Gorgo Starting C++

April 11, 2026

Contents

13. Memory Management

Every variable you have created so far lives on the **stack** — a region of memory that is managed automatically. When a variable goes out of scope, its memory is reclaimed for you. This is simple and reliable, but it has limits.

Sometimes you need memory that outlives the current scope, or you need to allocate a size that is not known until the program is running. For that, C++ gives you the **heap** — a larger region of memory that you control explicitly.

In this chapter, you will learn how heap memory works, why manual management is error-prone, and how modern C++ smart pointers make it safe and easy.

Stack vs. Heap

The **stack** is fast and automatic. Every time you declare a local variable, it goes on the stack. When the function returns, all its stack variables are destroyed:

```
void play() {
    int volume = 11;  // lives on the stack
}  // volume is destroyed here
```

The **heap** (also called **free store**) is a separate pool of memory. You request memory from the heap at runtime, and it stays allocated until you explicitly release it — or until the program ends.

The stack is like a concert venue's coat check: you hand in your coat, get a ticket, and pick it up on the way out. The heap is like renting a storage unit: it is yours until you cancel the lease.

Why Not Just Use the Stack?

Stack memory has two limitations that come up in real programs.

The size must be known at compile time. If the user decides how many items to store, you cannot create a stack array for it:

```
void record_scores() {
    int count;
    std::cout << "How many scores? ";
    std::cin >> count;

    // int scores[count];  // NOT standard C++ --- size must be a constant
}
```

Trap: Most compilers will actually accept `int scores[count];` without complaint. This is a **variable-length array** (VLA), a feature from C99 that some C++ compilers support as an extension. Do not use it. VLAs were never part of the C++ standard, they allocate on the stack so a large `count` can overflow it and crash your program, and their support varies across compilers and flags. Use `std::vector<int>` (Chapter 8) or heap allocation instead.

The heap lets you allocate exactly as much memory as you need at runtime.

Stack variables die when the function returns. If a function creates an object and needs to hand it back to the caller, a stack variable will not work — it is destroyed before the caller can use it:

```
#include <string>

std::string* make_greeting() {
    std::string local = "Don't Speak";
```

```
    return &local;   // BUG: local is destroyed when the function returns
}                     // the caller gets a dangling pointer
```

The caller receives a pointer to memory that no longer exists. The heap solves this because heap memory persists until you explicitly free it.

Tip: Prefer stack allocation whenever possible. It is faster, automatic, and less error-prone. Only use the heap when you need memory to outlive the current scope or when the size is not known at compile time.

Pointers

Before you can work with the heap, you need to understand **pointers**. A pointer is a variable that holds the memory address of another variable. You have made it this far without needing pointers directly because references, smart pointers, and standard containers handle most situations. But `new` returns a pointer, so you need the basics.

Getting an Address

The **address-of operator** `&` gives you the memory address of a variable:

```
int volume = 11;
std::cout << &volume << std::endl;  // prints something like 0x7ffd3a2c
```

The exact number depends on where the operating system placed `volume` in memory.

Declaring a Pointer

A pointer variable is declared by putting `*` after the type. It stores an address, not a value:

```
int volume = 11;
int *ptr = &volume;   // ptr holds the address of volume
```

`int *ptr` reads as "ptr is a pointer to an int."

You can also have a pointer to a pointer:

```
int **pptr = &ptr;   // pptr holds the address of ptr
```

This comes up when you need to modify a pointer itself through a function, or when dealing with arrays of pointers (like `argv` from Chapter 1, which is really `char **`).

Dereferencing

To access the value a pointer points to, use the **dereference operator** `*`:

```
int volume = 11;
int *ptr = &volume;

std::cout << *ptr << std::endl;   // prints 11 --- the value at the address

*ptr = 5;                         // changes volume through the pointer
std::cout << volume << std::endl;  // prints 5
```

Wut: The `*` symbol does three different things depending on context. In a declaration like `int *ptr`, it means "pointer to." In an expression like `*ptr`, it means "dereference." In an expression like `a * b`, it means multiplication. The compiler always knows which is which from context, even if the reader has to think for a moment.

