C++26
THE NEXT STANDARD

constexpr
concept
requires
co_await
noexcept
alignas
static_assert
thread_local
contract_assert

C++26 KEYWORD ATLAS

// The Complete Field Guide to
Reserved Words in Modern C++

82 keywords • alternative tokens • special identifiers • compiler support

```C++
consteval auto answer() {
  return 42;
}
static_assert(answer() == 42);
// Modern C++
```

Prepared by
Ayman Alheraki

Under the patronage of
https://simplifycpp.org

C++98 → C++26 | quick reference booklet

C++26
THE NEXT STANDARD

C++26 KEYWORD ATLAS

// The Complete Field Guide to Reserved Words in Modern C++

This compact booklet is designed as a fast and practical guide for modern C++ programmers. It brings together the language vocabulary through C++26, groups the words by function, explains what each one means, and illustrates them with tiny examples and concise support notes.

Fast reference
Find what you need quickly and keep coding.

Functional grouping
Keywords organized by purpose for clear understanding.

Tiny examples
Short, focused examples that make sense fast.

Compiler notes
Platform-aware support and important caveats.

Scope of this guide: 82 core keywords, 11 alternative operator tokens, 6 special identifiers, and key reserved naming rules.

Prepared by
Ayman Alheraki

Under the patronage of
https://simplifycpp.org

Contents

A structured route through the C++ reserved vocabulary, modern feature families, and practical appendices.

FOUNDATIONS

QUICK REFERENCE

Fast navigation: use page 29 to jump by keyword family and page 31 for alternative operator tokens.

APPENDICES

Scope

82 unconditional keywords, 11 alternative operator representations, and 6 identifiers with special meaning through the current C++26 draft.

Prepared by **Ayman Alheraki** | Under the patronage of **https://simplifycpp.org**

C++26
THE NEXT STANDARD

Author's Preface

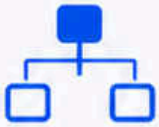
01 Modern C++ is rich, but its vocabulary can feel scattered across decades of evolution. A programmer meets long-standing structural keywords beside newer compile-time, coroutine, module, and contract vocabulary. This booklet was created to bring those words into one fast and practical reference.

02 Instead of presenting the vocabulary as one long, flat list, this guide groups words by function. That approach makes the language easier to browse, teach, remember, and compare.

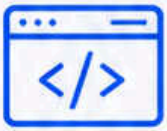
03 Each entry aims to answer a small set of useful questions quickly: What does this word mean? Since which standard does it matter? What does a tiny example look like? Does compiler support deserve a note?

04 Use this booklet as a desk reference, a revision guide, or a compact teaching companion. My hope is that it saves time for every programmer who works with modern C++.

fast reference



functional grouping



tiny examples

Prepared by
Ayman Alheraki

Under the patronage of
https://simplifycpp.org

How to Use This Booklet

A quick guide to get the most out of the atlas.

1 Read each keyword card quickly

1 Keyword name
KW
The word being explained.

2 Since
The standard in which it matters most.

3 Meaning
The concise functional definition.

4 Example
</>
A tiny illustrative code line.

5 Compiler note
Only when support history matters.

6 Warning or note
Useful caveats when needed.

2 Keep the distinctions clear

Keyword
KW
A reserved word in the language grammar.

Alternative token
&&
A word form of an operator such as `and` or `not`.

Special identifier
A context-sensitive identifier such as `override` or `module`.

Reserved name pattern
__X
Identifiers reserved to the implementation, such as `__name` or `_Name`.

Best reading path:
start with the functional chapters,
then use the alphabetical index for fast lookup.

The Reserved Vocabulary Map

An overview of the 99 reserved words and tokens in Modern C++.

1 82 Core Keywords

- Types and core values
- Declarations and type forms
- Classes and object model
- Control flow and execution
- Memory, lifetime, and storage
- Compile-time and constant evaluation
- Templates and concepts
- Casts and type introspection
- Exceptions
- Coroutines
- Modules and contracts

2 11 Alternative Operator Tokens

`and`	`and_eq`	`bitand`
`bitor`	`compl`	`not`
`not_eq`	`or`	`or_eq`
`xor`	`xor_eq`	

3 6 Special Identifiers

`final`	`import`
`module`	`override`
`post`	`pre`

4 Reserved Naming Rules

- `__name`
- `_Name`
- `_name in the global namespace`

These forms are reserved to the implementation.

Understanding these distinctions prevents many inaccurate keyword lists.

PART I

Fundamental Types and Core Values

These keywords define the built-in scalar types, type modifiers, and truth/null literals that appear constantly in C++ source code.

✓ `bool`	A `char`	8 `char8_t`	16 `char16_t`	32 `char32_t`
W `wchar_t`	int `int`	1.0f `float`	1.0 `double`	S `short`
L `long`	+/- `signed`	u `unsigned`	∅ `void`	T `true`
F `false`	∅ `nullptr`			

Why this group matters

These words form the most common building blocks of everyday C++.

What to watch

Some names denote types, while `true`, `false`, and `nullptr` are literals.

Modern additions

`char8_t` and `nullptr` arrived later than the classic C++98 core types.

At a Glance: How They Fit Together

Fundamental Keywords

- **Scalar Types**: `bool, char, char8_t, char16_t, char32_t, wchar_t, int, float, double, short, long`
- **Type Modifiers**: `signed, unsigned`
- **Literals**: `true, false, nullptr`
- **Special Type**: `void`

Fundamental Types and Core Values

These keywords define the built-in scalar building blocks and core truth/null literals of C++.

bool

Since: C++98

The built-in Boolean type.
Its values are true and false.

```
bool ready = true;
```

char

Since: C++98

The ordinary character type.
It is distinct from both
signed char and unsigned char.

```
char grade = 'A';
```

char8_t

Since: C++20

A distinct character type
for UTF-8 code units.

Compilers GCC 9 Clang 7 MSVC 19.22

```
const char8_t* text = u8"Hello";
```

char16_t

Since: C++11

A character type intended
for 16-bit Unicode code units.

Compilers GCC 4.4 Clang 2.9 MSVC 19.0

```
char16_t ch = u'\u03A9';
```

char32_t

Since: C++11

A character type intended
for 32-bit Unicode code units.

Compilers GCC 4.4 Clang 2.9 MSVC 19.0

```
char32_t ch = U'\U000 1F642';
```

wchar_t

Since: C++98

The implementation-defined
wide-character type used by
wide-character strings.

```
const wchar_t* text = L"wide";
```

NOTE Its width differs across platforms.

int

Since: C++98

The standard signed integer type.

```
int count = 42;
```

float

Since: C++98

Single-precision floating-point type.

```
float pi = 3.14f;
```

Fundamental Types and Core Values

These keywords define the built-in scalar building blocks and core truth/null literals of C++.

01 double

Since: C++98

Double-precision floating-point type.

```
double e = 2.718281828;
```

02 true

Since: C++98

The Boolean literal representing truth.

```
bool ok = true;
```

03 false

Since: C++98

The Boolean literal representing falsity.

```
bool ok = false;
```

04 nullptr

Since: C++11

The null pointer literal with a dedicated type.

```
int* p = nullptr;
```

Compilers

GCC 4.6 | Clang 2.9 | MSVC 16.0

05 short

Since: C++98

An integer type modifier producing a type no wider than `int`.

```
short temperature = -12;
```

06 long

Since: C++98

An integer or floating-point type modifier used in `long`, `long long`, and `long double`.

```
long value = 100000L;
```

07 signed

Since: C++98

The signed integer type modifier.

```
signed int delta = -5;
```

08 unsigned

Since: C++98

The unsigned integer type modifier.

```
unsigned count = 100u;
```

09 void

Since: C++98

Represents no value, or a typeless pointer target when used as `void*`.

```
void print();
```

At a glance

Type names:	`bool, char, int, float, double, wchar_t, char8_t, char16_t, char32_t`
Type modifiers:	`short, long, signed, unsigned`
Literals:	`true, false, nullptr`

Declarations, Namespaces, and Type Forms

1 auto

Since: C++98 / modern type deduction since C++11

Meaning: Introduces a declaration with type deduction in modern C++.

Example:

```
auto value = 42;
```

2 const

Since: C++98

Meaning: Marks an object or view as non-modifiable through that path.

Example:

```
const int max_users = 100;
```

3 volatile

Since: C++98

Meaning: Marks accesses as observable in ways relevant to the implementation.

Example:

```
volatile unsigned* reg;
```

4 mutable

Since: C++98

Meaning: Allows a data member to be modified even inside a const member function.

Example:

```
mutable int hits = 0;
```

5 namespace

Since: C++98

Meaning: Introduces a named scope that helps organize code and avoid collisions.

Example:

```
namespace math { }
```

6 typedef

Since: C++98

Meaning: Creates a traditional type alias.

Example:

```
typedef unsigned long ulong;
```

7 using

Since: C++98

Meaning: Introduces aliases, declarations, and namespace imports depending on context.

Example:

```
using Index = std::size_t;
```

8 enum

Since: C++98

Meaning: Introduces an enumeration type.

Example:

```
enum Color { Red, Green, Blue };
```

Modern tip

In everyday modern C++, `using` often replaces `typedef`, and `auto` has become a central readability tool.

