A Mini Guide to Modern C++

From C++11 to C++20

Adam Abed Abud

June 7, 2025

"Modern C++: a difficult match between high performance and high-level abstraction."

Abstract

Modern C++ (from C++11 onward) introduced numerous features that greatly improve code expressiveness, safety, and performance compared to C++98/03. Below is a guide organized by C++ standard version (C++11, C++14, C++17, C++20), highlighting key new language features and idioms of each. It is designed for developers already comfortable with C++98/03 and aims to highlight the new standards and features of C++.

Contents

1 C++11: A New Era

C++11 was a major update—the first big revision since C++98—with many additions to standardize common practice and improve abstraction power. It significantly changed the way we write C++ by introducing features for type inference, range-based loops, lambdas, improved resource management, move semantics for performance, and more. Below are some of the most important C++11 features.

1.1 Auto Type Inference (`auto`)

C++11 allows automatic type deduction with `auto`. No more verbose iterator or type declarations!

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

int main() {
    // Old way (C++98): explicitly specify iterator type
    std::vector<std::string> names = {"Alice", "Bob", "Eve"};
    std::vector<std::string>::iterator it = names.begin();
    std::cout << *it << "\n";  // prints "Alice"

    // Modern way (C++11): use auto to deduce the iterator type
    auto it2 = names.begin();
    std::cout << *it2 << "\n"; // prints "Alice"

    // auto can deduce any type from initializer
    auto n = 42;             // n is int
    auto d = 3.14;           // d is double
    auto name = names[1];    // name is std::string (copy of "Bob")
}
```

1.2 Range-Based For Loops

C++11 introduced a new for loop syntax to iterate over ranges (like arrays, vectors, containers) more easily. The range-based for loop automatically iterates through each element of a range without needing an index or explicit iterator. It's a more readable equivalent to the traditional loop that iterates over a container's elements

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

```cpp

int main() {
    std::vector<int> nums = {10, 20, 30};

    // Old C++98 style: explicit index or iterator
    for (size_t i = 0; i < nums.size(); ++i) {
        std::cout << nums[i] << " ";
    }
    std::cout << "\n";

    // C++11 range-based for loop: simpler iteration
    for (int value : nums) {
        std::cout << value << " ";
    }
    std::cout << "\n";

    // You can also use references to avoid copies or modify elements
    for (int &value : nums) {
        value += 5;  // modify in place
    }
    for (const int &value : nums) {
        std::cout << value << " ";  // prints modified values: 15 25 35
    }
}

```

1.3 Lambda Expressions

C++11 added lambda expressions, which are essentially inline anonymous function objects (closures) that you can define and invoke on the fly. Lambdas provide a convenient way to pass custom logic to algorithms (like `std::sort`, `std::for_each`) without writing separate function objects or function definitions. They are particularly useful when used in combination with STL functions.

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

int main() {
    std::vector<int> data = {3, 1, 4, 1, 5, 9};

```

```
    // Sort in descending order using a lambda as the comparison
    std::sort(data.begin(), data.end(), [](int a, int b) {
        return a > b;  // lambda returns true if a should come before b
    });

    // Print sorted result
    for (int x : data) {
        std::cout << x << " ";  // Output: 9 5 4 3 1 1
    }
}
```

1.4 Rvalue References & Move Semantics

One of the most impactful C++11 features for performance is rvalue references (&&) and move semantics. These address the inefficiency of unnecessary copies. An rvalue reference is a reference that can bind to temporary objects (rvalues), which C++98 could not do (except with const references). Using rvalue references, C++11 introduced move constructors and move assignment operators that transfer resources (like heap memory) from one object to another instead of copying them. Move semantics allow objects to be moved cheaply, avoiding expensive deep copies when the original object is a temporary.

In practice, move semantics mean you can std::move an object to explicitly indicate you want to transfer its contents elsewhere. After moving, the source object is left in a valid but unspecified state (often empty). This is especially beneficial for classes managing resources (e.g., vectors, strings, file handles) because it can eliminate needless allocations and copies.

```
#include <iostream>
#include <string>
#include <utility>  // for std::move

int main() {
    std::string a = "This is a large string.";
    std::string b;

    // Copy semantics (C++98): b gets a copy of a (expensive for large data)
    b = a;
    std::cout << "After copy, a = \"" << a << "\"\n";

    // Move semantics (C++11): b takes ownership of a's data (no copy of
    ↪ contents)
    b = std::move(a);
    std::cout << "After move, a = \"" << a << "\"\n";
```

```
16      std::cout << "After move, b = \"" << b << "\"\n";
17  }
```

Output:

```
After copy, a = "This is a large string."
After move, a = ""
After move, b = "This is a large string."
```

In the copy, **a** remained unchanged and **b** got a duplicate. In the move, b took over the data from a, and a was left empty (since its content was moved out). No costly copy of the buffer occurred. Move semantics thus "avoid unnecessary copies of temporary objects" and can significantly improve performance, especially in containers like std::vector when resizing or returning large objects from functions. To enable moves, classes can define a move constructor/assignment (or use =default to let the compiler generate them if the class has no custom copy logic). Standard library containers and types adopted move semantics in C++11, so they automatically use moves in many situations (e.g., `v.push_back(std::move(obj))` will move `obj` into the vector).

