Gorgo Starting C++

April 11, 2026

Contents

15. Odds and Ends

You have come a long way. You can write programs with variables, control flow, functions, classes, containers, and file I/O. But there are practical gaps that come up in real programs. What if you need to terminate a program from deep inside a nested function call? What if you need to call a C library from C++? What if you need to convert between types safely, measure how long something takes, or generate random numbers? This chapter covers those remaining topics: `exit()` for program termination, `extern "C"` for C interoperability, C++ casting operators, the `<chrono>` time library, and the `<random>` library.

exit()

You already know that `return 0;` in `main()` ends the program successfully. But what if you need to stop the program from deep inside a function, not just from `main()`?

The `exit()` function, declared in `<cstdlib>`, terminates the program immediately from anywhere. Its signature is:

```
void exit(int status);
```

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>

void load_config(const std::string& filename)
{
    std::ifstream infile(filename);

    if (!infile) {
        std::cerr << "Fatal: could not open " << filename << std::endl;
        exit(EXIT_FAILURE);
    }

    std::cout << "Config loaded." << std::endl;
}

int main()
{
    load_config("settings.cfg");
    std::cout << "Program running..." << std::endl;

    return EXIT_SUCCESS;
}
```

If `settings.cfg` does not exist, the program prints an error and stops immediately. The line "Program running..." never executes.

`EXIT_SUCCESS` and `EXIT_FAILURE` are constants defined in `<cstdlib>`. `EXIT_SUCCESS` is typically `0` and `EXIT_FAILURE` is typically `1`, but using the named constants makes your intent clearer.

exit() vs return

When you call `return` from `main()`, the program exits in a controlled way — local variables in `main()` are cleaned up, destructors are called. `exit()` also performs cleanup: it flushes output streams and calls functions registered with `atexit()`. The signature of `atexit()` is:

```
int atexit(void (*func)());
```

However, local variables on the stack do not have their destructors called when `exit()` is used.

Tip: Prefer `return` from `main()` when possible. Use `exit()` when you need to terminate from a function deep in the call stack and returning an error code all the way up to `main()` would be impractical.

Trap: Because `exit()` does not call destructors for local variables on the stack, resources managed by RAII (like file handles or smart pointers in local scope) may not be cleaned up properly. Use `exit()` sparingly and with awareness of this limitation.

extern "C"

C++ grew out of C, and there is a massive amount of existing C code in the world. Sometimes you need to call C functions from C++, or make C++ functions callable from C. The key to this is `extern "C"`.

Name Mangling

When you write a function in C++, the compiler does not store the function name as-is in the compiled output. Instead, it **mangles** the name — it encodes the function name along with its parameter types into a unique symbol.

This is necessary because C++ supports function overloading. Consider:

```
void play(int track);
void play(const std::string& song);
void play(int track, bool repeat);
```

All three functions are named `play`, but they take different parameters. The compiler needs to tell them apart in the compiled output, so it might mangle them into something like `_Z4playi`, `_Z4playRKSs`, and `_Z4playib`. The exact mangled names depend on the compiler, but the point is that each overload gets a unique name.

C does not have function overloading, so C compilers do not mangle names. A C function called `play` is simply stored as `play` in the compiled output.

This creates a problem: if you try to call a C function from C++, the C++ compiler will look for a mangled name that does not exist.

Disabling Name Mangling

`extern "C"` tells the C++ compiler: "do not mangle this function name — use C-style naming."