The Arrow Operator

When you have a pointer to a structure or class, you need to dereference it before accessing a member. The parentheses are required because `.` has higher precedence than `*`:

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

Song s = {"Popular", 1996};
Song *ptr = &s;

std::cout << (*ptr).title << std::endl;  // works but awkward
```

Because `(*ptr).member` is tedious to write, C++ provides the **arrow operator** `->` as a shorthand:

```
std::cout << ptr->title << std::endl;   // same thing, much cleaner
std::cout << ptr->year << std::endl;
```

`ptr->member` is exactly equivalent to `(*ptr).member`. You will see `->` everywhere in C++ — it is the standard way to access members through a pointer.

nullptr

A pointer that does not point to anything should be set to `nullptr`:

```
int *ptr = nullptr;   // points to nothing
```

Historically C++ used `NULL` to indicate a pointer to nothing. In C++, `NULL` is literally the integer `0`, which is recognized as an invalid address. C still uses `NULL`, and many older C++ code bases do too, but `nullptr` is preferred in modern C++ because it can be distinguished from an `int`. For example:

```
void look_up(int index);
void look_up(void *addr);
```

You would expect `look_up(NULL)` to invoke `void look_up(void *addr);` however, since `NULL` is the integer `0`, it matches the first `look_up` instead. `look_up(nullptr)` unambiguously calls the pointer overload.

Dereferencing a null pointer is undefined behavior — your program will almost certainly crash. Always check before dereferencing a pointer you are not sure about:

```
if (ptr != nullptr) {
    std::cout << *ptr << std::endl;
}
```

Tip: Modern C++ reduces the need for raw pointers significantly. References (Chapter 6) are safer for "point to an existing object" cases, and smart pointers (covered later in this chapter) are safer for heap memory. Raw pointers still appear in older code, C library interfaces, and `argv`, so you need to recognize them.

`new` and `delete`

The `new` operator allocates memory on the heap and returns a pointer to it. The `delete` operator releases that memory:

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

```cpp
int main()
{
    std::string *song = new std::string("Under the Bridge");
    std::cout << *song << std::endl;
    delete song;  // free the memory

    return 0;
}
```

Output:

```
Under the Bridge
```

After `delete`, the pointer `song` still exists, but the memory it points to has been freed. Using the pointer after `delete` is **undefined behavior**.

`new[]` and `delete[]` for Arrays

To allocate an array on the heap, use `new[]` and `delete[]`:

```cpp
int *scores = new int[5];  // allocate array of 5 ints

scores[0] = 10;
scores[1] = 20;
// ...

delete[] scores;  // free the array
```

Trap: If you allocate with `new[]`, you *must* free with `delete[]`. Using plain `delete` on an array allocated with `new[]` is undefined behavior. The compiler will not warn you — it will just silently corrupt memory.

Memory Leaks and Dangling Pointers

Manual memory management with `new` and `delete` is notoriously error-prone. Two of the most common bugs are **memory leaks** and **dangling pointers**.

Memory Leaks

A **memory leak** occurs when you allocate memory but never free it:

```cpp
void leak() {
    std::string *s = new std::string("Nothing Compares 2 U");
    // oops --- we never delete s
}  // s is destroyed, but the string on the heap lives on
```

Every time `leak()` is called, it allocates memory that is never released. Over time, your program eats more and more memory until it runs out.

Dangling Pointers

A **dangling pointer** is a pointer that refers to memory that has already been freed:

