Gorgo Continuing C++

April 11, 2026

Contents

Author Intro

if you're reading this, you have already survived *Gorgo Starting C++*. congratulations! you know classes, smart pointers, exceptions, and the basics of the standard library. now it's time to go deeper.

this booklet picks up where *Gorgo Starting C++* left off. it covers the topics that working C++ programmers use every day: inheritance and polymorphism, templates, the full standard library, concurrency, and the tools and practices that make real projects manageable. these are the things that separate someone who has learned C++ from someone who uses C++ professionally.

just like *Gorgo Starting C++* this booklet is to be experimented with. programming is learned by writing code; there is just no other way that comes close to it. not everyone starts out as a great programmer, but everyone can become one by writing code either from scratch or by modifying code of others.

i hope this booklet will be something that can help you learn to love the challenge/satisfaction/power/rewards that come with being a great programmer.

ben

0. How to Use This Book

This book picks up where *Gorgo Starting C++* left off. It assumes you are comfortable with variables, strings, control flow, functions, containers, I/O, exceptions, classes, and memory management. If any of those topics feel shaky, revisit the relevant chapter in *Gorgo Starting C++* before continuing.

Each chapter introduces topics that working C++ programmers use regularly, with explanations, code examples, and exercises.

Tips

Tips call out details that you need to pay special attention to. **Traps** warn you of common mistakes made. **Wut** calls out a detail that is counter-intuitive, so make sure you pay attention.

Function Signatures

When this book introduces a new function, it shows the function's **signature** and return type. A function's signature is its name and parameter list — it uniquely identifies the function. We also show the return type so you know what the function gives back. For example:

```
std::optional<T> find_value(const std::map<K, V>& m, const K& key);
```

This tells you that `find_value` takes a map and a key and returns an `std::optional`. You do not need to understand every detail the first time you see it, but it gives you three things at a glance: what the function is called, what goes in, and what comes out.

Try It

As the intro to the most amazing programming language book ever written starts out:

> The only way to learn a new programming language is by writing programs in it.

You need to write some code. Make sure you try writing some programs from scratch. At the end of most sections is a starter program that you can type in and modify to play with. Don't use it as an excuse to avoid writing some of your own starter programs. It's the only way to master a language.

Exercises

Don't skip the exercises at the end of the chapters. You can get the answer key, but don't look at the answer key before you work out the answer yourself. If you look at the answer key first, the concepts will not sink in.

1. Object-Oriented Programming

In *Gorgo Starting C++* you learned to define classes with constructors, destructors, member functions, and operator overloads. Those classes stand alone — each type is independent, with no formal relationship to any other. But real programs often have types that share behavior. A `Circle`, a `Rectangle`, and a `Triangle` are all shapes. An `MP3File` and a `WAVFile` are both audio files. Without a way to express these relationships, you end up duplicating code across similar types or writing awkward `if/else` chains to handle each one. **Object-oriented programming** (OOP) gives you tools to model shared behavior: **inheritance** lets one class build on another, and **polymorphism** lets you write code that works with an entire family of types without knowing the specific type at compile time. In this chapter you will learn inheritance, virtual functions, abstract classes, run-time type identification, and multiple inheritance.

Inheritance

Inheritance lets you define a new class based on an existing one. The existing class is the **base class** (sometimes called the parent class), and the new class is the **derived class** (or child class).

The derived class inherits all the members of the base class and can add new members or override existing behavior.

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

class Song {
public:
    Song(const std::string& title, const std::string& artist)
        : title_(title), artist_(artist) {}

    std::string title() const { return title_; }
    std::string artist() const { return artist_; }

    void print() const
    {
        std::cout << title_ << " by " << artist_ << "\n";
    }

private:
    std::string title_;
    std::string artist_;
};

class KaraokeSong : public Song {
public:
    KaraokeSong(const std::string& title, const std::string& artist,
                const std::string& lyrics)
        : Song(title, artist), lyrics_(lyrics) {}

    std::string lyrics() const { return lyrics_; }

private:
    std::string lyrics_;
};

int main()
{
```

```
    KaraokeSong ks("Toxic", "Britney Spears", "Baby, can't you see...");
    ks.print();  // inherited from Song
    std::cout << "Lyrics: " << ks.lyrics() << "\n";

    return 0;
}
```

```
Toxic by Britney Spears
Lyrics: Baby, can't you see...
```

The `: public Song` after `KaraokeSong` means "inherit publicly from `Song`." `KaraokeSong` gets `title()`, `artist()`, and `print()` automatically. The constructor uses a **member initializer list** to call `Song`'s constructor before initializing its own members.

Access Specifiers in Inheritance

You already know `public` and `private` from *Gorgo Starting C++*. Inheritance introduces a third access level: `protected`.

Access specifier	Accessible in the class	Accessible in derived classes	Accessible outside
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
`private`	Yes	No	No

`protected` members are like `private` members that derived classes can see.

The keyword after `:` in the class definition controls how the base class's members appear in the derived class:

Inheritance type	Base `public` becomes	Base `protected` becomes	Base `private` becomes
`public`	`public`	`protected`	inaccessible
`protected`	`protected`	`protected`	inaccessible
`private`	`private`	`private`	inaccessible

Tip: Almost all inheritance in C++ uses `public`. The other forms are rare and usually a sign that composition (having a member of that type) is a better design.

Constructors and Destructors in Derived Classes

When you create a derived object, constructors run from base to derived. When the object is destroyed, destructors run from derived to base:

```
#include <iostream>

class Base {
public:
    Base() { std::cout << "Base constructed\n"; }
    ~Base() { std::cout << "Base destroyed\n"; }
};

class Derived : public Base {
public:
    Derived() { std::cout << "Derived constructed\n"; }
```

```
    ~Derived() { std::cout << "Derived destroyed\n"; }
};

int main()
{
    Derived d;
    return 0;
}
```

```
Base constructed
Derived constructed
Derived destroyed
Base destroyed
```

If the base class constructor takes parameters, you must call it explicitly in the derived class's member initializer list, as shown in the `KaraokeSong` example above.

Polymorphism and Virtual Functions

Inheritance alone lets you reuse code, but the real power comes from **polymorphism** — the ability to use a base class pointer or reference to call a function that behaves differently depending on the actual type of the object.

Consider this problem:

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

class Shape {
public:
    Shape(const std::string& name) : name_(name) {}

    std::string name() const { return name_; }

    double area() const { return 0.0; }  // not useful

private:
    std::string name_;
};

class Circle : public Shape {
public:
    Circle(double radius) : Shape("Circle"), radius_(radius) {}

    double area() const { return 3.14159 * radius_ * radius_; }

private:
    double radius_;
};

int main()
{
    Circle c(5.0);
    Shape& ref = c;  // base reference to derived object

    std::cout << c.area() << "\n";    // 78.5397 — calls Circle::area
```

```
    std::cout << ref.area() << "\n";  // 0 — calls Shape::area!

    return 0;
}
```

```
78.5397
0
```

When you call `ref.area()`, the compiler sees `ref` is a `Shape&` and calls `Shape::area()`. It does not know that `ref` actually refers to a `Circle`. This is because the function is resolved at **compile time** based on the declared type.

To fix this, make `area()` a **virtual function**:

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

class Shape {
public:
    Shape(const std::string& name) : name_(name) {}
    virtual ~Shape() = default;

    std::string name() const { return name_; }

    virtual double area() const { return 0.0; }

private:
    std::string name_;
};

class Circle : public Shape {
public:
    Circle(double radius) : Shape("Circle"), radius_(radius) {}

    double area() const override { return 3.14159 * radius_ * radius_; }

private:
    double radius_;
};

int main()
{
    Circle c(5.0);
    Shape& ref = c;

    std::cout << c.area() << "\n";    // 78.5397
    std::cout << ref.area() << "\n";  // 78.5397 — now calls Circle::area!

    return 0;
}
```

```
78.5397
78.5397
```

The `virtual` keyword tells the compiler to resolve the call at **run time** based on the actual type of the object, not the declared type of the pointer or reference.

The `override` Keyword

The `override` keyword on `Circle::area()` tells the compiler "I intend to override a virtual function from the base class." If the base class does not have a matching virtual function (maybe you misspelled the name or got the parameters wrong), the compiler will give you an error. Without `override`, you would silently create a new function instead of overriding the base one — a bug that is very hard to find.

Tip: Always use `override` when overriding virtual functions. It catches mistakes at compile time instead of run time.

Virtual Destructors

Notice the `virtual ~Shape() = default;` in the example above. When you delete a derived object through a base pointer, the destructor must be virtual. Otherwise only the base destructor runs and the derived part leaks:

```
Shape* s = new Circle(5.0);
delete s;  // without virtual ~Shape(), Circle's destructor never runs!
```

Trap: If a class has any virtual functions, its destructor should be virtual too. This is one of the most common sources of memory leaks in C++ programs.

Object Slicing

When you assign a derived object to a base object **by value**, the derived part is cut off:

```
Circle c(5.0);
Shape s = c;  // slicing! only the Shape part is copied
std::cout << s.area() << "\n";  // 0 — Shape::area, not Circle::area
```

This is called **slicing**. The `Circle`-specific data (`radius_`) is lost because `s` is a `Shape` object, not a `Circle`.

To get polymorphic behavior, always use pointers or references to base classes, never copies.

Abstract Classes and Pure Virtual Functions

In the `Shape` example, `Shape::area()` returns `0.0`, which is meaningless. A plain `Shape` does not have a real area — only specific shapes like `Circle` and `Rectangle` do. You can express this by making `area()` a **pure virtual function**:

```
class Shape {
public:
    Shape(const std::string& name) : name_(name) {}
    virtual ~Shape() = default;

    std::string name() const { return name_; }

    virtual double area() const = 0;  // pure virtual

private:
    std::string name_;
};
```

The `= 0` says "this function has no implementation in the base class." A class with at least one pure virtual function is an **abstract class** and cannot be instantiated:

```
Shape s("generic");  // error: cannot instantiate abstract class
```

Derived classes must override all pure virtual functions to be instantiable:

```
class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : Shape("Rectangle"), w_(w), h_(h) {}

    double area() const override { return w_ * h_; }

private:
    double w_, h_;
};
```

Abstract classes are C++'s way of defining **interfaces** — contracts that derived classes must fulfill.

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

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Rectangle>(4.0, 6.0));

    for (const auto& s : shapes) {
        std::cout << s->name() << ": " << s->area() << "\n";
    }

    return 0;
}
```

```
Circle: 78.5397
Rectangle: 24
```

The loop works with any `Shape` — it does not know or care whether it is a `Circle`, `Rectangle`, or something that has not been written yet. This is the power of polymorphism.

RTTI and `dynamic_cast`

Sometimes you need to know the actual type of an object behind a base pointer. C++ provides **Run-Time Type Information** (RTTI) for this through two mechanisms: `typeid` and `dynamic_cast`.

`typeid`

`typeid` returns a `std::type_info` object that identifies the type. You need `#include <typeinfo>` to use it:

```
#include <iostream>
#include <typeinfo>

int main()
{
    Circle c(3.0);
    Shape& ref = c;

    std::cout << typeid(ref).name() << "\n";  // implementation-defined, but identifies Circle
```

```
    return 0;
}
```

The output of `typeid(...).name()` is compiler-specific (GCC prints mangled names like `6Circle`, MSVC prints `class Circle`), so it is mainly useful for debugging.

dynamic_cast

`dynamic_cast` safely converts a base pointer or reference to a derived type at run time. It checks whether the conversion is valid:

```
void describe(Shape& s)
{
    if (auto* c = dynamic_cast<Circle*>(&s)) {
        std::cout << "Circle with area " << c->area() << "\n";
    } else if (auto* r = dynamic_cast<Rectangle*>(&s)) {
        std::cout << "Rectangle with area " << r->area() << "\n";
    } else {
        std::cout << "Unknown shape\n";
    }
}
```

- For pointers: `dynamic_cast` returns `nullptr` if the conversion fails.
- For references: `dynamic_cast` throws `std::bad_cast` if the conversion fails.

`dynamic_cast` only works with polymorphic types (types that have at least one virtual function). You learned `static_cast` in *Gorgo Starting C++* — `static_cast` does not check at run time and is unsafe for downcasting. Use `dynamic_cast` when you are not sure of the actual type.

Tip: If you find yourself using `dynamic_cast` frequently, it may be a sign that your class hierarchy needs redesigning. A well-designed hierarchy uses virtual functions to dispatch behavior, avoiding the need to check types manually.

Multiple Inheritance

C++ allows a class to inherit from more than one base class:

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

class Printable {
public:
    virtual ~Printable() = default;
    virtual void print() const = 0;
};

class Serializable {
public:
    virtual ~Serializable() = default;
    virtual std::string serialize() const = 0;
};

class Track : public Printable, public Serializable {
public:
    Track(const std::string& title) : title_(title) {}
```

```
    void print() const override
    {
        std::cout << title_ << "\n";
    }

    std::string serialize() const override
    {
        return "track:" + title_;
    }

private:
    std::string title_;
};

int main()
{
    Track t("Clocks");
    t.print();
    std::cout << t.serialize() << "\n";

    return 0;
}
```

```
Clocks
track:Clocks
```

This works well when the base classes are abstract interfaces with no shared state.

The Diamond Problem

Problems arise when two base classes share a common ancestor:

```
class A {
public:
    int value = 42;
};

class B : public A {};
class C : public A {};
class D : public B, public C {};
```

`D` now has **two** copies of `A::value` — one through `B` and one through `C`. Accessing `d.value` is ambiguous:

```
D d;
// d.value;       // error: ambiguous
d.B::value = 1;   // OK: specifies which copy
d.C::value = 2;   // OK: specifies which copy
```

This is called the **diamond problem** because the inheritance diagram looks like a diamond shape.

The fix is **virtual inheritance**:

```
class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};
```

Now `D` has only one copy of `A`, shared between `B` and `C`. Virtual inheritance adds overhead and complexity, so it should be used sparingly.

Tip: Most experienced C++ programmers avoid deep inheritance hierarchies and prefer **composition** (having a member of another type) over inheritance when possible. Multiple inheritance works best with abstract interface classes that have no data members.

The `final` Keyword

You can prevent a class from being inherited or a virtual function from being overridden further using `final`:

```
class Base {
public:
    virtual void process() const {}
};

class Derived final : public Base {  // no one can inherit from Derived
    void process() const final {}     // no one can override process further
};
```

`final` is useful when a class represents a complete, concrete implementation that should not be extended.

Try It: Inheritance and Polymorphism

Here is a program that uses the concepts from this chapter. Type it in, compile with `g++ -std=c++23`, and experiment:

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

class Instrument {
public:
    Instrument(const std::string& name) : name_(name) {}
    virtual ~Instrument() = default;

    std::string name() const { return name_; }
    virtual std::string play() const = 0;

private:
    std::string name_;
};

class Guitar : public Instrument {
public:
    Guitar() : Instrument("Guitar") {}
    std::string play() const override { return "Am I ever gonna see your face again?"; }
};

class Drums : public Instrument {
public:
    Drums() : Instrument("Drums") {}
    std::string play() const override { return "boom-tss-boom-tss"; }
};

class Synth : public Instrument {
public:
```

```
    Synth() : Instrument("Synth") {}
    std::string play() const override { return "wub wub wub"; }
};

int main()
{
    std::vector<std::unique_ptr<Instrument>> band;
    band.push_back(std::make_unique<Guitar>());
    band.push_back(std::make_unique<Drums>());
    band.push_back(std::make_unique<Synth>());

    std::cout << "The band is playing:\n";
    for (const auto& inst : band) {
        std::cout << "  " << inst->name() << ": " << inst->play() << "\n";
    }

    return 0;
}
```

```
The band is playing:
  Guitar: Am I ever gonna see your face again?
  Drums: boom-tss-boom-tss
  Synth: wub wub wub
```

Try adding a `Keyboard` class. Experiment with what happens if you remove `override` and misspell a function name. Try deleting the virtual destructor and running with AddressSanitizer (`-fsanitize=address`) to see if it catches the leak.

Key Points

- **Inheritance** lets a derived class reuse and extend the behavior of a base class.
- Use `public` inheritance to model "is-a" relationships.
- `protected` members are accessible in derived classes but not outside the hierarchy.
- Constructors run base-first, destructors run derived-first.
- **Virtual functions** enable polymorphism: the actual function called depends on the object's run-time type, not the declared type of the pointer or reference.
- Always use `override` when overriding virtual functions to catch mistakes at compile time.
- If a class has virtual functions, its destructor should be `virtual`.
- **Object slicing** happens when you copy a derived object into a base object by value — use pointers or references instead.
- A **pure virtual function** (`= 0`) makes a class abstract and uninstantiable.
- Abstract classes define interfaces that derived classes must implement.
- `dynamic_cast` safely converts base pointers/references to derived types at run time.
- **Multiple inheritance** works best with abstract interface classes; avoid it with data-carrying base classes.
- The **diamond problem** occurs when a class inherits the same base through two paths; **virtual inheritance** solves it.
- The `final` keyword prevents further inheritance or overriding.

Exercises

1. **Think about it:** Why does C++ require you to explicitly write `virtual` on a function instead of making all member functions virtual by default, the way Java and Python do?
2. **What does this print?**

   ```
   class A {
   ```

```
public:
    virtual std::string who() const { return "A"; }
};

class B : public A {
public:
    std::string who() const override { return "B"; }
};

A* ptr = new B();
std::cout << ptr->who() << "\n";
delete ptr;
```

3. **Where is the bug?**

```
class Base {
public:
    ~Base() { std::cout << "Base destroyed\n"; }
    virtual void greet() const { std::cout << "Hola\n"; }
};

class Derived : public Base {
public:
    ~Derived() { delete data_; }
    void greet() const override { std::cout << "Buenos dias\n"; }
private:
    int* data_ = new int(42);
};

Base* b = new Derived();
delete b;
```

4. **Think about it:** When would you use an abstract class instead of a regular base class with default implementations?

5. **What does this print?**

```
class Animal {
public:
    virtual ~Animal() = default;
    virtual std::string sound() const { return "..."; }
};

class Cat : public Animal {
public:
    std::string sound() const override { return "Meow"; }
};

Cat c;
Animal a = c;
std::cout << c.sound() << "\n";
std::cout << a.sound() << "\n";
```

6. **Calculation:** Given this hierarchy:

```
class A { int x; };
class B : public A { int y; };
```

```
class C : public B { int z; };
```

Assuming `int` is 4 bytes with no padding, what is the minimum `sizeof(C)`?

7. **Where is the bug?**

```
class Shape {
public:
    virtual double area() const = 0;
};

class Square : public Shape {
public:
    Square(double side) : side_(side) {}
    double area() const { return side_ * side_; }
private:
    double side_;
};
```

8. **What does this print?**

```
class Base {
public:
    Base() { std::cout << "1 "; }
    virtual ~Base() { std::cout << "4 "; }
};

class Derived : public Base {
public:
    Derived() { std::cout << "2 "; }
    ~Derived() override { std::cout << "3 "; }
};

{ Derived d; }
```

9. **Think about it:** The text recommends preferring composition over inheritance. Give an example of a situation where inheritance is clearly the right choice and another where composition would be better.

10. **Write a program** that defines an abstract `MediaPlayer` class with a pure virtual `play()` method and at least two derived classes (e.g., `MP3Player` and `StreamPlayer`). Store them in a `std::vector<std::unique_ptr<MediaPlay` and call `play()` on each.

2. Templates

In Chapter 1 you saw how virtual functions let you write code that works with a family of related types at run time. Templates solve a different problem: writing code that works with **any** type at compile time.

Suppose you need a function that returns the larger of two values. Without templates you would write one version for `int`, another for `double`, another for `std::string`, and so on — identical logic repeated for every type. Templates let you write the logic once and let the compiler generate the type-specific versions for you.

In this chapter you will learn function templates, class templates, specialization, class template argument deduction (CTAD), variadic templates, and concepts.

Function Templates

A **function template** is a blueprint for a function. The compiler generates a concrete function for each type you use it with:

```
template<typename T>
T max_of(T a, T b)
{
    return (a > b) ? a : b;
}
```

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

template<typename T>
T max_of(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    std::cout << max_of(3, 7) << "\n";                          // int
    std::cout << max_of(3.14, 2.72) << "\n";                    // double
    std::cout << max_of<std::string>("Crazy", "Beautiful") << "\n";  // std::string

    return 0;
}
```

```
7
3.14
Crazy
```

The compiler **deduces** the type `T` from the arguments. When it sees `max_of(3, 7)`, it generates `int max_of(int a, int b)`. You can also specify the type explicitly with `max_of<std::string>(...)` when deduction is ambiguous or you want a specific type.

Each generated version is called a **template instantiation**. The compiler creates only the instantiations you actually use.

Multiple Template Parameters

You can have more than one template parameter:

```
template<typename T, typename U>
void print_pair(const T& first, const U& second)
```

```
{
    std::cout << first << ", " << second << "\n";
}

print_pair("Usher", 2004);       // T = const char*, U = int
print_pair(3.14, "Yeah!");       // T = double, U = const char*
```

Non-Type Template Parameters

Template parameters do not have to be types. They can also be compile-time constants:

```
template<typename T, int N>
T sum(const T (&arr)[N])
{
    T total = 0;
    for (int i = 0; i < N; ++i) {
        total += arr[i];
    }
    return total;
}

int scores[] = {90, 85, 92, 88};
std::cout << sum(scores) << "\n";  // 355 — N is deduced as 4
```

`std::array<T, N>` uses a non-type template parameter for its size — that is why the size is part of the type.

Class Templates

A **class template** lets you define a class that works with any type. You have already used class templates from the standard library: `std::vector<T>`, `std::array<T, N>`, `std::unique_ptr<T>`.

Here is a simple stack:

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

template<typename T>
class Stack {
public:
    void push(const T& value)
    {
        data_.push_back(value);
    }

    T pop()
    {
        if (data_.empty()) {
            throw std::runtime_error("pop from empty stack");
        }
        T top = data_.back();
        data_.pop_back();
        return top;
    }
```

```
    bool empty() const { return data_.empty(); }
    int size() const { return static_cast<int>(data_.size()); }

private:
    std::vector<T> data_;
};

int main()
{
    Stack<std::string> songs;
    songs.push("Since U Been Gone");
    songs.push("Umbrella");

    while (!songs.empty()) {
        std::cout << songs.pop() << "\n";
    }

    return 0;
}
```

```
Umbrella
Since U Been Gone
```

Member Functions Outside the Class

When you define a member function outside the class template, you repeat the template header:

```
template<typename T>
class Stack {
public:
    void push(const T& value);
    T pop();
    // ...
};

template<typename T>
void Stack<T>::push(const T& value)
{
    data_.push_back(value);
}

template<typename T>
T Stack<T>::pop()
{
    if (data_.empty()) {
        throw std::runtime_error("pop from empty stack");
    }
    T top = data_.back();
    data_.pop_back();
    return top;
}
```

Wut: Template definitions (not just declarations) must be visible at the point of use. This is why template code usually lives in header files, not `.cpp` files. If you put a template definition in a `.cpp` file, the compiler cannot see it when other files try to instantiate the template, and you will get linker errors.