```
extern "C" void c_function();
```

Now the C++ compiler knows to look for `c_function` without mangling, matching what a C compiler would produce.

Calling C Functions from C++

Imagine you have a C library with a function you want to use. You would declare it with `extern "C"`:

```
#include <iostream>

// Tell C++ this function uses C naming conventions
extern "C" {
    double sqrt(double x);
    int abs(int n);
}
```

```cpp
int main()
{
    std::cout << "La copa de la vida: sqrt(1998) = "
              << sqrt(1998) << std::endl;
    std::cout << "abs(-1) = " << abs(-1) << std::endl;

    return 0;
}
```

The `extern "C"` block tells the compiler that all functions declared inside it use C linkage.

Tip: In practice, you rarely need to write `extern "C"` declarations yourself. Most C library headers already handle this. But understanding why it exists helps you debug linker errors when mixing C and C++ code.

Wrapping C Headers

If you write a header that needs to work in both C and C++ code, you can use a common pattern:

```cpp
#ifdef __cplusplus
extern "C" {
#endif

void mi_funcion(int x);
int otra_funcion(const char* s);

#ifdef __cplusplus
}
#endif
```

`__cplusplus` is a macro that is defined only when compiling with a C++ compiler. When compiled as C++, the functions get `extern "C"` linkage. When compiled as C, the `extern "C"` parts are skipped entirely because C does not understand that syntax.

Wut: `extern "C"` does not mean "compile this as C code." The code inside `extern "C"` is still C++ — you can use C++ features. It only affects how the function name is stored in the compiled output.

Numbers and Casting

Everything is a Number

To the CPU, there are no strings, no classes, no booleans. There are only numbers — sequences of bits stored in memory. The types you use in C++ tell the compiler how to interpret those bits.

An `int` is a number you want to do arithmetic with. A `char` is also a number — just a smaller one. When you write `'A'`, the compiler stores the number `65`. When you write `'0'`, it stores `48`.

```cpp
#include <iostream>

int main()
{
    char letter = 'A';
```

```
    std::cout << "As char: " << letter << std::endl;
    std::cout << "As int:  " << static_cast<int>(letter) << std::endl;
    std::cout << "'A' + 1 = " << static_cast<char>(letter + 1) << std::endl;

    return 0;
}
```

Output:

```
As char: A
As int:  65
'A' + 1 = B
```

The same bits that represent the character `'A'` also represent the integer `65`. The type is just a label that tells the compiler what to do with the number.

Bit Widths and Ranges

Different types use different numbers of bits, which determines the range of values they can hold.

Type	Bits	Minimum	Maximum
`int8_t`	8	-128	127
`uint8_t`	8	0	255
`int16_t`	16	-32,768	32,767
`uint16_t`	16	0	65,535
`int32_t`	32	-2,147,483,648	2,147,483,647
`uint32_t`	32	0	4,294,967,295

These fixed-width types from `<cstdint>` guarantee exactly how many bits they use. The regular `int` is at least 16 bits but usually 32 bits on modern systems.

Trap: If you store a value too large for a type, it wraps around. For `uint8_t`, `255 + 1` becomes `0`. For `int8_t`, `127 + 1` becomes `-128`. This is a common source of subtle bugs.

C++ Casts

Sometimes you need to convert a value from one type to another. This is called **casting**. C++ provides four named cast operators that are safer and more expressive than the old C-style cast.

static_cast `static_cast` is the most common cast. Its syntax is:

```
static_cast<new_type>(expression)
```

Use it for well-defined, compile-time conversions between related types.

```
double pi = 3.14159;
int truncated = static_cast<int>(pi);  // 3 --- decimal part is lost

int score = 98;
double pct = static_cast<double>(score) / 100;  // 0.98
```

This is the cast you will use most often. It handles numeric conversions, conversions between related pointer types in a class hierarchy, and other conversions the compiler can verify at compile time.

dynamic_cast `dynamic_cast` is used for safe downcasting in class hierarchies with virtual functions. Its syntax is:

```
dynamic_cast<new_type>(expression)
```

It checks at runtime whether the cast is valid.

