You can declare multiple variables of the same type on one line:

```
int x = 0, y = 0, z = 0;
```

You can also declare a variable without initializing it:

```
int count;
```

Trap: An uninitialized variable contains whatever garbage was previously in that memory. Using it before assigning a value leads to unpredictable behavior. Always initialize your variables.

In C++, a region of memory with a type is called an **object**. A variable is a name — a label — for an object. When you write `int score = 99`, the compiler allocates memory for an `int` object and gives it the label `score`. This is different from languages like Java or Python, where "object" specifically means an instance of a class. In C++, every variable is a label for an object — `int score = 99` creates an `int` object, `char grade = 'A'` creates a `char` object, and an array of ten `doubles` is an object too. You will hear "object" used this way throughout C++ documentation and error messages, so it is worth knowing early that the word does not imply classes or object-oriented programming.

The `auto` Keyword

When you initialize a variable, the compiler already knows the type of the value on the right-hand side. The `auto` keyword lets you tell the compiler to figure out the type for you:

```
auto waterfalls = 3;        // int (integer literal)
auto speed = 88.0;          // double (floating-point literal)
```

```cpp
auto initial = 'T';         // char (character literal)
auto name = std::string("No Scrubs");  // std::string
```

`auto` does not mean the variable has no type — C++ is still strictly typed. It just means the compiler fills in the type based on the initializer.

This is most useful when the type is long or obvious from context:

```cpp
std::vector<std::string> songs;
auto it = songs.begin();    // instead of std::vector<std::string>::iterator
```

`auto` only works when the compiler has enough information to deduce the type. If you declare a variable without initializing it, there is nothing for the compiler to work with:

```cpp
auto count;    // error: no initializer --- what type is this?
int count;     // OK: the type is explicitly int
```

Using `auto` for function parameters is allowed in C++20, but it turns the function into a template (a topic for a more advanced book). For now, spell out the type in function parameters:

```cpp
void print(auto x);    // valid C++20, but creates a template
void print(int x);     // clearer for now
```

And `auto` can pick a type you did not intend. For example, `auto volume = 11;` gives you an `int`, not a `short` or a `long` — if you need a specific type, spell it out.

You will see `auto` used more in later chapters when types get verbose. For simple declarations like `int count = 0`, spelling out the type is clearer.

Tip: Use `auto` when the type is obvious from the right-hand side or when it is painfully long to write out. Spell the type explicitly when the compiler cannot deduce the type or when the deduced type might not be what you want.

The sizeof Operator

The `sizeof` operator tells you how many bytes a type or variable occupies in memory. It has two forms:

```cpp
sizeof(type)
sizeof expression
```

When used with a type, parentheses are required. When used with a variable or expression, they are optional. `sizeof` returns a value of type `std::size_t`, which is an unsigned integer type.

```cpp
#include <iostream>

int main()
{
    std::cout << "char:      " << sizeof(char) << " bytes" << std::endl;
    std::cout << "int:       " << sizeof(int) << " bytes" << std::endl;
    std::cout << "double:    " << sizeof(double) << " bytes" << std::endl;
    std::cout << "long long: " << sizeof(long long) << " bytes" << std::endl;

    int score = 100;
    std::cout << "score:     " << sizeof(score) << " bytes" << std::endl;

    return 0;
}
```

On a typical 64-bit system, this might print:

```
char:      1 bytes
int:       4 bytes
double:    8 bytes
long long: 8 bytes
score:     4 bytes
```

`sizeof` is evaluated at compile time, not when the program runs. You can use it with a type name (like `sizeof(int)`) or with a variable (like `sizeof(score)`).

Wut: `sizeof(char)` is always 1, by definition. That does not mean a char is always 8 bits — it means the size of everything else is measured in multiples of `char`. On virtually all modern systems, a char is 8 bits, but the C++ standard does not require it.

std::numeric_limits

The `sizeof` operator tells you how many bytes a type occupies, but not what range of values it can hold. The `<limits>` header provides `std::numeric_limits<T>`, which lets you query the minimum and maximum values of any numeric type:

```
static constexpr T min();      // smallest value (integers) or smallest
                                // positive normalized value (floating-point)
static constexpr T max();      // largest value
static constexpr T lowest();   // most negative value
```

```
#include <iostream>
#include <limits>

int main()
{
    std::cout << "int min:    "
              << std::numeric_limits<int>::min()
              << std::endl;
    std::cout << "int max:    "
              << std::numeric_limits<int>::max()
              << std::endl;
    std::cout << "double max: "
              << std::numeric_limits<double>::max()
              << std::endl;

    return 0;
}
```

Wut: For floating-point types, `std::numeric_limits<double>::min()` is *not* the most negative `double` — it is the smallest *positive* normalized value (about 2.2e-308). If you want the most negative value, use `lowest()`. For integer types, `min()` and `lowest()` return the same value.

