Gorgo Continuing C++

April 11, 2026

Contents

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>`.