```
#include <iostream>

class Base {
public:
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void special() { std::cout << "Do you believe?" << std::endl; }
};

int main()
{
    Base* bp = new Derived();
    Derived* dp = dynamic_cast<Derived*>(bp);

    if (dp != nullptr) {
        dp->special();
    } else {
        std::cout << "Cast failed" << std::endl;
    }

    delete bp;
    return 0;
}
```

If the object pointed to by `bp` is not actually a `Derived`, `dynamic_cast` returns `nullptr` instead of producing undefined behavior.

Tip: `dynamic_cast` only works with polymorphic types — classes that have at least one `virtual` function. If you have not studied inheritance yet, just know that this cast exists for safely converting between related class types at runtime.

const_cast `const_cast` adds or removes `const` from a pointer or reference. Its syntax is:

```
const_cast<new_type>(expression)
```

This is rarely needed and usually a sign that something in the design should be reconsidered.

```
#include <iostream>

void legacy_print(char* s)
{
    std::cout << s << std::endl;
}

int main()
{
```

```
    const char* song = "Believe";
    // legacy_print(song);  // Error: cannot convert const char* to char*
    legacy_print(const_cast<char*>(song));  // Compiles, but be careful

    return 0;
}
```

The main legitimate use is interfacing with old C APIs that take non-const pointers but promise not to modify the data.

Trap: If you use `const_cast` to remove `const` and then actually modify the data, the behavior is undefined if the original object was declared as `const`. Only use `const_cast` when you are certain the data will not be modified.

reinterpret_cast `reinterpret_cast` tells the compiler to treat the bits of one type as if they were another type entirely. Its syntax is:

```
reinterpret_cast<new_type>(expression)
```

This is the most dangerous cast and should be used rarely.

```
#include <iostream>
#include <cstdint>

int main()
{
    int value = 42;
    uintptr_t addr = reinterpret_cast<uintptr_t>(&value);

    std::cout << "Address of value: " << addr << std::endl;

    return 0;
}
```

This cast performs no conversion — it just reinterprets the bit pattern. It is used in low-level code like memory allocators or hardware interfaces.

Wut: `reinterpret_cast` does not change the bits at all. A `static_cast` from `float` to `int` actually converts the value (3.14 becomes 3). A `reinterpret_cast` would take the raw bits of the `float` and pretend they are an `int`, producing a completely different and probably meaningless number.

Why C++ Casts Over C-Style Casts?

In C (and in C++), you can cast with the syntax `(type)value`:

```
double pi = 3.14;
int n = (int)pi;  // C-style cast --- works but not recommended in C++
```

C++ also allows a **functional-style cast** that looks like a function call:

```
int n = int(pi);  // functional-style cast --- same thing, different syntax
```

Both forms are equivalent — `(int)pi` and `int(pi)` do exactly the same thing. Neither is recommended in new C++ code.

The problem with both forms is that they are blunt instruments. They can silently perform any of the four C++ casts, and you cannot tell which one just by looking at the code.

The C++ named casts are preferred because:

- **They express intent.** `static_cast` says "this is a safe, well-defined conversion." `reinterpret_cast` says "I know this is dangerous."
- **They are searchable.** You can search your codebase for `reinterpret_cast` to find all the dangerous casts. Good luck finding all the C-style casts with a search.
- **They are restrictive.** Each C++ cast only allows certain conversions. A C-style cast can do anything, including things you did not intend.

Tip: Use `static_cast` for safe conversions between numeric types. Use `dynamic_cast` for safe downcasting in class hierarchies. Avoid `const_cast` and `reinterpret_cast` unless you have a very specific reason. Never use C-style casts in new C++ code.

Time

Programs often need to work with time — measuring how long something takes, pausing execution, or converting between time units. C++ provides the `<chrono>` library for this.

Measuring Elapsed Time

The most common use of `<chrono>` is measuring how long a piece of code takes to run. `std::chrono::steady_clock` is the right clock for this because it never jumps forward or backward. The key functions are:

```
static time_point steady_clock::now();          // current time
Duration duration_cast<Duration>(duration d);   // convert between time units
Rep duration::count() const;                    // get the numeric value
void this_thread::sleep_for(duration d);        // pause execution
```