Arrays

An **array** is a collection of values of the same type, stored in contiguous memory. You declare an array by specifying its size in square brackets.

```
int scores[5] = {99, 85, 73, 91, 100};
```

Array indices start at 0, so `scores[0]` is 99 and `scores[4]` is 100.

```
std::cout << scores[0] << std::endl;   // prints 99
std::cout << scores[4] << std::endl;   // prints 100
```

Trap: Accessing an array out of bounds (like `scores[5]` in a 5-element array) is undefined behavior. C++ does not check array bounds for you — your program might crash, corrupt memory, or appear to work fine until it does not.

You can let the compiler figure out the size from the initializer:

```
int primes[] = {2, 3, 5, 7, 11};   // size is 5
```

To find the number of elements in an array, use `sizeof` on the array divided by `sizeof` one element:

```
int count = sizeof(primes) / sizeof(primes[0]);   // 5
```

The "value" of an array name is the address of its first element. This is important and will come up again in the Containers chapter.

Multidimensional Arrays

You can create arrays of arrays to represent grids, tables, or matrices.

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

`grid[0][0]` is 1, `grid[1][2]` is 7, and `grid[2][3]` is 12. The first index selects the row, the second selects the column.

You can think of `int grid[3][4]` as "3 rows of 4 ints each." The elements are laid out in memory row by row, so `grid[0][3]` and `grid[1][0]` are neighbors in memory.

Here is a program that prints a multiplication table using a 2D array:

```
#include <iostream>

int main()
{
    int table[5][5];

    for (int r = 0; r < 5; r++) {
        for (int c = 0; c < 5; c++) {
            table[r][c] = (r + 1) * (c + 1);
        }
    }

    for (int r = 0; r < 5; r++) {
        for (int c = 0; c < 5; c++) {
            std::cout << table[r][c] << "\t";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Tip: The Containers chapter will introduce `std::array` and `std::vector`, which are safer and more flexible alternatives to raw arrays. Prefer those in modern C++ whenever possible.

const

The `const` keyword marks a variable as **read-only**. Once initialized, its value cannot be changed.

```
const double PI = 3.14159265358979;
const int MAX_LIVES = 3;
```

If you try to modify a `const` variable, the compiler will give you an error:

```
const int limit = 10;
limit = 20;   // ERROR: cannot assign to a const variable
```

Use `const` for values that should never change. It makes your intent clear to anyone reading your code and lets the compiler catch accidental modifications.

const with Pointers

When `const` meets pointers, things get interesting. We have not covered pointers in detail yet, but it is worth previewing this because it trips up many programmers.

There are two things that can be const: the **pointer itself** or the **data it points to**.

```
int vida = 99;

const int *p1 = &vida;     // pointer to const int
int *const p2 = &vida;     // const pointer to int
const int *const p3 = &vida; // const pointer to const int
```

With `const int *p1`, you cannot change the value through `p1` (`*p1 = 42` is an error), but you can make `p1` point somewhere else.

With `int *const p2`, you cannot make `p2` point somewhere else (`p2 = &other` is an error), but you can change the value through `p2` (`*p2 = 42` is fine).

With `const int *const p3`, you cannot do either.

Tip: Read pointer declarations from right to left. `const int *p` reads as "p is a pointer to int that is const" — the int is const. `int *const p` reads as "p is a const pointer to int" — the pointer is const.

Structures

A **structure** lets you group related data together under one name.

```
struct Song {
    std::string title;
    std::string artist;
    int year;
};
```

This defines a new type called `Song` with three **members**: `title`, `artist`, and `year`.

You access members using the **dot operator** (`.`):

```cpp
#include <iostream>
#include <string>

struct Song {
    std::string title;
    std::string artist;
    int year;
};

int main()
{
    Song favorite;
    favorite.title = "Waterfalls";
    favorite.artist = "TLC";
    favorite.year = 1995;

    std::cout << favorite.title << " by " << favorite.artist
              << " (" << favorite.year << ")" << std::endl;

    return 0;
}
```

```
Waterfalls by TLC (1995)
```

You can also initialize a structure using curly braces:

```cpp
Song hit = {"No Scrubs", "TLC", 1999};
```

Structure Assignment

When you assign one structure to another, all members are **copied**:

```cpp
Song a = {"Livin' La Vida Loca", "Ricky Martin", 1999};
Song b = a;   // b is now a copy of a

b.year = 2000;

std::cout << a.year << std::endl;   // prints 1999 (unchanged)
std::cout << b.year << std::endl;   // prints 2000
```

Modifying `b` does not affect `a` because `b` has its own copy of all the data. This is an important detail — assignment copies the entire structure, member by member.

Wut: Structure assignment copies everything, which is usually what you want. But if the structure is very large, copying it can be expensive. We will revisit this when we talk about passing structures to functions.

Try It

Here is a program that puts several concepts from this chapter together. Type it in, compile it, and experiment with changes.