```cpp
int *p = new int(42);
delete p;
std::cout << *p << std::endl;  // DANGER: p is dangling
```

Dereferencing a dangling pointer is undefined behavior. Your program might crash, print garbage, or appear to work fine — until it does not.

Trap: After `delete`, set the pointer to `nullptr` if you plan to keep the pointer variable around. This does not prevent all dangling pointer bugs, but it makes it easier to check if a pointer is valid.

These problems are why modern C++ strongly discourages using raw `new` and `delete`. The solution is smart pointers.

Smart Pointers

Smart pointers are objects that manage heap memory for you. When a smart pointer goes out of scope, it automatically deletes the memory it owns. This pattern is called **RAII** — Resource Acquisition Is Initialization. The idea is that a resource (like heap memory) is tied to an object's lifetime: acquired in the constructor, released in the destructor.

Smart pointers live in the `<memory>` header.

`std::unique_ptr`

A `std::unique_ptr` represents **sole ownership** of a heap-allocated object. Only one `unique_ptr` can own a given piece of memory at a time. When the `unique_ptr` is destroyed, the memory is automatically freed.

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

int main()
{
    auto song =
        std::make_unique<std::string>("Don't Speak");
    std::cout << *song << std::endl;

    // no delete needed --- memory is freed when song goes out of scope
    return 0;
}
```

Output:

```
Don't Speak
```

Its signature is:

```cpp
std::unique_ptr<T> make_unique(Args... args);
```

`std::make_unique<T>(args...)` allocates a new `T` on the heap, passes `args` to its constructor, and wraps the result in a `unique_ptr`. Always prefer `make_unique` over `new`.

Because ownership is exclusive, you cannot copy a `unique_ptr`:

```cpp
std::unique_ptr<int> a = std::make_unique<int>(42);
std::unique_ptr<int> b = a;  // ERROR: cannot copy a unique_ptr
```

But you can **move** it (more on this shortly):

```cpp
std::unique_ptr<int> a = std::make_unique<int>(42);
std::unique_ptr<int> b = std::move(a);  // OK: ownership transferred to b
// a is now empty (nullptr)
```

Tip: `std::unique_ptr` should be your default choice for heap allocation. It has zero overhead compared to a raw pointer — the compiler generates the same code, but with automatic cleanup.

`std::shared_ptr`

Sometimes multiple parts of your code need to share ownership of the same object. A `std::shared_ptr` uses **reference counting** to track how many `shared_ptr`s point to the same memory. The memory is freed only when the last `shared_ptr` owning it is destroyed.

You create a `shared_ptr` with `std::make_shared`, whose signature mirrors `make_unique`:

```
std::shared_ptr<T> make_shared(Args... args);
```

Two useful member functions for inspecting and managing a `shared_ptr` are `use_count()` and `reset()`:

```
long use_count() const;   // returns the current reference count
void reset();             // releases this shared_ptr's ownership
```

```
#include <iostream>
#include <memory>
#include <string>

int main()
{
    auto song1 =
        std::make_shared<std::string>("Under the Bridge");
    // both point to the same string
    std::shared_ptr<std::string> song2 = song1;

    std::cout << *song1 << std::endl;
    std::cout << *song2 << std::endl;
    std::cout << "ref count: " << song1.use_count() << std::endl;

    song1.reset();  // song1 gives up ownership
    std::cout << "ref count: " << song2.use_count() << std::endl;

    // memory is freed when song2 goes out of scope
    return 0;
}
```

Output:

```
Under the Bridge
Under the Bridge
ref count: 2
ref count: 1
```

`std::make_shared` is the preferred way to create a `shared_ptr`, just as `make_unique` is for `unique_ptr`.

Tip: Use `shared_ptr` only when you truly need shared ownership. If one owner is enough, use `unique_ptr` instead — it is simpler and has no reference-counting overhead.