1.5 Smart Pointers (`std::unique_ptr` and `std::shared_ptr`)

C++11 introduced smart pointer types `unique_ptr` and `std::shared_ptr` (in $< memory >$) to improve memory management and replace raw pointers for owning memory. `unique_ptr<T>` represents exclusive ownership of a heap-allocated object (at most one `unique_ptr` can own a given object), while `std::shared_ptr<T>` is a reference-counting pointer that allows shared ownership (multiple pointers to the same object, which is deleted when the last reference goes away). These smart pointers automatically delete the managed object in their destructor, following RAII principles, which helps prevent memory leaks and makes exception-safe code easier.

In modern C++, using raw new and delete is strongly discouraged; instead, you allocate with a smart pointer (or use factory functions like `std::make_unique`, shown later) and let it manage the lifetime. Smart pointers also integrate with move semantics (`unique_ptr` can be moved but not copied).

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

struct Song {
    std::string title;
    Song(std::string t) : title(std::move(t)) {
        std::cout << "Song \"" << title << "\" created.\n";
    }
```

```
    ~Song() {
        std::cout << "Song \"" << title << "\" destroyed.\n";
    }
};

int main() {
    // Using a raw pointer (C++98 style):
    Song* songPtr = new Song("Imagine");
    // ... use songPtr ...
    delete songPtr;  // must manually delete to avoid leak

    // Using unique_ptr (C++11 RAII style):
    std::unique_ptr<Song> songPtr2(new Song("Let It Be"));
    // ... use songPtr2 ...
}   // songPtr2 goes out of scope here, automatically deleting the Song
```

Output:

```
Song "Imagine" created.
Song "Imagine" destroyed.
Song "Let It Be" created.
Song "Let It Be" destroyed.
```

In this example, songPtr2 is a `std::unique_ptr`$< Song >$ that owns the Song object. When songPtr2 exits scope, its destructor deletes the Song object automatically, so we don't need to call delete. This automatic cleanup is essential for exception safety: even if an exception occurs, the `unique_ptr` will clean up the resource. By contrast, with the raw pointer songPtr, if an exception had occurred before the delete, the program would leak memory. `std::shared_ptr` works similarly but allows multiple pointers to share the same object (and uses reference counting to delete when appropriate). Prefer `unique_ptr` when you don't need shared ownership. Smart pointers make memory management safer and easier, embracing the RAII idiom ("resource acquisition is initialization").

1.6 Standard Threads (`std::thread`)

C++11 introduced `std::thread` to provide a standard way to launch and manage threads. Previously, developers relied on platform-specific APIs or third-party libraries for multithreading. With `std::thread`, you can easily create threads that execute functions, lambdas, or function objects.

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

```
3
4  void hello() {
5      std::cout << "Hello from thread!\n";
6  }
7
8  int main() {
9      std::thread t(hello);    // Start a thread running the 'hello' function
10     t.join();                // Wait for the thread to finish
11     std::cout << "Back in main.\n";
12 }
```

You can also use lambdas with `std::thread`:

```
1  std::thread t([] {
2      std::cout << "Hello from lambda thread!\n";
3  });
4  t.join();
```

`std::thread::join()` waits for the thread to finish. If you forget to call `join()` or `detach()` on a thread before the thread object is destroyed, the program will terminate.

1.7 Compile-Time Constants and Functions (`constexpr`)

C++11 introduced the constexpr keyword, allowing certain variables and functions to be evaluated at compile time. A constexpr variable is effectively a constant expression (like an enum or #define in old C++, but with type safety), and a constexpr function can produce compile-time results if called with constant inputs. This enables performing computations at compile time (for efficiency or to use results in contexts that require compile-time constants, like array sizes or template parameters). In C++11, constexpr functions had strict limitations (only a single return statement, etc.), but even this allowed things like computing Fibonacci or factorial at compile time. C++14 and C++20 further relaxed and extended constexpr (allowing loops, multiple statements, and even heap allocation by C++20).

```
1  #include <iostream>
2  constexpr int factorial(int n) {
3      // Compute factorial at compile time (C++14 allows loops in constexpr)
4      int result = 1;
5      for(int i = 1; i <= n; ++i) {
6          result *= i;
7      }
8      return result;
```

```
}

int main() {
    constexpr int val = factorial(5);           // computed at compile time
    std::array<int, factorial(3)> arr;          // factorial(3)=6, array of
    ↪ length 6
    std::cout << "5! = " << val << "\n";        // prints "5! = 120"
    std::cout << "Size of arr = " << arr.size();
}
```

In this snippet, factorial is declared constexpr, so calls with constant arguments (like factorial(5)) are evaluated by the compiler at compile time. We even use factorial(3) in an array bound, something not possible with a regular function. In C++11, the factorial function would have to be written recursively or as a single return expression (because C++11 constexpr functions couldn't have loops), but C++14 relaxed those rules, allowing the loop shown above. constexpr is very useful for meta-programming and for performance (compute once at compile time, reuse at runtime without cost).

1.8 A Type-Safe Null Pointer (`nullptr`)

C++11 introduced nullptr as a new keyword to represent a null pointer constant (replacing the traditional NULL macro or literal 0). Unlike NULL (which is typically defined as 0), nullptr is of type std::nullptr_t and can only be converted to pointer types, making it type-safe. This prevents ambiguous situations (e.g., choosing between overloaded functions or constructors that accept integers vs pointers) and makes code intent clearer.

