Gorgo Continuing C++

April 11, 2026

Contents

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?**

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

8. **What does this print?**

```cpp
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.