PART II

Classes, Objects, and OOP Keywords

These keywords shape classes, object relationships, access control, and polymorphism in modern C++.

class
Defines a user-defined type with data and member functions.

struct
Defines a lightweight aggregate type, members are public by default.

union
Defines a type where all members share the same memory location.

public
Specifies that members are accessible from anywhere.

private
Restricts member access to within the class.

protected
Allows access within the class and its derived classes.

this
Pointer to the current object instance.

friend
Grants access to non-member functions or classes.

virtual
Enables runtime polymorphism via virtual functions.

explicit
Prevents implicit conversions and constructions.

operator
Defines or overloads operators for user-defined types.

Why this group matters
These keywords let you model real-world entities, control access, establish relationships, and define clean, expressive interfaces.

What to watch
Misusing access specifiers or friends can break encapsulation. Overusing `virtual` or `operator` overloads can add hidden complexity and cost.

Modern tip
Prefer `explicit` for safe and clear APIs, use `virtual` only when polymorphism is needed, and design small, focused interfaces.

At a Glance: How They Fit Together

OOP Keywords

- **Type Forms**: `class` `struct` `union`
- **Access Control**: `public` `private` `protected`
- **Object Relationships**: `this` `friend` `virtual`
- **Interface Behavior**: `explicit` `operator`

Classes, Objects, and OOP Keywords

Core object-model vocabulary for everyday C++.

1 class

Since: C++98

Meaning: Defines a class type with members and member functions.

Example:

```
class Car { };
```

2 struct

Since: C++98

Meaning: Defines a simple aggregate type; like `class` with different defaults.

Example:

```
struct Point { int x; int y; };
```

3 public

Since: C++98

Meaning: Specifies that the following members are publicly accessible.

Example:

```
public: void run();
```

4 private

Since: C++98

Meaning: Specifies that the following members are accessible only within the class.

Example:

```
private: int secret;
```

5 protected

Since: C++98

Meaning: Specifies that the following members are accessible in derived classes.

Example:

```
protected: int value;
```

6 this

Since: C++98

Meaning: Pointer to the current object.

Example:

```
this->value = value;
```

7 friend

Since: C++98

Meaning: Grants a function or class access to private and protected members.

Example:

```
friend void inspect(const Box&);
```

8 virtual

Since: C++98

Meaning: Enables runtime polymorphism via dynamic binding.

Example:

```
virtual void draw() const;
```

Quick distinction

`struct` and `class` differ mainly by default access and default inheritance.

Control Flow and Execution

These keywords direct branching, looping, and flow transfer to control how a program executes in C++.

if

Executes a block if a condition is true.

else

Executes a block when the preceding condition is false.

switch

Selects one block to execute from many cases.

case

Defines a case label within a switch.

default

Defines the block to run when no case matches.

for

Repeats a block using init, condition, and update.

while

Repeats a block while a condition is true.

do

Repeats a block until a condition becomes false.

break

Exits the nearest loop or switch statement.

continue

Skips to the next iteration of the nearest loop.

return

Exits the current function and (optionally) returns a value.

goto

Jumps to a label within the same function.

Why this group matters

Control flow constructs decide which code runs, how often, and when execution moves elsewhere.

What to watch

Mind fall-through in `switch` statements, infinite loops, and the overuse of `goto`, which reduces readability.

Modern tip

Prefer structured control flow. Use `break`/`continue` intentionally and `return` early to simplify logic.

At a Glance: How They Fit Together

Control Flow & Execution

Selection

`if` `else` `switch` `case` `default`

Choose which block executes based on conditions.

Iteration

`for` `while` `do`

Repeat a block of code while a condition holds.

Jumps / Flow Transfer

`break` `continue` `return` `goto`

Alter the normal flow of execution within a function.

Control Flow and Execution

These keywords control program flow, make decisions, and manage loops and exits.

1 if

Since: C++98

Meaning: Executes a block only if the condition evaluates to true.

Example:

```
if (ok) run();
```

2 switch

Since: C++98

Meaning: Selects one of many blocks to execute based on a value.

Example:

```
switch (state) { }
```

3 case

Since: C++98

Meaning: Labels a block inside a switch that matches a value.

Example:

```
case 1:
```

4 default

Since: C++98

Meaning: Specifies the block to execute if no cases match.

Example:

```
default:
```

5 for

Since: C++98

Meaning: Repeats a block a known number of times.

Example:

```
for (int i=0; i<10; ++i) { }
```

6 while

Since: C++98

Meaning: Repeats a block while the condition remains true.

Example:

```
while (running) { }
```

7 do

Since: C++98

Meaning: Executes a block once, then repeats while the condition is true.

Example:

```
do { work(); }
while (ok);
```

8 break

Since: C++98

Meaning: Immediately exits the nearest loop or switch.

Example:

```
break;
```

9 continue

Since: C++98

Meaning: Skips the rest of the current iteration and continues with the next.

Example:

```
continue;
```

Also in this family

`return` ends a function; `goto` jumps to a label and is best used sparingly.

Memory, Lifetime, and Storage

These keywords govern storage duration, linkage, dynamic memory, and low-level object layout.

`static`
Gives static storage duration or internal linkage.

`extern`
Declares a name with external linkage.

`thread_local`
Gives thread storage duration to an object.

`register`
Suggests storage in a CPU register (hint only).

`new`
Allocates memory dynamically on the free store.

`delete`
Deallocates memory previously allocated by new.

`sizeof`
Yields the size (in bytes) of a type or object.

`alignas`
Specifies the alignment requirement of a type.

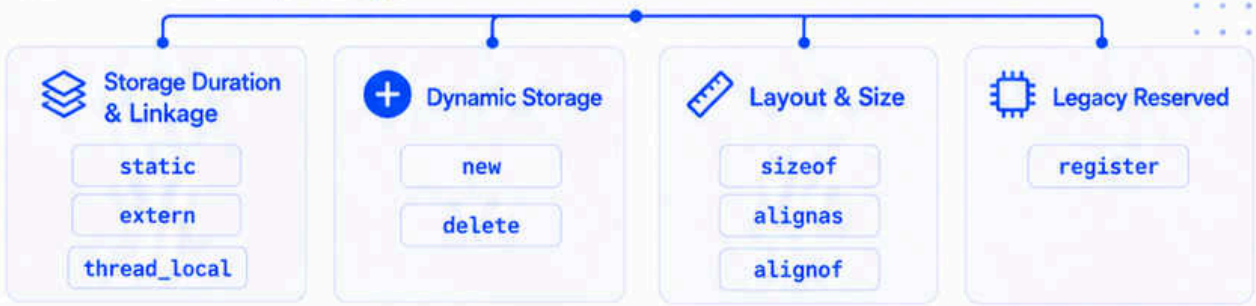
`alignof`
Yields the alignment requirement of a type.

Why this group matters
These keywords control how and where data lives, for how long, and how it's accessed efficiently and safely.

What to watch
Linkage affects visibility across translation units, duration affects lifetime, and layout affects performance and ABI.

Modern tip
Prefer RAII and smart pointers over manual new/delete, and use alignas for performance-sensitive types.

At a Glance: How They Fit Together



Memory, Lifetime, and Storage

1 static

Since: C++98

Meaning: Gives static storage duration. At namespace scope, has internal linkage by default.

Example:

```
static int calls = 0;
```

2 extern

Since: C++98

Meaning: Declares a name defined in another translation unit or with external linkage.

Example:

```
extern int global_count;
```

3 thread_local

Since: C++11

Meaning: Gives thread storage duration; the object has a separate instance per thread.

Example:

```
thread_local int error_code = 0;
```

GCC 4.8* | Clang 3.3 | MSVC 19.0*

4 new

Since: C++98

Meaning: Dynamically allocates storage and returns a pointer to an object.

Example:

```
int* p = new int(42);
```

5 delete

Since: C++98

Meaning: Deallocates storage obtained by new (or new[]), calling the destructor.

Example:

```
delete p;
```

6 sizeof

Since: C++98

Meaning: Yields the size in bytes of a type or expression at compile time.

Example:

```
auto bytes = sizeof(double);
```

7 alignas

Since: C++11

Meaning: Specifies a stricter alignment requirement for an object or type.

Example:

```
alignas(64) int data[16];
```

8 alignof

Since: C++11

Meaning: Yields the alignment requirement of a type at compile time.

Example:

```
auto a = alignof(double);
```

9 register

Since: C++98

Meaning: Suggests storage in a CPU register. The keyword is reserved but obsolete.

Example:

```
register is reserved but
obsolete in modern C++;
```

`register` remains a reserved keyword, but its historical storage-class role is obsolete.

Compile-Time and Constant Evaluation

These keywords move logic into translation time and make compile-time intent explicit.

constexpr

Defines functions or variables that can be evaluated at compile time.

consteval

Requires immediate compile-time evaluation; no runtime fallback is allowed.

constinit

Ensures a variable with static or thread storage duration is constant-initialized.

static_assert

Performs a compile-time check and produces a custom diagnostic if false.

decltype

Produces the type of an expression without evaluating it.

noexcept

Specifies that a function does not throw exceptions, enabling stronger guarantees.

Why this group matters

These keywords help shift work to compile time, catch errors earlier, and document guarantees to the compiler and readers.