```
#include <iostream>

void foo(int x) { std::cout << "foo(int)\n"; }
void foo(void* p) { std::cout << "foo(void*)\n"; }

int main() {
    foo(0);        // calls foo(int) { 0 is int, might not be intended
    // foo(NULL);  // would call foo(int) as NULL is 0 (int), potentially
    ↪ surprising
    foo(nullptr);  // calls foo(void*) { nullptr is a null pointer, calls
    ↪ pointer overload
}
```

1.9 Strongly-Typed Enums (`enum class`)

C++98 enums were plain enums that would implicitly convert to int and could cause name conflicts (all enumerators lived in the enclosing scope). C++11 introduced scoped enums via the enum class syntax. These are strongly-typed and scoped: the enumerators do not implicitly convert to integers, and they are accessed with their scope. This makes enums safer and better encapsulated.

```cpp
#include <iostream>

// C++98 style enum
enum Color { RED, GREEN, BLUE };       // RED, GREEN, BLUE are in global scope, implicitly int

// C++11 enum class
enum class Animal { Dog, Cat, Lion };  // Animal::Dog, Animal::Cat, Animal::Lion are scoped and strongly typed

int main() {
    Color c = RED;
    int n = BLUE;             // allowed in old enums (BLUE converts to int 2)
    // Animal a = Lion;      // ERROR: Lion is not in this scope (must use Animal::Lion)
    Animal a = Animal::Lion; // OK
    // int x = Animal::Dog; // ERROR: no implicit conversion from Animal to int
    if (a == Animal::Lion) {
        std::cout << "It's a lion!\n";
    }
}
```

1.10 Compile-Time Assertions (`static_assert`)

static_assert is a compile-time assertion introduced in C++11. It allows you to check conditions at compile time and produce a compilation error with a message if the condition is false. This is useful for validating template parameters, constants, or platform-specific assumptions (like type sizes) during compilation rather than at runtime.

```cpp
#include <type_traits>

template<typename T>
void processIntegralType(T value) {
    static_assert(std::is_integral_v<T>, "T must be an integral type");
```

```
    // Function implementation assumes T is integral...
}

int main() {
    processIntegralType(42);        // OK, int is integral

    // processIntegralType(3.14);   // ERROR at compile time: static
    ↪  assertion failed
}
```

Here, static_assert(std::is_integral_v¡T¿, "T must be an integral type"); checks at compile time that the template type T is an integral type. If we try to call processIntegralType with a non-integral type (like double), the static assertion fails and the compiler emits the provided error message. In C++98, similar checks had to be done with tricky template techniques or runtime asserts. static_assert makes intent clear and fails early during compilation if conditions are not met.

C++11 introduced many other features as well (such as override/final for virtual functions, variadic templates for templates with variable number of arguments, new std::thread library for concurrency, noexcept specifier, std::initializer_list, etc.), but the above features are among the most impactful in day-to-day C++ development.

2 C++14: Small Enhancements

C++14 was a smaller update (a follow-on to C++11) that added some convenient language improvements and relaxed certain rules, making C++11 features easier to use. Key C++14 features include generic lambdas, lambda capture initializers, return type deduction for normal functions, constexpr enhancements, and some new standard library utilities. Here are important C++14 additions.

2.1 Generic Lambdas

While C++11 lambdas required specifying the type of each parameter, C++14 allows auto in lambda parameter lists. Generic lambdas make the lambda a template, deducing the parameter types at call sites. This is especially useful for writing function objects that work with any type, similar to templates, but with lambda convenience.

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

int main() {
    // A generic lambda that adds two elements of any type
```

```
    auto add = [](auto x, auto y) {
        return x + y;
    };

    std::cout << add(2, 3) << "\n";          // x and y deduced as int,
    ↪ outputs 5
    std::cout << add(2.5, 3.1) << "\n";      // x and y deduced as double,
    ↪ outputs 5.6

    // Using generic lambda in an algorithm (e.g., transform to double each
    ↪ element)
    std::vector<int> v = {1, 2, 3};
    std::vector<int> result;
    result.resize(v.size());
    std::transform(v.begin(), v.end(), result.begin(), [](auto n) { return n
    ↪ * 2; });
    // result now contains {2, 4, 6}
}
```

In this example, the lambda [](auto x, auto y) return x + y; can add two integers or two doubles (or any types supporting +). The compiler generates the appropriate function for each call. This avoids writing overloaded lambdas or specifying templates explicitly – the lambda itself is polymorphic. C++14's generic lambdas thus provide a succinct way to create function templates on the fly.

2.2 Lambda Capture Initializers

C++14 also improved lambdas by allowing initialized captures (sometimes called lambda capture by value with initialization or generalized lambda capture). This lets you capture an expression or move-only object by value directly within the capture clause. In C++11, you could capture this or variables by value or reference, but you couldn't capture the result of an expression or a move-only object (like std::unique_ptr) without first storing it in a named variable. C++14's capture init syntax solves that by allowing forms like [name = expr] in the capture list.