Template Specialization

Sometimes a template's general implementation does not work well for a particular type. **Template specialization** lets you provide a custom implementation for specific types.

Full Specialization

A **full specialization** provides an implementation for one specific type:

```
#include <cstring>
#include <iostream>

template<typename T>
T max_of(T a, T b)
{
    return (a > b) ? a : b;
}

// Full specialization for const char*
template<>
const char* max_of<const char*>(const char* a, const char* b)
{
    return (std::strcmp(a, b) > 0) ? a : b;
}

int main()
{
    std::cout << max_of(3, 7) << "\n";              // uses general template
    std::cout << max_of("Hola", "Adios") << "\n";   // uses specialization

    return 0;
}
```

```
7
Hola
```

Without the specialization, `max_of("Hola", "Adios")` would compare pointer addresses, not the string contents.

Partial Specialization

Partial specialization customizes a class template for a category of types. It only works with class templates, not function templates:

```
#include <iostream>

template<typename T>
class Wrapper {
public:
    void describe() const { std::cout << "General wrapper\n"; }
```

```
};

// Partial specialization for pointer types
template<typename T>
class Wrapper<T*> {
public:
    void describe() const { std::cout << "Pointer wrapper\n"; }
};

int main()
{
    Wrapper<int> w1;
    Wrapper<int*> w2;
    w1.describe();  // General wrapper
    w2.describe();  // Pointer wrapper

    return 0;
}
```

```
General wrapper
Pointer wrapper
```

CTAD (Class Template Argument Deduction)

Before C++17, you always had to specify template arguments when creating objects:

```
std::pair<int, std::string> p(1, "Complicated");  // verbose
std::vector<int> v = {1, 2, 3};                    // had to write <int>
```

C++17 introduced **CTAD** — the compiler can deduce the template arguments from the constructor arguments:

```
std::pair p(1, std::string("Complicated"));  // deduces pair<int, string>
std::vector v = {1, 2, 3};                   // deduces vector<int>
```

CTAD works with your own class templates too:

```
template<typename T>
class Holder {
public:
    Holder(T value) : value_(value) {}
    T get() const { return value_; }
private:
    T value_;
};

Holder h(42);          // deduces Holder<int>
Holder s("All the Small Things");  // deduces Holder<const char*>
```

Trap: CTAD deduces `const char*` for string literals, not `std::string`. If you want `Holder<std::string>`, pass a `std::string` explicitly: `Holder h(std::string("All the Small Things"))`.

Deduction Guides

You can provide **deduction guides** to control how CTAD works:

```cpp
template<typename T>
class Holder {
public:
    Holder(T value) : value_(value) {}
    T get() const { return value_; }
private:
    T value_;
};

// Deduction guide: const char* should become std::string
Holder(const char*) -> Holder<std::string>;

Holder h("Boulevard of Broken Dreams");  // now deduces Holder<std::string>
```

Variadic Templates

Variadic templates accept any number of template arguments. They use **parameter packs** — a way to represent zero or more types or values.

```cpp
#include <iostream>

template<typename... Args>
void print_all(const Args&... args)
{
    ((std::cout << args << " "), ...);
    std::cout << "\n";
}

int main()
{
    print_all(1, "Shakira", 3.14, "Drops of Jupiter");

    return 0;
}
```

```
1 Shakira 3.14 Drops of Jupiter
```

The `...` after `typename` declares a parameter pack. The `((std::cout << args << " "), ...)` is a **fold expression** (C++17) — it expands the pack by applying the comma operator between each element.

Fold Expressions

C++17 fold expressions provide a clean syntax for expanding parameter packs with an operator:

Syntax	Expansion
`(args + ...)`	`a1 + (a2 + (a3 + a4))` (right fold)
`(... + args)`	`((a1 + a2) + a3) + a4` (left fold)
`(args + ... + init)`	`a1 + (a2 + (a3 + init))` (right fold with init)
`(init + ... + args)`	`((init + a1) + a2) + a3` (left fold with init)

```cpp
template<typename... Args>
auto sum(const Args&... args)
{
    return (args + ...);
}
```

```cpp
std::cout << sum(1, 2, 3, 4) << "\n";  // 10
```

sizeof...

`sizeof...` returns the number of elements in a parameter pack:

```cpp
template<typename... Args>
void count_args(const Args&... args)
{
    std::cout << "Got " << sizeof...(args) << " arguments\n";
}

count_args(1, "two", 3.0);  // Got 3 arguments
```

Concepts (C++20)

Templates accept any type, and when a type does not support the operations used inside the template, you get an error. Before C++20, these errors were notoriously long and cryptic.

Concepts let you specify what a template type must support, giving clear errors when a type does not qualify.

Using Standard Concepts

The `<concepts>` header provides ready-made concepts:

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

template<std::integral T>
T double_it(T value)
{
    return value * 2;
}

int main()
{
    std::cout << double_it(21) << "\n";   // 42 — int is integral
    // double_it(3.14);                   // error: double is not integral

    return 0;
}
```

Some commonly used standard concepts:

Concept	Requires
`std::integral`	Integer type (int, long, char, etc.)
`std::floating_point`	Floating-point type (float, double)
`std::same_as<T, U>`	T and U are the same type
`std::convertible_to<From, To>`	From is convertible to To
`std::copyable`	Type can be copied
`std::movable`	Type can be moved

`requires` Clauses

You can write ad-hoc constraints with `requires`:

```
template<typename T>
    requires std::integral<T> || std::floating_point<T>
T absolute(T value)
{
    return (value < 0) ? -value : value;
}
```

Or use a trailing requires clause:

```
template<typename T>
T absolute(T value) requires std::integral<T> || std::floating_point<T>
{
    return (value < 0) ? -value : value;
}
```

Writing Custom Concepts

You can define your own concepts:

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

template<typename T>
concept Printable = requires(T t) {
    { std::cout << t } -> std::same_as<std::ostream&>;
};

template<Printable T>
void display(const T& value)
{
    std::cout << value << "\n";
}

int main()
{
    display(42);
    display("Viva la Vida");
    display(3.14);

    return 0;
}
```

```
42
Viva la Vida
3.14
```

The `requires` expression lists operations the type must support. The `->` syntax constrains the return type of the expression.

`requires` Expressions

A `requires` expression can test multiple things:

```cpp
template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::same_as<T>;     // can add two T values
    { a += b };                         // supports +=
};
```

You can also use `requires` in `constexpr if` to branch at compile time:

```cpp
template<typename T>
void process(const T& value)
{
    if constexpr (std::integral<T>) {
        std::cout << "Integer: " << value << "\n";
    } else if constexpr (std::floating_point<T>) {
        std::cout << "Float: " << value << "\n";
    } else {
        std::cout << "Other: " << value << "\n";
    }
}
```

Tip: Concepts make templates easier to use and debug. When a type does not satisfy a concept, the compiler tells you exactly which requirement failed instead of producing pages of nested template errors.

Try It: Template Playground

Here is a program that exercises several template features. Type it in, compile with `g++ -std=c++23`, and experiment:

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

// Function template with concept
template<typename T>
    requires std::integral<T> || std::floating_point<T>
T clamp(T value, T lo, T hi)
{
    if (value < lo) return lo;
    if (value > hi) return hi;
    return value;
}

// Class template
template<typename T>
class Playlist {
public:
    void add(const T& item) { items_.push_back(item); }

    void print() const
    {
        for (const auto& item : items_) {
            std::cout << "  " << item << "\n";
        }
```

```
    }

    auto size() const { return items_.size(); }

private:
    std::vector<T> items_;
};

// Variadic template
template<typename... Args>
void log(const Args&... args)
{
    ((std::cout << args << " "), ...);
    std::cout << "\n";
}

int main()
{
    // Concepts
    std::cout << "Clamped: " << clamp(150, 0, 100) << "\n";
    std::cout << "Clamped: " << clamp(3.14, 0.0, 1.0) << "\n";

    // CTAD
    Playlist<std::string> songs;
    songs.add("Crazy in Love");
    songs.add("Maps");
    songs.add("Seven Nation Army");

    std::cout << "\nPlaylist (" << songs.size() << " songs):\n";
    songs.print();

    // Variadic template
    log("Track", 1, "playing at", 44100, "Hz");

    return 0;
}
```

```
Clamped: 100
Clamped: 1
Playlist (3 songs):
  Crazy in Love
  Maps
  Seven Nation Army
Track 1 playing at 44100 Hz
```

Try adding a `Playlist<int>` for track numbers. Write a concept called `HasSize` that requires a type to have a `.size()` method, and write a function template constrained by it.

Key Points

- **Function templates** let you write a function once and use it with any type. The compiler generates type-specific versions (instantiations) as needed.
- **Class templates** work the same way for classes. `std::vector`, `std::array`, and `std::unique_ptr` are all class templates.
- The compiler **deduces** template arguments from function arguments. You can also specify them

explicitly with `f<int>(...)`.
- **Non-type template parameters** are compile-time constants like `int N` in `std::array<T, N>`.
- Template definitions must be in headers because the compiler needs to see them at every instantiation point.
- **Full specialization** provides a custom implementation for one specific type. **Partial specialization** (class templates only) customizes for a category of types.
- **CTAD** (C++17) lets the compiler deduce class template arguments from constructor arguments. Deduction guides can customize this behavior.
- **Variadic templates** accept any number of arguments using parameter packs (`typename... Args`).
- **Fold expressions** (C++17) expand parameter packs concisely: `(args + ...)`.
- **Concepts** (C++20) constrain what types a template accepts, producing clear error messages.
- Use `requires` clauses for ad-hoc constraints or define reusable named concepts.

Exercises

1. **Think about it:** Templates generate code at compile time, while virtual functions dispatch at run time. What are the trade-offs between these two approaches to polymorphism?

2. **What does this print?**

```cpp
template<typename T>
T add(T a, T b) { return a + b; }

std::cout << add(3, 4) << "\n";
std::cout << add(std::string("Hola"), std::string(" mundo")) << "\n";
```

3. **Where is the bug?**

```cpp
template<typename T>
T max_of(T a, T b)
{
    return (a > b) ? a : b;
}

std::cout << max_of(3, 4.5) << "\n";
```

4. **Calculation:** How many template instantiations are generated by this code?

```cpp
template<typename T>
T identity(T x) { return x; }

identity(1);
identity(2);
identity(3.0);
identity(std::string("test"));
identity(42);
```

5. **What does this print?**

```cpp
template<typename... Args>
auto sum(Args... args)
{
    return (args + ...);
}

std::cout << sum(1, 2, 3, 4, 5) << "\n";
```

6. **Think about it:** Why must template definitions live in header files? What would happen if you put a template function's definition in a `.cpp` file and tried to use it from another `.cpp` file?

7. **Where is the bug?**

```cpp
template<typename T>
class Holder {
public:
    Holder(T val) : value_(val) {}
    T get() const { return value_; }
private:
    T value_;
};

Holder h = "Lose Yourself";
std::cout << h.get() << "\n";
```

   What type does CTAD deduce for `T`? Is this likely what the programmer intended?

8. **What does this print?**

```cpp
template<typename T>
void describe(T) { std::cout << "general\n"; }

template<>
void describe<int>(int) { std::cout << "int\n"; }

describe(42);
describe(3.14);
describe("hello");
```

9. **Calculation:** What does `sizeof...(args)` return for this call?

```cpp
template<typename... Args>
int count(Args... args) { return sizeof...(args); }

std::cout << count(1, "two", 3.0, '4', true) << "\n";
```

10. **Write a program** that defines a class template `Pair<T, U>` with two members `first` and `second`, a constructor, and a `print()` method. Test it with `Pair<std::string, int>` storing song names and release years. Add a deduction guide so that `Pair("I Gotta Feeling", 2009)` deduces `Pair<std::string, int>`.

3. The Standard Template Library

In *Gorgo Starting C++* you learned `std::vector` and `std::array` — the two workhorses of sequential storage. But not every problem is a list. Sometimes you need to look up a value by key, enforce uniqueness, or process items in priority order. The **Standard Template Library** (STL) provides a rich set of containers, each optimized for different access patterns. In Chapter 2 you learned that these containers are class templates — they work with any type. In this chapter you will learn the associative containers, unordered containers, and container adaptors, and how to choose the right one for the job.

Associative Containers

Associative containers store elements in **sorted order** and provide efficient lookup, insertion, and deletion — typically O(log n). They are implemented as balanced binary search trees (usually red-black trees).

`std::map`

`std::map<Key, Value>` stores key-value pairs sorted by key. Each key is unique:

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

int main()
{
    std::map<std::string, int> album_years;
    album_years["Elephant"] = 2003;
    album_years["Hot Fuss"] = 2004;
    album_years["Funeral"] = 2004;

    for (const auto& [album, year] : album_years) {
        std::cout << album << ": " << year << "\n";
    }

    return 0;
}
```

```
Elephant: 2003
Funeral: 2004
Hot Fuss: 2004
```

The output is sorted alphabetically by key. The `const auto& [album, year]` syntax is a **structured binding** that unpacks each `std::pair<const std::string, int>` in the map.

Key operations:

```
// Insert or update
album_years["X&Y"] = 2005;
album_years.insert({"Absolution", 2003});

// Lookup
if (album_years.count("Funeral") > 0) {
    std::cout << "Found\n";
}

// C++20: contains()
//   bool contains(const Key& key) const;
if (album_years.contains("Funeral")) {
```

```
    std::cout << "Found\n";
}

// Safe access with find()
//   iterator find(const Key& key);
auto it = album_years.find("Elephant");
if (it != album_years.end()) {
    std::cout << it->second << "\n";  // 2003
}
```

Trap: Using `operator[]` on a key that does not exist will **insert** a default-constructed value. If you just want to check whether a key exists, use `find()` or `contains()` instead.

std::set

`std::set<T>` stores unique values in sorted order. It is like a `map` with only keys:

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

int main()
{
    std::set<std::string> genres;
    genres.insert("Rock");
    genres.insert("Pop");
    genres.insert("Rock");  // duplicate — ignored
    genres.insert("Indie");

    for (const auto& g : genres) {
        std::cout << g << "\n";
    }

    std::cout << "Size: " << genres.size() << "\n";

    return 0;
}
```

```
Indie
Pop
Rock
Size: 3
```

`insert()` returns a `std::pair<iterator, bool>` — the `bool` tells you whether the insertion actually happened:

```
auto [it, inserted] = genres.insert("Pop");
if (!inserted) {
    std::cout << "Pop was already in the set\n";
}
```

std::multimap and std::multiset

`std::multimap` and `std::multiset` are like `map` and `set` but allow **duplicate keys**:

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

int main()
{
    std::multimap<std::string, std::string> songs_by_genre;
    songs_by_genre.insert({"Rock", "Seven Nation Army"});
    songs_by_genre.insert({"Rock", "Mr. Brightside"});
    songs_by_genre.insert({"Pop", "Crazy in Love"});
    songs_by_genre.insert({"Pop", "Toxic"});

    // equal_range returns a pair of iterators: [first match, past last match)
    //   pair<iterator, iterator> equal_range(const Key& key);
    auto [begin, end] = songs_by_genre.equal_range("Rock");
    std::cout << "Rock songs:\n";
    for (auto it = begin; it != end; ++it) {
        std::cout << "  " << it->second << "\n";
    }

    return 0;
}
```

```
Rock songs:
  Seven Nation Army
  Mr. Brightside
```

Tip: `multimap` does not have `operator[]` because a key can map to multiple values. Use `find()`, `equal_range()`, or a range-based loop to access elements.

Unordered Containers

Unordered containers use **hash tables** instead of trees. They provide O(1) average-case lookup, insertion, and deletion — faster than the O(log n) of ordered containers. The trade-off is that elements are not stored in any particular order.

`std::unordered_map`

`std::unordered_map<Key, Value>` is the hash-table equivalent of `std::map`:

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

int main()
{
    std::unordered_map<std::string, int> play_counts;
    play_counts["Rehab"] = 42;
    play_counts["Poker Face"] = 87;
    play_counts["Use Somebody"] = 55;

    play_counts["Rehab"] += 1;

    for (const auto& [song, count] : play_counts) {
```

```
        std::cout << song << ": " << count << " plays\n";
    }

    return 0;
}
```

The output order is not alphabetical — it depends on the hash function and internal bucket layout.

The API is nearly identical to `std::map`: `operator[]`, `find()`, `contains()`, `insert()`, `erase()` all work the same way.

`std::unordered_set`

`std::unordered_set<T>` is the hash-table equivalent of `std::set`:

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

int main()
{
    std::unordered_set<std::string> tags = {"chill", "summer", "dance", "chill"};

    std::cout << "Tags (" << tags.size() << "):\n";
    for (const auto& t : tags) {
        std::cout << "  " << t << "\n";
    }

    return 0;
}
```

Duplicates are ignored, just like `std::set`, but the elements are not sorted.

Wut: To use a custom type as a key in an unordered container, you must provide a hash function and an equality operator. Standard types like `std::string`, `int`, and `double` already have hash functions.

Ordered vs. Unordered

Feature	`map/set`	`unordered_map/unordered_set`
Order	Sorted by key	No guaranteed order
Lookup	O(log n)	O(1) average, O(n) worst
Insertion	O(log n)	O(1) average, O(n) worst
Requires	`operator<` or comparator	Hash function + `operator==`
Memory	Tree nodes (pointer overhead)	Hash buckets (may waste space)

Use ordered containers when you need sorted iteration or range queries. Use unordered containers when you only need fast lookup by key.

Container Adaptors

Container adaptors wrap an underlying container and restrict its interface to provide a specific behavior. They are not full containers — you cannot iterate through them.

std::stack

`std::stack<T>` provides LIFO (last in, first out) access:

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

int main()
{
    std::stack<std::string> history;
    history.push("Chasing Cars");
    history.push("How to Save a Life");
    history.push("Fix You");

    while (!history.empty()) {
        std::cout << history.top() << "\n";
        history.pop();
    }

    return 0;
}
```

```
Fix You
How to Save a Life
Chasing Cars
```

Key methods:

```cpp
void push(const T& value);   // add to top
void pop();                   // remove from top (does not return it)
T& top();                     // access top element
bool empty() const;
size_t size() const;
```

Trap: `pop()` does not return the removed element. You must call `top()` first to get the value, then `pop()` to remove it.

std::queue

`std::queue<T>` provides FIFO (first in, first out) access:

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

int main()
{
    std::queue<std::string> playlist;
    playlist.push("Crazy");
    playlist.push("Gold Digger");
    playlist.push("Single Ladies");

    while (!playlist.empty()) {
        std::cout << "Now playing: " << playlist.front() << "\n";
        playlist.pop();
```

```
    }

    return 0;
}
```

```
Now playing: Crazy
Now playing: Gold Digger
Now playing: Single Ladies
```

Key methods:

```
void push(const T& value);   // add to back
void pop();                   // remove from front
T& front();                   // access front element
T& back();                    // access back element
bool empty() const;
size_t size() const;
```

std::priority_queue

`std::priority_queue<T>` is a queue where the highest-priority element is always at the top. By default, it is a max-heap — the largest element has the highest priority:

```
#include <iostream>
#include <queue>

int main()
{
    std::priority_queue<int> pq;
    pq.push(30);
    pq.push(10);
    pq.push(50);
    pq.push(20);

    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << "\n";

    return 0;
}
```

```
50 30 20 10
```

To get a min-heap (smallest first), use `std::greater<T>` as the comparator:

```
std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;
```

Choosing the Right Container

Here is a quick decision guide:

Need	Container
Sequential access, dynamic size	`std::vector`
Fixed-size array	`std::array`
Fast lookup by key (sorted)	`std::map`

Need	Container
Fast lookup by key (unsorted)	`std::unordered_map`
Unique values (sorted)	`std::set`
Unique values (unsorted)	`std::unordered_set`
Duplicate keys allowed (sorted)	`std::multimap` / `std::multiset`
LIFO (stack behavior)	`std::stack`
FIFO (queue behavior)	`std::queue`
Priority-based access	`std::priority_queue`

When in doubt, start with `std::vector`. It has the best cache performance and works well for most tasks. Switch to a different container only when you have a specific need that `std::vector` does not serve well.

Tip: `std::vector` is almost always faster than you expect. Even linear search through a small vector often beats hash table or tree lookup because of CPU cache effects. Profile before switching containers.