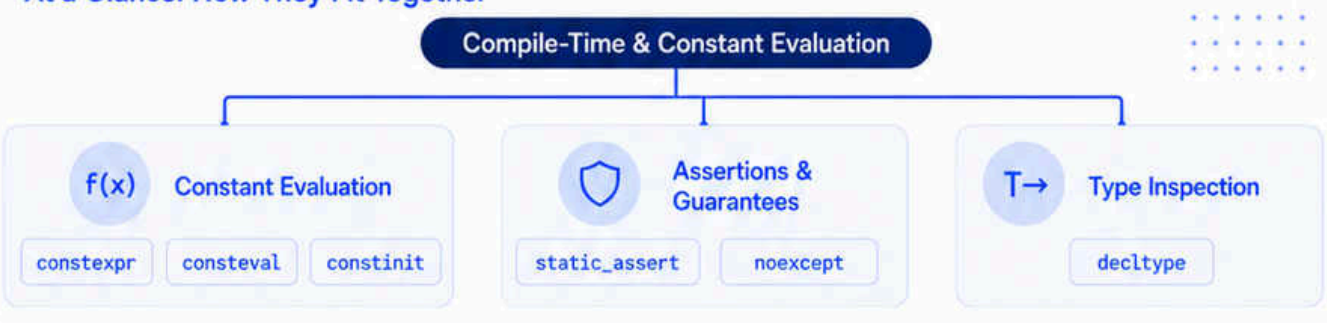
What to watch

`consteval` has no runtime fallback, `constinit` only guarantees constant initialization, and `noexcept` is part of a function's type.

Modern tip

Prefer `constexpr` and `consteval` for compile-time algorithms, `constinit` for globals, and `static_assert` to lock in invariants early.

At a Glance: How They Fit Together



Compile-Time and Constant Evaluation

Control what is evaluated at compile time, enforce requirements, and guarantee safe initialization.

1 constexpr

A function or variable that can be evaluated at compile time when the arguments are constant expressions.

```
constexpr int square(int x) {
    return x * x;
}

constexpr int v = square(6);
static_assert(v == 36);
```

Usable at compile time and at runtime.

2 consteval

A function that must be evaluated at compile time. Calling it at runtime is ill-formed.

```
consteval int cube(int x) {
    return x * x * x;
}

constexpr int c = cube(3); // OK
int r = cube(3);           // Error
```

Always compile-time only. No runtime fallback.

3 constinit

Ensures a variable with static or thread storage duration is initialized at compile time (constant initialization).

```
constinit int g = 42;      // OK

constinit std::array<int, 2> a
    = {1, 2};              // OK

// constinit int* p = new int(7);
// Error: dynamic initialization
```

Does not imply const; it enforces static initialization.

static_assert

Perform compile-time checks on constant expressions.

```
static_assert(sizeof(int) == 4,
        "int must be 4 bytes");
static_assert(square(5) == 25);
```

No runtime cost. Fails the build with a clear message.

T= decltype

Yields the type of an expression without evaluating it.

```
int x = 0;
decltype(x) y = 1;      // int
decltype((x)) z = x;    // int&
```

Essential for generic code and deducing exact types.

noeccept

Specifies and queries whether a function is guaranteed not to throw.

```
void f() noexcept { }
static_assert(noexcept(f()));
constexpr bool ok = noexcept(f(1));
```

Enables optimizations and stronger exception-safety guarantees.

At a glance: when to use which

constexpr

Can be compile-time

Use when a function or variable can be evaluated at compile time when possible, but may run at runtime.

Flexibility with optimization.

consteval

Must be compile-time

Use when a function must never run at runtime (e.g., for metaprogramming or validation).

Compile-time only, no fallback.

constinit

Static initialization guarantee

Use for variables that must be initialized at compile time with static storage duration.

No dynamic initialization.

Templates, Concepts, and Generic Programming

These keywords define reusable abstractions and constraints that enable generic, type-safe, and efficient code in modern C++.

template

Introduces a template to define a family of types or functions.

typename

Disambiguates dependent names that refer to types inside templates.

concept

Defines a compile-time predicate that constrains template arguments.

requires

Applies constraints to templates or functions using Boolean expressions and concepts.

auto

Enables type deduction for variables, return types, and parameters.

using

Introduces type aliases and simplifies names for types and templates.

Why this group matters

These keywords let you build reusable, constraint-aware abstractions that work across many types with zero overhead.

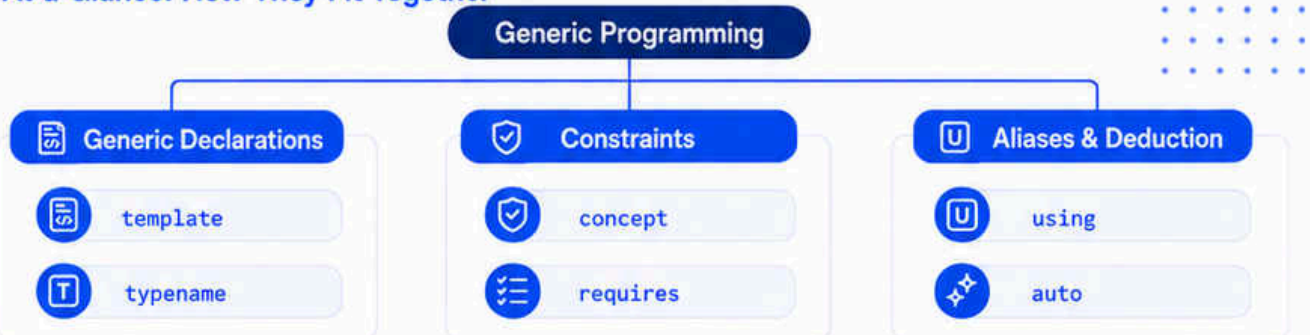
What to watch

Use concepts and `requires` to express intent and get clearer diagnostics when constraints are not met.

Modern tip

Prefer `concepts + requires` over SFINAE for constraints. Use `auto` and `using` to keep code clean and intention-revealing.

At a Glance: How They Fit Together



Templates, Concepts, and Generic Programming

Tools for writing reusable, expressive, and constraint-aware code.

1 template

Since: C++98

Meaning: Introduces a template for functions or classes.

Example:

```
template<typename T>
struct Box { T value; };
```

2 typename T

Since: C++98

Meaning: Disambiguates nested dependent names that refer to types.

Example:

```
typename T::value_type x;
```

3 concept

Since: C++20

Meaning: Declares a constraint that expresses a semantic requirement.

Example:

```
template<typename T>
concept Addable =
requires(T a, T b) { a + b; };
```

GCC 10 | Clang 10 | MSVC 19.30*

4 requires

Since: C++20

Meaning: Applies a constraint to a template, function, or class, enabling constrained overloads.

Example:

```
template<typename T>
requires std::integral<T>
T add(T a, T b) { return a + b; }
```

GCC 10 | Clang 10 | MSVC 19.30*

5 auto A

Since: C++98

Meaning: Enables type deduction from an initializer or return expression.

Example:

```
auto value = 42;
```

6 using U

Since: C++98

Meaning: Introduces an alias for a type or imports names into the current scope.

Example:

```
using Index = std::size_t;
```

Important note

In a type-parameter list, `template<typename T>` and `template<class T>` are functionally equivalent. Choose whichever you prefer—use them consistently.

Casts and Type Inquiry

Safe conversions, low-level reinterpretation, and runtime type information in modern C++.

static_cast

Performs compile-time checked conversions between related types. No runtime type checking.

dynamic_cast

Performs runtime checked conversions for polymorphic types. Returns null (pointer) or throws `bad_cast` (reference) on failure.

const_cast

Adds or removes const or volatile qualification. Does not cast away underlying constness.

reinterpret_cast

Reinterprets the bit pattern of a value as a different type. No guarantees about portability or safety.

typeid

Returns a `std::type_info` object describing the static or dynamic type of an expression (may use RTTI).

Why this group matters

These keywords give precise control over conversions and introspection—balancing safety, intent, and low-level power.

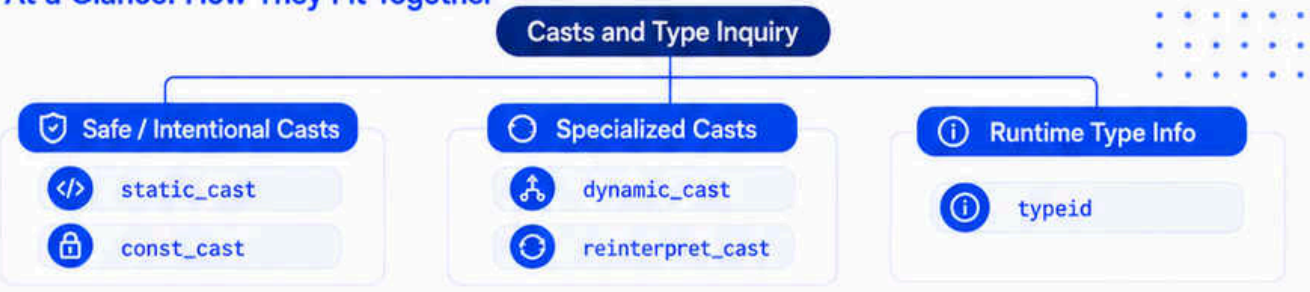
What to watch

Only `dynamic_cast` works across a polymorphic hierarchy. `reinterpret_cast` can break strict aliasing and object lifetime rules.

Modern tip

Prefer `static_cast` by default, use `dynamic_cast` for safe downcasts, and `typeid` for diagnostics and reflection-like capabilities.

At a Glance: How They Fit Together



`dynamic_cast` and `typeid` rely on polymorphic types and RTTI. If RTTI is disabled (`-fno-rtti`), they are unavailable.

Cast Operators and RTTI Keywords

Meaning, intent, and examples for the main C++ cast-related keywords.