Getting a Raw Pointer from a Smart Pointer

Sometimes you need to pass a raw pointer to a function that does not understand smart pointers — a C library function, for example. Both `std::unique_ptr` and `std::shared_ptr` provide a `.get()` method that returns the raw pointer without releasing ownership:

```
T* get() const;
```

```
auto song = std::make_unique<std::string>("Under the Bridge");

// pass the raw pointer to a function that expects std::string*
std::string *raw = song.get();
std::cout << *raw << std::endl;   // "Under the Bridge"

// song still owns the memory --- do NOT delete raw
```

Trap: Never `delete` a pointer obtained from `.get()`. The smart pointer still owns the memory and will `delete` it when it goes out of scope. Deleting it yourself causes a double-free bug.

Move Semantics

When you copy a large object — like a `std::string` with a long value — the program has to duplicate all the data. **Move semantics** offer an alternative: instead of copying the data, you *transfer* it from one object to another, leaving the source in a valid but empty state.

Think of it like giving someone your notebook instead of photocopying every page.

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

int main()
{
    std::string a = "Nothing Compares 2 U";
    std::cout << "a: " << a << std::endl;

    std::string b = std::move(a);  // move a's contents into b
    std::cout << "b: " << b << std::endl;
    std::cout << "a: " << a << std::endl;  // a is now empty

    return 0;
}
```

Output:

```
a: Nothing Compares 2 U
b: Nothing Compares 2 U
a:
```

After the move, `a` is in a valid but unspecified state — for `std::string`, that means it is empty. The actual string data was not copied; ownership of the internal buffer was transferred to `b`.

Trap: After moving from an object, do not use it unless you assign a new value to it first. The object is in a valid state, but its contents are unspecified.

Its simplified signature is:

```
T&& move(T&& value);
```

`std::move` does not actually move anything — it simply casts its argument to an **rvalue reference**, which tells the compiler "it is OK to move from this." The actual moving is done by the receiving object's move constructor or move assignment operator. You will learn about the special member functions (copy constructor, move constructor, and their assignment counterparts) in Chapter 14.

Putting It All Together

Here is a complete program that demonstrates smart pointers and move semantics:

```
#include <iostream>
#include <memory>
#include <string>

class Song {
private:
    std::string title;
    std::string artist;

public:
    Song(const std::string &t, const std::string &a) : title(t), artist(a) {
        std::cout << "  created: " << title << std::endl;
    }

    ~Song() {
        std::cout << "  destroyed: " << title << std::endl;
    }

    void print() const {
        std::cout << "  " << title << " by " << artist << std::endl;
    }
};

int main()
{
    std::cout << "--- unique_ptr ---" << std::endl;
    {
        auto song = std::make_unique<Song>("Don't Speak", "No Doubt");
        song->print();
    }  // song is destroyed here
    std::cout << std::endl;

    std::cout << "--- shared_ptr ---" << std::endl;
    {
        std::shared_ptr<Song> s1;
        {
            auto s2 = std::make_shared<Song>("Under the Bridge", "RHCP");
            s1 = s2;
            std::cout << "  ref count: " << s1.use_count() << std::endl;
        }  // s2 destroyed, but Song lives on
        std::cout << "  ref count: " << s1.use_count() << std::endl;
        s1->print();
    }  // s1 destroyed, Song is finally freed
    std::cout << std::endl;
```

```
    std::cout << "--- move ---" << std::endl;
    std::string lyrics = "Nada se compara contigo";
    std::cout << "  before: " << lyrics << std::endl;
    std::string moved = std::move(lyrics);
    std::cout << "  moved:  " << moved << std::endl;
    std::cout << "  after:  " << lyrics << std::endl;

    return 0;
}
```

Output:

```
--- unique_ptr ---
  created: Don't Speak
  Don't Speak by No Doubt
  destroyed: Don't Speak

--- shared_ptr ---
  created: Under the Bridge
  ref count: 2
  ref count: 1
  Under the Bridge by RHCP
  destroyed: Under the Bridge