Try It: Container Sampler

Here is a program that exercises multiple container types. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <unordered_map>

int main()
{
    // map: album ratings
    std::map<std::string, double> ratings;
    ratings["Hot Fuss"] = 4.5;
    ratings["Elephant"] = 4.8;
    ratings["Funeral"] = 4.7;
    ratings["X&Y"] = 3.9;

    std::cout << "Albums (sorted):\n";
    for (const auto& [album, rating] : ratings) {
        std::cout << "  " << album << ": " << rating << "\n";
    }

    // set: unique genres
    std::set<std::string> genres = {"Rock", "Indie", "Pop", "Rock", "Indie"};
    std::cout << "\nGenres: ";
    for (const auto& g : genres) {
        std::cout << g << " ";
    }
    std::cout << "\n";

    // unordered_map: fast lookup
    std::unordered_map<std::string, int> track_num;
```

```
    track_num["Intro"] = 1;
    track_num["Verse"] = 2;
    track_num["Chorus"] = 3;

    if (track_num.contains("Chorus")) {
        std::cout << "\nChorus is track " << track_num["Chorus"] << "\n";
    }

    // stack: undo history
    std::stack<std::string> undo;
    undo.push("add track");
    undo.push("change volume");
    undo.push("delete track");

    std::cout << "\nUndo stack:\n";
    while (!undo.empty()) {
        std::cout << "  undo: " << undo.top() << "\n";
        undo.pop();
    }

    // priority_queue: highest rated first
    std::priority_queue<std::pair<double, std::string>> top_albums;
    for (const auto& [album, rating] : ratings) {
        top_albums.push({rating, album});
    }

    std::cout << "\nTop albums:\n";
    while (!top_albums.empty()) {
        auto [rating, album] = top_albums.top();
        std::cout << "  " << album << " (" << rating << ")\n";
        top_albums.pop();
    }

    return 0;
}
```

Try adding a `std::multimap` that maps genres to multiple songs. Experiment with `std::unordered_set` and see how the iteration order differs from `std::set`.

Key Points

- **Associative containers** (`std::map`, `std::set`, `std::multimap`, `std::multiset`) store elements in sorted order using balanced trees, with O(log n) operations.
- `std::map` stores unique key-value pairs; `std::set` stores unique values only.
- `std::multimap` and `std::multiset` allow duplicate keys.
- **Unordered containers** (`std::unordered_map`, `std::unordered_set`) use hash tables for O(1) average-case operations, but elements have no guaranteed order.
- Use `contains()` (C++20) or `find()` to check for existence without inserting. `operator[]` on a `map` inserts a default value if the key is missing.
- **Container adaptors** (`std::stack`, `std::queue`, `std::priority_queue`) wrap other containers to provide restricted interfaces.
- `stack` is LIFO, `queue` is FIFO, `priority_queue` is a max-heap by default.
- `pop()` on adaptors does not return the removed element — call `top()` or `front()` first.
- When choosing a container, start with `std::vector` and switch only when you have a specific need.

- Use structured bindings (`auto [key, value]`) to iterate over maps concisely.

Exercises

1. **Think about it:** Why does `std::map::operator[]` insert a default value when the key is missing, instead of throwing an exception? What would the implications be if it threw instead?
2. **What does this print?**

```cpp
std::map<std::string, int> m;
m["b"] = 2;
m["a"] = 1;
m["c"] = 3;

for (const auto& [k, v] : m) {
    std::cout << k << v;
}
std::cout << "\n";
```

3. **Where is the bug?**

```cpp
std::map<std::string, int> scores;
scores["Alice"] = 95;
scores["Bob"] = 87;

int charlie_score = scores["Charlie"];
std::cout << "Charlie: " << charlie_score << "\n";
```

4. **What does this print?**

```cpp
std::set<int> s = {5, 3, 1, 4, 1, 5, 3};
std::cout << s.size() << "\n";
```

5. **Calculation:** A `std::map<std::string, int>` has 1,000,000 entries. Approximately how many comparisons does a lookup require? (Hint: O(log n) with base 2.)
6. **Think about it:** When would you choose `std::map` over `std::unordered_map`? Give two concrete scenarios.
7. **What does this print?**

```cpp
std::stack<int> s;
s.push(10);
s.push(20);
s.push(30);
s.pop();
std::cout << s.top() << "\n";
s.pop();
std::cout << s.top() << "\n";
```

8. **Where is the bug?**

```cpp
std::priority_queue<int> pq;
pq.push(5);
pq.push(15);
pq.push(10);

int smallest = pq.top();
std::cout << "Smallest: " << smallest << "\n";
```

9. **What does this print?**

```
std::multiset<int> ms = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
std::cout << ms.count(5) << "\n";
```

10. **Write a program** that reads words from the user (one per line, until they type "done") and uses a `std::map<std::string, int>` to count how many times each word was entered. Print the word counts in alphabetical order.

4. Ranges, Algorithms, and Lambdas

In *Gorgo Starting C++* you learned to store data in `std::vector` and `std::array` and to iterate through them with range-based for loops. But when you need to sort, search, or filter that data, you end up writing loops by hand. Hand-written loops are repetitive, easy to get wrong (off-by-one errors, forgotten edge cases), and they bury your intent behind boilerplate. Every time you write a loop to find the maximum element, you are re-implementing logic that has already been written and tested. The standard library provides **algorithms** — pre-built, well-tested functions for the operations you need most. Combined with **lambdas** (inline functions you pass as arguments) and C++20 **ranges**, they let you say *what* you want done instead of spelling out *how* to do it. In this chapter you will learn the most common algorithms, how to write lambdas to customize their behavior, and how ranges and views make composing operations cleaner.

The <algorithm> Header

The `<algorithm>` header provides dozens of functions that operate on containers. Most of them take a pair of iterators (remember `.begin()` and `.end()` from *Gorgo Starting C++*) to define the range of elements to work on.

Let's start with the most commonly used algorithms.

Sorting

`std::sort` arranges elements in ascending order:

```
void sort(Iterator first, Iterator last);
void sort(Iterator first, Iterator last, Compare comp);
```

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> scores = {88, 42, 95, 67, 73};

    std::sort(scores.begin(), scores.end());

    for (const auto& s : scores) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
42 67 73 88 95
```

It works with strings too — they sort alphabetically:

```
std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};
std::sort(songs.begin(), songs.end());
// songs is now {"Hey Ya!", "Hips Don't Lie", "Mr. Brightside"}
```

Finding Elements

`std::find` searches for a value and returns an iterator to the first match, or `.end()` if not found:

```
Iterator find(Iterator first, Iterator last, const T& value);
```

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

int main()
{
    std::vector<std::string> playlist = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};

    auto it = std::find(playlist.begin(), playlist.end(), "Mr. Brightside");
    if (it != playlist.end()) {
        std::cout << "Found: " << *it << "\n";
    } else {
        std::cout << "Not found\n";
    }

    return 0;
}
```

```
Found: Mr. Brightside
```

Tip: Always check the result of `std::find` against `.end()` before dereferencing the iterator. Dereferencing `.end()` is undefined behavior.

Counting

`std::count` counts how many times a value appears:

```cpp
int count(Iterator first, Iterator last, const T& value);
```

```cpp
std::vector<int> votes = {1, 2, 1, 3, 1, 2, 1};
int ones = std::count(votes.begin(), votes.end(), 1);
std::cout << "Number of 1s: " << ones << "\n";  // 4
```

`for_each`

`std::for_each` applies a function to every element in a range. It is similar to a range-based for loop, but is useful when you want to pass a function directly:

```cpp
Function for_each(Iterator first, Iterator last, Function f);
```

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

void print_song(const std::string& song)
{
    std::cout << "  " << song << "\n";
}

int main()
{
    std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside"};
```

```
    std::cout << "Playlist:\n";
    std::for_each(songs.begin(), songs.end(), print_song);

    return 0;
}
```

```
Playlist:
  Hey Ya!
  Mr. Brightside
```

That third argument is a function — `std::for_each` calls it once for every element. But writing a separate named function for something this simple is tedious. This is where lambdas come in.

Lambdas

A **lambda** is an anonymous function that you define right where you need it. Instead of writing a separate function like `print_song` above, you can write:

```
std::for_each(songs.begin(), songs.end(), [](const std::string& song) {
    std::cout << "  " << song << "\n";
});
```

The lambda syntax is:

```
[captures](parameters) { body }
```

- **`[captures]`** — what variables from the surrounding scope the lambda can access
- **`(parameters)`** — just like function parameters
- **`{ body }`** — the code to execute

Here is a simple example:

```
auto greet = [](const std::string& name) {
    std::cout << "Hola, " << name << "!\n";
};

greet("Shakira");  // Hola, Shakira!
greet("OutKast");  // Hola, OutKast!
```

You can store a lambda in a variable using `auto` and call it like a regular function.

Captures

Lambdas can access variables from the surrounding scope through **captures**:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int threshold = 5;

    // std::count_if: like std::count, but with a predicate
    //   int count_if(Iterator first, Iterator last, Predicate pred);
    auto count = std::count_if(nums.begin(), nums.end(),
        [threshold](int n) { return n > threshold; });
```

```
    std::cout << count << " numbers above " << threshold << "\n";

    return 0;
}
```

```
5 numbers above 5
```

The `[threshold]` capture makes a copy of `threshold` available inside the lambda.

Here are the capture options:

Syntax	Meaning
`[]`	Capture nothing
`[x]`	Capture x by value (copy)
`[&x]`	Capture x by reference
`[=]`	Capture everything by value
`[&]`	Capture everything by reference
`[=, &x]`	Capture everything by value, but x by reference

```
int total = 0;
std::vector<int> prices = {10, 20, 30};

std::for_each(prices.begin(), prices.end(), [&total](int p) {
    total += p;
});

std::cout << "Total: " << total << "\n";  // Total: 60
```

Here `[&total]` captures `total` by reference, so the lambda can modify it.

Tip: Prefer specific captures like `[x]` or `[&x]` over blanket captures like `[=]` or `[&]`. Explicit captures make it clear what the lambda depends on and help prevent accidental captures.

Transform

`std::transform` applies a function to each element and stores the result. It can write the results back into the same container or into a different one:

```
OutputIterator transform(Iterator first, Iterator last, OutputIterator result, Function f);
```

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5};
    std::vector<int> doubled(nums.size());

    std::transform(nums.begin(), nums.end(), doubled.begin(),
        [](int n) { return n * 2; });

    for (const auto& d : doubled) {
        std::cout << d << " ";
```

```
    }
    std::cout << "\n";

    return 0;
}
```

```
2 4 6 8 10
```

Trap: When writing results to a different container, that container must already have enough space. In the example above, `doubled` is created with `nums.size()` elements. If you use an empty vector, you will write past its end — undefined behavior.

Accumulate

`std::accumulate` reduces a range to a single value by applying an operation. It lives in `<numeric>`, not `<algorithm>`:

```
T accumulate(Iterator first, Iterator last, T init);
T accumulate(Iterator first, Iterator last, T init, BinaryOp op);
```

```
#include <iostream>
#include <numeric>
#include <vector>

int main()
{
    std::vector<int> scores = {90, 85, 92, 88};

    int sum = std::accumulate(scores.begin(), scores.end(), 0);
    std::cout << "Sum: " << sum << "\n";
    std::cout << "Average: " << sum / static_cast<int>(scores.size()) << "\n";

    return 0;
}
```

```
Sum: 355
Average: 88
```

The third argument (`0`) is the initial value. You can also pass a custom operation as a fourth argument:

```
int product = std::accumulate(scores.begin(), scores.end(), 1,
    [](int a, int b) { return a * b; });
```

Min and Max

`std::min_element` and `std::max_element` return iterators to the smallest and largest elements:

```
Iterator min_element(Iterator first, Iterator last);
Iterator max_element(Iterator first, Iterator last);
```

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> temps = {72, 68, 85, 61, 79};
```

```
    auto coldest = std::min_element(temps.begin(), temps.end());
    auto hottest = std::max_element(temps.begin(), temps.end());

    std::cout << "Coldest: " << *coldest << "\n";
    std::cout << "Hottest: " << *hottest << "\n";

    return 0;
}
```

```
Coldest: 61
Hottest: 85
```

Ranges (C++20)

The algorithms above all take pairs of iterators: `container.begin(), container.end()`. C++20 introduced the `std::ranges` namespace, which lets you pass the container directly:

```
#include <algorithm>
#include <iostream>
#include <ranges>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};

    // void ranges::sort(Range& r);
    std::ranges::sort(songs);

    for (const auto& s : songs) {
        std::cout << s << "\n";
    }

    return 0;
}
```

```
Hey Ya!
Hips Don't Lie
Mr. Brightside
```

Compare `std::sort(songs.begin(), songs.end())` with `std::ranges::sort(songs)`. The ranges version is simpler and less error-prone — you cannot accidentally pass mismatched iterators.

`std::ranges::find` works the same way:

```
// Iterator ranges::find(Range& r, const T& value);
auto it = std::ranges::find(songs, "Mr. Brightside");
if (it != songs.end()) {
    std::cout << "Encontrado: " << *it << "\n";
}
```

Tip: When your compiler supports C++20, prefer `std::ranges::` versions of algorithms. They are simpler, safer, and often provide better error messages when something goes wrong.

Views

Views are one of the most powerful features added in C++20. A **view** is a lightweight wrapper that transforms or filters elements **lazily** — it does not create a new container or copy data. Instead, it computes each element on demand as you iterate.

Think of it like looking through a filter: the original data does not change, you just see it differently.

Pipe Syntax

Views use the pipe operator `|` to chain operations together, much like Unix pipes:

```cpp
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // auto views::filter(Predicate pred);   // keeps elements where pred is true
    for (int n : nums | std::views::filter([](int n) { return n % 2 == 0; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
2 4 6 8 10
```

The expression `nums | std::views::filter(...)` creates a view that only yields even numbers. No new vector is created — the filter is applied on the fly as you iterate.

Common Views

Here are the views you will use most often:

```cpp
auto views::filter(Predicate pred);      // keeps matching elements
auto views::transform(Function f);       // applies f to each element
auto views::take(int n);                 // first n elements
auto views::drop(int n);                 // skip first n elements
auto views::reverse;                     // iterate in reverse
```

```cpp
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // filter: keep only elements matching a condition
    std::cout << "Even: ";
    for (int n : nums | std::views::filter([](int n) { return n % 2 == 0; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";
```

```
    // transform: apply a function to each element
    std::cout << "Tripled: ";
    for (int n : nums | std::views::transform([](int n) { return n * 3; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // take: keep only the first N elements
    std::cout << "First 3: ";
    for (int n : nums | std::views::take(3)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // drop: skip the first N elements
    std::cout << "Skip 7: ";
    for (int n : nums | std::views::drop(7)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // reverse: iterate in reverse order
    std::cout << "Reversed: ";
    for (int n : nums | std::views::reverse) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
Even: 2 4 6 8 10
Tripled: 3 6 9 12 15 18 21 24 27 30
First 3: 1 2 3
Skip 7: 8 9 10
Reversed: 10 9 8 7 6 5 4 3 2 1
```

Chaining Views

The real power of views shows when you chain them together:

```
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Get the first 3 even numbers, doubled
    std::cout << "Result: ";
    for (int n : nums
            | std::views::filter([](int n) { return n % 2 == 0; })
            | std::views::transform([](int n) { return n * 2; })
```

```
            | std::views::take(3)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
Result: 4 8 12
```

This reads almost like English: take `nums`, filter the even ones, double them, and take the first 3. Each `|` passes the result of the left side as input to the right side.

Wut: Views are lazy. In the chained example above, the `transform` and `filter` are not applied to all 10 elements. Once `take(3)` has yielded 3 elements, the pipeline stops — elements 8, 10, and beyond are never even looked at. This makes views very efficient when you only need a subset of results.

Views with Strings

Views work well with any container, including vectors of strings:

```
#include <iostream>
#include <ranges>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> songs = {
        "Hey Ya!", "Mr. Brightside", "Hips Don't Lie"
    };

    std::cout << "Long titles:\n";
    for (const auto& s : songs
            | std::views::filter([](const std::string& s) {
                return s.size() > 10;
            })) {
        std::cout << "  " << s << "\n";
    }

    return 0;
}
```

```
Long titles:
  Mr. Brightside
  Hips Don't Lie
```

Try It: Algorithm Starter

Here is a program that exercises several algorithms and views. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <algorithm>
#include <iostream>
#include <numeric>
```

```cpp
#include <ranges>
#include <string>
#include <vector>

int main()
{
    std::vector<int> scores = {72, 95, 88, 61, 84, 90, 77};

    // Sort
    std::ranges::sort(scores);
    std::cout << "Sorted: ";
    for (int s : scores) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    // Find
    // std::distance returns the number of steps between two iterators:
    //   int distance(Iterator first, Iterator last);
    auto it = std::ranges::find(scores, 88);
    if (it != scores.end()) {
        std::cout << "Found 88 at position "
                  << std::distance(scores.begin(), it) << "\n";
    }

    // Accumulate (still needs begin/end)
    int sum = std::accumulate(scores.begin(), scores.end(), 0);
    std::cout << "Sum: " << sum << ", Average: "
              << sum / static_cast<int>(scores.size()) << "\n";

    // Min and max — ranges::minmax returns the smallest and largest elements:
    //   auto ranges::minmax(Range& r);   // returns {min, max}
    auto [lo, hi] = std::ranges::minmax(scores);
    std::cout << "Min: " << lo << ", Max: " << hi << "\n";

    // Lambda with count_if
    //   int ranges::count_if(Range& r, Predicate pred);
    int above_80 = std::ranges::count_if(scores,
        [](int s) { return s > 80; });
    std::cout << "Scores above 80: " << above_80 << "\n";

    // Views pipeline
    std::cout << "Top 3 scores doubled: ";
    for (int s : scores
            | std::views::reverse
            | std::views::take(3)
            | std::views::transform([](int s) { return s * 2; })) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Key Points

- The `<algorithm>` header provides reusable functions like `std::sort`, `std::find`, `std::count`, `std::for_each`, and `std::transform`.
- `std::accumulate` (from `<numeric>`) reduces a range to a single value.
- Lambdas are anonymous functions written as `[captures](params) { body }`. They are the primary way to customize algorithm behavior.
- Captures control what a lambda can access from its surrounding scope: `[x]` by value, `[&x]` by reference, `[=]` all by value, `[&]` all by reference.
- C++20 `std::ranges::` algorithms accept containers directly instead of iterator pairs.
- Views (`std::views::filter`, `transform`, `take`, `drop`, `reverse`) apply transformations lazily without copying data.
- Views chain together with the `|` pipe operator, creating readable data pipelines.
- Lazy evaluation means views only compute elements as they are consumed, making them efficient for large data sets.

Exercises

1. **Think about it:** Why do you think the standard library provides both `std::sort(v.begin(), v.end())` and `std::ranges::sort(v)`? If the ranges version is simpler, why keep the iterator version?

2. **What does this print?**

```cpp
std::vector<int> v = {5, 3, 8, 1, 9, 2};
std::sort(v.begin(), v.end());
auto it = std::find(v.begin(), v.end(), 8);
std::cout << *it << " " << *(it - 1) << "\n";
```

3. **What does this print?**

```cpp
std::vector<int> v = {1, 2, 3, 4, 5};
auto result = std::count_if(v.begin(), v.end(),
    [](int n) { return n % 2 != 0; });
std::cout << result << "\n";
```

4. **Where is the bug?**

```cpp
std::vector<int> nums = {10, 20, 30};
std::vector<int> doubled;

std::transform(nums.begin(), nums.end(), doubled.begin(),
    [](int n) { return n * 2; });
```

5. **Calculation:** Given this code:

```cpp
std::vector<int> v = {4, 7, 2, 9, 1};
int x = std::accumulate(v.begin(), v.end(), 10);
```

   What is the value of x?

6. **What does this print?**

```cpp
int factor = 3;
auto multiply = [factor](int n) { return n * factor; };
std::cout << multiply(5) << " " << multiply(10) << "\n";
```

7. **Where is the bug?**

```cpp
std::vector<int> nums = {1, 2, 3, 4, 5};
int total = 0;
```

```cpp
std::for_each(nums.begin(), nums.end(), [total](int n) {
    total += n;
});

std::cout << "Total: " << total << "\n";
```

8. **Think about it:** Views are "lazy" — they do not process elements until you iterate. Why is this an advantage? Can you think of a situation where processing all elements upfront would be better?

9. **What does this print?** (Assume C++20)

```cpp
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8};
for (int n : v
        | std::views::filter([](int n) { return n > 3; })
        | std::views::take(3)) {
    std::cout << n << " ";
}
std::cout << "\n";
```

10. **Write a program** that stores a list of test scores in a `std::vector<int>`, then uses algorithms and/or views to:

    - Sort the scores
    - Print only scores above 70
    - Print the average score
    - Print the highest and lowest scores

5. Enums, constexpr, and Compile-Time Programming

Catching mistakes at compile time is always better than catching them at run time. A program that fails to compile is annoying but harmless. A program that compiles and then crashes in production is a disaster. C++ gives you several tools to move work from run time to compile time: **scoped enumerations** give type-safe named constants, **`constexpr`** and **`consteval`** let you compute values during compilation, **`static_assert`** checks conditions before the program ever runs, and **type aliases** make complex types readable. In this chapter you will learn how to use these features to write code that is safer, faster, and clearer.

Scoped Enumerations: `enum class`

You may have seen traditional C-style enums in older code:

```
enum Color { Red, Green, Blue };
enum TrafficLight { Red, Yellow, Green };  // error: Red and Green already defined!
```

The names leak into the surrounding scope and collide. They also implicitly convert to `int`, which can cause subtle bugs.

C++11 introduced **scoped enumerations** (`enum class`) to fix both problems:

```
#include <iostream>

enum class Color { Red, Green, Blue };
enum class TrafficLight { Red, Yellow, Green };

int main()
{
    Color c = Color::Blue;
    TrafficLight t = TrafficLight::Red;

    // int x = c;           // error: no implicit conversion to int
    int x = static_cast<int>(c);  // OK: explicit conversion gives 2

    if (t == TrafficLight::Red) {
        std::cout << "Stop!\n";
    }

    return 0;
}
```

```
Stop!
```

Each enumerator is scoped to its enum, so `Color::Red` and `TrafficLight::Red` do not collide. There is no implicit conversion to `int` — you must use `static_cast` if you need the numeric value.

Underlying Type

By default, the underlying type of an `enum class` is `int`. You can change it:

```
enum class Status : uint8_t { OK = 0, Error = 1, Pending = 2 };
```

This is useful when memory matters (embedded systems, network protocols) or when you need to match an external format.

`using enum` (C++20)

If you get tired of writing the enum name repeatedly, C++20 lets you bring enumerators into scope:

```cpp
#include <iostream>

enum class Direction { North, South, East, West };

void navigate(Direction d)
{
    using enum Direction;  // bring all enumerators into scope

    switch (d) {
    case North: std::cout << "Going north\n"; break;
    case South: std::cout << "Going south\n"; break;
    case East:  std::cout << "Going east\n"; break;
    case West:  std::cout << "Going west\n"; break;
    }
}

int main()
{
    navigate(Direction::North);
    return 0;
}
```

```
Going north
```

Tip: Use `using enum` only in limited scopes (like inside a function or switch statement) to avoid polluting the outer namespace — that would defeat the purpose of scoped enums.

`constexpr`

A `constexpr` variable or function can be evaluated at **compile time**. The compiler computes the result and bakes it into the binary, so there is no run-time cost:

`constexpr` Variables

```cpp
constexpr int max_tracks = 100;
constexpr double pi = 3.14159265358979;
```

A `constexpr` variable must be initialized with a value the compiler can compute. It is implicitly `const`.

`constexpr` Functions

A `constexpr` function **can** be evaluated at compile time if all its arguments are compile-time constants. If called with run-time values, it runs at run time like a normal function:

```cpp
#include <iostream>

constexpr int factorial(int n)
{
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

```cpp
int main()
{
    constexpr int f5 = factorial(5);  // computed at compile time
    std::cout << f5 << "\n";          // 120

    int n = 6;
    int f6 = factorial(n);            // computed at run time (n is not constexpr)
    std::cout << f6 << "\n";          // 720

    return 0;
}
```

```
120
720
```

`constexpr` functions can use loops, conditionals, and local variables. The restrictions are that everything must be evaluable at compile time: no I/O, no dynamic allocation (mostly), and no undefined behavior.