1 static_cast

Since: C++98

Meaning: Converts between related or compatible types at compile time.

Example:

```
static_cast<double>(n)
```

2 dynamic_cast

Since: C++98

Meaning: Performs safe runtime-checked casts in polymorphic type hierarchies.

Example:

```
dynamic_cast<Derived*>(basePtr)
```

3 const_cast

Since: C++98

Meaning: Adds or removes constness (or volatility) from a type.

Example:

```
const_cast<char*>(p)
```

4 reinterpret_cast

Since: C++98

Meaning: Converts between unrelated types by reinterpreting the bit pattern.

Example:

```
reinterpret_cast<std::uintptr_t>(ptr)
```

5 typeid

Since: C++98

Meaning: Returns a `std::type_info` object describing the type of an expression.

Example:

```
typeid(obj).name()   // or: typeid(*ptr)
```

Quick guidance

Prefer static_cast

Use for safe, well-defined conversions known at compile time (numeric conversions, up/down casts within a hierarchy, etc.).

Use dynamic_cast

Use in polymorphic hierarchies when you need a checked downcast or cross-cast. Returns `nullptr` (or throws `std::bad_cast` by reference) on failure.

Use reinterpret_cast sparingly

Only for low-level, implementation- or hardware-related conversions. Avoid unless you fully understand the representation.

Exceptions and Error Handling

The core keywords that transfer, handle, and constrain exceptional control flow.

try

Defines a block of code that may throw exceptions.

Since: C++98

catch

Handles an exception thrown from a try block, optionally matching its type.

Since: C++98

throw

Transfers control to the nearest matching catch by throwing an exception object.

Since: C++98

noexcept

Specifies that a function will not throw, or checks this property in an expression.

Since: C++11

`noexcept` is both a specifier (declares a non-throwing function) and an operator (evaluates whether an expression is non-throwing).

Why this group matters

They form the backbone of exceptional control flow, enabling robust error recovery and clear separation of error paths.

What to watch

Always match throws with appropriate catches, avoid throwing from destructors, and be mindful of exception safety guarantees.

Modern tip

Use `noexcept` to communicate intent, enable optimizations, and strengthen interfaces. Prefer the operator form in static assertions and concepts.

At a Glance: How They Fit Together

Throwing

throw
Signals an exceptional condition and transfers control outward.

Handling

try
Encapsulates code that might throw.

catch
Handles exceptions and restores control locally.

Guarantees

noexcept
Specifies non-throwing behavior or queries it in expressions.
Both specifier and operator in practice.

try, catch, throw, and noexcept

A practical quick-reference for exception-related keywords.

1 try

Since: C++98

Meaning: Starts a block of code to be monitored for exceptions.

Example:

```
try {
    risky();
}
```

2 catch

Since: C++98

Meaning: Handles an exception of a specific type.

Example:

```
catch (const std::exception& e) {
    // handle the exception
}
```

3 throw

Since: C++98

Meaning: Raises an exception to signal an error condition.

Example:

```
throw std::runtime_error("fail");
```

4 noeccept

Since: C++11

Meaning: Specifies that a function is guaranteed not to throw exceptions.

Example:

```
void f() noexcept {
   // function body
}
```

GCC 4.6+ | Clang 3.0+ | MSVC VS 2015

5 Notes & Comparisons

- **Rethrowing**
  Inside a `catch` block, use `throw;` to rethrow the current exception, preserving its type and information.

  Example:

```
catch (const std::exception& e) {
   log(e.what());
   throw;    // rethrow the same exception
}
```

- **noexcept as an Operator**
  `noexcept` can also be used as an operator to query whether an expression is noexcept.

  Examples:

```
noexcept(f());                  // true or false
noexcept(f() noexcept);         // true
noexcept(std::vector<int>().push_back(1));
                                // typically false
```

Coroutines and Asynchronous Flow

The three C++20 keywords that suspend, resume, and produce values incrementally.

co_await

Since: C++20

Suspends the current coroutine until the awaited expression completes. Resumes with the result.

co_yield

Since: C++20

Produces an intermediate value from a coroutine and suspends. Resumes when the consumer requests more.

co_return

Since: C++20

Returns the final value from a coroutine and completes it, waking any waiters on the result.

Why this group matters

Coroutines enable asynchronous, non-blocking, and generator-style code with straightforward syntax and clear control over suspension and resumption.

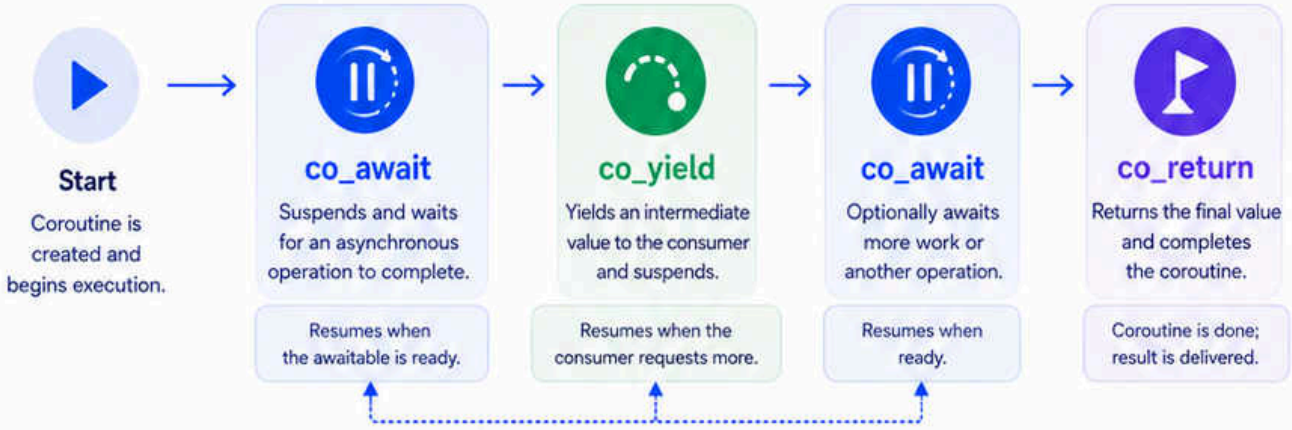
What to watch

Understand lifetime and ownership across suspension points. Avoid holding references to locals that may be destroyed while the coroutine is suspended.

Modern tip

Combine `co_await` for async operations with `co_yield` to stream results, and `co_return` for the final result—building expressive, efficient async pipelines.

At a Glance: Coroutine Flow



Since C++20

Support snapshot

GCC 10	GCC 11	Clang	MSVC
(with `-fcoroutines`)	(standard mode)	Partial support	VS 2019 16.8

Coroutine Keywords in Practice

Meaning, examples, and relationships for co_await, co_yield, and co_return.

1 co_await

Since: C++20

Meaning:

Suspends the current coroutine until an awaitable operation completes, then resumes with the result.

Example:

```
co_await socket.async_read();
```

GCC 10/11 | Clang partial | MSVC 16.8

2 co_yield

Since: C++20

Meaning:

Suspends the coroutine and produces a value to the caller (e.g., via a generator), which can be consumed.

Example:

```
co_yield value;
```

GCC 10/11 | Clang partial | MSVC 16.8

3 co_return

Since: C++20

Meaning:

Sets the final result of the coroutine and completes it, resuming the caller with that result.

Example:

```
co_return result;
```

GCC 10/11 | Clang partial | MSVC 16.8

How they cooperate

Await

The coroutine starts and hits `co_await`. It suspends and waits for an async operation to finish.

```
co_await socket.async_read();
```

→

Yield

When ready to produce a value (e.g., in a generator), the coroutine uses `co_yield` to deliver it and suspend.

```
co_yield value;
```

→

Return

When complete, the coroutine uses `co_return` to set the final result and finish execution.

```
co_return result;
```

ⓘ Coroutines may use `await`, `yield`, and `return` multiple times before completion, depending on the design.

Important note

These keywords are valid only in coroutine contexts—inside coroutine functions or bodies of coroutine-enabled constructs.

Modules and Program Boundaries

Modern keywords for exporting interfaces and importing named program units.

1 module

Meaning: Declares a named module. A module partitions a program into interface and implementation units.

Example:

```
export module math;
```

Since: C++20

2 import

Meaning: Imports declarations from a named module into the current translation unit.

Example:

```
import math;
```

Since: C++20

3 export

Meaning: Makes declarations part of a module's interface so they can be imported.

Example:

```
export int add(int a, int b);
```

Since: C++20

Why this group matters

Modules improve build times, enforce clear boundaries, and provide stable, explicit interfaces for larger codebases.

What to watch

Support is still evolving across compilers. Tooling and build system integration may require extra configuration.

Modern tip

Export only the minimal API surface. Keep implementation details in module partitions to protect encapsulation.

At a Glance: How They Fit Together

Interface Unit

math.ixx

```
export module math;
export int add(
  int, int);
export int sub(
  int, int);
```

Declares the module and exports its API.

→

Importing Code

main.cpp

```
import math;

int main() {
  int r = add(2, 3);
  return r;
}
```

Imports the module to use its exported declarations.

→

Exported API

```
int add(int, int)
int sub(int, int)
...
```

Only exported names form the module's public interface.

Compiler & Tooling Notes

Since: C++20

GCC	Clang	MSVC
11 (-fmodules-ts)	modules TS / evolving standard modules support	VS 2019 16.8

Contracts and Assertions in C++26

New contract vocabulary plus its relationship to compile-time assertions.