--- move ---
  before: Nada se compara contigo
  moved:  Nada se compara contigo
  after:
```

Key Points

- A **pointer** holds the address of another variable. Use `&` to get an address, `*` to dereference, and `->` to access members through a pointer.
- The **stack** is fast and automatic; the **heap** requires manual management.
- `new` allocates on the heap; `delete` frees it. Use `new[]`/`delete[]` for arrays.
- Forgetting `delete` causes **memory leaks**. Using a pointer after `delete` creates a **dangling pointer**.
- **`std::unique_ptr`** provides sole ownership with automatic cleanup — use it as your default.
- **`std::shared_ptr`** provides shared ownership via reference counting — use it when multiple owners are needed.
- Always prefer `std::make_unique` and `std::make_shared` over raw `new`.
- **Move semantics** transfer resources instead of copying them, which is more efficient.
- **RAII** ties resource lifetimes to object lifetimes — acquire in the constructor, release in the destructor.

Exercises

1. What is the difference between stack and heap memory? Give one situation where you would need to use the heap.
2. What does the following program print?

```
#include <iostream>
#include <memory>

int main()
{
```

```
    auto p = std::make_shared<int>(99);
    auto q = p;
    auto r = p;

    std::cout << p.use_count() << std::endl;

    q.reset();
    std::cout << p.use_count() << std::endl;

    r.reset();
    std::cout << p.use_count() << std::endl;

    return 0;
}
```

3. What is the bug in the following code?

```
void play() {
    int *volumes = new int[3];
    volumes[0] = 7;
    volumes[1] = 9;
    volumes[2] = 11;
    delete volumes;
}
```

4. Why can you not copy a `std::unique_ptr`? What should you do instead if you want to transfer ownership?

5. After `std::move(a)` is called, is it safe to use `a`? What state is `a` in?

6. What is wrong with the following code?

```
#include <memory>
#include <iostream>

int main()
{
    int *raw = new int(42);
    std::unique_ptr<int> a(raw);
    std::unique_ptr<int> b(raw);

    std::cout << *a << std::endl;
    std::cout << *b << std::endl;
    return 0;
}
```

7. If a `std::shared_ptr` is copied 4 times (so there are 5 `shared_ptr`s total pointing to the same object), what is the reference count? How many of those `shared_ptr`s need to be destroyed before the object is freed?

8. Write a program that creates a `std::unique_ptr<std::string>` holding your favorite 90s song title. Move it to a second `unique_ptr`, then print from the second and verify the first is empty (check with `if (!ptr)`).

9. What does the following code print?

```
int x = 10;
int *p = &x;
```

```
*p = 20;
std::cout << x << std::endl;
```

10. Given a `struct Song { std::string title; int year; };` and a pointer `Song *ptr`, write two equivalent expressions to access `title` — one using `*` and `.`, the other using `->`.

11. **Where is the bug?**

```
void play(Song *song)
{
    std::cout << song->title << " (" << song->year << ")\n";
}

int main()
{
    Song *s = nullptr;
    play(s);
    return 0;
}
```

    Why is this dangerous, and what is the smallest change to `play` that makes it safe?

12. **Think about it:** Explain RAII in your own words. Why is `std::unique_ptr` an RAII wrapper around `new`/`delete`? Name two other RAII types you have already seen earlier in this book.

13. **Where is the bug?**

```
void make_playlist()
{
    std::string *fav = new std::string("Wonderwall");
    if (fav->size() > 100) {
        return;
    }
    std::cout << *fav << "\n";
    delete fav;
}
```

    The function looks fine in the common case but leaks memory in one specific path. Identify the leak and rewrite the function so it cannot leak no matter which return path is taken (without sprinkling extra `delete` calls everywhere).

14. **Write a program** that uses `std::unique_ptr<int>` to wrap a heap integer and then passes the underlying raw pointer to a small C-style function

```
void c_api(int *p)
{
    *p += 1;
}
```

    Use `.get()` to obtain the raw pointer, call `c_api`, and then print the value. Why is it important that the `unique_ptr` *keeps* ownership across the call to `c_api` — in particular, why must `c_api` not call `delete` on its parameter?