`if constexpr`

`if constexpr` evaluates a condition at compile time and discards the unused branch entirely. This is especially useful in templates:

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
void describe(T value)
{
    if constexpr (std::is_integral_v<T>) {
        std::cout << value << " is an integer\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << value << " is a float\n";
    } else {
        std::cout << value << " is something else\n";
    }
}

int main()
{
    describe(42);
    describe(3.14);
    describe("Complicated");

    return 0;
}
```

```
42 is an integer
3.14 is a float
Complicated is something else
```

Unlike a regular `if`, the discarded branch does not need to compile for the given type. This is what makes `if constexpr` essential for template metaprogramming.

`consteval`

`consteval` (C++20) is stricter than `constexpr`. A `consteval` function **must** be evaluated at compile time. If you try to call it with run-time values, the compiler rejects the code:

```
consteval int square(int n)
{
    return n * n;
}

constexpr int x = square(5);  // OK: compile time
// int y = 6; square(y);      // error: y is not a compile-time constant
```

Use `consteval` when a function only makes sense at compile time — like computing array sizes, lookup table entries, or hash values for compile-time string matching.

Tip: Use `constexpr` when a function *can* run at compile time. Use `consteval` when it *must* run at compile time.

`constinit`

`constinit` (C++20) ensures that a variable with **static storage duration** (globals, file-scope variables, `static` locals) is initialized at compile time, avoiding the "static initialization order fiasco":

```
constinit int global_max = 100;  // guaranteed compile-time initialization
```

Unlike `constexpr`, `constinit` does not make the variable `const` — you can modify it after initialization:

```
constinit int counter = 0;  // initialized at compile time

void increment()
{
    counter++;  // OK: counter is not const
}
```

Wut: `constinit` only affects initialization, not the entire lifetime of the variable. A `constinit` variable is mutable after initialization (unlike `constexpr`, which is always `const`).

`static_assert`

`static_assert` checks a condition at compile time. If the condition is false, compilation fails with the message you provide:

```
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
static_assert(sizeof(void*) == 8, "this code assumes 64-bit pointers");
```

It is useful for documenting and enforcing assumptions about the platform, and for checking template parameters:

```
template<typename T>
class NumericBuffer {
    static_assert(std::is_arithmetic_v<T>, "T must be a numeric type");
    // ...
};
```

```
NumericBuffer<int> ok;            // compiles
// NumericBuffer<std::string> bad; // error: T must be a numeric type
```

`static_assert` has zero run-time cost — it exists only during compilation.

Type Aliases

Type aliases give a new name to an existing type, making complex types readable and easier to change later.

`using` vs. `typedef`

C++ has two ways to create type aliases. The modern `using` syntax is preferred:

```
// Modern (preferred)
using Playlist = std::vector<std::string>;
using SongMap = std::map<std::string, int>;

// Old-style typedef (still works, same effect)
typedef std::vector<std::string> Playlist;
typedef std::map<std::string, int> SongMap;
```

`using` is easier to read, especially for function pointer types:

```
// using
using Callback = void(*)(int);

// typedef — harder to parse
typedef void(*Callback)(int);
```

Alias Templates

`using` can also create templated type aliases, which `typedef` cannot:

```
template<typename T>
using Vec = std::vector<T>;

Vec<int> numbers = {1, 2, 3};
Vec<std::string> words = {"Estoy", "aqui"};
```

This is especially useful for simplifying nested template types:

```
template<typename K, typename V>
using HashMap = std::unordered_map<K, V>;

HashMap<std::string, int> scores;
```

Try It: Compile-Time Playground

Here is a program that exercises the compile-time features from this chapter. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <array>
#include <iostream>
#include <type_traits>

enum class Note { C = 0, D = 2, E = 4, F = 5, G = 7, A = 9, B = 11 };

constexpr int note_to_midi(Note n, int octave)
```

```
{
    return (octave + 1) * 12 + static_cast<int>(n);
}

consteval int bpm_to_ms(int bpm)
{
    return 60'000 / bpm;
}

template<typename T>
using Grid = std::array<std::array<T, 4>, 4>;

int main()
{
    // constexpr
    constexpr int middle_c = note_to_midi(Note::C, 4);
    std::cout << "Middle C MIDI: " << middle_c << "\n";

    // consteval — must be compile time
    constexpr int beat_ms = bpm_to_ms(120);
    std::cout << "120 BPM = " << beat_ms << " ms per beat\n";

    // static_assert
    static_assert(note_to_midi(Note::C, 4) == 60, "Middle C should be MIDI 60");
    static_assert(bpm_to_ms(120) == 500, "120 BPM should be 500 ms");

    // enum class with using enum
    using enum Note;
    constexpr auto a440 = note_to_midi(A, 4);
    std::cout << "A440 MIDI: " << a440 << "\n";

    // Type alias template
    Grid<int> pattern = {{
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1}
    }};

    std::cout << "Pattern[0][2]: " << pattern[0][2] << "\n";

    return 0;
}
```

```
Middle C MIDI: 60
120 BPM = 500 ms per beat
A440 MIDI: 69
Pattern[0][2]: 1
```

Try changing the `bpm_to_ms` call to use a non-constexpr variable and see the error. Add more notes and experiment with `static_assert` to verify your MIDI calculations.

Key Points

- **`enum class`** creates scoped enumerations with no implicit conversion to `int` and no name leakage. Use `static_cast` for explicit conversion.
- You can specify the underlying type: `enum class Foo : uint8_t`.
- **`using enum`** (C++20) brings enumerators into scope within a limited region.
- **`constexpr`** variables are compile-time constants. `constexpr` functions can run at compile time or run time depending on their arguments.
- **`consteval`** (C++20) functions must run at compile time — calling them with run-time values is a compile error.
- **`constinit`** (C++20) guarantees compile-time initialization for static-duration variables without making them `const`.
- **`if constexpr`** evaluates conditions at compile time and discards the unused branch, which is essential for templates.
- **`static_assert`** checks conditions at compile time with zero run-time cost.
- **Type aliases** with `using` are preferred over `typedef`. `using` supports alias templates; `typedef` does not.

Exercises

1. **Think about it:** Why does `enum class` require `static_cast` to convert to `int`, when the old `enum` converted implicitly? What bugs does this prevent?

2. **What does this print?**

```cpp
enum class Suit : int { Hearts = 0, Diamonds, Clubs, Spades };
int x = static_cast<int>(Suit::Spades);
std::cout << x << "\n";
```

3. **Where is the bug?**

```cpp
enum class Priority { Low, Medium, High };

void handle(Priority p)
{
    if (p == 2) {
        std::cout << "High priority!\n";
    }
}
```

4. **Calculation:** What is the value of `result`?

```cpp
constexpr int power(int base, int exp)
{
    int result = 1;
    for (int i = 0; i < exp; ++i) {
        result *= base;
    }
    return result;
}

constexpr int result = power(2, 10);
```

5. **Think about it:** What is the practical difference between `constexpr` and `consteval`? When would you use one over the other?

6. **Where is the bug?**

```cpp
consteval int compute(int x) { return x * x; }
```

```cpp
int main()
{
    int n;
    std::cin >> n;
    int result = compute(n);
    std::cout << result << "\n";
    return 0;
}
```

7. **What does this print?**

```cpp
template<typename T>
void check(T value)
{
    if constexpr (std::is_integral_v<T>) {
        std::cout << value * 2 << "\n";
    } else {
        std::cout << value << "\n";
    }
}

check(5);
check(3.14);
```

8. **Think about it:** Why does `constinit` exist as a separate keyword from `constexpr`? What problem does it solve that `constexpr` does not?

9. **Where is the bug?**

```cpp
using StringPair = std::pair<std::string, std::string>;

StringPair get_pair()
{
    return {"Hola", "mundo"};
}

auto [a, b] = get_pair();
std::cout << a << " " << b << "\n";
```

(Trick question — is there actually a bug?)

10. **Write a program** that defines a `constexpr` function to convert Fahrenheit to Celsius (`(f - 32) * 5 / 9.0`). Use `static_assert` to verify that `212` F is `100.0` C and `32` F is `0.0` C. Then define an `enum class Season { Spring, Summer, Fall, Winter }` and a `constexpr` function that returns a typical temperature for each season. Print all four seasons and their temperatures.

6. Advanced Strings

In *Gorgo Starting C++* you learned `std::string` for storing and manipulating text. `std::string` is versatile, but it owns its data — every copy creates a new allocation, and passing a string to a function that only needs to read it can be needlessly expensive. Sometimes you need pattern matching to validate input or extract structured data from text. Other times you need to convert between strings and numbers efficiently. In this chapter you will learn `std::string_view` for lightweight non-owning references to strings, `std::regex` for pattern matching, and the standard library's string-to-number and number-to-string conversion functions.

`std::string_view`

`std::string_view` (C++17, `#include <string_view>`) is a non-owning, read-only view of a character sequence. It stores a pointer and a length — no allocation, no copy:

```
void string_view(const char* str, size_t len);  // conceptually
```

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

void greet(std::string_view name)
{
    std::cout << "Hola, " << name << "!\n";
}

int main()
{
    std::string s = "Beyonce";
    greet(s);         // no copy — views into s
    greet("Shakira"); // no copy — views the string literal

    return 0;
}
```

```
Hola, Beyonce!
Hola, Shakira!
```

Without `string_view`, you would either pass `const std::string&` (which requires a `std::string` to exist) or `const char*` (which loses the length). `string_view` works with both — it is the best parameter type for functions that only need to read a string.

Common Operations

`string_view` supports most of the read-only operations you know from `std::string`:

```
std::string_view sv = "Lose Yourself";
std::cout << sv.size() << "\n";            // 13
std::cout << sv.substr(5, 8) << "\n";      // Yourself
std::cout << sv.find("Your") << "\n";      // 5
std::cout << sv.starts_with("Lose") << "\n";  // 1 (true)
std::cout << sv[0] << "\n";                // L
```

`substr()` on a `string_view` returns another `string_view` — no allocation. This is much cheaper than `std::string::substr()`, which creates a new string.

You can also narrow a view from either end:

```cpp
std::string_view sv = "  trimmed  ";
sv.remove_prefix(2);  // sv is now "trimmed  "
sv.remove_suffix(2);  // sv is now "trimmed"
```

Lifetime Dangers

Because `string_view` does not own its data, the underlying string must outlive the view:

```cpp
std::string_view dangerous()
{
    std::string s = "temporary";
    return s;  // BUG: s is destroyed, view becomes dangling
}
```

Trap: Never return a `string_view` to a local `std::string`. The string is destroyed when the function returns, and the view becomes a dangling pointer. Return `std::string` from functions that create new strings.

```cpp
// Also dangerous:
std::string_view sv;
{
    std::string temp = "gone soon";
    sv = temp;
}
// sv is dangling here — temp was destroyed
```

The rule is simple: use `string_view` for parameters and local variables when you know the source outlives the view. Store `std::string` when you need ownership.

Regular Expressions

The `<regex>` header provides pattern matching for strings. Regular expressions (regex) let you search for patterns instead of exact strings.

`std::regex_match`

`std::regex_match` checks whether an **entire** string matches a pattern:

```cpp
bool regex_match(const string& s, const regex& pattern);
bool regex_match(const string& s, smatch& match, const regex& pattern);
```

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

int main()
{
    std::regex email_pattern(R"(\w+@\w+\.\w+)");

    std::string s1 = "fan@correo.com";
    std::string s2 = "not-an-email";

    std::cout << std::regex_match(s1, email_pattern) << "\n";  // 1 (true)
    std::cout << std::regex_match(s2, email_pattern) << "\n";  // 0 (false)
```

```
    return 0;
}
```

```
1
0
```

The `R"(...)"` syntax is a **raw string literal** — backslashes are not escape characters, which makes regex patterns much easier to read.

std::regex_search

`std::regex_search` finds the first match **within** a string (it does not require the whole string to match):

```
bool regex_search(const string& s, smatch& match, const regex& pattern);
```

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

int main()
{
    std::string text = "Released in 2003, it sold 2 million copies by 2005";
    std::regex year_pattern(R"(\d{4})");
    std::smatch match;

    std::string::const_iterator start = text.cbegin();
    while (std::regex_search(start, text.cend(), match, year_pattern)) {
        std::cout << "Found year: " << match[0] << "\n";
        start = match.suffix().first;
    }

    return 0;
}
```

```
Found year: 2003
Found year: 2005
```

std::regex_replace

`std::regex_replace` replaces matches with a new string:

```
string regex_replace(const string& s, const regex& pattern, const string& replacement);
```

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

int main()
{
    std::string text = "Call me at 555-1234 or 555-5678";
    std::regex phone(R"(\d{3}-\d{4})");

    std::string redacted = std::regex_replace(text, phone, "XXX-XXXX");
    std::cout << redacted << "\n";

    return 0;
}
```

```
Call me at XXX-XXXX or XXX-XXXX
```

Capture Groups

Parentheses in a regex create **capture groups** that let you extract parts of a match:

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

int main()
{
    std::string entry = "Nelly Furtado - Say It Right (2006)";
    std::regex pattern(R"((.+) - (.+) \((\d{4})\))");
    std::smatch match;

    if (std::regex_match(entry, match, pattern)) {
        std::cout << "Artist: " << match[1] << "\n";
        std::cout << "Title:  " << match[2] << "\n";
        std::cout << "Year:   " << match[3] << "\n";
    }

    return 0;
}
```

```
Artist: Nelly Furtado
Title:  Say It Right
Year:   2006
```

`match[0]` is the entire match, `match[1]` is the first group, `match[2]` the second, and so on.

Tip: `std::regex` can be slow — it compiles the pattern at run time. If you use the same pattern repeatedly, create the `std::regex` object once and reuse it. For performance-critical code, consider a dedicated regex library.

Common Regex Patterns

Pattern	Matches
`\d`	A digit (0-9)
`\w`	A word character (letter, digit, or underscore)
`\s`	Whitespace
`.`	Any character
`*`	Zero or more of the preceding
`+`	One or more of the preceding
`?`	Zero or one of the preceding
`{n}`	Exactly n of the preceding
`{n,m}`	Between n and m of the preceding
`[abc]`	Any one of a, b, or c
`^`	Start of string
`$`	End of string
`(...)`	Capture group

String Conversions

Converting between strings and numbers is one of the most common operations in programming. C++ provides several approaches, each with different trade-offs.

`std::stoi`, `std::stod`, and Friends

The `<string>` header provides functions to convert strings to numbers:

```
int stoi(const string& str, size_t* pos = nullptr, int base = 10);
long stol(const string& str, size_t* pos = nullptr, int base = 10);
long long stoll(const string& str, size_t* pos = nullptr, int base = 10);
float stof(const string& str, size_t* pos = nullptr);
double stod(const string& str, size_t* pos = nullptr);
```

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

int main()
{
    std::string s = "2003";
    int year = std::stoi(s);
    std::cout << year + 1 << "\n";  // 2004

    double pi = std::stod("3.14159");
    std::cout << pi << "\n";  // 3.14159

    // Parsing hex
    int hex_val = std::stoi("FF", nullptr, 16);
    std::cout << hex_val << "\n";  // 255

    return 0;
}
```

These functions throw `std::invalid_argument` if the string cannot be parsed and `std::out_of_range` if the value is too large for the target type.

`std::to_string`

`std::to_string` converts numbers to strings:

```
string to_string(int value);
string to_string(double value);
// ... and other numeric types
```

```
int bpm = 128;
std::string msg = "Playing at " + std::to_string(bpm) + " BPM";
std::cout << msg << "\n";  // Playing at 128 BPM
```

Tip: For formatted output, `std::format` (Chapter 9 of *Gorgo Starting C++*) is more flexible than `std::to_string`. Use `std::to_string` when you just need a plain number-to-string conversion.

`std::from_chars` and `std::to_chars` (C++17)

For high-performance, locale-independent conversions, C++17 provides `std::from_chars` and `std::to_chars` in `<charconv>`:

```
from_chars_result from_chars(const char* first, const char* last, int& value, int base = 10);
to_chars_result to_chars(char* first, char* last, int value, int base = 10);
```

```
#include <charconv>
#include <iostream>

int main()
{
    // String to number
    const char* str = "42";
    int value = 0;
    auto [ptr, ec] = std::from_chars(str, str + 2, value);
    if (ec == std::errc{}) {
        std::cout << "Parsed: " << value << "\n";
    }

    // Number to string
    char buf[20];
    auto [end, ec2] = std::to_chars(buf, buf + sizeof(buf), 2006);
    if (ec2 == std::errc{}) {
        std::cout << "Formatted: " << std::string_view(buf, end - buf) << "\n";
    }

    return 0;
}
```

```
Parsed: 42
Formatted: 2006
```

`from_chars` and `to_chars` never allocate memory, never throw exceptions, and are not affected by locale settings. They are the fastest standard conversion functions available.

Comparison

Feature	`stoi/to_string`	`from_chars/to_chars`
Locale-dependent	Yes	No
Throws exceptions	Yes	No (uses error codes)
Allocates memory	`to_string` does	Never
Speed	Good	Fastest
Ease of use	Easy	Verbose

For most code, `stoi` and `to_string` are fine. Reach for `from_chars/to_chars` when you need maximum performance or locale independence (e.g., parsing data files or network protocols).

Try It: String Processing

Here is a program that exercises string_view, regex, and conversions. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <charconv>
#include <cstring>
#include <iostream>
#include <regex>
#include <string>
```

```cpp
#include <string_view>
#include <vector>

// Uses string_view — no copies
bool starts_with_the(std::string_view s)
{
    return s.starts_with("The");
}

int main()
{
    // string_view
    std::vector<std::string> bands = {
        "The Strokes", "Arcade Fire", "The Killers", "Interpol"
    };

    std::cout << "Bands starting with 'The':\n";
    for (const auto& b : bands) {
        if (starts_with_the(b)) {
            std::cout << "  " << b << "\n";
        }
    }

    // Regex: parse "Artist - Song (Year)" entries
    std::regex entry_re(R"((.+?) - (.+?) \((\d{4})\))");
    std::vector<std::string> entries = {
        "Gorillaz - Feel Good Inc (2005)",
        "Yeah Yeah Yeahs - Maps (2003)",
        "Keane - Somewhere Only We Know (2004)"
    };

    std::cout << "\nParsed entries:\n";
    for (const auto& e : entries) {
        std::smatch m;
        if (std::regex_match(e, m, entry_re)) {
            std::cout << "  " << m[2] << " by " << m[1] << " (" << m[3] << ")\n";
        }
    }

    // from_chars
    const char* nums[] = {"120", "140", "160"};
    std::cout << "\nBPM values:\n";
    for (const char* n : nums) {
        int bpm = 0;
        auto [ptr, ec] = std::from_chars(n, n + std::strlen(n), bpm);
        if (ec == std::errc{}) {
            std::cout << "  " << bpm << " BPM = " << (60'000 / bpm) << " ms\n";
        }
    }

    return 0;
}
```

Try adding more regex patterns — for example, one that extracts hashtags from a string or validates a date

format.

Key Points

- **`std::string_view`** is a lightweight, non-owning reference to a string. It avoids copies and works with both `std::string` and `const char*`.
- `string_view::substr()` returns another view (no allocation), unlike `string::substr()`.
- Never return a `string_view` to a local `std::string` — the view becomes dangling.
- **`std::regex`** provides pattern matching: `regex_match` for full-string matches, `regex_search` for partial matches, and `regex_replace` for substitution.
- Capture groups in parentheses let you extract parts of a match.
- Use raw string literals (`R"(...)"`) for readable regex patterns.
- **`std::stoi/std::stod`** convert strings to numbers; they throw on invalid input.
- **`std::to_string`** converts numbers to strings.
- **`std::from_chars/std::to_chars`** (C++17) are the fastest conversion functions: no allocation, no exceptions, no locale dependency.

Exercises

1. **Think about it:** Why is `std::string_view` a better function parameter type than `const std::string&` for functions that only read the string? What is the main risk of using `string_view`?
2. **What does this print?**

```cpp
std::string_view sv = "Estoy aqui";
sv.remove_prefix(6);
std::cout << sv << "\n";
std::cout << sv.size() << "\n";
```

3. **Where is the bug?**

```cpp
std::string_view get_greeting()
{
    std::string s = "Buenos dias";
    return s;
}

std::cout << get_greeting() << "\n";
```

4. **What does this print?**

```cpp
std::regex pattern(R"(\d+)");
std::string text = "Track 7 of 12";
std::smatch match;

if (std::regex_search(text, match, pattern)) {
    std::cout << match[0] << "\n";
}
```

5. **Calculation:** What does `std::stoi("0xFF", nullptr, 16)` return?
6. **Think about it:** Why do `std::from_chars` and `std::to_chars` not use exceptions? What advantage does this give for performance-critical code?
7. **Where is the bug?**

```cpp
std::string s = "not a number";
int n = std::stoi(s);
std::cout << n << "\n";
```

8. **What does this print?**

```
std::string entry = "Daft Punk - One More Time (2000)";
std::regex re(R"((.+) - (.+) \((\d+)\))");
std::smatch m;
std::regex_match(entry, m, re);
std::cout << m[2] << "\n";
```

9. **Think about it:** The `<regex>` library can be slow for complex patterns. What alternatives exist in the C++ ecosystem for high-performance regex matching?

10. **Write a program** that takes a list of strings in the format "Name:Score" (e.g., "Alice:95", "Bob:87") stored in a `std::vector<std::string>`. Use `std::regex` or `string_view::find` to parse each entry, convert the score to an `int`, and print the name and score. Also print the average score.

7. Utilities

Real programs deal with messy situations: a lookup might find nothing, a value could be one of several types, and functions sometimes need to return multiple values at once. Before C++17, programmers used raw pointers for "maybe no value," unions for "one of several types," and output parameters or custom structs for "return multiple things." All of these are clunky and error-prone. The modern standard library provides clean, type-safe alternatives: `std::optional`, `std::variant`, `std::any`, `std::tuple`, and `std::pair`. In this chapter you will learn when and how to use each one.

`std::optional`

`std::optional<T>` (C++17, `#include <optional>`) holds either a value of type `T` or nothing at all. It is the right tool when "no value" is a valid result — like a database lookup that might not find a match or a configuration setting that might not be set:

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

std::optional<std::string> find_artist(int track_id)
{
    if (track_id == 1) return "Outkast";
    if (track_id == 2) return "Missy Elliott";
    return std::nullopt;  // nothing found
}

int main()
{
    auto result = find_artist(1);
    if (result.has_value()) {
        std::cout << "Found: " << result.value() << "\n";
    }

    auto missing = find_artist(99);
    if (!missing) {  // same as !missing.has_value()
        std::cout << "Not found\n";
    }

    return 0;
}
```

```
Found: Outkast
Not found
```

Accessing the Value

There are several ways to get the value out of an optional:

```
std::optional<int> opt = 42;

int a = opt.value();        // throws std::bad_optional_access if empty
int b = *opt;               // undefined behavior if empty — no check!
int c = opt.value_or(0);    // returns 0 if empty
```

Trap: `*opt` does not check whether the optional contains a value. Use `value()` when you want an exception on empty access, or check with `has_value()` / `if (opt)` first.