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

int main() {
    std::unique_ptr<int> ptr(new int(42));

    // C++11 needed a workaround to capture move-only ptr (e.g., make it
    ↪ global or use reference wrapper)
```

```
    // C++14: capture ptr by moving it into the lambda
    auto lam = [p = std::move(ptr)]() {
        // inside lambda, use p as a unique_ptr<int>
        if (p) {
            std::cout << "Value = " << *p << "\n";
        }
    };

    // ptr is now null (ownership moved into lambda's p)
    lam();  // prints "Value = 42"
    // lam();  // calling again would print "Value = 42" again (p still holds
    ↪  the value inside lambda)
}
```

In this code, the lambda capture `[p = std::move(ptr)]` creates a new data member p inside the lambda closure, initialized by moving ptr into it. This means the lambda takes ownership of the dynamic int. We couldn't do this in C++11 directly. Similarly, you could capture `[value = someFunction()]` to execute an expression and store its result in the lambda. This feature makes lambdas more powerful and convenient, especially with asynchronous code or callback-based code where you often need to capture move-only types or computed values.

2.3 Return Type Deduction

C++11 allowed auto in function definitions only in the context of trailing return type syntax (e.g., auto $func() -> int$). C++14 extends type deduction to normal function return types (not just lambdas). You can declare a function with auto as its return type, without specifying $->$ ..., and the compiler will deduce the return type from the function's return statements. This works if all return statements yield the same type. It's especially handy for template code or when the return type is complicated or dependent on template parameters.

`auto` return type for regular functions:

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

// C++14 allows this: return type will be deduced.
auto combineVectors(const std::vector<int>& a, const std::vector<int>& b) {
    std::vector<int> result = a;
    result.insert(result.end(), b.begin(), b.end());
    return result;  // compiler deduces return type as std::vector<int>
}

```

```cpp
int main() {
    std::vector<int> v1 = {1, 2}, v2 = {3, 4};
    auto combined = combineVectors(v1, v2);
    // combined is deduced as std::vector<int>
    for(int x : combined) std::cout << x << " ";  // Output: 1 2 3 4
}
```

2.4 Binary Literals and Digit Separators

C++14 introduced minor but helpful improvements for numeric literals:

- Binary literals: You can write integers in binary using the prefix $0b$ or $0B$. For example, $0b1011$ represents the binary number 1011 (which is 11 in decimal). In C++98/11, you would have to use octal or hexadecimal or manually convert from binary strings.
- Digit separators: You can use the single quote ' as a digit separator in numeric literals to improve readability. It's ignored by the compiler, but helps humans read long literals (for example, 1'000'000 is one million). This works in integer and floating literals.

```cpp
#include <iostream>
int main() {
    int mask = 0b1100'1101;       // binary literal for 0xCD (205 in decimal)
    int bigNumber = 1'000'000;    // 1000000, easier to read with separators
    double avogadro = 6.022'140'76e23;  // Avogadro's number, using
    ↪  separators for readability

    std::cout << "mask = " << mask << "\n";           // prints 205
    std::cout << "bigNumber = " << bigNumber << "\n"; // prints 1000000
    std::cout << "Avogadro  " << avogadro << "\n";
}
```

2.5 Safer Object Creation (`std::make_unique`)

In C++11, we got `std::unique_ptr`, but a corresponding factory function `std::make_unique` was only added in C++14.

The `std::make_unique<T>(args...)` function allocates a new `T` with the given constructor arguments and returns a `std::unique_ptr<T>` to it.

It is not a core language feature, but an important library idiom. Using `make_unique` is recommended because it avoids certain potential issues (such as memory leaks if an exception is thrown in the middle of a `new` expression with multiple arguments) and it reduces typing by not needing to repeat the type on both sides of an assignment.

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

struct Widget {
    Widget(const std::string& name) : name(name) {
        std::cout << "Widget " << name << " constructed\n";
    }
    std::string name;
};

int main() {
    // C++11 style unique_ptr creation:
    std::unique_ptr<Widget> w1(new Widget("Alpha"));

    // C++14 style using make_unique:
    auto w2 = std::make_unique<Widget>("Beta");

    std::cout << "w1->name = " << w1->name << "\n";
    std::cout << "w2->name = " << w2->name << "\n";
}
```

Output

```
Widget Alpha constructed
Widget Beta constructed
w1->name = Alpha
w2->name = Beta
```

Both `w1` and `w2` are `unique_ptrs` owning a `Widget`, but `make_unique` makes the code shorter (`auto w2 = std::make_unique<Widget>("Beta")`) and more exception-safe.

`std::make_unique` was missing in C++11, and developers often wrote their own or used `std::shared_ptr<T> p(new T(...))` as a workaround, but now it is part of the standard. Always prefer `make_unique` to construct `unique_ptrs`.

3 C++17: More Expressive, More Powerful

C++17 continued the evolution with a mix of big and small features that cleaned up language edges and introduced new ways to decompose and conditionally compile code. Notable

C++17 features include structured bindings, selection statements with initializers, if constexpr for compile-time branching, inline variables, improvements to templates, and new library types like std::optional, std::variant, std::any, std::string_view, as well as the filesystem library. Here are the key C++17 features:

3.1 Structured Bindings

Structured bindings provide a convenient syntax to unpack tuples, pairs, structs, and other multi-member objects into separate variables. In C++98/11, one might use std::tie or manual unpacking, but with structured bindings you can declare multiple variables that directly bind to the elements of a tuple-like object. This greatly simplifies code that works with functions returning multiple values (via std::tuple or struct) or iterating maps, etc.

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

int main() {
    // Example 1: Unpacking a tuple
    std::tuple<int, std::string, double> personData(42, "Alice", 170.5);
    auto [id, name, height] = personData;
    std::cout << "ID=" << id << ", Name=" << name << ", Height=" << height <<
    ↪  "\n";

    // Example 2: Iterating a map with structured bindings (key, value)
    std::map<std::string, int> scores = { {"Bob", 10}, {"Alice", 20} };
    for (const auto& [player, score] : scores) {
        std::cout << player << " has score " << score << "\n";
    }
}
```