```
#include <chrono>
#include <iostream>
#include <thread>

int main()
{
    auto start = std::chrono::steady_clock::now();

    // Simulate some work
    std::this_thread::sleep_for(std::chrono::milliseconds(150));

    auto end = std::chrono::steady_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::milliseconds>(
            end - start);

    std::cout << "That took " << elapsed.count() << " ms" << std::endl;

    return 0;
}
```

Output (approximately):

```
That took 150 ms
```

`steady_clock::now()` returns a time point. Subtracting two time points gives a duration. `duration_cast` converts that duration to the units you want — milliseconds, microseconds, seconds, etc.

Duration Arithmetic

Durations are type-safe. You cannot accidentally mix up seconds and milliseconds because they are different types. The library handles conversions automatically when it is safe.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::chrono;

    seconds two_min = minutes(2);
    milliseconds half_sec = milliseconds(500);

    std::cout << "2 minutes = "
              << two_min.count()
              << " seconds" << std::endl;
    std::cout << "500 ms = " << duration_cast<seconds>(half_sec).count()
              << " seconds" << std::endl;

    auto mixed = seconds(3) + milliseconds(250);
    std::cout << "3s + 250ms = "
              << duration_cast<milliseconds>(mixed).count()
              << " ms" << std::endl;

    return 0;
}
```

Output:

```
2 minutes = 120 seconds
500 ms = 0 seconds
3s + 250ms = 3250 ms
```

Notice that converting 500 milliseconds to seconds gives `0`, not `0.5`. `duration_cast` truncates — it does not round. This is the same behavior as integer division.

Tip: Use `std::chrono::steady_clock` for measuring elapsed time. `system_clock` can jump forward or backward (e.g., when the system clock is adjusted), which would throw off your measurements.

Trap: `duration_cast` truncates toward zero. If you need to know that an operation took 1.7 seconds, cast to milliseconds (1700) rather than seconds (1).

Random Numbers

Generating random numbers comes up surprisingly often — games, simulations, testing, shuffling data. C++ provides a proper random number library in `<random>`.

The Old Way: rand()

C provides `rand()` and `srand()` in `<cstdlib>`. Their signatures are:

```
int rand();                          // returns a pseudo-random integer
void srand(unsigned int seed);       // seeds the random number generator
// returns current calendar time (from <ctime>)
time_t time(time_t* arg);
```

You might see them in older code:

```
#include <cstdlib>
#include <ctime>
#include <iostream>

int main()
{
    srand(static_cast<unsigned>(time(nullptr)));  // Seed with current time
    std::cout << rand() % 10 << std::endl;        // 0-9, but biased!

    return 0;
}
```

This works but has problems. `rand()` produces low-quality random numbers on many systems. Using `%` to get a range introduces bias — some numbers come up more often than others. And `srand(time(nullptr))` means two programs started in the same second get the same sequence.

Trap: Avoid `rand()` and `srand()` in new C++ code. They exist for C compatibility but produce poor randomness and make it easy to introduce subtle bias.

The C++ Way: Engines and Distributions

The `<random>` library separates two concerns: generating raw random bits (the **engine**) and shaping those bits into the range and distribution you want (the **distribution**). The key components and their signatures are:

```
// std::random_device
unsigned int operator()();   // produces a random seed

// std::mt19937
mt19937(unsigned int seed);  // construct with seed

// distributions
uniform_int_distribution(IntType a, IntType b);     // integers in [a, b]
uniform_real_distribution(RealType a, RealType b);   // reals in [a, b)
ResultType operator()(Generator& gen);               // generate a value
```

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<int> track(1, 12);
```

```
    std::uniform_real_distribution<double> score(0.0, 10.0);

    std::cout << "Random track: " << track(gen) << std::endl;
    std::cout << "Random score: " << score(gen) << std::endl;

    std::cout << "Cinco tracks al azar: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << track(gen) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Possible output:

```
Random track: 7
Random score: 3.14159
Cinco tracks al azar: 11 3 7 1 9
```