Monadic Operations (C++23)

C++23 added three methods that let you chain operations on optionals without nested `if` checks:

```
std::optional<T> transform(F func);   // apply func if has value, return optional<Result>
std::optional<U> and_then(F func);    // apply func that returns optional<U>
std::optional<T> or_else(F func);     // if empty, call func to provide fallback
```

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

std::optional<std::string> lookup(int id)
{
    if (id == 1) return "Nelly";
    return std::nullopt;
}

int main()
{
    auto result = lookup(1)
        .transform([](const std::string& s) { return s + " Furtado"; })
        .value_or("Unknown");

    std::cout << result << "\n";  // Nelly Furtado

    auto empty = lookup(99)
        .transform([](const std::string& s) { return s + " Furtado"; })
        .value_or("Unknown");

    std::cout << empty << "\n";  // Unknown

    return 0;
}
```

```
Nelly Furtado
Unknown
```

`transform` applies the function only if the optional has a value, propagating `nullopt` otherwise. This avoids the pyramid of `if (opt)` checks.

`std::variant`

`std::variant<Types...>` (C++17, `#include <variant>`) holds exactly one value from a fixed set of types. It is a type-safe alternative to C unions:

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

int main()
{
```

```
    std::variant<int, double, std::string> v;

    v = 42;
    std::cout << std::get<int>(v) << "\n";       // 42

    v = "In the End";
    std::cout << std::get<std::string>(v) << "\n";  // In the End

    v = 3.14;
    std::cout << std::get<double>(v) << "\n";    // 3.14

    return 0;
}
```

```
42
In the End
3.14
```

A `variant` always holds exactly one of its types. Assigning a new value changes the active type.

Checking the Active Type

```
std::variant<int, std::string> v = "Hola";

if (std::holds_alternative<std::string>(v)) {
    std::cout << "It's a string: " << std::get<std::string>(v) << "\n";
}

// std::get throws std::bad_variant_access if the wrong type is active
// std::get_if returns a pointer (nullptr if wrong type)
if (auto* p = std::get_if<int>(&v)) {
    std::cout << "It's an int: " << *p << "\n";
}
```

`std::visit`

`std::visit` calls a function with the currently active value, whatever its type. The function must handle all possible types:

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

int main()
{
    std::variant<int, double, std::string> v = "Complicated";

    std::visit([](auto&& val) {
        std::cout << val << "\n";
    }, v);

    return 0;
}
```

```
Complicated
```

The lambda uses `auto&&` to accept any type. For type-specific behavior, you can use an **overload set**:

```cpp
struct Visitor {
    void operator()(int i) const { std::cout << "int: " << i << "\n"; }
    void operator()(double d) const { std::cout << "double: " << d << "\n"; }
    void operator()(const std::string& s) const { std::cout << "string: " << s << "\n"; }
};

std::visit(Visitor{}, v);
```

Tip: `std::variant` is especially useful for representing states (e.g., `variant<Loading, Loaded, Error>`) or heterogeneous data (e.g., a JSON value that could be a number, string, bool, or null).

std::any

`std::any` (C++17, `#include <any>`) can hold a value of **any** type. Unlike `variant`, you do not need to list the possible types upfront:

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

int main()
{
    std::any a = 42;
    std::cout << std::any_cast<int>(a) << "\n";  // 42

    a = std::string("Lean on Me");
    std::cout << std::any_cast<std::string>(a) << "\n";  // Lean on Me

    // Wrong type throws std::bad_any_cast
    try {
        std::cout << std::any_cast<double>(a) << "\n";
    } catch (const std::bad_any_cast& e) {
        std::cout << "Bad cast: " << e.what() << "\n";
    }

    return 0;
}
```

```
42
Lean on Me
Bad cast: bad any_cast
```

`std::any` uses type erasure internally and can hold any copyable type. It is useful for plugin systems or generic containers where the set of types is not known at compile time.

Wut: Prefer `std::variant` over `std::any` when you know the possible types. `variant` checks types at compile time; `any` defers all checking to run time. `any` is essentially a type-safe `void*` — use it only when you genuinely do not know the type in advance.

std::tuple

`std::tuple<Types...>` (C++11, `#include <tuple>`) groups a fixed number of values of different types. It is a generalization of `std::pair` to any number of elements:

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

int main()
{
    std::tuple<std::string, int, double> track("Yeah!", 2004, 4.5);

    std::cout << std::get<0>(track) << "\n";  // Yeah!
    std::cout << std::get<1>(track) << "\n";  // 2004
    std::cout << std::get<2>(track) << "\n";  // 4.5

    return 0;
}
```

```
Yeah!
2004
4.5
```

Structured Bindings

Accessing tuple elements by index is awkward. C++17 **structured bindings** let you unpack a tuple into named variables:

```cpp
auto [title, year, rating] = track;
std::cout << title << " (" << year << ") - " << rating << " stars\n";
```

Structured bindings work with tuples, pairs, arrays, and structs:

```cpp
// With std::pair
std::pair<std::string, int> album = {"Elephunk", 2003};
auto [name, yr] = album;

// With arrays
int arr[] = {10, 20, 30};
auto [a, b, c] = arr;

// With structs
struct Point { double x, y; };
Point p = {3.0, 4.0};
auto [px, py] = p;
```

`std::make_tuple` and `std::tie`

`std::make_tuple` creates a tuple with deduced types:

```cpp
auto t = std::make_tuple("Breathe Me", 2005, true);
```

`std::tie` creates a tuple of references, useful for unpacking into existing variables or for comparison:

```cpp
std::string title;
int year;
bool favorite;

std::tie(title, year, favorite) = t;
std::cout << title << "\n";  // Breathe Me
```

With C++17 structured bindings, you rarely need `std::tie` anymore.

Returning Multiple Values

Tuples are a natural way to return multiple values from a function:

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

std::tuple<std::string, int> parse_track(const std::string& entry)
{
    auto dash = entry.find(" - ");
    return {entry.substr(0, dash), std::stoi(entry.substr(dash + 3))};
}

int main()
{
    auto [artist, year] = parse_track("Snow Patrol - 2003");
    std::cout << artist << ", " << year << "\n";

    return 0;
}
```

```
Snow Patrol, 2003
```

Tip: For functions that return two or three related values, a named struct is often clearer than a tuple. `auto [name, age, score]` is fine; `auto [a, b, c, d, e, f]` is not — the reader has no idea what each element means.

`std::pair` Revisited

You have already used `std::pair` with `std::map` in Chapter 3. A pair is just a two-element tuple with named members `first` and `second`:

```
std::pair<std::string, int> song("Lollipop", 2008);
std::cout << song.first << ": " << song.second << "\n";  // Lollipop: 2008

// C++17: CTAD
std::pair p("Stacy's Mom", 2003);  // deduces pair<const char*, int>
```

You can also create pairs with `std::make_pair`:

```
auto p = std::make_pair("Float On", 2004);
```

Structured bindings make pairs much more readable than accessing `.first` and `.second`:

```
for (const auto& [song, year] : my_map) {
    std::cout << song << " (" << year << ")\n";
}
```

Try It: Utility Sampler

Here is a program that exercises the utility types from this chapter. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <any>
#include <iostream>
#include <optional>
#include <string>
```

```cpp
#include <tuple>
#include <variant>
#include <vector>

// optional: safe lookup
std::optional<int> find_year(const std::string& title)
{
    if (title == "Take Me Out") return 2004;
    if (title == "Float On") return 2004;
    if (title == "Naive") return 2006;
    return std::nullopt;
}

// variant: a value that could be several types
using JsonValue = std::variant<int, double, std::string, bool>;

void print_json(const JsonValue& v)
{
    std::visit([](auto&& val) {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, bool>) {
            std::cout << (val ? "true" : "false");
        } else if constexpr (std::is_same_v<T, std::string>) {
            std::cout << "\"" << val << "\"";
        } else {
            std::cout << val;
        }
    }, v);
}

int main()
{
    // optional
    for (const auto& title : {"Take Me Out", "Unknown Song", "Naive"}) {
        auto year = find_year(title);
        std::cout << title << ": " << year.value_or(0) << "\n";
    }

    // variant
    std::cout << "\nJSON values: ";
    std::vector<JsonValue> values = {42, 3.14, std::string("Hola"), true};
    for (const auto& v : values) {
        print_json(v);
        std::cout << " ";
    }
    std::cout << "\n";

    // tuple
    auto [artist, album, year] = std::make_tuple("Franz Ferdinand", "Franz Ferdinand", 2004);
    std::cout << "\n" << artist << " - " << album << " (" << year << ")\n";

    // any
    std::any wild = 42;
    std::cout << "\nany<int>: " << std::any_cast<int>(wild) << "\n";
```

```
    wild = std::string("Anything goes");
    std::cout << "any<string>: " << std::any_cast<std::string>(wild) << "\n";

    return 0;
}
```

Try adding a function that returns `std::optional<std::tuple<...>>` for a lookup that returns multiple values or nothing.

Key Points

- **`std::optional<T>`** represents a value that may or may not be present. Use it instead of sentinel values (`-1`, `nullptr`) or output parameters.
- Access optional values with `value()` (throws if empty), `*opt` (UB if empty), or `value_or(default)`.
- C++23 monadic operations (`transform`, `and_then`, `or_else`) let you chain operations on optionals without nested checks.
- **`std::variant<Types...>`** holds one value from a fixed set of types. It is a type-safe replacement for C unions.
- `std::visit` dispatches to a function based on the active type. Use `std::holds_alternative` or `std::get_if` for type checks.
- **`std::any`** can hold any copyable type, checked only at run time. Prefer `variant` when you know the possible types.
- **`std::tuple`** groups multiple values of different types. **Structured bindings** (`auto [a, b, c]`) make tuples readable.
- **`std::pair`** is a two-element tuple with `first` and `second` members. Structured bindings are preferred over accessing `.first`/`.second` directly.
- For functions returning multiple values, prefer a named struct when there are more than two or three elements.

Exercises

1. **Think about it:** Why is `std::optional` better than returning a magic value like `-1` or an empty string to indicate "no result"?
2. **What does this print?**

   ```
   std::optional<int> opt;
   std::cout << opt.value_or(42) << "\n";
   opt = 7;
   std::cout << opt.value_or(42) << "\n";
   ```

3. **Where is the bug?**

   ```
   std::optional<std::string> name;
   std::cout << *name << "\n";
   ```

4. **What does this print?**

   ```
   std::variant<int, std::string> v = 42;
   v = "changed";
   std::cout << std::holds_alternative<int>(v) << "\n";
   std::cout << std::holds_alternative<std::string>(v) << "\n";
   ```

5. **Think about it:** When would you use `std::any` instead of `std::variant`? Give a concrete example.
6. **Calculation:** Given:

   ```
   auto t = std::make_tuple(10, 20, 30, 40);
   auto [a, b, c, d] = t;
   ```

What are the values of `a`, `b`, `c`, and `d`?

7. **Where is the bug?**

```
std::variant<int, double, std::string> v = 3.14;
int x = std::get<int>(v);
```

8. **What does this print?**

```
std::pair p(std::string("Naive"), 2006);
auto [title, year] = p;
year = 2007;
std::cout << p.second << "\n";
```

9. **Think about it:** The text suggests using a named struct instead of a large tuple for function return values. Why? What are the advantages of each approach?

10. **Write a program** that defines a function `parse_color` which takes a string like `"rgb(255,128,0)"` and returns `std::optional<std::tuple<int, int, int>>`. Return `std::nullopt` if the format is wrong. Test it with valid and invalid inputs, and use structured bindings to unpack successful results.

8. Namespaces and the Preprocessor

As projects grow, name collisions become inevitable. Two libraries might both define a `log` function, or your `Error` class might clash with one from a dependency. Namespaces solve this by grouping names into distinct scopes. The preprocessor, inherited from C, controls what code the compiler sees — include guards prevent double-inclusion, macros define constants and conditional blocks, and conditional compilation lets you target different platforms. C++20 modules aim to replace much of the preprocessor's job. In this chapter you will learn namespace design, the preprocessor's key features, and a preview of modules.

Namespace Basics

You have used `std::` since the beginning of *Gorgo Starting C++*. `std` is a namespace — a named scope that contains the entire standard library. You can create your own:

```
#include <iostream>

namespace audio {

void play(const char* track)
{
    std::cout << "Playing: " << track << "\n";
}

int volume = 80;

}  // namespace audio

int main()
{
    audio::play("Yellow");
    std::cout << "Volume: " << audio::volume << "\n";

    return 0;
}
```

```
Playing: Yellow
Volume: 80
```

Everything inside `namespace audio { ... }` is accessed with the `audio::` prefix.

Nested Namespaces

Before C++17, nesting namespaces required separate blocks:

```
namespace company {
    namespace audio {
        namespace codec {
            void decode() {}
        }
    }
}
```

C++17 lets you write this in one line:

```
namespace company::audio::codec {
    void decode() {}
}
```

```
company::audio::codec::decode();
```

Anonymous Namespaces

An **anonymous namespace** gives its contents internal linkage — they are visible only within the current file:

```
namespace {
    int helper_count = 0;

    void internal_helper()
    {
        helper_count++;
    }
}
```

This is the modern C++ replacement for the `static` keyword at file scope. Other files cannot see `helper_count` or `internal_helper` even if they try to declare them with `extern`.

Tip: Use anonymous namespaces instead of `static` for file-local functions and variables. The `static` keyword at file scope is a holdover from C and is considered less idiomatic in C++.

Inline Namespaces

Inline namespaces make their contents accessible as if they were in the enclosing namespace. They are primarily used for API versioning:

```
#include <iostream>

namespace mylib {

inline namespace v2 {
    void greet() { std::cout << "Hola from v2!\n"; }
}

namespace v1 {
    void greet() { std::cout << "Hola from v1!\n"; }
}

}  // namespace mylib

int main()
{
    mylib::greet();       // calls v2::greet (inline namespace)
    mylib::v1::greet();   // explicitly calls v1
    mylib::v2::greet();   // explicitly calls v2

    return 0;
}
```

```
Hola from v2!
Hola from v1!
Hola from v2!
```

By marking `v2` as `inline`, users who call `mylib::greet()` automatically get the latest version. Users who need the old version can explicitly qualify `mylib::v1::greet()`.

`using` Declarations vs. `using` Directives

There are two ways to bring namespace members into the current scope.

A **`using` declaration** imports a single name:

```
using std::cout;
using std::string;

cout << "No prefix needed\n";
string s = "clean";
```

A **`using` directive** imports an entire namespace:

```
using namespace std;

cout << "Everything from std is visible\n";
```

Trap: Never put `using namespace std;` (or any `using` directive) in a header file. It pollutes the namespace of every file that includes the header, causing unexpected name collisions. `using namespace` in a `.cpp` file or inside a function is acceptable but use it with care.

The guideline:

Context	Recommendation
Header files	Never use `using namespace`. Use full qualification.
Source files (top level)	`using` declarations for frequently used names
Inside functions	`using namespace` is acceptable for convenience

Include Guards

When a header file is `#included` from multiple places, the compiler can see the same declarations twice. **Include guards** prevent this:

```
// audio.h — traditional include guard
#ifndef AUDIO_H
#define AUDIO_H

void play(const char* track);

#endif  // AUDIO_H
```

The first time `audio.h` is included, `AUDIO_H` is not defined, so the content is processed and `AUDIO_H` gets defined. The second time, `AUDIO_H` is already defined, so the entire content is skipped.

`#pragma once` is a simpler alternative supported by all major compilers:

```
// audio.h — pragma once
#pragma once

void play(const char* track);
```

Feature	Include guards	`#pragma once`
Standard	Yes (part of the language)	Not standard, but universally supported
Syntax	Verbose (3 lines)	One line
Edge cases	Works everywhere	May fail with symlinks or network drives

Tip: Either approach works. `#pragma once` is simpler and is the de facto standard in modern codebases. Use traditional include guards if your build environment has exotic filesystem issues.

Macros and Conditional Compilation

The C preprocessor runs before the compiler sees your code. Macros are text substitutions — they replace one sequence of tokens with another.

`#define`

```
#define MAX_TRACKS 100
#define PI 3.14159

int tracks[MAX_TRACKS];
```

The preprocessor replaces every occurrence of `MAX_TRACKS` with `100` before compilation.

Tip: Prefer `constexpr` variables over `#define` for constants. `constexpr` is type-safe and respects scopes; macros do not:

```
constexpr int max_tracks = 100;  // preferred
#define MAX_TRACKS 100            // avoid when possible
```

Function-Like Macros

```
#define SQUARE(x) ((x) * (x))

int a = SQUARE(5);    // expands to ((5) * (5)) = 25
int b = SQUARE(2+3);  // expands to ((2+3) * (2+3)) = 25
```

The extra parentheses are critical — without them, `SQUARE(2+3)` would expand to `2+3 * 2+3 = 11`.

Trap: Macros are pure text replacement. They do not understand types, scopes, or expressions. Prefer `constexpr` functions or templates over function-like macros.

Conditional Compilation

Conditional compilation lets you include or exclude code based on compile-time conditions:

```
#ifdef _WIN32
    #include <windows.h>
#elif defined(__linux__)
    #include <unistd.h>
#elif defined(__APPLE__)
    #include <mach/mach.h>
#endif
```

Common predefined macros:

Macro	Meaning
`_WIN32`	Windows
`__linux__`	Linux
`__APPLE__`	macOS / iOS
`__cplusplus`	C++ standard version (e.g., `202302L` for C++23)
`NDEBUG`	Release mode (disables `assert`)

```
#if __cplusplus >= 202002L
    // C++20 or later
    #include <ranges>
#else
    // Fallback for older compilers
#endif
```

When to Use Macros

Macros still have legitimate uses: - Include guards - Platform-specific conditional compilation - `assert()` (needs to capture `__FILE__` and `__LINE__`) - Compile-time feature detection

For everything else — constants, inline functions, type-safe generics — use `constexpr`, `inline`, and templates.

Modules Preview (C++20)

C++20 introduced **modules** as a modern replacement for the `#include` / header-file model. Modules solve several long-standing problems: - Headers are processed every time they are included, slowing compilation - Include order can matter (macros leak across headers) - Include guards / `#pragma once` are workarounds, not solutions

Basic Syntax

A module is defined with `export module`:

```
// greeting.cppm (module interface file)
export module greeting;

import <string>;

export std::string greet(const std::string& name)
{
    return "Hola, " + name + "!";
}
```

And consumed with `import`:

```
// main.cpp
import greeting;
import <iostream>;

int main()
{
    std::cout << greet("Mundo") << "\n";

    return 0;
}
```

Current State

Module support is improving across compilers, but as of this writing: - MSVC has the most mature support - GCC and Clang support modules but with some limitations - Build system support (CMake, etc.) is still evolving

Tip: Modules are the future of C++ code organization, but the ecosystem is not fully there yet. Learn the concepts now, and start using them when your toolchain supports them well. For now, headers with `#pragma once` remain the practical choice for most projects.

Try It: Namespace Organization

Here is a program that demonstrates namespace design. Type it in, compile with `g++ -std=c++23`, and experiment:

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

namespace studio {

namespace audio {
    void play(const std::string& track)
    {
        std::cout << "Playing: " << track << "\n";
    }
}

namespace video {
    void play(const std::string& clip)
    {
        std::cout << "Showing: " << clip << "\n";
    }
}

inline namespace v2 {
    std::string format_title(const std::string& title)
    {
        return "[" + title + "]";
    }
}

namespace v1 {
    std::string format_title(const std::string& title)
    {
        return title;
    }
}

}  // namespace studio

namespace {
    int internal_counter = 0;
    void tick() { internal_counter++; }
}
```

```cpp
int main()
{
    // Nested namespaces
    studio::audio::play("Speed of Sound");
    studio::video::play("music video");

    // Inline namespace (v2 is default)
    std::cout << studio::format_title("Harder Better Faster Stronger") << "\n";
    std::cout << studio::v1::format_title("Harder Better Faster Stronger") << "\n";

    // Anonymous namespace
    tick();
    tick();
    std::cout << "Counter: " << internal_counter << "\n";

    // using declaration
    using studio::audio::play;
    play("Around the World");

    // Conditional compilation
    #ifdef NDEBUG
        std::cout << "Release build\n";
    #else
        std::cout << "Debug build\n";
    #endif

    return 0;
}
```

Try renaming the `play` functions to the same name in different namespaces and see how the compiler resolves them. Try removing the `inline` from `v2` and see what happens when you call `studio::format_title`.

Key Points

- **Namespaces** group names to avoid collisions. Use `::` to access members.
- **Nested namespaces** can be declared with `namespace a::b::c { }` (C++17).
- **Anonymous namespaces** give contents internal linkage (file-local visibility), replacing `static` at file scope.
- **Inline namespaces** make their contents accessible as if they were in the enclosing namespace, useful for API versioning.
- A **`using` declaration** imports a single name; a **`using` directive** imports an entire namespace. Never use `using namespace` in header files.
- **Include guards** (`#ifndef`/`#define`/`#endif`) or **`#pragma once`** prevent double-inclusion of headers.
- **Macros** are text substitution. Prefer `constexpr` for constants and templates/`inline` for function-like macros.
- **Conditional compilation** (`#ifdef`, `#if`) is useful for platform-specific code and feature detection.
- **Modules** (C++20) are the modern alternative to headers, offering faster compilation and better isolation. Ecosystem support is still maturing.

Exercises

1. **Think about it:** Why is `using namespace std;` in a header file dangerous? Give a specific example of a name collision it could cause.

2. **What does this print?**

```cpp
namespace a {
    int x = 1;
    namespace b {
        int x = 2;
    }
}

std::cout << a::x << " " << a::b::x << "\n";
```

3. **Where is the bug?**

```cpp
// file1.cpp
namespace { int count = 0; }

// file2.cpp
extern int count;
void increment() { count++; }
```

4. **Think about it:** When would you use an inline namespace? Describe a real-world scenario where it would be useful.

5. **What does this print?**

```cpp
#define DOUBLE(x) x * 2

int result = DOUBLE(3 + 4);
std::cout << result << "\n";
```

6. **Where is the bug?** (And what is the fix?)

```cpp
// utils.h
using namespace std;

string format(int x);
```

7. **Think about it:** What advantages do C++20 modules have over traditional headers? Why hasn't the industry fully adopted them yet?