Output

```
ID=42, Name=Alice, Height=170.5
Bob has score 10
Alice has score 20
```

3.2 Compile-time Conditional (`if constexpr`)

C++17 introduced if constexpr which is a compile-time conditional inside templates. It allows you to conditionally compile code based on a constant expression (often a trait of a template

parameter). If the condition is false, the branch is discarded at compile time, so even if it contains code that would be ill-formed for some types, it won't cause a compilation error (similar to SFINAE techniques, but more straightforward). This is extremely useful for writing templated code that needs to do different things depending on type properties.

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

template<typename T>
void printNumberOrPointer(T value) {
    if constexpr (std::is_pointer_v<T>) {
        // This branch is compiled only if T is a pointer type
        std::cout << "Pointer value: " << value
                  << ", pointing to " << *value << "\n";
    } else {
        // This branch is compiled if T is not a pointer
        std::cout << "Non-pointer value: " << value << "\n";
    }
}

int main() {
    int x = 42;
    printNumberOrPointer(x);    // T deduced as int, non-pointer branch runs
    printNumberOrPointer(&x);   // T deduced as int*, pointer branch runs
}
```

Output:

```
Non-pointer value: 42
Pointer value: 0x7ffe... , pointing to 42
```

In printNumberOrPointer, we use `if constexpr` (`std::is_pointer_v`$< T >$). For $T = int$, the constexpr condition is false, so the else branch is compiled and the pointer branch is discarded (even though *value would be invalid for non-pointers, it's not compiled in that case). For $T = int*$, the condition is true, so the pointer branch is compiled. This feature enables cleaner template code without needing tricks like tag dispatch or partial specialization for simple cases. It significantly improves template metaprogramming readability.

3.3 Inline Variables

Prior to C++17, the inline keyword could not be applied to variables (only to functions), which made defining certain constants in header files cumbersome (the header would need an

extern declaration and a definition in a .cpp file to avoid ODR violations). C++17 introduced inline variables, allowing the definition of a variable in a header such that it's treated as if it has internal linkage (avoiding multiple definition errors). This is mainly useful for constants or singletons that you want to define in header-only libraries.

```cpp
// config.h (header file)
#pragma once
#include <string>
inline const int MaxConnections = 8;          // inline constant
inline std::string defaultName = "Guest";     // inline variable (non-const
↪  also allowed)

// main.cpp
#include <iostream>
#include "config.h"

int main() {
    std::cout << "MaxConnections=" << MaxConnections << "\n";
    defaultName = "Admin";
    std::cout << "Name=" << defaultName << "\n";
}
```

Here, MaxConnections and defaultName are defined as inline. You can include config.h in multiple source files and still have only one instance of these variables in the program. The inline variable tells the compiler that this is the same object across all inclusions (similar to how inline functions work). This feature is primarily for convenience in defining global constants or single-instance objects in header-only modules.

3.4 New std functions: `std::optional`, `std::variant`, `std::any`

C++17 introduced several new std types in the <optional>, <variant>, and <any> headers that are very useful for certain idioms:

- `std::optional`: Represents an optional value – either contains a T or is "empty". This is a type-safe way to indicate "nullable" or "optional" return values instead of using pointers or special values. It's useful for functions that might not return a value (e.g., a search that might fail) – in C++98 one might return a boolean and an output parameter or something, but now returning optional<T> is more expressive.
- `std::variant`: A type-safe union that can hold one of several types (from a fixed set). It's like a tagged union or an enum+union combination, replacing the need for boost::variant or manual unions with tag variables. You can retrieve the value with std::get or visit it with std::visit (providing a lambda for each type).
- `std::any`: A type-safe container for single values of any type. It can hold one value of any type (type-erased). You can query the stored type and attempt to cast it back.

It's useful for when you need a truly generic container for one value, replacing the need for boost::any.

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

std::optional<std::string> findKeyword(const std::string& text) {
    if (text.find("C++") != std::string::npos) {
        return "C++";       // found the keyword
    }
    return std::nullopt;    // std::nullopt indicates "no value"
}

int main() {
    auto result = findKeyword("I love C++17 features");
    if (result) {  // or result.has_value()
        std::cout << "Found keyword: " << *result << "\n";
    } else {
        std::cout << "Keyword not found\n";
    }
}
```

In `findKeyword`, we return `std::optional<std::string>` it either contains the found word "C++" or nothing (`std::nullopt`). The caller checks if (result) to see if a value was returned, and uses *result to access the string if present. In C++98, we might have returned an index or a boolean plus output parameter, but optional provides a clearer intention: the result might or might not be there. Similarly, variant and any (not shown here for brevity) open up new ways to design APIs: for instance, a variant¡int, std::string¿ function result could mean "it might return either an int or a string", and the caller must handle both. These types make C++ more expressive and less error-prone by providing standard solutions to common needs (nullable values, union types).