1 pre

Contract annotation: expresses a precondition for a function.

What it means

States what must be true before the function is entered.

Example:

```
pre(x > 0);
```

2 post

Contract annotation: expresses a postcondition for a function.

What it means

States what will be true after successful function completion.

Example:

```
post(result > 0);
```

3 contract_assert

C++26 runtime contract assertion keyword. Evaluated at runtime.

What it means

Checks a condition during execution; triggers the active contract violation handler on failure.

Example:

```
contract_assert(
  ptr != nullptr);
```

4 static_assert

Compile-time assertion keyword (established since C++11).

What it means

Checks a constant expression at compile time; prevents invalid programs from compiling.

Example:

```
static_assert(
  sizeof(int) == 4);
```

pre_post
Contract annotations (declarative)

contract_assert
Runtime assertion (imperative)

static_assert
Compile-time assertion (compile-time)

Different purposes, complementary in practice

Compiler Support Snapshot

Feature	GCC	Clang	MSVC
Contracts (C++26)	✓ GCC 16	⊘ No	− Public conformance page not yet listing support
`static_assert` (since C++11)	✓ Yes (longstanding)	✓ Yes (longstanding)	✓ Yes (longstanding)

ⓘ Support status reflects public information as of May 2025 and may change.

Why this group matters

Contracts describe intent and behavior, while `contract_assert` enforces it at runtime. `static_assert` complements them by validating assumptions early at compile time.

What to watch

Contracts are a new facility in C++26 and toolchain support will evolve. Design your contracts to be clear, stable, and consistent across build configurations.

Modern tip

Use all three tools together: `pre/post` to document intent, `contract_assert` to enforce at runtime, and `static_assert` to lock in compile-time guarantees.

Compiler Support Snapshot

A practical high-level view of when major modern keyword families became available in GCC, Clang, and MSVC.

Feature	Keywords	Standard	GCC	Clang	MSVC	Notes
Concepts & requires	`concept`, `requires`	C++20	GCC 10	Clang 10	MSVC VS 2019 16.3–16.7	Broad C++20 support family.
Coroutines	`co_await`, `co_yield`, `co_return`	C++20	GCC 10 with `-fcoroutines` GCC 11 in standard mode	Clang partial support	MSVC VS 2019 16.8	Windows/ABI caveats on Clang.
Modules	`module`, `import`, `export`	C++20	GCC 11 with `-fmodules-ts`	Clang modules TS / evolving standard support	MSVC VS 2019 16.8	Implementation maturity varies.
Contracts	`pre`, `post`, `contract_assert`	C++26	GCC 16	No	Not publicly listed	Newest and least portable.
Core classic keywords	`class`, `if`, `try`, `throw`, `new`, `delete`, etc.	C++98	Yes	Yes	Yes	Long-established.

Note
Consult current release notes for edge cases and partial support.

Master Keyword Roadmap

A fast navigation page that groups the booklet's reserved keywords by theme and points the reader to the most relevant pages.

1 Object Model

- class
- struct
- union
- public
- private
- protected
- this
- friend
- virtual
- explicit
- operator

Pages 10–11

2 Control Flow

- if
- else
- switch
- case
- default
- for
- while
- do
- break
- continue
- return
- goto

Pages 12–13

3 Memory & Storage

- static
- extern
- thread_local
- new
- delete
- sizeof
- alignas
- alignof
- register

Pages 14–15

4 Compile-Time

- constexpr
- consteval
- constinit
- static_assert
- decltype
- noexcept

Pages 16–17

5 Generic Programming

- template
- typename
- concept
- requires
- auto
- using

Pages 18–19

6 Casts & RTTI

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast
- typeid
- decltype(auto)

Pages 20–21

7 Exceptions

- try
- catch
- throw
- noexcept
- exception_ptr

Pages 22–23

8 Coroutines

- co_await
- co_yield
- co_return
- co_suspend
- co_handle
- generator

Pages 24–25

9 Modules

- module
- export
- import
- module: partition

Page 26

10 Contracts

- contract_assert
- [[expects]]
- [[ensures]]
- [[pre]]
- [[post]]
- audit

Page 27

11 Compiler Support Appendix

- feature-test macros
- implementation
- __has_include
- __has_cpp_attribute
- diagnostics

Page 28

How to use this atlas

- Find a keyword (or group) you need.
- Jump to the indicated page for the full entry.
- Use cross-references inside entries to explore related topics.
- Return here anytime for a fast reorientation.

Fast recall tips

- Think in themes: keywords are easier to remember by purpose.
- Spot the family: similar keywords often share patterns.
- See it in context: read the examples on the target page.
- Practice & revisit: repetition builds fluent recall.

Keyword Evolution Timeline

How the reserved-word vocabulary expanded from C++98 to C++26.

1

C++98 Foundation

`class`, `struct`, `union`, `enum`, `public`, `private`, `protected`, `virtual`, `try`, `catch`, `throw`, `new`, `delete`, `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, `typeid`

1998

Established the core object model, access control, exceptions, RTTI, and the classic cast operators.

2

C++11 Expansion

`nullptr`, `constexpr`, `decltype`, `static_assert`, `noexcept`, `thread_local`, `alignas`, `alignof`, `char16_t`, `char32_t`

2011

Added compile-time power, better safety checks, and support for concurrency and Unicode.

3

C++14 / C++17 Refinement

These standards improved many features while adding few or no major new reserved keywords. Emphasis was on refinement, not vocabulary growth.

2014 / 2017

Library, language, and tooling evolutions—without expanding the reserved-word set.

4

C++20 Big Leap

`concept`, `requires`, `consteval`, `constinit`, `co_await`, `co_yield`, `co_return`, `char8_t`, `module`, `import`

2020

Introduced concepts, coroutines, modules, and compile-time evaluation foundations.

5

C++23

Broad language and library progress with no major new keyword family.

2023

Focused on usability, ranges, pattern matching, and library enhancements.

6

C++26 Contracts

`pre`, `post`, `contract_assert`

2026

Brings contracts to the core language with three dedicated keywords.

Key insight

Most standards add many capabilities without adding many new reserved words.
C++ grows by strengthening and connecting features, not by swelling the keyword list.

Why this matters

- Reserved words can't be used as identifiers.
- Knowing when keywords appeared helps when reading older code or targeting specific standards.
- Feature sets evolve faster than the keyword set.
- Design clarity improves with a stable vocabulary.

Reading hint

- Use the timeline to place a keyword in its historical context.
- Look up details in the atlas pages for each keyword group.
- When in doubt, assume code portability is limited by the oldest keyword you use.
- Prefer modern features, but respect the standard level your codebase must support.

Alternative Tokens and Operator Words

The textual operator keywords that can replace symbolic operators in C++.

1 and = &&

Meaning
Logical AND

Example

```
if (a and b) { /* ... */ }
```

2 and_eq = &=

Meaning
Bitwise AND assignment

Example

```
flags and_eq MASK;
```

3 bitand = &

Meaning
Bitwise AND

Example

```
auto c = a bitand b;
```

4 bitor = |

Meaning
Bitwise OR

Example

```
auto c = a bitor b;
```

5 compl = ~

Meaning
Bitwise NOT (complement)

Example

```
auto c = compl a;
```

6 not = !

Meaning
Logical NOT

Example

```
if (not ready) { /* ... */ }
```

7 not_eq = !=

Meaning
Not equal

Example

```
if (x not_eq y) { /* ... */ }
```

8 or = ||

Meaning
Logical OR

Example

```
if (a or b) { /* ... */ }
```

9 or_eq = |=

Meaning
Bitwise OR assignment

Example

```
flags or_eq MASK;
```

10 xor = ^

Meaning
Bitwise XOR (exclusive OR)

```
auto c = a xor b;
```

11 xor_eq = ^=

Meaning
Bitwise XOR assignment

```
flags xor_eq MASK;
```

12 = ^=

Meaning
Bitwise XOR assignment

```
flags xor_eq MASK;
```

Note
In C++, these are keywords, not macros.
They are useful for readability, coding standards, or environments where symbolic forms are inconvenient.

Caution
These alternative tokens are less common in modern code but are fully standard, portable, and supported by all conforming C++ compilers.

Keyword Placement Map

Where different C++ keyword families usually appear.

1 Declarations & Types

Introduce types, aliases, and relationships.

- `class` Class definitions
- `struct` Struct definitions
- `enum` Enumerations
- `union` Unions
- `typedef` Type aliases
- `using` Type and alias declarations
- `typename` Dependent type name
- `friend` Friend declarations
- `explicit` Prevent implicit conversions

```
class X { … };    using Id = int;
friend class Y;   explicit X(int);
```

2 Expressions & Operators

Appear inside expressions and expression contexts.

- `new` Dynamic allocation
- `delete` Dynamic deallocation
- `sizeof` Size in bytes
- `typeid` Type information
- `decltype` Type of an expression
- `static_cast` Static cast
- `dynamic_cast` Dynamic cast
- `const_cast` Const cast
- `reinterpret_cast` Reinterpret cast
- `noexcept` No-throw specification

```
auto p = new X;  auto n = sizeof(T);
auto c = static_cast<Base*>(p);
```

3 Control Flow

Control the order and flow of execution.