8. **What does this print?**

```cpp
namespace outer {
    inline namespace inner {
        int value = 42;
    }
}

std::cout << outer::value << "\n";
std::cout << outer::inner::value << "\n";
```

9. **Calculation:** Given the macro:

```cpp
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

What is the value of `MAX(3+1, 2+3)`?

10. **Write a program** that defines a namespace `music` with sub-namespaces `rock` and `pop`, each containing a function `top_song()` that returns a different string. Use a `using` declaration to bring one into scope and call both. Add an anonymous namespace with a helper function used by both sub-namespaces.

9. RAII and Resource Management

In *Gorgo Starting C++* you learned that `std::unique_ptr` and `std::shared_ptr` manage memory automatically by freeing it when the pointer goes out of scope. That pattern — acquiring a resource in a constructor and releasing it in a destructor — is one of the most important ideas in C++. It has a name: **RAII** (Resource Acquisition Is Initialization). RAII applies to far more than memory: file handles, network connections, mutex locks, database transactions, and any other resource that must eventually be released. In this chapter you will learn the RAII pattern in depth, exception safety guarantees, scope guards, and how to use custom deleters with smart pointers.

The RAII Pattern

RAII ties the lifetime of a resource to the lifetime of an object:

1. The **constructor** acquires the resource.
2. The **destructor** releases the resource.
3. Because C++ guarantees that destructors run when objects leave scope (even when exceptions are thrown), the resource is always released.

Here is RAII applied to a file handle:

```
#include <cstdio>
#include <stdexcept>
#include <string>

class FileHandle {
public:
    FileHandle(const std::string& path, const char* mode)
        : fp_(std::fopen(path.c_str(), mode))
    {
        if (!fp_) {
            throw std::runtime_error("Cannot open: " + path);
        }
    }

    ~FileHandle()
    {
        if (fp_) {
            std::fclose(fp_);
        }
    }

    // Prevent copying (two objects should not close the same file)
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Allow moving
    FileHandle(FileHandle&& other) noexcept : fp_(other.fp_)
    {
        other.fp_ = nullptr;
    }

    FILE* get() const { return fp_; }

private:
```

```
    FILE* fp_;
};
```

With this class, a file is always closed, no matter how the function exits:

```
void process_file(const std::string& path)
{
    FileHandle file(path, "r");
    // ... use file.get() ...
    // If an exception is thrown here, ~FileHandle still runs
}
// ~FileHandle runs when 'file' goes out of scope
```

Without RAII, you would need to remember to call `fclose()` on every exit path — including the ones created by exceptions.

Tip: If you find yourself writing cleanup code in multiple places, you probably need an RAII wrapper. If the resource already has a standard wrapper (like `std::fstream` for files, `std::lock_guard` for mutexes), use that instead of writing your own.

RAII Beyond Memory

RAII works with any resource:

Resource	RAII Wrapper
Heap memory	`std::unique_ptr`, `std::shared_ptr`
Files	`std::fstream`, `std::ofstream`, `std::ifstream`
Mutex locks	`std::lock_guard`, `std::unique_lock`
Database connections	Custom wrapper
Network sockets	Custom wrapper
Temporary files	Custom wrapper (create in ctor, delete in dtor)

Exception Safety Guarantees

When a function throws an exception, what state does it leave the program in? C++ defines three levels of **exception safety**:

Basic Guarantee

The **basic guarantee** promises: - No resources are leaked. - The program is in a valid state (no undefined behavior). - But the state may have changed — partial modifications may be visible.

Most well-written C++ code provides at least the basic guarantee. RAII gives you this almost for free: if every resource is managed by an object, destructors clean up automatically.

Strong Guarantee

The **strong guarantee** promises: - If the function throws, the program state is unchanged — as if the function was never called. - This is "commit or rollback" semantics.

Providing the strong guarantee usually means doing all work on a copy, then swapping:

```
void update_playlist(std::vector<std::string>& playlist, const std::string& song)
{
    std::vector<std::string> temp = playlist;  // copy
```

```
    temp.push_back(song);                    // modify copy
    // If push_back throws (bad_alloc), playlist is untouched
    playlist = std::move(temp);              // commit (noexcept)
}
```

Nothrow Guarantee

The **nothrow guarantee** promises the function never throws. Mark such functions with `noexcept`:

```
void swap(int& a, int& b) noexcept
{
    int temp = a;
    a = b;
    b = temp;
}
```

Destructors are implicitly `noexcept`. Move constructors and move assignment operators should be `noexcept` whenever possible — the standard library containers rely on this for efficiency.

Trap: If a `noexcept` function does throw, `std::terminate` is called and the program crashes. Only use `noexcept` when you are certain the function cannot throw.

Which Guarantee to Aim For

Situation	Recommendation
Destructors	Always nothrow
Move operations	Nothrow whenever possible
Simple operations	Basic guarantee is usually sufficient
Operations that modify shared state	Consider strong guarantee
Swap functions	Nothrow

Scope Guards

Sometimes you need cleanup that does not fit neatly into a class destructor. A **scope guard** is a small RAII object that runs a function when it goes out of scope:

```
#include <functional>
#include <iostream>

class ScopeGuard {
public:
    explicit ScopeGuard(std::function<void()> cleanup)
        : cleanup_(std::move(cleanup)) {}

    ~ScopeGuard()
    {
        if (cleanup_) {
            cleanup_();
        }
    }

    void dismiss() { cleanup_ = nullptr; }
```

```
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;

private:
    std::function<void()> cleanup_;
};
```

Usage:

```
void process()
{
    auto* raw = acquire_resource();
    ScopeGuard guard([raw]() {
        release_resource(raw);
        std::cout << "Resource released\n";
    });

    // ... do work that might throw ...

    // If we get here successfully, maybe we want to keep the resource:
    // guard.dismiss();  // cancel the cleanup
}
// guard's destructor releases the resource if not dismissed
```

The `dismiss()` method lets you cancel the cleanup if the operation succeeds — useful for commit/rollback patterns.

Tip: The C++ standard library does not have a scope guard yet, but `<experimental/scope>` provides `scope_exit`, `scope_success`, and `scope_fail` in some implementations. Writing your own is straightforward, as shown above.

Custom Deleters with Smart Pointers

`std::unique_ptr` and `std::shared_ptr` call `delete` by default, but you can provide a **custom deleter** for resources that need different cleanup.

`unique_ptr` with Custom Deleter

The deleter is part of the type:

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

int main()
{
    auto file_deleter = [](FILE* fp) {
        if (fp) {
            std::fclose(fp);
            std::cout << "File closed\n";
        }
    };

    std::unique_ptr<FILE, decltype(file_deleter)> file(
        std::fopen("playlist.txt", "w"), file_deleter);
```

```
    if (file) {
        std::fprintf(file.get(), "1. Clocks\n2. Yellow\n");
    }

    return 0;
}
// file_deleter runs here, closing the file
```

```
File closed
```

`shared_ptr` with Custom Deleter

With `shared_ptr`, the deleter is **not** part of the type — you pass it as a constructor argument:

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

int main()
{
    std::shared_ptr<void> memory(
        std::malloc(1024),
        [](void* ptr) {
            std::free(ptr);
            std::cout << "Memory freed\n";
        }
    );

    // Use memory.get() ...

    return 0;
}
```

```
Memory freed
```

Practical Example: C Library Handles

Many C libraries return opaque handles that must be freed with a specific function. Custom deleters let you manage them with smart pointers:

```
// Example with a hypothetical C library
// Handle create_session();
// void destroy_session(Handle h);

auto deleter = [](Handle* h) { destroy_session(*h); delete h; };
std::unique_ptr<Handle, decltype(deleter)> session(
    new Handle(create_session()), deleter);
```

Tip: When wrapping C library resources, prefer `unique_ptr` with a custom deleter. It has zero overhead compared to manual cleanup and guarantees the resource is released exactly once.

Try It: RAII in Action

Here is a program that demonstrates RAII with different resource types. Type it in, compile with `g++ -std=c++23`, and experiment:

```cpp
#include <cstdio>
#include <functional>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <string>

// Simple scope guard
class ScopeGuard {
public:
    explicit ScopeGuard(std::function<void()> fn) : fn_(std::move(fn)) {}
    ~ScopeGuard() { if (fn_) fn_(); }
    void dismiss() { fn_ = nullptr; }
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;
private:
    std::function<void()> fn_;
};

// RAII file wrapper using unique_ptr with custom deleter
using FilePtr = std::unique_ptr<FILE, decltype([](FILE* f) { std::fclose(f); })>;

FilePtr open_file(const std::string& path, const char* mode)
{
    FILE* fp = std::fopen(path.c_str(), mode);
    if (!fp) throw std::runtime_error("Cannot open: " + path);
    return FilePtr(fp);
}

int main()
{
    // RAII file handle
    try {
        auto file = open_file("/tmp/raii_test.txt", "w");
        std::fprintf(file.get(), "Somebody That I Used to Know\n");
        std::cout << "Wrote to file\n";
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << "\n";
    }
    // file is automatically closed here

    // Scope guard
    std::cout << "Starting operation...\n";
    {
        ScopeGuard guard([]() {
            std::cout << "Cleanup complete\n";
        });

        std::cout << "Doing work...\n";
        // guard.dismiss();  // uncomment to skip cleanup
    }
    // guard runs cleanup here

    // shared_ptr with custom deleter
```

```
    {
        auto ptr = std::shared_ptr<int>(
            new int(42),
            [](int* p) {
                std::cout << "Custom delete: " << *p << "\n";
                delete p;
            }
        );
        std::cout << "Value: " << *ptr << "\n";
    }

    return 0;
}
```

Try adding a function that throws an exception after opening a file and verify the file is still closed. Try the scope guard with `dismiss()` to see the difference.

Key Points

- **RAII** (Resource Acquisition Is Initialization) ties resource lifetime to object lifetime. The constructor acquires, the destructor releases.
- C++ guarantees destructors run when objects leave scope, even during exception unwinding. This makes RAII the foundation of exception-safe code.
- The **basic guarantee** promises no leaks and valid state but allows partial modifications.
- The **strong guarantee** promises rollback on failure (copy, modify, swap).
- The **nothrow guarantee** (`noexcept`) promises no exceptions. Use it for destructors, moves, and swaps.
- **Scope guards** are lightweight RAII objects that run a cleanup function at scope exit. `dismiss()` can cancel the cleanup for commit/rollback patterns.
- **Custom deleters** let `unique_ptr` and `shared_ptr` manage non-memory resources (file handles, C library objects).
- `unique_ptr` custom deleters are part of the type; `shared_ptr` custom deleters are not.
- When wrapping C resources, prefer `unique_ptr` with a custom deleter — zero overhead, guaranteed cleanup.

Exercises

1. **Think about it:** Why is RAII considered one of the most important patterns in C++? How does it compare to try/finally in languages like Java and Python?
2. **What happens here?**

   ```
   void risky()
   {
       FILE* fp = fopen("data.txt", "r");
       process(fp);  // might throw
       fclose(fp);
   }
   ```

   What goes wrong if `process` throws an exception? How would you fix it with RAII?
3. **Think about it:** Why should move constructors and move assignment operators be `noexcept`? What happens if they are not?
4. **Where is the bug?**

   ```
   class Connection {
   public:
   ```

```
    Connection() { connect(); }
    ~Connection() { disconnect(); }
};

void transfer()
{
    Connection c1;
    Connection c2 = c1;  // copy
    // ... work ...
}
```

5. **Calculation:** How many times is `fclose` called?

```
{
    auto d = [](FILE* f) { fclose(f); };
    std::unique_ptr<FILE, decltype(d)> f1(fopen("a.txt", "r"), d);
    auto f2 = std::move(f1);
}
```

6. **Think about it:** What is the difference between the basic guarantee and the strong guarantee? Give an example where the basic guarantee is sufficient and one where you would want the strong guarantee.

7. **Where is the bug?**

```
void process()
{
    auto ptr = std::make_unique<int[]>(100);
    // ... do work ...
    ptr.release();  // "release" the memory
}
```

8. **What does this print?**

```
{
    ScopeGuard g1([]() { std::cout << "A "; });
    ScopeGuard g2([]() { std::cout << "B "; });
    ScopeGuard g3([]() { std::cout << "C "; });
}
```

   (Using the ScopeGuard class from this chapter.)

9. **Think about it:** Why does `shared_ptr` not include the deleter in its type, while `unique_ptr` does? What design trade-off does this represent?

10. **Write a program** that wraps `malloc`/`free` in a `unique_ptr` with a custom deleter. Allocate an array of 10 integers, fill them with values, print them, and verify the memory is freed by printing a message in the deleter.

10. Concurrency

Modern CPUs have multiple cores, but the programs you have written so far use only one. **Concurrency** lets your program do several things at the same time — processing data on one thread while waiting for I/O on another, or splitting a large computation across cores to finish faster. Concurrency is powerful but treacherous: shared data accessed from multiple threads without coordination leads to **data races** — subtle, non-deterministic bugs that are among the hardest to diagnose. In this chapter you will learn to create threads, protect shared data with mutexes, coordinate threads with condition variables, use `std::async` for higher-level concurrency, and avoid common pitfalls.

`std::thread`

`std::thread` (`#include <thread>`) creates a new thread of execution:

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

void hello()
{
    std::cout << "Hola from a thread!\n";
}

int main()
{
    std::thread t(hello);  // start a new thread running hello()
    t.join();              // wait for it to finish

    return 0;
}
```

```
Hola from a thread!
```

`t.join()` blocks the calling thread until `t` finishes. You **must** either `join()` or `detach()` every thread before it is destroyed, or the program calls `std::terminate`.

Passing Arguments

You can pass arguments to the thread function:

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

void play(const std::string& song, int track_num)
{
    std::cout << "Track " << track_num << ": " << song << "\n";
}

int main()
{
    std::thread t(play, "Hung Up", 1);
    t.join();

    return 0;
}
```

```
Track 1: Hung Up
```

Trap: Arguments are **copied** into the thread by default. If you need to pass by reference, use `std::ref()`:

```
int count = 0;
std::thread t([](int& c) { c++; }, std::ref(count));
t.join();
// count is now 1
```

Without `std::ref`, the thread gets its own copy and `count` stays 0.

Detaching Threads

`t.detach()` lets the thread run independently. The thread continues even after the `std::thread` object is destroyed:

```
std::thread t(background_task);
t.detach();  // thread runs on its own
// Be careful: if t accesses local variables, they may be destroyed!
```

Trap: A detached thread has no way to report back. If it accesses variables from the creating scope, those variables may already be destroyed. Prefer `join()` unless you have a specific reason to detach.

Lambdas as Thread Functions

Lambdas are the most common way to write thread functions:

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

int main()
{
    int result = 0;

    std::thread t([&result]() {
        result = 42;
    });
    t.join();

    std::cout << "Result: " << result << "\n";

    return 0;
}
```

```
Result: 42
```

`std::jthread` (C++20)

`std::jthread` automatically joins in its destructor, so you never forget:

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

int main()
{
    std::jthread t([]() {
        std::cout << "Auto-joining thread\n";
```

```
    });
    // No need to call t.join() — destructor handles it

    return 0;
}
```

Mutexes and Locks

When two threads access the same data and at least one writes, you have a **data race** — undefined behavior. A **mutex** (mutual exclusion) prevents this by ensuring only one thread accesses the protected data at a time.

`std::mutex`

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

std::mutex mtx;
int shared_count = 0;

void increment(int times)
{
    for (int i = 0; i < times; ++i) {
        mtx.lock();
        shared_count++;
        mtx.unlock();
    }
}

int main()
{
    std::thread t1(increment, 100000);
    std::thread t2(increment, 100000);
    t1.join();
    t2.join();

    std::cout << "Count: " << shared_count << "\n";  // 200000

    return 0;
}
```

```
Count: 200000
```

Without the mutex, the count would be unpredictable — both threads could read the same value, increment it, and write back, losing an increment.

`std::lock_guard`

Calling `lock()` and `unlock()` manually is error-prone — if an exception is thrown between them, the mutex stays locked forever. `std::lock_guard` uses RAII (Chapter 9) to lock on construction and unlock on destruction:

```
void increment(int times)
{
    for (int i = 0; i < times; ++i) {
```

```
        std::lock_guard<std::mutex> guard(mtx);
        shared_count++;
        // mutex unlocked when guard goes out of scope
    }
}
```

Tip: Always use `lock_guard` (or `unique_lock`, `scoped_lock`) instead of calling `lock()`/`unlock()` directly. This is RAII applied to mutex locking.

`std::unique_lock`

`std::unique_lock` is like `lock_guard` but more flexible — you can unlock and relock it, and it is movable:

```
std::unique_lock<std::mutex> lock(mtx);
// ... critical section ...
lock.unlock();
// ... non-critical work ...
lock.lock();
// ... another critical section ...
```

`unique_lock` is required for condition variables (covered next).

`std::scoped_lock` (C++17)

`std::scoped_lock` can lock multiple mutexes at once without risking deadlock:

```
std::mutex mtx1, mtx2;

void transfer()
{
    std::scoped_lock lock(mtx1, mtx2);  // locks both, deadlock-free
    // ... modify data protected by both mutexes ...
}
```

If you tried to lock them separately, two threads could each lock one mutex and wait for the other — a **deadlock**.

Condition Variables

A **condition variable** lets one thread wait until another thread signals that something has happened. The classic use is the **producer-consumer** pattern:

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <queue>
#include <string>
#include <thread>
#include <vector>

std::queue<std::string> work_queue;
std::mutex mtx;
std::condition_variable cv;
bool done = false;
```

```cpp
void producer()
{
    std::vector<std::string> songs = {"Toxic", "Crazy", "Rehab"};
    for (const auto& song : songs) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            work_queue.push(song);
        }
        cv.notify_one();
    }
    {
        std::lock_guard<std::mutex> lock(mtx);
        done = true;
    }
    cv.notify_one();
}

void consumer()
{
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []() { return !work_queue.empty() || done; });

        while (!work_queue.empty()) {
            std::cout << "Processing: " << work_queue.front() << "\n";
            work_queue.pop();
        }

        if (done) break;
    }
}

int main()
{
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

```
Processing: Toxic
Processing: Crazy
Processing: Rehab
```

Key points about condition variables: - `cv.wait(lock, predicate)` unlocks the mutex, sleeps until notified, relocks, and checks the predicate. If the predicate is false, it goes back to sleep. - Always use the predicate form of `wait` to handle **spurious wakeups** — the OS can wake a thread without a notification. - `notify_one()` wakes one waiting thread; `notify_all()` wakes all of them.

Wut: Condition variables can experience **spurious wakeups** — the thread wakes up even though no one called `notify`. This is why you must always use the predicate form: `cv.wait(lock, predicate)`. Without it, your thread may proceed when it should still be waiting.

`std::async` and `std::future`

Threads are low-level. `std::async` (`#include <future>`) provides a higher-level way to run a task asynchronously and get its result:

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

std::string fetch_lyrics(const std::string& song)
{
    // Simulate slow operation
    return "Lyrics for: " + song;
}

int main()
{
    // Start async task
    std::future<std::string> result = std::async(std::launch::async,
        fetch_lyrics, "Somewhere Only We Know");

    std::cout << "Doing other work...\n";

    // Get the result (blocks if not ready yet)
    std::cout << result.get() << "\n";

    return 0;
}
```

```
Doing other work...
Lyrics for: Somewhere Only We Know
```

`std::async` returns a `std::future` that holds the result. Calling `future.get()` blocks until the result is ready and returns it. If the async task threw an exception, `get()` rethrows it.

Launch Policies

Policy	Behavior
`std::launch::async`	Guaranteed new thread
`std::launch::deferred`	Runs when `get()` is called (lazy)
Default (both)	Implementation chooses

Tip: `std::async` is the easiest way to parallelize independent tasks. Use it when you need a result from a background computation. Use `std::thread` when you need more control over the thread's lifetime.

Atomics

For simple operations on shared variables (counters, flags), mutexes are overkill. **Atomic operations** guarantee that reads and writes happen indivisibly, without locks:

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::atomic<int> counter(0);

void count_up(int times)
{
    for (int i = 0; i < times; ++i) {
        counter++;  // atomic increment — no mutex needed
    }
}

int main()
{
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(count_up, 100000);
    }
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Counter: " << counter << "\n";  // 400000

    return 0;
}
```

```
Counter: 400000
```

Common atomic operations:

```
std::atomic<int> a(0);
a.store(5);             // write
int x = a.load();       // read
int old = a.exchange(10);  // swap and return old value
a.fetch_add(1);         // atomic increment
a.fetch_sub(1);         // atomic decrement
```

Tip: Use `std::atomic` for simple shared counters and flags. For anything more complex (protecting multiple variables or a data structure), use a mutex.

Thread Safety Pitfalls

Data Races

A data race occurs when two threads access the same variable, at least one writes, and there is no synchronization. Data races are **undefined behavior** — anything can happen:

```
// BUG: data race on 'count'
int count = 0;

void bad_increment()
{
    for (int i = 0; i < 100000; ++i) {
        count++;  // not atomic, not protected
    }
}
```

Fix with a mutex, `lock_guard`, or `std::atomic<int>`.

Deadlock

A deadlock occurs when two threads each hold a lock the other needs:

```
// Thread 1: lock A, then lock B
// Thread 2: lock B, then lock A
// Both wait forever!
```

Prevention strategies: - Always lock mutexes in the same order. - Use `std::scoped_lock` to lock multiple mutexes simultaneously. - Minimize the time you hold locks.

False Sharing

When two threads modify different variables that happen to share a cache line, the CPU invalidates the cache on every write, destroying performance. This is **false sharing**:

```
// Both on the same cache line — slow!
struct Counters {
    int a;  // thread 1 writes this
    int b;  // thread 2 writes this
};
```

Fix by padding:

```
struct alignas(64) Counters {
    alignas(64) int a;
    alignas(64) int b;
};
```

Try It: Concurrent Counter

Here is a program that demonstrates threads, mutexes, and atomics. Type it in, compile with `g++ -std=c++23 -pthread`, and experiment:

```
#include <atomic>
#include <chrono>
#include <future>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

// Shared state
std::mutex mtx;
int mutex_count = 0;
std::atomic<int> atomic_count(0);
```

```
void mutex_increment(int n)
{
    for (int i = 0; i < n; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        mutex_count++;
    }
}

void atomic_increment(int n)
{
    for (int i = 0; i < n; ++i) {
        atomic_count++;
    }
}

template<typename Func>
long long benchmark(Func f, int threads, int per_thread)
{
    auto start = std::chrono::high_resolution_clock::now();

    std::vector<std::thread> workers;
    for (int i = 0; i < threads; ++i) {
        workers.emplace_back(f, per_thread);
    }
    for (auto& t : workers) {
        t.join();
    }

    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
}

int main()
{
    const int threads = 4;
    const int per_thread = 1000000;

    auto ms1 = benchmark(mutex_increment, threads, per_thread);
    std::cout << "Mutex count:  " << mutex_count
              << " (" << ms1 << " ms)\n";

    auto ms2 = benchmark(atomic_increment, threads, per_thread);
    std::cout << "Atomic count: " << atomic_count
              << " (" << ms2 << " ms)\n";

    // std::async example
    auto fut = std::async(std::launch::async, []() {
        return std::string("Async result ready");
    });

    std::cout << fut.get() << "\n";

    return 0;
```

```
}
```

Compare the performance of mutex-based and atomic-based counting. Try removing the synchronization entirely and see how the count becomes incorrect.