3.5 std::string_view

`std::string_view` (in <string_view>) is a new lightweight view into a string (added in C++17). It basically acts like a pointer to a segment of a string plus a length, and it doesn't own the string data. The benefit is that it allows functions to accept a string view to read from strings or character arrays without copying. In C++98, if you wanted a function to accept both std::string and C-string inputs efficiently, you might write overloads or accept $const char*$ and $const std::string\&$ separately. With string_view, a single function can accept a std::string_view, and you can pass it a std::string, a string literal, or a $char*$ + length easily. It's great for read-only string operations where you want to avoid unnecessary allocations.

```cpp
#include <string_view>
#include <iostream>
void printMessage(std::string_view message) {
    std::cout << "Message: " << message << "\n";
}

int main() {
    std::string hello = "Hello";
    const char* world = "World";
    printMessage(hello);                  // std::string can be passed
    printMessage("Goodbye string_view"); // string literal can be passed
    printMessage(std::string( world ));  // C-string or anything convertible
}
```

The printMessage function can take any string-like input without creating a new std::string internally. It simply views the passed data. One must be careful that the data pointed to remains valid for the duration of use (e.g., don't pass a string_view that points to a temporary that goes out of scope), but in many cases it leads to more efficient code by avoiding copies. std::string_view is often used for function parameters and for slicing strings.

3.6 Filesystem Library

C++17 integrates the filesystem library (previously experimental/Boost.Filesystem) into ¡filesystem¿. This library provides facilities to work with files and directories: path manipulation, directory iteration, file status (existence, size, permissions), and file operations (create directories, remove, rename, etc.). In C++98, one had to use platform-specific APIs or third-party libraries for these tasks. Now it's standardized.

```cpp
#include <filesystem>
#include <iostream>
namespace fs = std::filesystem;
int main() {
    fs::path filePath("test.txt");

    // Check if file exists
    if (fs::exists(filePath)) {
        std::cout << filePath << " exists, size = "
                  << fs::file_size(filePath) << " bytes\n";
    } else {
        std::cout << filePath << " does not exist\n";
    }

```

```
    // Create a directory
    fs::create_directory("example_dir");
    // Iterate over current directory
    for (const auto& entry : fs::directory_iterator(".")) {
        std::cout << "Found " << entry.path().filename() << "\n";
    }
}
```

C++17 had other improvements as well (inline if/switch as we saw, folding for parameter packs, template argument deduction for class templates so you can instantiate templates like `std::pair(1,2.5)` without specifying types, new attributes like `[[nodiscard]]`, and more), but the above features are among the most useful for day-to-day development.

4 C++20: New and modern features

C++20 is another significant milestone, often considered one of the largest updates since C++11. It introduced major features that further empower C++ developers: modules for better encapsulation and build times, concepts for template constraints, coroutines for asynchronous and lazily-evaluated code, the ranges library for more expressive collection processing, the three-way comparison (spaceship) operator to simplify comparisons, and several other improvements (like designated initializers, expanded constexpr capabilities, consteval and constinit, calendar/time library, etc.). Below are the key C++20 features and their usage.

4.1 Concepts and Template Constraints

Concepts introduce a way to specify constraints on template parameters, making templates safer and error messages clearer. In essence, a concept is a compile-time predicate (a set of requirements) that a type must satisfy. You can use concepts in template declarations (using the concept keyword to define one, or using library concepts from ¡concepts¿ like std::integral) to require that template arguments meet certain criteria (e.g., "this template type must have a + operator" or "must be an integral type"). This helps catch misuse of templates at compile time with friendly errors, and it documents the intended usage of a template directly in its interface.

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

// Use a standard library concept to constrain template (std::integral means
↪  integer types)
template <std::integral T>
T addNumbers(T a, T b) {
    return a + b;
}
```

```

template <typename T>
concept HasSize = requires(T x) { x.size(); };
// custom concept: type T must have a size() method

template <HasSize T>
void printSize(const T& container) {
    std::cout << "Size is " << container.size() << "\n";
}

int main() {
    std::cout << addNumbers(3, 4) << "\n";      // OK, int is integral
    ↪ (prints 7)
    // std::cout << addNumbers(2.5, 3.5) << "\n";  // Error: double not
    ↪ allowed (not integral)

    std::string s = "hello";
    printSize(s);   // OK, std::string has size()
    // printSize(42); // Error: int doesn't have size()
}
```

In addNumbers, we constrained the template with std::integral concept (from <concepts> library) to accept only integral types. If someone tries to call addNumbers with a non-integral type like double, they get a clear compile-time error about constraints not satisfied. We also defined a custom concept HasSize which uses a requires expression to ensure the type has a .size() member. The function template printSize then only accepts types satisfying HasSize. If you try to call it with an int, the error will say that int doesn't satisfy HasSize (because it lacks size()). Concepts thus allow template authors to explicitly state requirements, and the compiler checks them before instantiation. This results in cleaner template code (no need for SFINAE tricks) and better compile-time diagnostics.

4.2 Ranges Library

The C++20 ranges library (in <ranges>) reimagines how we work with collections and algorithms, making them more composable and declarative. Ranges allow you to create pipeline of operations on collections using range adapters (such as `views::filter`, `views::transform`) and to use algorithms in a way that integrates with these adapters. The ranges library adds a lot of expressive power to C++ by letting you write code that clearly expresses intent (like "filter this sequence to even numbers, then transform to squares, then iterate"). It also builds on concepts to ensure type safety of these compositions.

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