```cpp
#include <iostream>
#include <string>

struct Cancion {
```

```
    std::string titulo;
    std::string artista;
    int anio;
};

int main()
{
    Cancion playlist[3] = {
        {"Waterfalls", "TLC", 1995},
        {"No Scrubs", "TLC", 1999},
        {"Livin' La Vida Loca", "Ricky Martin", 1999}
    };

    int count = sizeof(playlist) / sizeof(playlist[0]);

    for (int i = 0; i < count; i++) {
        std::cout << playlist[i].titulo << " - " << playlist[i].artista
                  << " (" << playlist[i].anio << ")" << std::endl;
    }

    return 0;
}
```

```
Waterfalls - TLC (1995)
No Scrubs - TLC (1999)
Livin' La Vida Loca - Ricky Martin (1999)
```

Key Points

- Every variable has a type that determines what it can hold and how much memory it uses.
- The fundamental types include `int`, `char`, `float`, `double`, `bool`, `short`, `long`, and their unsigned variants.
- A `char` is just a small integer; **ASCII** is the standard mapping between numbers 0–127 and the characters they represent. `std::cout` displays a `char` as a glyph and an `int` as digits, even when both hold the same value.
- Always initialize your variables — uninitialized variables contain garbage.
- `sizeof` tells you how many bytes a type or variable occupies.
- `std::numeric_limits<T>` from `<limits>` lets you query the min, max, and lowest values of any numeric type.
- Arrays store multiple values of the same type in contiguous memory, indexed starting at 0.
- Accessing an array out of bounds is undefined behavior — C++ will not catch it for you.
- `const` marks a value as read-only and makes your intent clear.
- With pointers, `const` can protect the pointer, the data, or both.
- Structures group related data together; assignment copies all members.

Exercises

1. On a system where `int` is 4 bytes, what is `sizeof(scores)` for `int scores[10]`?
2. What does the following program print?

```
#include <iostream>

int main()
{
    char c = 'C';
```

```
    c = c + 3;
    std::cout << c << std::endl;
    return 0;
}
```

3. What is wrong with the following code?

```
int data[3] = {10, 20, 30};
std::cout << data[3] << std::endl;
```

4. Consider the following declarations:

```
const int *p1 = nullptr;
int x = 42;
int *const p2 = &x;
```

Which one prevents you from changing the value being pointed to? Which one prevents you from changing where the pointer points?

5. What does the following program print?

```
#include <iostream>

struct Punto {
    int x;
    int y;
};

int main()
{
    Punto a = {3, 7};
    Punto b = a;
    b.x = 10;
    std::cout << a.x << " " << b.x << std::endl;
    return 0;
}
```

6. Why is it important to initialize variables before using them? What could happen if you read from an uninitialized `int`?

7. If `short` is 2 bytes, what is the maximum value an `unsigned short` can hold? How does this differ from a signed `short`?

8. Write a program that declares a structure to hold information about a car (make, model, year) and creates an array of 3 cars. Print out each car's information.

9. What does `std::numeric_limits<uint8_t>::max()` return? What about `std::numeric_limits<double>::min()` — is it a large negative number?

10. **What does this print?**

```
#include <iostream>

int main()
{
    auto a = 42;
    auto b = 42.0;
    auto c = 42 / 5;
    auto d = 42.0 / 5;
    std::cout << a << " " << b << " " << c << " " << d << "\n";
```

```
    return 0;
}
```

What is the deduced type of each variable?

11. **Calculation:** Given this declaration, what is the value at `grid[1][2]`?

```
int grid[3][4] = {
    {0, 1, 2, 3},
    {4, 5, 6, 7},
    {8, 9, 10, 11},
};
```

What is `sizeof(grid)` on a system where `int` is 4 bytes? How many `int` elements does `grid` hold in total?

12. **What does this print?**

```
#include <iostream>

int main()
{
    unsigned char x = 250;
    x = x + 10;
    std::cout << static_cast<int>(x) << "\n";
    return 0;
}
```

Why does `unsigned char` produce that result instead of `260`?

13. **What does this print?** Use the ASCII table to figure it out without running the code.

```
#include <iostream>

int main()
{
    char a = 'a';
    char b = a + 4;
    std::cout << b << " " << static_cast<int>(b) << std::endl;
    return 0;
}
```