Key Points

- **`std::thread`** creates a new thread of execution. Every thread must be `join()`ed or `detach()`ed before destruction.
- **`std::jthread`** (C++20) automatically joins on destruction.
- Arguments are copied into threads by default; use `std::ref()` for references.
- **`std::mutex`** provides mutual exclusion. Use **`std::lock_guard`** or **`std::scoped_lock`** (RAII) instead of manual `lock()`/`unlock()`.
- **`std::unique_lock`** is more flexible than `lock_guard` — it can be unlocked, relocked, and moved.
- **`std::scoped_lock`** (C++17) locks multiple mutexes simultaneously to prevent deadlock.
- **Condition variables** let threads wait for a signal. Always use the predicate form of `wait` to handle spurious wakeups.
- **`std::async`** and **`std::future`** provide high-level asynchronous computation. `future.get()` blocks until the result is ready.
- **`std::atomic`** provides lock-free operations for simple types (counters, flags).
- **Data races** (unsynchronized access) are undefined behavior. Use mutexes, atomics, or other synchronization primitives.
- **Deadlocks** occur when threads wait for each other's locks. Lock in consistent order or use `scoped_lock`.

Exercises

1. **Think about it:** Why must every `std::thread` be either joined or detached? What happens if you destroy a joinable thread?
2. **Where is the bug?**

   ```
   int total = 0;

   void add(int n) { total += n; }

   std::thread t1(add, 100);
   std::thread t2(add, 200);
   t1.join();
   t2.join();

   std::cout << total << "\n";
   ```

3. **Think about it:** Why does `std::lock_guard` not have `unlock()` and `lock()` methods, while `std::unique_lock` does? When would you need the extra flexibility?
4. **What does this program print?** (Approximately — exact output depends on scheduling.)

   ```
   std::atomic<int> x(0);

   std::thread t1([&]() { x++; x++; x++; });
   std::thread t2([&]() { x++; x++; x++; });
   t1.join();
   t2.join();

   std::cout << x << "\n";
   ```

5. **Where is the deadlock?**

```cpp
std::mutex m1, m2;

void thread_a()
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
    // ...
}

void thread_b()
{
    std::lock_guard<std::mutex> lock1(m2);
    std::lock_guard<std::mutex> lock2(m1);
    // ...
}
```

How would you fix it?

6. **Think about it:** When should you use `std::async` instead of creating a `std::thread` manually? What are the advantages?

7. **What value does `result` hold?**

```cpp
auto fut = std::async(std::launch::deferred, []() { return 6 * 7; });
// ... other work ...
int result = fut.get();
```

8. **Where is the bug?**

```cpp
std::mutex mtx;

void process()
{
    mtx.lock();
    if (some_condition()) {
        return;  // oops
    }
    // ... more work ...
    mtx.unlock();
}
```

9. **Think about it:** Why is `std::atomic` faster than using a mutex for simple counters? When would you still prefer a mutex over an atomic?

10. **Write a program** that uses four threads to compute the sum of a large vector (1 million elements). Each thread should sum one quarter of the vector. Use `std::async` and `std::future` to collect the partial sums, then print the total.

11. The Filesystem Library

In *Gorgo Starting C++* you learned to read and write file contents with `std::fstream`. But working with files means more than reading and writing: you often need to check whether a file exists, list directory contents, copy or rename files, or build paths that work across operating systems. Before C++17, you needed platform-specific APIs (POSIX or Windows) for all of this. C++17 introduced `<filesystem>`, a portable library for working with paths, directories, and file metadata. In this chapter you will learn `std::filesystem::path`, directory iteration, file operations, and file status queries.

All filesystem types and functions live in the `std::filesystem` namespace. We will use the common alias:

```
#include <filesystem>
namespace fs = std::filesystem;
```

`std::filesystem::path`

A `fs::path` represents a file path in a platform-independent way. It handles separator differences (`/` on Unix, `\` on Windows) automatically:

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "/home/user/music/playlist.m3u";

    std::cout << "Full path:  " << p << "\n";
    std::cout << "Filename:   " << p.filename() << "\n";
    std::cout << "Stem:       " << p.stem() << "\n";
    std::cout << "Extension:  " << p.extension() << "\n";
    std::cout << "Parent:     " << p.parent_path() << "\n";

    return 0;
}
```

```
Full path:  "/home/user/music/playlist.m3u"
Filename:   "playlist.m3u"
Stem:       "playlist"
Extension:  ".m3u"
Parent:     "/home/user/music"
```

Building Paths with /

The `/` operator concatenates paths, inserting the correct separator:

```
fs::path base = "/home/user";
fs::path full = base / "music" / "album" / "track01.mp3";
std::cout << full << "\n";
// "/home/user/music/album/track01.mp3"
```

This is much cleaner than string concatenation and handles platform differences automatically.

Other Path Operations

```
fs::path p = "/home/user/docs/../music/./track.mp3";

// Normalize the path (resolve . and ..)
std::cout << p.lexically_normal() << "\n";
// "/home/user/music/track.mp3"

// Make relative to another path
fs::path base = "/home/user";
std::cout << p.lexically_relative(base) << "\n";
// "docs/../music/./track.mp3"

// Check if path is absolute or relative
std::cout << p.is_absolute() << "\n";  // true
std::cout << fs::path("music/track.mp3").is_relative() << "\n";  // true
```

Checking File Status

Before operating on a file, you often need to check whether it exists and what kind of entry it is:

```
fs::path p = "/home/user/music";

// bool exists(const path& p);
if (fs::exists(p)) {
    std::cout << p << " exists\n";
}

// bool is_regular_file(const path& p);
// bool is_directory(const path& p);
// bool is_symlink(const path& p);
if (fs::is_directory(p)) {
    std::cout << p << " is a directory\n";
}

if (fs::is_regular_file(p / "track.mp3")) {
    std::cout << "It's a file\n";
}
```

File Size

```
// uintmax_t file_size(const path& p);
auto size = fs::file_size("/home/user/music/track.mp3");
std::cout << "Size: " << size << " bytes\n";
```

Trap: `file_size` throws `filesystem_error` if the file does not exist or you do not have permission. Check `exists()` first, or use the overload that takes a `std::error_code` parameter.

Error Handling

Most filesystem functions have two overloads: - One that throws `fs::filesystem_error` on failure. - One that takes a `std::error_code&` parameter and sets it instead of throwing.

```cpp
std::error_code ec;
auto size = fs::file_size("nonexistent.txt", ec);
if (ec) {
    std::cout << "Error: " << ec.message() << "\n";
} else {
    std::cout << "Size: " << size << "\n";
}
```

Directory Iteration

`directory_iterator`

`fs::directory_iterator` lists the entries in a single directory:

```cpp
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path dir = "/home/user/music";

    for (const auto& entry : fs::directory_iterator(dir)) {
        std::cout << entry.path().filename();
        if (entry.is_directory()) {
            std::cout << "/";
        }
        std::cout << "\n";
    }

    return 0;
}
```

Each `entry` is a `fs::directory_entry` with methods like `path()`, `is_regular_file()`, `is_directory()`, and `file_size()`.

`recursive_directory_iterator`

`fs::recursive_directory_iterator` walks the entire directory tree:

```cpp
for (const auto& entry : fs::recursive_directory_iterator(dir)) {
    if (entry.is_regular_file() && entry.path().extension() == ".mp3") {
        std::cout << entry.path() << "\n";
    }
}
```

Tip: `recursive_directory_iterator` follows symlinks by default on some platforms. Use `fs::directory_options::skip_permission_denied` to avoid exceptions on directories you cannot read:

```cpp
for (const auto& entry : fs::recursive_directory_iterator(
        dir, fs::directory_options::skip_permission_denied)) {
    // ...
}
```

File Operations

Creating Directories

```
// bool create_directory(const path& p);
fs::create_directory("/home/user/music/new_album");

// bool create_directories(const path& p);  — creates parent dirs too
fs::create_directories("/home/user/music/2005/singles");
```

`create_directory` fails if the parent does not exist. `create_directories` creates the entire path.

Copying Files

```
// void copy(const path& from, const path& to);
// void copy_file(const path& from, const path& to);
fs::copy_file("track01.mp3", "backup/track01.mp3");

// With options
fs::copy_file("track01.mp3", "backup/track01.mp3",
              fs::copy_options::overwrite_existing);
```

Copy options:

Option	Effect
`none`	Fail if destination exists (default)
`overwrite_existing`	Replace the destination
`skip_existing`	Silently skip if destination exists
`update_existing`	Replace only if source is newer

Renaming and Moving

```
// void rename(const path& old_p, const path& new_p);
fs::rename("track01.mp3", "01_intro.mp3");
fs::rename("old_dir", "new_dir");  // works for directories too
```

`rename` can also move files across directories on the same filesystem.

Removing Files

```
// bool remove(const path& p);
fs::remove("temp.txt");

// uintmax_t remove_all(const path& p);  — removes directory and all contents
auto count = fs::remove_all("old_backup");
std::cout << "Removed " << count << " entries\n";
```

Trap: `remove_all` recursively deletes everything inside a directory. Double-check the path before calling it — there is no "undo."

File Permissions

You can query and modify file permissions:

```cpp
// fs::perms permissions(const path& p);
auto perms = fs::status("script.sh").permissions();

if ((perms & fs::perms::owner_exec) != fs::perms::none) {
    std::cout << "Owner can execute\n";
}

// void permissions(const path& p, perms prms, perm_options opts);
fs::permissions("script.sh",
    fs::perms::owner_exec,
    fs::perm_options::add);
```

Common permission flags:

Flag	Meaning
`owner_read`	Owner can read
`owner_write`	Owner can write
`owner_exec`	Owner can execute
`group_read`	Group can read
`others_read`	Others can read
`all`	All permissions

Try It: File Manager

Here is a program that exercises the filesystem library. Type it in, compile with `g++ -std=c++23`, and experiment:

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

namespace fs = std::filesystem;

void create_test_files(const fs::path& dir)
{
    fs::create_directories(dir / "rock");
    fs::create_directories(dir / "pop");

    std::ofstream(dir / "rock" / "boulevard.txt") << "Boulevard of Broken Dreams\n";
    std::ofstream(dir / "rock" / "numb.txt") << "Numb\n";
    std::ofstream(dir / "pop" / "umbrella.txt") << "Umbrella\n";
    std::ofstream(dir / "readme.txt") << "Test files\n";
}

int main()
{
    fs::path test_dir = fs::temp_directory_path() / "fs_test";

    // Clean up from previous runs
    fs::remove_all(test_dir);

    // Create test structure
    create_test_files(test_dir);
```

```
    std::cout << "Created files in: " << test_dir << "\n\n";

    // List all files recursively
    std::cout << "All files:\n";
    for (const auto& entry : fs::recursive_directory_iterator(test_dir)) {
        auto rel = fs::relative(entry.path(), test_dir);
        if (entry.is_regular_file()) {
            std::cout << "  " << rel << " (" << entry.file_size() << " bytes)\n";
        } else if (entry.is_directory()) {
            std::cout << "  " << rel << "/\n";
        }
    }

    // Copy a file
    fs::copy_file(test_dir / "rock" / "numb.txt",
                  test_dir / "pop" / "numb_copy.txt");
    std::cout << "\nCopied numb.txt to pop/\n";

    // Rename
    fs::rename(test_dir / "readme.txt", test_dir / "README.txt");
    std::cout << "Renamed readme.txt -> README.txt\n";

    // Check existence
    std::cout << "\nREADME.txt exists: "
              << fs::exists(test_dir / "README.txt") << "\n";
    std::cout << "readme.txt exists: "
              << fs::exists(test_dir / "readme.txt") << "\n";

    // Count .txt files
    int txt_count = 0;
    for (const auto& entry : fs::recursive_directory_iterator(test_dir)) {
        if (entry.path().extension() == ".txt") {
            txt_count++;
        }
    }
    std::cout << "\nTotal .txt files: " << txt_count << "\n";

    // Cleanup
    auto removed = fs::remove_all(test_dir);
    std::cout << "Cleaned up " << removed << " entries\n";

    return 0;
}
```

Try modifying this to find the largest file, list only directories, or filter by extension.

Key Points

- **`std::filesystem::path`** represents file paths portably. Use `/` operator to concatenate paths.
- `filename()`, `stem()`, `extension()`, and `parent_path()` decompose paths.
- `lexically_normal()` resolves `.` and `..` segments.
- **`exists()`**, **`is_regular_file()`**, **`is_directory()`** check file status.
- **`file_size()`** returns the size of a regular file in bytes.
- **`directory_iterator`** lists a single directory; **`recursive_directory_iterator`** walks the entire tree.

- **`create_directory`** creates one directory; **`create_directories`** creates the entire path.
- **`copy_file`** copies a file; use `copy_options` to control overwrite behavior.
- **`rename`** renames or moves entries; **`remove`** deletes a file; **`remove_all`** deletes a directory tree.
- Most functions have a throwing overload and a `std::error_code` overload for error handling.
- File permissions can be queried with `status().permissions()` and modified with `permissions()`.

Exercises

1. **Think about it:** Why does `std::filesystem::path` use `/` as the concatenation operator instead of `+`? What would go wrong with `+`?

2. **What does this print?**

```
fs::path p = "/home/user/docs/report.pdf";
std::cout << p.stem() << "\n";
std::cout << p.extension() << "\n";
std::cout << p.parent_path().filename() << "\n";
```

3. **Where is the bug?**

```
auto size = fs::file_size("maybe_missing.txt");
std::cout << "Size: " << size << "\n";
```

4. **Think about it:** What is the difference between `directory_iterator` and `recursive_directory_iterator`? When would you use each?

5. **What does this print?**

```
fs::path p = "/home/user/./docs/../music/track.mp3";
std::cout << p.lexically_normal() << "\n";
```

6. **Calculation:** You call `create_directories("/a/b/c/d")` on a system where only `/a` exists. How many new directories are created?

7. **Where is the bug?**

```
fs::create_directory("/new_project/src/main");
```

   (Assume `/new_project` does not exist yet.)

8. **Think about it:** Why does `remove_all` return the number of entries removed? When would this be useful?

9. **What happens?**

```
fs::copy_file("a.txt", "b.txt");
fs::copy_file("a.txt", "b.txt");
```

   What happens on the second call? How would you fix it?

10. **Write a program** that takes a directory path as a command-line argument and prints a summary: the total number of files, the total number of directories, and the total size of all regular files in bytes. Use `recursive_directory_iterator` to walk the tree.

12. Best Practices and Common Idioms

You now have a solid foundation in C++: classes, templates, the standard library, concurrency, and more. Knowing the language features is necessary, but knowing how to use them well is what separates a competent programmer from a great one. This chapter covers the coding standards, idioms, and patterns that experienced C++ developers rely on, and closes with a look at what is coming in C++26.

Coding Standards and Style

Naming Conventions

There is no single naming convention in C++. The standard library uses `snake_case` for everything. Google style uses `CamelCase` for types and `snake_case` for functions. The most important rule is **consistency within your project**:

Element	Common styles
Types / classes	`PascalCase` or `snake_case`
Functions	`snake_case` or `camelCase`
Variables	`snake_case`
Constants	`kConstantName` or `UPPER_CASE`
Private members	`name_` (trailing underscore)
Macros	`UPPER_CASE` (the one universal convention)

Const-Correctness

Mark everything `const` that should not change. This communicates intent, catches bugs at compile time, and enables optimizations:

```
// Parameters
void process(const std::string& name);   // does not modify name
void modify(std::string& name);           // may modify name

// Member functions
class Playlist {
public:
    int size() const;                     // does not modify the object
    void add(const std::string& song);    // modifies the object
};

// Variables
const int max_tracks = 100;
```

Tip: Make everything `const` by default. Only remove `const` when you have a reason to mutate. This is the opposite of what most beginners do, and it prevents a large class of bugs.

The Rule of Zero

If your class does not manage resources directly (it uses `std::string`, `std::vector`, smart pointers, etc.), you should not write any special member functions — the compiler-generated defaults do the right thing:

```
class Song {
public:
    Song(std::string title, std::string artist)
```

```
        : title_(std::move(title)), artist_(std::move(artist)) {}

    // No destructor, no copy/move constructors, no assignment operators.
    // The compiler generates correct versions automatically.

private:
    std::string title_;
    std::string artist_;
};
```

This is the **Rule of Zero**: let the compiler do the work.

The Rule of Five

If your class manages a resource directly (raw pointers, file handles, etc.), you must define all five special member functions:

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator

```
#include <cstring>

class Buffer {
public:
    Buffer(size_t size) : data_(new char[size]), size_(size) {}

    ~Buffer() { delete[] data_; }

    Buffer(const Buffer& other) : data_(new char[other.size_]), size_(other.size_)
    {
        std::memcpy(data_, other.data_, size_);
    }

    Buffer& operator=(const Buffer& other)
    {
        if (this != &other) {
            delete[] data_;
            data_ = new char[other.size_];
            size_ = other.size_;
            std::memcpy(data_, other.data_, size_);
        }
        return *this;
    }

    Buffer(Buffer&& other) noexcept : data_(other.data_), size_(other.size_)
    {
        other.data_ = nullptr;
        other.size_ = 0;
    }

    Buffer& operator=(Buffer&& other) noexcept
    {
        if (this != &other) {
```

```
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }

private:
    char* data_;
    size_t size_;
};
```

Tip: If you find yourself writing the Rule of Five, ask whether a standard type (like `std::vector<char>`) could manage the resource for you instead, bringing you back to the Rule of Zero.

When to Use `auto`

`auto` deduces the type from the initializer. Use it when the type is obvious or verbose, not when it obscures meaning:

```
// Good: type is clear from context
auto it = my_map.find("key");
auto [name, score] = get_result();
auto ptr = std::make_unique<Widget>();

// Bad: what type is this?
auto x = compute();          // unclear — reader must find compute()
auto result = process(data); // unclear
```

Common C++ Idioms

PIMPL (Pointer to Implementation)

The **PIMPL** idiom hides implementation details behind a pointer, reducing compile-time dependencies. Changes to the implementation do not require recompiling users of the class:

```
// playlist.h
#include <memory>
#include <string>

class Playlist {
public:
    Playlist();
    ~Playlist();
    void add(const std::string& song);
    int size() const;

private:
    struct Impl;                // forward declaration
    std::unique_ptr<Impl> pimpl_;
};
```

```cpp
// playlist.cpp
#include "playlist.h"
#include <vector>

struct Playlist::Impl {
    std::vector<std::string> songs;
};

Playlist::Playlist() : pimpl_(std::make_unique<Impl>()) {}
Playlist::~Playlist() = default;

void Playlist::add(const std::string& song)
{
    pimpl_->songs.push_back(song);
}

int Playlist::size() const
{
    return static_cast<int>(pimpl_->songs.size());
}
```

The header does not include `<vector>`, so files that include `playlist.h` do not depend on `<vector>`. This can significantly reduce build times in large projects.

Wut: The destructor must be defined in the `.cpp` file (even as `= default`) because `unique_ptr` needs the complete type to destroy it. If you let the compiler generate the destructor in the header, it will fail because `Impl` is incomplete there.

CRTP (Curiously Recurring Template Pattern)

The **CRTP** provides **static polymorphism** — polymorphic behavior resolved at compile time instead of run time:

```cpp
#include <iostream>

template<typename Derived>
class Player {
public:
    void play()
    {
        static_cast<Derived*>(this)->play_impl();
    }
};

class MP3Player : public Player<MP3Player> {
public:
    void play_impl()
    {
        std::cout << "Playing MP3\n";
    }
};

class WAVPlayer : public Player<WAVPlayer> {
public:
    void play_impl()
```

```
    {
        std::cout << "Playing WAV\n";
    }
};

template<typename T>
void start_playback(Player<T>& player)
{
    player.play();
}

int main()
{
    MP3Player mp3;
    WAVPlayer wav;
    start_playback(mp3);  // Playing MP3
    start_playback(wav);  // Playing WAV

    return 0;
}
```

```
Playing MP3
Playing WAV
```

Unlike virtual functions, CRTP has no vtable overhead. The compiler inlines the call because it knows the exact type at compile time.

The trade-off: you cannot store different `Player` types in the same container (no common base class), so CRTP does not replace virtual functions when you need run-time polymorphism (Chapter 1).

Tag Dispatch

Tag dispatch selects a function overload at compile time using empty tag types:

```
#include <iostream>

struct fast_tag {};
struct safe_tag {};

void process(int x, fast_tag)
{
    std::cout << "Fast path: " << x << "\n";
}

void process(int x, safe_tag)
{
    if (x < 0) {
        std::cout << "Error: negative\n";
        return;
    }
    std::cout << "Safe path: " << x << "\n";
}

int main()
{
    process(42, fast_tag{});
```

```
    process(-1, safe_tag{});

    return 0;
}
```

```
Fast path: 42
Error: negative
```

Tag dispatch was more common before C++20. Today, `if constexpr` and concepts (Chapter 2) often achieve the same result more cleanly.

Code Review Checklist

When reviewing your own or others' C++ code, check for:

Correctness

- ☐ Does the code handle all edge cases?
- ☐ Are there potential off-by-one errors?
- ☐ Is error handling complete (exceptions, error codes)?

Resource Management

- ☐ Are all resources managed by RAII (Chapter 9)?
- ☐ Is the Rule of Zero or Rule of Five followed?
- ☐ Are smart pointers used instead of raw `new`/`delete`?

Safety

- ☐ Is `const` used wherever possible?
- ☐ Are `override` and `final` used on virtual functions?
- ☐ Are data races avoided with mutexes or atomics (Chapter 10)?

Performance

- ☐ Are large objects passed by `const&` or moved?
- ☐ Are unnecessary copies avoided?
- ☐ Is `reserve()` called on vectors when the size is known?

Style

- ☐ Is naming consistent with the project convention?
- ☐ Are comments explaining *why*, not *what*?
- ☐ Is the code simple and readable?

What's Next: C++26 Preview

C++ continues to evolve. Here are some features expected in C++26 and beyond:

`std::execution` (Senders/Receivers)

A structured framework for asynchronous and parallel programming, replacing ad-hoc thread management with composable operations.

Pattern Matching (post-C++26)

A cleaner alternative to `if`/`else` chains and `std::visit`, still under active development:

```
// Proposed syntax (not final)
inspect (value) {
    0 => std::cout << "zero\n";
```

```
    int i if i > 0 => std::cout << "positive\n";
    _ => std::cout << "other\n";
};
```

Reflection

The ability to inspect types at compile time — querying member names, types, and attributes programmatically. This will enable automatic serialization, ORM generation, and much more.

Contracts

Preconditions and postconditions as part of function declarations:

```
int sqrt_of(int x)
    pre (x >= 0)
    post (r: r >= 0)
{
    // ...
}
```

Tip: C++ evolves roughly every three years. Keeping up with the latest standards ensures you can write cleaner, safer code. Follow the ISO C++ committee's progress and experiment with new features as your compiler supports them.