- `if` Conditional branch
- `else` Else branch
- `switch` Multi-way branch
- `case` Switch case label
- `default` Default case label
- `for` For loop
- `while` While loop
- `do` Do-while loop
- `break` Exit loop/switch
- `continue` Skip to next iteration
- `return` Return from function
- `goto` Unconditional jump

```
for (int i = 0; i < n; ++i) { … }
if (ok) return value; else { … }
```

4 Compile-Time & Constraints

Evaluate, check, and constrain at compile time.

- `constexpr` Constant evaluation
- `consteval` Immediate evaluation
- `constinit` Constant initialization
- `static_assert` Compile-time assertion
- `concept` Concept definition
- `requires` Requirement expression

```
template <typename T>
concept Addable = requires(T a, T b) { a + b; };
static_assert(Addable<int>);
```

5 Modules & Program Boundaries

Define and manage program structure across translation units.

- `module` Define a module
- `import` Import a module
- `export` Export declarations

```
export module math.core;
import math.core;
export int add(int, int);
```

6 Coroutines & Contracts

Enable coroutines and contract-based guarantees.

- `co_await` Await a coroutine result
- `co_yield` Yield from coroutine
- `co_return` Return from coroutine

- `pre` Precondition annotation
- `post` Postcondition annotation
- `contract_assert` Contract assertion

```
co_await task;
pre(x > 0); post(result >= 0);
contract_assert(ptr != nullptr);
```

How to read this map

Connect the dot	Start from the code you're writing and jump to the relevant keyword family.
Spot the context	Each group shows the typical places its keywords appear.
Use the mini examples	They hint at real-world placement and common patterns.
Go deeper	Turn to the referenced pages to explore details and best practices.

Cross-reference shortcut

Find the full keyword pages quickly.

Tip: Use the Master Keyword Roadmap on page 29 for a fast jump to any area.

Keyword Decision Guide

A quick path to the right keyword family for a given job.

START

What are you trying to do?

1 Need compile-time evaluation or checking?

→ **constexpr • consteval • constinit • static_assert**

Compute, enforce, or validate things at compile time.

2 Need type deduction or type introspection?

→ **auto • decltype • typename • using • typedef**

Let the compiler deduce types or name type-related entities clearly.

3 Need safe or specialized conversion?

→ **static_cast • dynamic_cast • const_cast • reinterpret_cast**

Convert values or references with explicit intent.

4 Need generic constraints?

→ **template • concept • requires**

Constrain templates and express requirements in a clear, modern way.

5 Need storage duration or linkage control?

→ **static • extern • thread_local • mutable**

Control lifetime, visibility, and mutability of objects or data.

6 Need interface boundaries?

→ **module • import • export**

Organize code into modules and manage visibility across translation units.

7 Need suspension or async flow?

→ **co_await • co_yield • co_return**

Build coroutines and asynchronous control flows with clear intent.

8 Need function conditions?

→ **pre • post • contract_assert**

Specify preconditions, postconditions, and check them at runtime or compile time.

Modern rule of thumb

Prefer intent-revealing keywords that make your code safer, clearer, and easier to reason about.

Express intent
Use the keyword that best communicates your goal.

Make it safe
Let the compiler help you prevent mistakes early.

Stay modern
Use C++26 features to write clearer, more robust code.

Compose well
These keywords work best when used together.

Revisit & refine
Refactor with better keywords as your code evolves.

Feature-Test Macros and Detection

How to check whether keyword-driven language features are available.

Feature	Keywords	Feature-Test Macro	First Standard	Notes
Concepts	`concept`, `requires`	`__cpp_concepts`	C++20	Concepts and requires expressions.
Coroutines	`co_await`, `co_yield`, `co_return`	`__cpp_impl_coroutine`	C++20	Coroutines support keywords.
Modules	`module`, `import`, `export`	`__cpp_modules`	C++20	Named modules and `module` interface units.
Consteval	`consteval`	`__cpp_consteval`	C++20	Immediate evaluation functions.
Constinit	`constinit`	`__cpp_constinit`	C++20	Constant initialization guarantee.
Static assert	`static_assert`	`__cpp_static_assert`	C++11	Improved `static_assert` (since C++11).
Decltype(auto)	`decltype(auto)`	`__cpp_decltype_auto`	C++14	`decltype(auto)` return type deduction.

Vendor Helper Macros

Useful detection helpers provided by many modern standard libraries.

`__has_include(<header>)`

Checks whether a header can be included.
Returns `1` if available, `0` otherwise.

`__has_cpp_attribute(attr)`

Checks whether a standard attribute is supported.
Returns a value `> 0` if supported, `0` otherwise.
Use in combination with feature-test macros.

Example: Conditional Compilation

Use feature-test macros to enable code paths only when the feature is available.

```
#if defined(__cpp_concepts) && __cpp_concepts >= 201907L
    template <typename T>
    concept Integral = requires (T t) { t + t; };
#else
    // Fallback for compilers without concepts
    template <typename T>
    struct is_integral : std::false_type { };
#endif

#if __has_include(<format>)
   #include <format>
#endif
```

Macros are defined in `<version>`.

Check the macro value (if non-zero) to determine support and the macro value to detect the revision.

Always provide a fallback path to keep code portable.

Compiler Modes and Flags

Practical switches and notes for GCC, Clang, and MSVC.

GCC

Standard selection
Use `-std` to select the C++ standard.

`-std=c++20` `-std=c++23` `-std=c++26`

Coroutines
GCC 10: enable with `-fcoroutines`

GCC 11+: available in standard mode.

Modules (TS)
Supported as Technical Specification.

`-fmodules-ts`

Contracts (C++26)
Implemented in GCC 16.

Enable with `-std=c++26`.

Notes
Feature maturity varies. Check release notes for bugs and limitations.

Clang

Standard selection
Use `-std` to select the C++ standard.

`-std=c++20` `-std=c++23` `-std=c++2c`

Concepts
Concepts are supported starting in Clang 10.

Modules (TS)
Supported as TS.

`-fmodules-ts`

Standard module support is evolving.

Coroutines
Supported since Clang 7. Generally works well, with caveats on some targets and standard libs.

Contracts (C++26)
No C++26 contracts support yet in Clang (as of 2025).

Notes
Use `-Wall -Wextra` and read release notes for implementation status.

MSVC

Standard selection
Use `/std` to select the C++ standard.

`/std:c++20` `/std:c++latest`

Concepts
Concepts are broadly available in VS 2019 (16.3–16.7 era).

Coroutines
Coroutines supported in VS 2019 16.8.

Modules (TS)
Modules (TS) supported in VS 2019 16.8.

Contracts (C++26)
No public contracts support listed (as of 2025).

Notes
Use `/W4` and `/permissive-` for best conformance. Check docs for updates.

Always verify current release notes for edge cases and partial implementations.

Common Keyword Pitfalls

Frequent misunderstandings and how to avoid them.

1 static_cast vs reinterpret_cast

Pitfall
Treating cast types as interchangeable or using reinterpret_cast to "convert" object layouts.

What actually happens
reinterpret_cast performs no conversion and can produce invalid values or violate strict aliasing.

Safer rule
Use static_cast for well-defined conversions; avoid reinterpret_cast unless you truly need bit reinterpretation.

2 dynamic_cast needs polymorphic types

Pitfall
Using dynamic_cast on types that do not have a virtual function.

What actually happens
It fails to compile. RTTI requires a polymorphic base.

Safer rule
Ensure the base class has at least one virtual function (often via a virtual destructor).

3 const_cast does not make undefined behavior safe

Pitfall
Using const_cast to modify a truly const object (e.g., const storage or a literal).

What actually happens
Modifying a const object is undefined behavior.

Safer rule
Only use const_cast to remove const from a non-const object.

4 noexcept specifier vs noexcept operator

Pitfall
Confusing the noexcept specifier (without parentheses) with the noexcept operator (with parentheses).

What actually happens
noexcept is a specifier; noexcept(expr) is an operator that evaluates to bool.

Safer rule
Use noexcept on functions; use noexcept(expr) in conditions or static_assert.

5 auto vs decltype(auto)

Pitfall
Using auto when you need to preserve reference or value category.

What actually happens
auto drops references and cv/refs; decltype(auto) preserves them.

Safer rule
Use auto for values; use decltype(auto) to forward exactly what an expression returns.

6 typename in dependent contexts

Pitfall
Omitting typename when referring to a type that depends on a template parameter.

What actually happens
The compiler may treat it as a value and produce confusing errors.

Safer rule
Use typename for dependent types: typename T::type or typename X<T>::Y.

7 module exports should stay minimal

Pitfall
Exporting too much (headers, macros, internal details) by default.

What actually happens
Increases build time, coupling, and breaks encapsulation.

Safer rule
Export only the stable interface. Keep implementation details unexported in module partitions.

8 co_await works only in coroutine contexts

Pitfall
Using co_await in a regular function or forgetting a promise type.

What actually happens
It fails to compile outside a coroutine or without a valid awaitable.

Safer rule
Use co_await only inside a coroutine (returning a coroutine type). Ensure the awaited type is awaitable.

9 pre/post/contract_assert serve different roles

Pitfall
Using these assertions interchangeably.

What actually happens
They differ in timing, intent, and typical builds.

Safer rule
Use pre for inputs, post for outputs, contract_assert for invariants. Reserve static_assert for compile-time.

Fast safety checklist

- Choose the smallest, most precise cast.
- Use RTTI only with polymorphic bases.
- Never const_cast away true const.
- Know the difference: noexcept vs noexcept(expr).
- Pick auto or decltype(auto) intentionally.
- Use typename in dependent contexts.
- Keep module exports minimal and stable.
- Use co_await only in coroutine functions.
- Use the right assertion for the job.