Here is what each piece does:

- **`std::random_device rd`** provides a seed from your operating system's entropy source — truly unpredictable.
- **`std::mt19937 gen(rd())`** creates a Mersenne Twister engine seeded with that random value. This engine produces high-quality pseudo-random numbers.
- **`std::uniform_int_distribution<int> track(1, 12)`** takes the engine's output and maps it to an integer in $[1, 12]$, with each value equally likely.
- **`std::uniform_real_distribution<double> score(0.0, 10.0)`** does the same for floating-point values in $[0.0, 10.0)$.

Tip: Create the engine once and reuse it. Creating a new `std::mt19937` for every random number is wasteful and can produce poor results if seeded with similar values.

Wut: `std::random_device` is not guaranteed to be truly random on all platforms. On some systems it may fall back to a pseudo-random generator. In practice, on Linux, macOS, and Windows, it reads from the OS entropy pool and is fine for seeding.

Other Distributions

`uniform_int_distribution` and `uniform_real_distribution` give every value in the range an equal chance. But sometimes you want values that cluster around a center — this is a **normal distribution** (also called a Gaussian or bell curve).

```
std::normal_distribution<RealType>(RealType mean, RealType stddev);
```

The `mean` is the center of the bell curve. The `stddev` (standard deviation) controls how spread out the values are — about 68% of values fall within one standard deviation of the mean, and about 95% within two.

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
```

```
    std::mt19937 gen(rd());

    std::normal_distribution<double> rating(7.0, 1.5);

    std::cout << "Diez puntuaciones al azar:" << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << rating(gen) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Most values will be close to 7.0, with occasional values farther away. The `<random>` header provides many other distributions (Bernoulli, Poisson, etc.), but uniform and normal cover most practical needs.

Key Points

- `exit()` terminates the program from any function; prefer `return` from `main()` when possible.
- `EXIT_SUCCESS` and `EXIT_FAILURE` are portable constants for exit codes.
- C++ mangles function names to support overloading; C does not.
- `extern "C"` disables name mangling so C and C++ code can link together.
- Use `#ifdef __cplusplus` guards to write headers that work in both C and C++.
- To the CPU, everything is a number — types tell the compiler how to interpret the bits.
- A `char` is just a small integer; `'A'` is 65.
- Different bit widths give different value ranges; overflow wraps around.
- Prefer `static_cast` for safe conversions, `dynamic_cast` for safe downcasting, and avoid `const_cast` and `reinterpret_cast` unless necessary.
- Never use C-style casts in new C++ code — use the named C++ casts instead.
- Use `std::chrono::steady_clock` to measure elapsed time; `duration_cast` converts between time units but truncates.
- Avoid `rand()` and `srand()` — use `<random>` with an engine (`std::mt19937`) and a distribution (`std::uniform_int_distribution`, etc.).
- Seed the engine with `std::random_device` for unpredictable results.
- `std::normal_distribution` generates values clustered around a mean with a given standard deviation (bell curve).

Exercises

1. What does the following program print if the file `data.txt` does not exist?

```
#include <cstdlib>
#include <fstream>
#include <iostream>

void read_file()
{
    std::ifstream f("data.txt");
    if (!f) {
        std::cout << "A" << std::endl;
        exit(EXIT_FAILURE);
    }
    std::cout << "B" << std::endl;
}
```

```cpp
int main()
{
    read_file();
    std::cout << "C" << std::endl;
    return EXIT_SUCCESS;
}
```

2. What is name mangling, and why does C++ do it but C does not?

3. A coworker writes the following C++ code to call a C library function but gets a linker error about an undefined symbol. What is the fix?

```cpp
// my_program.cpp
#include <iostream>

void c_library_init();

int main()
{
    c_library_init();
    std::cout << "Ready" << std::endl;
    return 0;
}
```

4. What is the value of `x` after this code runs?

```cpp
uint8_t x = 250;
x = x + 10;
```

5. What does the following program print?

```cpp
#include <iostream>

int main()
{
    char c = 48;
    std::cout << c << std::endl;
    std::cout << static_cast<int>(c) << std::endl;
    return 0;
}
```

6. Explain why this C-style cast is dangerous and what C++ cast you should use instead:

```cpp
void* ptr = get_some_pointer();
int* ip = (int*)ptr;
```

7. What is the difference between `static_cast<int>(3.14)` and `reinterpret_cast<int>(3.14)`? Will the second one even compile?