```cpp
#include <iostream>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

    // Use ranges to create a view of even squares
    auto evenSquares = numbers
        | std::views::filter([](int x) { return x % 2 == 0; })
        | std::views::transform([](int x) { return x * x; });

    // evenSquares is a lazily evaluated view: no new container created
    for (int value : evenSquares) {
        std::cout << value << " ";  // will print: 4 16 36 (squares of 2,4,6)
    }
    std::cout << "\n";
}
```

In this snippet, numbers `std::views::filter(...)` `std::views::transform(...)` creates a pipeline: first filter the numbers to even ones, then transform each to its square. The result evenSquares is a view – it doesn't store new integers, it will iterate through the original vector and apply the operations on the fly. The range-based for loop then prints the results. This kind of code is very expressive compared to equivalent manual loops in C++98. The ranges library also includes range versions of many ¡algorithm¿ functions (in <algorithm> or <ranges>), for example std::ranges::sort can directly sort a container (no need to pass begin/end). Overall, ranges allow "composable transformations on collections of data", greatly increasing expressive power

4.3 Coroutines

Coroutines in C++20 are a major feature enabling asynchronous and lazy computations. A coroutine is a function that can suspend and resume execution while maintaining its state. In C++20, a function becomes a coroutine if it uses co_await, co_yield, or co_return. Coroutines allow writing code in a synchronous style that actually executes asynchronously or incrementally (for example, to implement generators or async I/O)

- co_return value; returns a value (or completes the coroutine).
- co_yield value; produces a value and suspends (for generator-style coroutines).
- co_await; suspends until the awaited thing is ready (for async tasks).

Unlike regular functions, coroutines don't unwind the stack on suspension; instead, they preserve state in a heap-allocated coroutine frame and can resume later.

A full coroutine example requires understanding promise types and coroutine handles, which is advanced. However, we can illustrate a simple use-case: a generator coroutine that yields a sequence of values.

```cpp
#include <coroutine>
#include <iostream>

// Pseudo-code: a simple generator type for demonstration
template<typename T>
struct Generator {
    struct promise_type {
        T currentValue;
        Generator get_return_object() {
            return Generator{
            ↪  std::coroutine_handle<promise_type>::from_promise(*this) };
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(T value) {
            currentValue = value;
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> coro;
    Generator(std::coroutine_handle<promise_type> h) : coro(h) {}
    ~Generator() { if (coro) coro.destroy(); }

    // Get next value from the generator
    T next() {
        if (!coro.done()) {
            coro.resume();
            return coro.promise().currentValue;
        }
        throw std::out_of_range("Generator finished");
    }
};

// A coroutine function that yields an infinite arithmetic sequence
Generator<int> countBy(int start, int step) {
    int value = start;
    while (true) {
        co_yield value;
        value += step;
```

```
    }
}

int main() {
    auto seq = countBy(5, 5);              // sequence: 5, 10, 15, 20, ...
    std::cout << seq.next() << " ";        // 5
    std::cout << seq.next() << " ";        // 10
    std::cout << seq.next() << "\n";       // 15
}
```

In this conceptual example, countBy is a coroutine that yields an infinite sequence starting from start and incrementing by step. Each co_yield value; produces a value and suspends the coroutine. The Generator¡int¿ type is a simplified coroutine return type that manages the coroutine's state (the actual implementation in production code might be more complex or use a library/TS). In main, we get a generator and then call next() to get subsequent values. The output demonstrates that state is preserved between calls (it keeps counting).

Coroutines allow writing asynchronous code (like waiting for I/O) in a sequential fashion without blocking threads, or implementing lazy generators as shown. They are a powerful feature for advanced scenarios like event loops, pipelines, or infinite sequences, enabling code that "executes asynchronously (e.g. non-blocking I/O without explicit callbacks) and supports lazy-computed infinite sequences"

4.4 Modules

Modules are a huge change in C++20 that aims to replace the traditional header/include mechanism with a more robust, scalable system. A module encapsulates code (like functions, classes, etc.) and explicitly exports the parts that should be visible outside. Other translation units can import the module rather than including textual headers. Modules can significantly improve compile times (no more re-parsing the same headers in every file) and avoid issues like macro collisions or violations from multiple includes.

Key aspects of modules:

- Defined with module declarations in special module interface units (usually .ixx or .mpp files, or designated by compiler options).
- Use export to specify which definitions (functions, classes, variables, etc.) are visible to importers.
- Import modules in other files with the import ModuleName; syntax (instead of #include).
- Modules do not allow cyclic dependencies and have their own internal linkage rules for non-exported parts.