Fast Revision Cheat Sheet

One-page memory cues for the major C++ keyword families.

Build types and interfaces

Define shapes and set access rules.

- **class** — Blueprint for objects.
- **struct** — Lightweight struct + public by default.
- **union** — Different views of the same memory.
- **enum** — Names for integral values.
- **public private protected** — Control visibility and inheritance.

Steer execution

Choose paths, repeat, and exit.

- **if** — Take a path when true.
- **switch** — Multi-way branch.
- **for** — Counted iterations.
- **while** — Repeat while true.
- **break** — Exit the nearest loop/switch.
- **continue** — Skip to next iteration.
- **return** — Give back and leave.

Manage storage and objects

Control object lifetime and linkage.

- **new** — Allocate dynamically.
- **delete** — Free dynamic memory.
- **static** — One instance / static storage.
- **extern** — Use a definition from elsewhere.
- **thread_local** — One instance per thread.

Think at compile time

Let the compiler do the thinking.

- **constexpr** — Usable in constant expressions.
- **consteval** — Must be evaluated at compile time.
- **constinit** — Constant initialization guaranteed.
- **static_assert** — Check a condition at compile time.

Write generic code

Abstract over types and constraints.

- **template** — Parameterize code.
- **typename** — Name a dependent type.
- **concept** — Define a set of requirements.
- **requires** — Constrain what can be used.
- **using** — Introduce aliases and clarity.

Convert and inspect

Cast between types and discover facts.

- **static_cast** — Safe, explicit conversions.
- **dynamic_cast** — Polymorphic, checked casts.
- **const_cast** — Add/remove const/volatile.
- **reinterpret_cast** — Low-level bit reinterpretation.
- **typeid** — Type information at runtime.

Suspend and resume

Write asynchronous logic clearly.

- **co_await** — Wait for an awaitable.
- **co_yield** — Produce a value lazily.
- **co_return** — Return from a coroutine.

Guard behavior

Specify contracts and guarantees.

- **noexcept** — Promise no exceptions.
- **pre** — State must hold before call.
- **post** — State must hold after call.
- **contract_assert** — Check contract at runtime.

Quick Study Plan

A practical route for mastering the reserved-word landscape.

1 Foundations – object model and control flow.

Focus on
`class`, `struct`, `union`, `public`, `private`, `protected`, `this`, `friend`, `virtual`, `explicit`, `operator`.

Practice by
Write small types and member functions. Explore access, constructors, and overrides.

Common trap
Forgetting that access affects design and APIs, not just encapsulation.

2 Storage and object lifetime.

Focus on
`static`, `extern`, `thread_local`, `new`, `delete`, `sizeof`, `alignas`, `alignof`, `register`.

Practice by
Manage resources in different scopes. Measure sizes and alignments.

Common trap
Assuming lifetime or storage where none exists.

3 Compile-time vocabulary.

Focus on
`constexpr`, `consteval`, `constinit`, `static_assert`, `decltype`, `noexcept`.

Practice by
Build expressions that run at compile time and add static assertions.

Common trap
Using constexpr where runtime is required or vice versa.

4 Generic programming and constraints.

Focus on
`template`, `typename`, `concept`, `requires`, `auto`, `using`.

Practice by
Constrain templates with concepts. Prefer clear, minimal interfaces.

Common trap
Over-constraining templates or leaking implementation details.

5 Casts and exceptions.

Focus on
`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, `typeid`, `exceptions`.

Practice by
Use the right cast for the job and handle failures with exceptions.

Common trap
Using `reinterpret_cast` to "force it to work" or swallowing errors.

6 Coroutines and modules.

Focus on
`co_await`, `co_yield`, `co_return`, `module`, `import`, `export`.

Practice by
Write a tiny coroutine generator and split a project into modules.

Common trap
Mixing coroutine lifetime with local references.

7 Contracts and compiler support appendix.

Focus on
`pre`, `post`, `contract_assert`, `[[expects]]`, `[[ensures]]`, `[[audit]]`.

Practice by
Add contracts to critical APIs and review support in your toolchain.

Common trap
Relying on contracts in builds where they aren't enabled.

Weekly / Session Plan

15–20 minutes per family

Mon	Tue	Wed	Thu	Fri	Sat	Sun
Foundations	Storage & lifetime	Compile-time vocabulary	Generic & constraints	Casts & exceptions	Coroutines & modules	Contracts & appendix

Keep this in mind

Repetition with small code examples is more valuable than memorizing isolated lists. Code, tweak, revisit, repeat.

```
template <typename T>
requires std::integral<T>
constexpr T square(T x) { return x * x; }
```

About This Booklet

A concise guide to C++ reserved words through C++26.

Purpose

To give modern C++ programmers a fast, structured, and practical map of the language keywords.

Who it is for

Learners, working developers, reviewers, and technical writers.

How to use it

- Read by family to understand keywords in context.
- Jump through the roadmap to find what you need.
- Revisit the cheat sheets for quick recall.
- Test ideas and validate assumptions in a compiler.

Prepared by

Prepared by Ayman Alheraki.

Under the patronage of

https://simplifycpp.org

Booklet at a Glance

Core Exceptions

Understand how exceptions and error handling work.

Contracts & Assertions

Use contracts to express intent and catch issues early.

Compiler Support Snapshot

Know what your compiler supports today.

Master Keyword Roadmap

Navigate keywords by theme to the right pages.

Quick Reference

Cheat sheets and tips for daily productivity.

> Modern C++ is easier to learn when its vocabulary is grouped by purpose, context, and evolution.

Thank you for being part of the C++ journey.

Keep exploring. Keep building. The best is yet to come.

APPENDIX C

Compiler Support for Modern Keyword Features

The table focuses on modern keyword additions and context-sensitive identifiers whose support is materially version-dependent.

Keyword	Std	GCC	Clang	MSVC
`alignas`	C++11	GCC 4.8	Clang 3.0	MSVC 19.0
`alignof`	C++11	GCC 4.5	Clang 2.9	MSVC 19.0
`constexpr`	C++11	GCC 4.6	Clang 3.1	MSVC 19.0
`decltype`	C++11	GCC 4.8.1*	Clang 2.9	MSVC 17.0*
`noexcept`	C++11	GCC 4.6	Clang 3.0	MSVC 19.0
`nullptr`	C++11	GCC 4.6	Clang 2.9	MSVC 16.0
`thread_local`	C++11	GCC 4.8*	Clang 3.3	MSVC 19.0*
`char8_t`	C++20	GCC 9	Clang 7	MSVC 19.22
`concept`	C++20	GCC 10	Clang 10	MSVC 19.30*
`requires`	C++20	GCC 10	Clang 10	MSVC 19.30*
`consteval`	C++20	GCC 11	Clang 17*	MSVC 19.29*
`constinit`	C++20	GCC 10	Clang 10	MSVC 19.29
`co_await`	C++20	GCC 10	Clang 17*	MSVC 19.28*
`co_yield`	C++20	GCC 10	Clang 17*	MSVC 19.28*
`co_return`	C++20	GCC 10	Clang 17*	MSVC 19.28*
`contract_assert`	C++26	GCC 16	Clang -	MSVC -

How to read the starred cells

- A star (*) indicates important partial-support history or toolchain caveats.
- Versions prior to those shown may have partial or experimental support.
- Support often matured in later point releases.
- Verify with your target toolchain and flags.
- '-' means no public support as of this milestone.

APPENDIX A

Keyword Support Matrix
Major Compilers

Practical minimum-version guidance for the modern language families most closely tied to this atlas. Snapshot checked 15 August 2026.

Feature	GCC	Clang	MSVC	Practical note
Concepts / `requires`	10+	10+	VS 2019 16.3+	Later defect fixes continued.
Coroutines	10 + `-fcoroutines`; 11 in C++20 mode	C++20 support; Windows target caveats	VS 2019 16.8+	Library/ABI/tooling also matter.
Modules	11+ via `-fmodules-ts`	C++20 modules from 15; later fixes	VS 2019 16.8+	Build integration varies.
C++26 Contracts	**16+**	**No**	Not publicly listed	GCC reports P2900R14 support.
C++26 Reflection	**16+** + `-freflection`	**No**	Not publicly listed	Very new; portability is limited.
Pack indexing	15+	19+	Not publicly listed	Verify feature-test macros.

Support means more than accepting a keyword.

Compiler parsing, library availability, ABI stability, IDE support, build-system integration, and target-specific limitations can mature at different times. For portable code, prefer feature-test macros and current vendor release notes over version assumptions alone.

Vendor sources used for this snapshot: GCC C++ status, Clang C++ status, and Microsoft C/C++ language conformance documentation. GCC lists Contracts and Reflection in GCC 16; Clang currently lists both as unsupported; the Microsoft conformance table does not yet list equivalent C++26 entries.

APPENDIX B

Keyword Evolution Timeline

The unconditional keyword set grew slowly even while the language gained major new capabilities.

C++98	63 keywords	Foundation of the language. Includes `class`, `template`, exceptions, RTTI, casts, storage/control-flow vocabulary, and `export`.
C++11	+10 = 73	Added `alignas`, `alignof`, `char16_t`, `char32_t`, `constexpr`, `decltype`, `noexcept`, `nullptr`, `static_assert`, `thread_local`.
C++14	+0	Important language refinements, but no new unconditional keywords.
C++17	+0	Major facilities such as `if constexpr` use existing keywords; no new unconditional keywords.
C++20	+8 = 81	Added `char8_t`, `concept`, `consteval`, `constinit`, `co_await`, `co_return`, `co_yield`, `requires`.
C++23	+0	Continued language evolution without expanding the unconditional keyword table.
C++26	+1 = 82	Added `contract_assert`. Contract identifiers `pre` and `post` have special contextual meaning rather than being unconditional keywords.