Key Points

- **Naming conventions** vary; consistency within a project matters most.
- **Const-correctness**: make everything `const` by default.
- **Rule of Zero**: if your class does not manage resources directly, do not write special member functions.
- **Rule of Five**: if you manage a resource directly, define all five special members.
- Use `auto` when the type is clear or verbose, not when it obscures meaning.
- **PIMPL** hides implementation details behind a pointer, reducing compile dependencies and build times. The destructor must be in the `.cpp` file.
- **CRTP** provides static polymorphism with no vtable overhead, but does not support run-time polymorphism.
- **Tag dispatch** selects overloads at compile time; largely superseded by `if constexpr` and concepts.
- A good code review checks correctness, resource management, safety, performance, and style.
- **C++26** will bring `std::execution`, pattern matching, reflection, and contracts.

Exercises

1. **Think about it:** Why does the text recommend starting with `const` and removing it only when needed, rather than the other way around?
2. **Think about it:** When should you follow the Rule of Zero vs. the Rule of Five? How do you decide?
3. **Where is the bug?**

   ```
   class Data {
   public:
       Data(int n) : ptr_(new int[n]), size_(n) {}
       ~Data() { delete[] ptr_; }
   private:
       int* ptr_;
       int size_;
   };
   ```

```
Data a(10);
Data b = a;
```

4. **Think about it:** What are the trade-offs between CRTP (static polymorphism) and virtual functions (dynamic polymorphism)? When would you choose each?

5. **What does this print?**

```
auto x = 42;
auto y = 3.14;
auto z = x + y;
std::cout << z << "\n";
```

6. **Think about it:** Why must the PIMPL destructor be defined in the `.cpp` file? What error do you get if you let the compiler generate it in the header?

7. **Where is the problem?**

```
void add_to_vector(std::vector<int> v, int x)
{
    v.push_back(x);
}

std::vector<int> data = {1, 2, 3};
add_to_vector(data, 4);
std::cout << data.size() << "\n";
```

8. **Think about it:** The code review checklist mentions checking that comments explain "why" not "what." Why is this distinction important? Give an example of a bad comment and a good one for the same code.

9. **Calculation:** A class has a `std::vector<std::string>` member and a `std::unique_ptr<Widget>` member. How many special member functions should you write for this class?

10. **Write a program** that uses the CRTP to create a `Printable<Derived>` base class with a `print()` method that calls `Derived::to_string()`. Create two derived classes (e.g., `Song` and `Album`) that each implement `to_string()`. Demonstrate calling `print()` on instances of each.

Appendix A: Build Systems and Tooling

Throughout this book you have compiled programs with a single `g++` command. That works for small programs, but real projects have dozens or hundreds of source files, external dependencies, and platform-specific requirements. A **build system** automates compilation so you do not have to type long commands or remember which files changed. This appendix covers CMake (the most widely used C++ build system), compiler flags, sanitizers, static analysis, and basic debugging.

CMake Basics

CMake is a **build system generator** — it reads a `CMakeLists.txt` file and generates the actual build files (Makefiles on Linux/macOS, Visual Studio projects on Windows).

A Minimal Project

Create a directory with two files:

```
my_project/
  CMakeLists.txt
  main.cpp
```

`CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.20)
project(MyProject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(myapp main.cpp)
```

`main.cpp`:

```
#include <iostream>

int main()
{
    std::cout << "Built with CMake!\n";
    return 0;
}
```

Build it:

```
mkdir build && cd build
cmake ..
make
./myapp
```

Multiple Source Files

```
add_executable(myapp
    main.cpp
    playlist.cpp
    audio.cpp
)
```

Libraries

```
# Create a library
add_library(audio audio.cpp codec.cpp)

# Link it to the executable
add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE audio)
```

`PRIVATE` means `audio` is only needed by `myapp`, not by anything that uses `myapp`. Use `PUBLIC` if the dependency is also needed by consumers of the target.

Compiler Warnings

```
target_compile_options(myapp PRIVATE -Wall -Wextra -pedantic)
```

Including Headers

```
target_include_directories(myapp PRIVATE ${CMAKE_SOURCE_DIR}/include)
```

External Dependencies with `find_package`

```
find_package(Threads REQUIRED)
target_link_libraries(myapp PRIVATE Threads::Threads)
```

For popular libraries (Boost, OpenSSL, etc.), CMake provides built-in `Find` modules. For others, use `FetchContent` to download them:

```
include(FetchContent)
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 10.2.1
)
FetchContent_MakeAvailable(fmt)
target_link_libraries(myapp PRIVATE fmt::fmt)
```

Tip: CMake has a steep learning curve, but it is the de facto standard for C++ projects. Start with the basics above and learn more as your projects grow.

Compiler Flags and Warnings

The compiler flags you choose affect correctness, performance, and debuggability.

Warning Flags

Always compile with warnings enabled:

```
g++ -Wall -Wextra -pedantic -Werror main.cpp
```

Flag	Effect
`-Wall`	Enable most common warnings
`-Wextra`	Enable additional warnings
`-pedantic`	Warn about non-standard extensions
`-Werror`	Treat warnings as errors

Tip: Use `-Werror` in CI/CD pipelines to prevent warnings from accumulating. In development, you may want warnings without the hard stop.

Standard Selection

```
g++ -std=c++23 main.cpp   # C++23
g++ -std=c++20 main.cpp   # C++20
g++ -std=c++17 main.cpp   # C++17
```

Optimization Levels

Flag	Effect
`-O0`	No optimization (fastest compile, best debugging)
`-O1`	Basic optimization
`-O2`	Standard optimization (good for release)
`-O3`	Aggressive optimization
`-Os`	Optimize for size
`-Og`	Optimize for debugging

Debug Information

```
g++ -g main.cpp           # Include debug symbols
g++ -g -O0 main.cpp       # Debug build (best for debuggers)
g++ -O2 -DNDEBUG main.cpp # Release build (disables assert)
```

Sanitizers

Sanitizers are compiler features that instrument your code to detect bugs at run time. They add overhead but catch problems that are otherwise invisible.

AddressSanitizer (ASan)

Detects memory errors: buffer overflows, use-after-free, double-free, memory leaks:

```
g++ -fsanitize=address -g main.cpp -o main
./main
```

If your program has a memory bug, ASan prints a detailed error report with the exact location.

UndefinedBehaviorSanitizer (UBSan)

Detects undefined behavior: signed integer overflow, null pointer dereference, misaligned access:

```
g++ -fsanitize=undefined -g main.cpp -o main
```

ThreadSanitizer (TSan)

Detects data races in multithreaded programs (Chapter 10):

```
g++ -fsanitize=thread -g main.cpp -o main -pthread
```

Tip: Run your tests with sanitizers regularly. Many bugs — especially memory and threading bugs — are silent until they corrupt data or crash under production load. Sanitizers catch them early.

Combining Sanitizers

You can combine ASan and UBSan:

```
g++ -fsanitize=address,undefined -g main.cpp
```

But ASan and TSan cannot be used together — they instrument memory differently.

Static Analysis

Static analysis examines your code without running it, catching bugs that the compiler's warnings miss.

clang-tidy

`clang-tidy` is the most popular C++ linter. It checks for common mistakes, style issues, and modernization opportunities:

```
clang-tidy main.cpp -- -std=c++23
```

Useful check categories:

Category	What it checks
`bugprone-*`	Common bug patterns
`modernize-*`	Suggest modern C++ replacements
`performance-*`	Performance issues
`readability-*`	Code readability
`cppcoreguidelines-*`	C++ Core Guidelines compliance

cppcheck

`cppcheck` is a standalone static analyzer:

```
cppcheck --enable=all main.cpp
```

It catches issues like unused variables, null pointer dereferences, and resource leaks.

Compiler Warnings as Analysis

With `-Wall -Wextra -pedantic -Werror`, the compiler itself is a basic static analyzer. Start there before adding external tools.

Debugging with gdb/lldb

When your program crashes or produces wrong results, a debugger lets you step through the code line by line, inspect variables, and examine the call stack.

Basic gdb Commands

```
g++ -g -O0 main.cpp -o main
gdb ./main
```

Command	Effect
`run`	Start the program
`break main`	Set a breakpoint at `main`
`break file.cpp:42`	Breakpoint at line 42
`next`	Execute next line (step over)
`step`	Step into function call
`continue`	Run until next breakpoint
`print x`	Print the value of `x`
`backtrace`	Show the call stack
`info locals`	Show local variables
`quit`	Exit gdb

lldb

`lldb` is the LLVM debugger, used primarily on macOS. Its commands are similar:

gdb	lldb
`run`	`run`
`break main`	`breakpoint set --name main`
`next`	`next`
`step`	`step`
`print x`	`frame variable x` or `p x`
`backtrace`	`thread backtrace`

Debugging Tips

- Compile with `-g -O0` for the best debugging experience. Optimizations can reorder code and eliminate variables.
- Use `valgrind` as an alternative to ASan for memory debugging: `valgrind ./main`
- Core dumps: if a program crashes, the OS can save a core dump. Load it with `gdb ./main core` to examine the state at the time of the crash.

Tip: Learn to use a debugger early. `std::cout` debugging is tempting but slow and unreliable. A debugger shows you exactly what is happening, where, and why.

Key Points

- **CMake** is the standard C++ build system. `CMakeLists.txt` defines targets, sources, and dependencies.
- Use `add_executable` for programs, `add_library` for libraries, and `target_link_libraries` to connect them.
- **Compiler flags**: `-Wall -Wextra -pedantic` for warnings, `-std=c++23` for the standard, `-O2` for release, `-g -O0` for debug.
- **Sanitizers** catch runtime bugs: ASan (memory), UBSan (undefined behavior), TSan (data races). Use them in testing.
- **Static analysis** tools like `clang-tidy` and `cppcheck` catch bugs without running the code.
- **gdb/lldb** let you step through code, set breakpoints, and inspect variables. Compile with `-g -O0` for best results.

Exercises

1. **Think about it:** Why is CMake called a "build system generator" rather than a "build system"? What does it generate?
2. **Write a CMakeLists.txt** for a project with `main.cpp`, `audio.cpp`, and `audio.h`. Set the C++ standard to 23 and enable `-Wall -Wextra -pedantic`.
3. **Think about it:** Why should you compile with `-Wall -Wextra -pedantic` from the start of a project rather than adding them later?
4. **Calculation:** You have a program with a buffer overflow that only corrupts memory silently. Which sanitizer would catch it? What compiler flag would you use?
5. **Think about it:** AddressSanitizer and ThreadSanitizer cannot be used together. Why might that be? How would you test for both memory and threading bugs?
6. **Write a gdb session** (sequence of commands) that:
   - Sets a breakpoint at `main`
   - Runs the program
   - Steps through three lines
   - Prints a local variable called `count`
   - Continues to the end
7. **Think about it:** What is the difference between `-O0`, `-O2`, and `-O3`? When would you use each?
8. **Where is the problem?**

   ```
   add_executable(myapp main.cpp)
   target_link_libraries(myapp fmt)
   ```

   What is missing compared to the example in this chapter?
9. **Think about it:** Why does the text recommend running tests with sanitizers enabled? What kinds of bugs do sanitizers catch that tests alone miss?
10. **Set up a project** with CMake that has a `main.cpp` and a `math_utils.cpp/math_utils.h` library. The library should have a function `int factorial(int n)`. Build it with CMake, run it, and then compile with AddressSanitizer enabled and verify it runs cleanly.

Appendix B: Testing

How do you know your code works? You run it and check the output. But as your program grows, manually checking every feature after every change becomes impossible. **Automated tests** solve this: small programs that exercise your code and verify the results. A good test suite catches bugs before your users do and gives you confidence to refactor without fear. This appendix covers unit testing concepts, two popular frameworks (Google Test and Catch2), test-driven development, and mocking.

Unit Testing Concepts

A **unit test** tests one small piece of code — a function, a class method, or a small module — in isolation.

Arrange, Act, Assert

Most unit tests follow the **Arrange-Act-Assert** pattern:

1. **Arrange** — set up the test data and dependencies.
2. **Act** — call the function or method under test.
3. **Assert** — check that the result matches what you expect.

```
// Arrange
std::vector<int> scores = {90, 85, 92, 88};

// Act
int sum = std::accumulate(scores.begin(), scores.end(), 0);

// Assert
assert(sum == 355);
```

What Makes a Good Test

- **Focused**: tests one thing. If it fails, you know exactly what broke.
- **Independent**: does not depend on other tests or shared state.
- **Fast**: runs in milliseconds, not seconds.
- **Deterministic**: produces the same result every time.

Google Test

Google Test (also called gtest) is the most widely used C++ testing framework.

Setting Up

With CMake (Appendix A):

```
include(FetchContent)
FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG v1.14.0
)
FetchContent_MakeAvailable(googletest)

add_executable(tests test_math.cpp)
target_link_libraries(tests PRIVATE gtest_main)
```

`gtest_main` provides a `main()` function, so your test file only needs test cases.

Writing Tests

```
#include <gtest/gtest.h>

int add(int a, int b) { return a + b; }

TEST(AddTest, PositiveNumbers)
{
    EXPECT_EQ(add(2, 3), 5);
}

TEST(AddTest, NegativeNumbers)
{
    EXPECT_EQ(add(-1, -2), -3);
}

TEST(AddTest, Zero)
{
    EXPECT_EQ(add(0, 0), 0);
    EXPECT_EQ(add(5, 0), 5);
}
```

`TEST(TestSuite, TestName)` defines a test case. The suite name groups related tests.

Assertions

Macro	Checks	On failure
`EXPECT_EQ(a, b)`	`a == b`	Continues
`EXPECT_NE(a, b)`	`a != b`	Continues
`EXPECT_LT(a, b)`	`a < b`	Continues
`EXPECT_GT(a, b)`	`a > b`	Continues
`EXPECT_TRUE(cond)`	`cond` is true	Continues
`EXPECT_FALSE(cond)`	`cond` is false	Continues
`EXPECT_THROW(expr, type)`	`expr` throws `type`	Continues
`ASSERT_EQ(a, b)`	`a == b`	Stops test

`EXPECT_*` macros report failures but continue the test. `ASSERT_*` macros abort the test immediately — use them when continuing would cause a crash (e.g., after a null check).

Fixtures

When multiple tests need the same setup, use a **fixture**:

```
class PlaylistTest : public ::testing::Test {
protected:
    void SetUp() override
    {
        playlist.push_back("Hey Ya!");
        playlist.push_back("Toxic");
        playlist.push_back("Crazy");
    }

    std::vector<std::string> playlist;
};
```

```
TEST_F(PlaylistTest, HasThreeSongs)
{
    EXPECT_EQ(playlist.size(), 3);
}

TEST_F(PlaylistTest, ContainsToxic)
{
    auto it = std::find(playlist.begin(), playlist.end(), "Toxic");
    EXPECT_NE(it, playlist.end());
}
```

`TEST_F` uses the fixture class. `SetUp()` runs before each test; `TearDown()` (optional) runs after each test. Each test gets a fresh instance — tests do not share state.

Running Tests

```
./tests                  # run all tests
./tests --gtest_filter="AddTest.*"  # run only AddTest suite
./tests --gtest_list_tests          # list all tests
```

Catch2

Catch2 is a header-friendly alternative to Google Test with a more concise syntax.

Setting Up

```
FetchContent_Declare(
    Catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.5.2
)
FetchContent_MakeAvailable(Catch2)

add_executable(tests test_math.cpp)
target_link_libraries(tests PRIVATE Catch2::Catch2WithMain)
```

Writing Tests

```
#include <catch2/catch_test_macros.hpp>

int add(int a, int b) { return a + b; }

TEST_CASE("add returns correct sums", "[add]")
{
    REQUIRE(add(2, 3) == 5);
    REQUIRE(add(-1, -2) == -3);
    REQUIRE(add(0, 0) == 0);
}
```

Sections

Catch2's **sections** replace fixtures for many use cases. Each section runs with a fresh state:

```
TEST_CASE("Playlist operations", "[playlist]")
{
```

```
    std::vector<std::string> playlist = {"Hey Ya!", "Toxic"};

    SECTION("adding a song increases size")
    {
        playlist.push_back("Crazy");
        REQUIRE(playlist.size() == 3);
    }

    SECTION("clearing removes all songs")
    {
        playlist.clear();
        REQUIRE(playlist.empty());
    }
}
```

Each `SECTION` runs independently — the `playlist` vector is reset between sections.

Assertions

Catch2	Behavior
`REQUIRE(expr)`	Fatal — stops test on failure
`CHECK(expr)`	Non-fatal — reports but continues
`REQUIRE_THROWS_AS(expr, type)`	Checks that `expr` throws `type`

Google Test vs. Catch2

Feature	Google Test	Catch2
Syntax	`TEST()`, `EXPECT_*`, `ASSERT_*`	`TEST_CASE`, `REQUIRE`, `CHECK`
Fixtures	Class-based (`TEST_F`)	Section-based
Setup	Requires linking	Header-friendly
Maturity	Industry standard	Popular, modern
Mocking	Built-in (Google Mock)	Separate libraries

Both are excellent choices. Google Test is more common in industry; Catch2 is often preferred for smaller projects.

Test-Driven Development

Test-Driven Development (TDD) flips the usual workflow: you write the test *first*, then write the code to make it pass.

The Red-Green-Refactor Cycle

1. **Red**: Write a test for a feature that does not exist yet. Run it — it fails (red).
2. **Green**: Write the simplest code that makes the test pass (green).
3. **Refactor**: Clean up the code while keeping the tests green.

Repeat for each new feature or behavior.

Example

Step 1 — Red:

```
TEST(Factorial, BaseCase)
{
    EXPECT_EQ(factorial(0), 1);
    EXPECT_EQ(factorial(1), 1);
}
```

This fails because `factorial` does not exist.

Step 2 — Green:

```
int factorial(int n)
{
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Tests pass.

Step 3 — Refactor: The code is already clean, so nothing to change. Add the next test:

```
TEST(Factorial, LargerValues)
{
    EXPECT_EQ(factorial(5), 120);
    EXPECT_EQ(factorial(10), 3628800);
}
```

When TDD Helps

- Well-defined requirements with clear inputs and outputs.
- Bug fixes: write a test that reproduces the bug, then fix the code.
- Library code where the API is designed upfront.

When TDD is Less Useful

- Exploratory or prototype code where requirements are unclear.
- UI code or systems with heavy external dependencies.
- Performance-sensitive code where the algorithm may change drastically.

Tip: Even if you do not follow strict TDD, writing tests alongside your code (rather than after) catches bugs earlier and keeps your code testable.

Mocking

A **mock** is a fake object that replaces a real dependency in a test. Mocking lets you test a class in isolation without needing a database, network, or filesystem.

Google Mock Basics

Google Mock comes with Google Test. It uses macros to create mock classes from interfaces (abstract classes, Chapter 1):

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
```

```cpp
// Interface
class Database {
public:
    virtual ~Database() = default;
    virtual std::string lookup(int id) = 0;
    virtual void save(int id, const std::string& data) = 0;
};

// Mock
class MockDatabase : public Database {
public:
    MOCK_METHOD(std::string, lookup, (int id), (override));
    MOCK_METHOD(void, save, (int id, const std::string& data), (override));
};
```

Using the Mock

```cpp
// The class under test
class MusicService {
public:
    MusicService(Database& db) : db_(db) {}

    std::string get_track_name(int id)
    {
        return db_.lookup(id);
    }

private:
    Database& db_;
};

// The test
TEST(MusicServiceTest, ReturnsTrackName)
{
    MockDatabase mock_db;
    EXPECT_CALL(mock_db, lookup(1))
        .WillOnce(::testing::Return("Bohemian Like You"));

    MusicService service(mock_db);
    EXPECT_EQ(service.get_track_name(1), "Bohemian Like You");
}
```

`EXPECT_CALL` sets an expectation: when `lookup(1)` is called, return `"Bohemian Like You"`. If the method is never called, or called with different arguments, the test fails.

Key Google Mock Features

Macro/Function	Purpose
`MOCK_METHOD(return, name, (args), (qualifiers))`	Define a mock method
`EXPECT_CALL(mock, method(args))`	Set an expectation
`.WillOnce(Return(value))`	Return a value once
`.WillRepeatedly(Return(value))`	Return a value every time

Macro/Function	Purpose
`.Times(n)`	Expect exactly n calls

Tip: Mocking works best when your code depends on **interfaces** (abstract classes) rather than concrete classes. This is another reason to use polymorphism (Chapter 1) in your designs.

Key Points

- **Unit tests** test small pieces of code in isolation using Arrange-Act-Assert.
- Good tests are focused, independent, fast, and deterministic.
- **Google Test**: `TEST()` for test cases, `EXPECT_*/ASSERT_*` for assertions, `TEST_F` for fixtures.
- **Catch2**: `TEST_CASE` for test cases, `REQUIRE/CHECK` for assertions, `SECTION` for shared setup.
- **TDD** (Test-Driven Development) follows the red-green-refactor cycle: write the test first, then the code.
- **Mocking** replaces real dependencies with fake objects for isolated testing. Google Mock provides `MOCK_METHOD` and `EXPECT_CALL`.
- Mocking works best with interfaces (abstract classes).
- Write tests alongside your code, not after. Test coverage gives you confidence to refactor.

Exercises

1. **Think about it:** Why should tests be independent of each other? What problems can arise when tests share state?
2. **Write a test** (using Google Test syntax) for a function `bool is_palindrome(const std::string& s)` that checks if a string reads the same forwards and backwards. Include tests for "racecar" (true), "hello" (false), and "" (true).
3. **Think about it:** What is the difference between `EXPECT_EQ` and `ASSERT_EQ` in Google Test? When would you choose one over the other?
4. **Where is the bug?**

   ```
   TEST_F(PlaylistTest, CanRemoveSong)
   {
       playlist.erase(playlist.begin());
       ASSERT_EQ(playlist.size(), 2);
       EXPECT_EQ(playlist[0], "Toxic");
   }
   ```

   (Hint: what if the `SetUp` adds songs in a different order than expected?)
5. **Think about it:** When is TDD most useful? When is it less useful? Give an example of each.
6. **Write a Catch2 test** for a function `int clamp(int value, int lo, int hi)` that restricts a value to a range. Use sections to test: value below range, value in range, and value above range.
7. **Think about it:** Why does mocking require interfaces (abstract classes)? What happens if you try to mock a class with no virtual functions?
8. **What does this test check?**

   ```
   TEST(StringTest, EmptyString)
   {
       std::string s;
       EXPECT_TRUE(s.empty());
       EXPECT_EQ(s.size(), 0);
   ```

```
    EXPECT_EQ(s, "");
}
```

9. **Think about it:** The text says "write tests alongside your code, not after." Why is writing tests after the code is finished less effective?

10. **Write a mock** (using Google Mock syntax) for this interface and a test that uses it:

    ```
    class Logger {
    public:
        virtual ~Logger() = default;
        virtual void log(const std::string& message) = 0;
        virtual int message_count() const = 0;
    };
    ```

    Write a class `App` that takes a `Logger&` and has a `run()` method that calls `log("Starting")`. Test that `run()` calls `log` exactly once with the message "Starting".

Index