8. What does the `#ifdef __cplusplus` guard accomplish in a C/C++ shared header? When would the code inside the `#ifdef` be skipped?

9. Write a program that takes an `int` and prints it as a `char`, and takes a `char` and prints its integer value. Use `static_cast` for both conversions. Test it with the value `65` and the character `'Z'`.

10. What does the following program print?

```cpp
#include <chrono>
#include <iostream>

int main()
```

```
{
    using namespace std::chrono;

    auto d = seconds(5) + milliseconds(750);
    std::cout << duration_cast<seconds>(d).count() << std::endl;

    return 0;
}
```

11. Why should you use `std::chrono::steady_clock` instead of `std::chrono::system_clock` when measuring how long a piece of code takes to run?

12. What is wrong with this code for generating a random number between 1 and 100?

```
#include <cstdlib>
#include <iostream>

int main()
{
    int r = rand() % 100 + 1;
    std::cout << r << std::endl;

    return 0;
}
```

13. Write a program that uses `<random>` to simulate rolling two six-sided dice 10 times and prints each roll.

14. Write a program that generates 10 random values using `std::normal_distribution` with a mean of 100 and a standard deviation of 15. Print each value. Are most values close to 100?

15. **What does this print?**

```
#include <iostream>

struct Track { virtual ~Track() = default; };
struct AudioTrack : Track { void play() { std::cout << "audio\n"; } };
struct VideoTrack : Track { void play() { std::cout << "video\n"; } };

void play(Track *t)
{
    if (auto *a = dynamic_cast<AudioTrack *>(t)) {
        a->play();
    } else if (auto *v = dynamic_cast<VideoTrack *>(t)) {
        v->play();
    } else {
        std::cout << "unknown\n";
    }
}

int main()
{
    AudioTrack a;
    VideoTrack v;
    Track      t;
    play(&a);
    play(&v);
    play(&t);
```

```
    return 0;
}
```

Why does the base class need a `virtual` destructor for `dynamic_cast` to work here?

16. **Where is the bug?**

```
void uppercase_first(const std::string &s)
{
    char &first = const_cast<char &>(s[0]);
    first = static_cast<char>(std::toupper(first));
}

int main()
{
    const std::string title = "wonderwall";
    uppercase_first(title);
    std::cout << title << "\n";
    return 0;
}
```

The function compiles, but it is undefined behavior at runtime. Explain what `const_cast` is doing here and why this particular use of it is broken.

17. **Calculation:** Given the fixed-width types from `<cstdint>`, how many bytes does each of the following take, and what is the largest value it can hold?

```
int8_t   a;
uint8_t  b;
int16_t  c;
uint32_t d;
int64_t  e;
```

Why prefer `int32_t` over `int` when you need exactly 32 bits, and why prefer `int` over `int32_t` for ordinary counters?

18. **Think about it:** What is the difference between calling `std::exit(0)` and writing `return 0;` from `main`? Specifically, which destructors run in each case? Sketch a small program with a class that prints from its destructor and predict what each version prints.

19. **Write a program** that seeds a `std::mt19937` from `std::random_device`, then uses it with a `std::uniform_int_distribution<int>(1, 100)` to print 10 random integers in the range `[1, 100]`. Now run the program twice and compare: do you get the same numbers each time? Then change the program to use a fixed seed (`std::mt19937 rng(42);`) and run it twice again. What changed, and why?