Separate vocabulary classes matter. The current draft lists **82 unconditional keywords**, **11 alternative operator representations**, and **6 identifiers with special meaning**: `final`, `import`, `module`, `override`, `post`, and `pre`.

This classification follows the current C++ draft sections [lex.key] and [lex.name].

APPENDIX C

How Compilers Treat Keywords

Understanding the lexical category explains why some words are always forbidden as names while others are contextual.

1. Unconditional keywords

Identifiers in the standard keyword table are always treated as keywords (outside the special attribute-token exception). They cannot be used as ordinary identifiers or macro names.

```
int class = 1;      // ill-formed
int value = 1;      // OK
```

2. Alternative operator representations

`and`, `or`, `not`, `bitand`, and their related forms are reserved representations of symbolic operators - not library macros.

```
if (ready and connected) { /* same as && */ }
```

3. Identifiers with special meaning

`final`, `override`, `module`, `import`, `pre`, and `post` receive special meaning only in defined grammatical contexts; otherwise ambiguity normally resolves toward an ordinary identifier.

4. Implementation-reserved identifiers

Names containing `__`, names beginning with underscore + uppercase, and leading-underscore names in the global namespace are reserved for implementations. No diagnostic is required for misuse.

Why this matters: lexical classification affects parsers, linters, code generators, macro systems, reflection tooling, and source-to-source transformations - not just what names a programmer may choose.

Reserved Identifier Rules

The standard reserves specific naming patterns for the implementation. These rules are easy to follow and prevent subtle portability conflicts.

Rule	Avoid	Prefer
Double underscore anywhere	`my__value`, `__count`	`my_value`, `count`
Underscore + uppercase at start	`_Index`, `_Node`	`Index`, `Node`
Leading underscore in global namespace	`int _value;`	`int value;`
Implementation-looking names (best practice)	`__gnu_helper`, `_MSC_check`	`gnuHelper`, `mscCheck`

Exact standard rules

- Every identifier containing a double underscore `__` is reserved to the implementation for any use.
- Every identifier beginning with underscore followed by an uppercase letter is reserved to the implementation for any use.
- Every identifier beginning with underscore is reserved to the implementation for use as a name in the global namespace.

Standard-library namespace caution

Do not add declarations to `namespace std` unless the standard explicitly permits that specific customization. Also avoid inventing global names that imitate implementation internals. A simple naming convention without leading underscores is the safest default.

Rule of thumb

If a name looks like it could belong to the compiler, standard library, operating system, or ABI implementation, choose a different name.

APPENDIX E

Quick Comparison Tables

A. Named casts

Keyword	Best fit	Tiny example
`static_cast`	Well-defined explicit conversions	`int n = static_cast<int>(d);`
`dynamic_cast`	RTTI-checked casts in polymorphic hierarchies	`auto* p = dynamic_cast<D*>(base);`
`const_cast`	Add/remove cv-qualification; does not make a truly const object modifiable	`auto* p = const_cast<int*>(cp);`
`reinterpret_cast`	Low-level representation/address reinterpretation	`auto u = reinterpret_cast<uintptr_t>(p);`

B. Compile-time vocabulary

Keyword	Meaning	Fast distinction
`constexpr`	Can participate in constant evaluation	May also run at runtime where allowed.
`consteval`	Immediate function	Calls must produce a constant-evaluated result.
`constinit`	Static initialization guarantee	Does **not** make the object const.
`static_assert`	Compile-time assertion	Rejects the program if its constant condition is false.

C. Contracts in C++26

Vocabulary	Role	Use
`pre`	Context-sensitive precondition identifier	State what must hold at function entry.
`post`	Context-sensitive postcondition identifier	State what must hold on successful function completion.
`contract_assert`	Unconditional C++26 keyword	Express a contract assertion at a point in a function body.

Portability note: Contracts are a very new C++26 facility. Compiler support and evaluation semantics should be checked against the exact toolchain and build configuration you target.

APPENDIX F

Concepts & Requires Cheatsheet

Three forms cover most day-to-day constraint code: concept declarations, requires-clauses, and requires-expressions.

1. Concept declaration

```
#include <concepts>

template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};
```

2. Requires-clause

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

3. Requires-expression

```
template <typename T>
concept HasSize = requires(T x) {
    x.size();
    typename T::value_type;
};
```

Fast guidance

- Name reusable semantic requirements with `concept`.
- Use `requires` to constrain overloads where intent is clearest.
- Prefer concepts to opaque SFINAE when both express the same rule cleanly.
- Feature baseline: GCC 10, Clang 10, and MSVC VS 2019 16.3 for the initial C++20 concepts implementation; later fixes continue to matter.

APPENDIX G

Coroutines Summary

The three C++20 coroutine keywords describe suspension, production, and completion; the surrounding coroutine type supplies the protocol.

`co_await`

Suspend / await

Suspends according to an awaitable and resumes with its result.

`co_yield`

Produce / suspend

Produces an intermediate value through the coroutine promise protocol.

`co_return`

Complete

Finishes the coroutine and supplies its final result where applicable.

```
// Conceptual generator-like body
generator<int> count_to(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}

// Conceptual async task body
task<int> fetch_value() {
    int v = co_await get_value_async();
    co_return v;
}
```

Typical lifecycle

create → suspend / await → resume → yield (optional) → return / complete

The compiler transforms a coroutine into a state machine around a promise type. Lifetimes across suspension points must be designed deliberately.

Compiler snapshot: GCC supports standard-mode coroutines from GCC 11 (GCC 10 with `-fcoroutines`); MSVC lists P0912R5 from VS 2019 16.8. Clang reports broad C++20 coroutine support but still documents Windows target caveats.

APPENDIX H

Modules at a Glance

Modules create explicit translation-unit boundaries and exported interfaces; they are not simply textual includes with a new spelling.

Module interface unit

```
export module math;

export int add(int a, int b);

int add(int a, int b) {
   return a + b;
}
```

Importer

```
import math;

int main() {
   int result = add(2, 3);
   return result == 5 ?  0 :  1;
}
```

Vocabulary	Category	Role
`export`	unconditional keyword	Makes declarations part of the exported module interface; also used in the module declaration.
`module`	special/contextual identifier	Introduces a module declaration in the grammar.
`import`	special/contextual identifier	Imports a module or importable unit according to module grammar.

Toolchain reality

GCC documents modules through `-fmodules-ts` beginning with GCC 11. Clang lists P1103R3 modules from Clang 15 with later fixes continuing. MSVC lists P1103R3 Modules from VS 2019 16.8. Build-system support and compiler maturity still vary, so test the exact workflow you ship.

Practical rule: Export the smallest stable interface you can; keep implementation details unexported.

APPENDIX I

Operator Precedence - Quick View

A compact reminder of the main operator families. Parentheses are usually clearer than relying on memory.

Lvl	Family	Operators / constructs	Assoc.
1	Scope resolution	`::`	L–R
2	Postfix	`() [] . -> ++ -`	L–R
3	Unary / creation	`++ - + - ! ~ * & sizeof alignof new delete co_await`	R–L
4	Pointer-to-member	`.* ->*`	L–R
5	Multiplicative	`* / \%`	L–R
6	Additive	`+ -`	L–R
7	Shift	`« »`	L–R
8	Relational	`< <= > >=`	L–R
9	Equality	`== !=`	L–R
10	Bitwise AND	`&`	L–R
11	Bitwise XOR	`^`	L–R
12	Bitwise OR	`|`	L–R
13	Logical AND	`&& / and`	L–R
14	Logical OR	`|| / or`	L–R
15	Conditional	`?:`	R–L
16	Assignment	`= += -= *= /= %= «= »= &= ^= |=`	R–L
17	Comma	`,`	L–R

Alternative tokens keep the same precedence. For example, `and` has exactly the precedence of `&&`, and `not` that of `!`. They are alternative lexical representations, not new operators.

Best practice

Use parentheses when an expression mixes several families or when a reviewer would need to recall the table. Clarity beats cleverness.

References & Closing

Primary standards/draft references and compiler documentation used to validate the vocabulary classifications and support notes.

Language and standards references

- Current C++ working draft: `https://eel.is/c++draft/`
- Keywords: `https://eel.is/c++draft/lex.key`
- Identifiers and reserved-name rules: `https://eel.is/c++draft/lex.name`
- C++ reference: `https://en.cppreference.com/`

Compiler conformance references

- GCC C++ status: `https://gcc.gnu.org/projects/cxx-status.html`
- Clang C++ status: `https://clang.llvm.org/cxx_status.html`
- MSVC language conformance: `https://learn.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance`

Keep the vocabulary close. Keep the code clear.

The goal of this booklet is not to make programmers memorize isolated words. It is to make the structure of modern C++ easier to navigate: recognize the family, understand the intent, then verify the exact rule when precision matters.

Prepared by

Ayman Alheraki

Modern C++ technical reference booklet.

Under the patronage of

https://simplifycpp.org

Practical C++ knowledge, references, and learning resources.

Thank you for being part of the C++ journey.

Keep learning. Keep building. Keep simplifying C++.