```
// math_utils.ixx (Module Interface)
export module math_utils;       // declare module name
export int add(int a, int b) {  // export a function
    return a + b;
}
// (could have non-exported/internal functions or includes here)

// main.cpp (Importing the module)
#include <iostream>
import math_utils;  // import the module (no .h file needed)

int main() {
    std::cout << "2 + 3 = " << add(2, 3) << "\n";
}
```

In this pseudo-code: math_utils.ixx is a module interface file declaring export module math_utils;. We export an add function. This module could be compiled to a binary module interface by the compiler. In `main.cpp`, we use import math_utils; to import that module and then call `add(2,3)` as if it were a normal function. No header files are included for add – the compiled module provides the necessary information.

The benefits are faster compilation (because the compiler processes each module once and can reuse the compiled interface) and better encapsulation (only exported symbols are visible, internal helper functions or includes won't leak out). For large projects, modules can be transformative by reducing compile times and increasing clarity of interfaces. Transitioning to modules from headers is a big change, but it's one of the directions C++ is moving towards for the long term.

4.5 Spaceship Operator (<=>)

The three-way comparison operator $'<=>'$, nicknamed the "spaceship operator", was introduced in C++20 to simplify and unify comparison logic. It automatically generates the conventional comparison operators ($==$, $!=$, $<$, $<=$, $>$, $>=$) if you default it, and provides a single operator that returns an ordering category (`std::strong_ordering`, `std::partial_ordering`, etc.) indicating less, equal, or greater. In earlier C++ versions, writing a class that supports all six comparison operators required a lot of boilerplate. With $<=>$, you can often just write one line.

```
#include <compare>
#include <iostream>

struct Point {
    int x, y;
```

```
    // Automatically generate all comparison operators (x then y
    ↪ lexicographical)
    auto operator<=>(const Point& other) const = default;
};

int main() {
    Point a{1, 5}, b{1, 10};
    if (a < b) {  // uses synthesized operator< from operator<=>
        std::cout << "Point a is less than b\n";
    }
    auto cmp = a <=> b;
    if (cmp < 0) {  // cmp is std::strong_ordering, can be compared to 0
        std::cout << "a < b (strong ordering)\n";
    }
}
```

Here, `Point::operator<=>` is defaulted, which means the compiler will compare a and b by their members in order. It will first compare x; if they differ, that decides the result, otherwise it compares y. By defaulting it, we also implicitly get operator $==$ (unless we explicitly define it differently). In main, we can use $a < b$ even though we didn't explicitly define $<$ – the compiler generated it. We can also use the result of $<=>$ directly: cmp will be a std::strong_ordering which can be checked for < 0, $== 0$, or > 0 to see if a is less, equal, or greater. The spaceship operator makes writing sortable types much easier and less error-prone. It also allows more efficient comparisons in some cases and supports different strengths of ordering (strong, weak, partial) for floating-point or custom orderings.

4.6 Designated Initializers

C++20 allows designated initializers for aggregates (structs, arrays, unions) similar to C99. This means you can specify the names of fields you want to initialize in curly brace initialization. This was not allowed in C++98/C++11 (where aggregate initialization had to be in order without skipping fields). With designated initializers, the code is clearer and you can omit some fields (they'll be value-initialized) or initialize out of order by naming the members.

```
#include <iostream>
struct Person {
    std::string name;
    int age;
    bool employed;
};

int main() {
    // C++20 designated initializers
```

```
    Person p1{ .name = "Alice", .age = 30, .employed = true };
    Person p2{ .age = 25, .name = "Bob" };  // employed will be false by
    ↪ default
    std::cout << p1.name << " is " << p1.age << "\n";
    std::cout << p2.name << " is " << p2.age << ", employed=" <<
    ↪ std::boolalpha << p2.employed << "\n";
}
```

Here we explicitly name name and age for p2 and omit employed, which will default to false (since Person is an aggregate, omitted fields are value-initialized). This feature can improve clarity for structs with many fields by labeling initializers, and it avoids mistakes with ordering.

4.7 Other C++20 features

Other notable C++20 Features:

- `consteval` and `constinit`: consteval can be used on a function to indicate it must be evaluated at compile time (a step beyond constexpr for immediate functions). `constinit` ensures a global or static variable is constant-initialized (catching if it isn't).
- Expanded constexpr: C++20 makes many standard library functions constexpr (even things like std::vector methods) and allows allocation in constexpr contexts, making compile-time computation far more powerful.
- Calendar and Time Zone library: ¡chrono¿ got calendar/timezone utilities (year, month, etc. types).
- `std::span`: A lightweight view similar to `string_view` but for arrays of any type (not owning, just pointer+size).
- Three-way comparison for primitives: The ¡=¿ operator is also defined for built-ins, and new comparison categories like std::strong_ordering indicate results of comparisons.

C++20 thus delivers a broad range of enhancements. From concepts (which revolutionize template programming by making constraints explicit) to coroutines (enabling a new style of async and generator code), to ranges (which add "huge amount of expressive power and flexibility" to how we work with collections), C++20 makes the language more powerful and expressive while also cleaner and easier to use in many respects.

Summary

Modern C++ (11/14/17/20) has introduced features that change how we write C++ compared to the C++98 era. auto and `decltype` reduce verbosity in code. Range-based loops and algorithms with lambdas make iteration and algorithms cleaner. Move semantics and smart pointers drastically improve performance and safety by handling resources better. `constexpr` and `consteval` enable more compile-time computation, while `static_assert` and concepts catch errors early at compile time with clear intent. New types like optional, variant, any,

`string_view`, and span extend the vocabulary for safer and more expressive code. `Modules` and `coroutines` are big modern features that pave the way for large-scale projects and asynchronous programming. Each C++ revision builds on the previous, so adopting these features will make your code more idiomatic, efficient, and easier to maintain in the long run, while still retaining C++'s power and flexibility.