

# Practical Production-proven Constexpr API Elements

**Marcus Boerger**

**2024**

# Practical Production-proven Constexpr API Elements

---

**Marcus Boerger**  
**C++ On Sea**  
**July 2024**  
**(updated)**



# What?

- Intro & Basics
- A Hash Function
- Objects
- Containers
- Glimpse into production/metrics

Learn about constexpr.

Compile-time & Run-time.

# Why?

Assume: Existing code can be improved!

- Code generated from schema.
- Code is hand optimized.
- Uses a lot of if-size-then blocks.
- Hard to maintain.
- Optimizations become harmful!

# How?

- Code uses light background.
- Explanations use darker background.
- Some slides link to Compiler Explorer:



- Some slides link to [helly25.com/mbo](https://helly25.com/mbo):



# Practical: mbo Library



[helly25.com/mbo](https://helly25.com/mbo)

[github.com/helly25/mbo](https://github.com/helly25/mbo)

License: Apache 2.0

Used in production.

Build using Bazel. Too lazy to add CMake support.

**Once upon a time...**



# Once upon a time...

- We celebrated that some compilers optimized strlen



# Once upon a time...

- We celebrated that some compilers optimized `strlen`.
  - Well, sometimes.
  - And it was not clear when it happened and when not.



# constexpr

C++11 introduced constexpr.



# constexpr

C++11 introduced constexpr - but it was pretty lame.



# constexpr

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr.



# constexpr

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr - but it was still lame.



# constexpr

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr - but it was still lame.

C++17 had better constexpr:

- Allowing crazy stuff like: `constexpr std::string_view Name = "Oh!";`



# constexpr

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr - but it was still lame.

C++17 had better constexpr - but it was still lame.



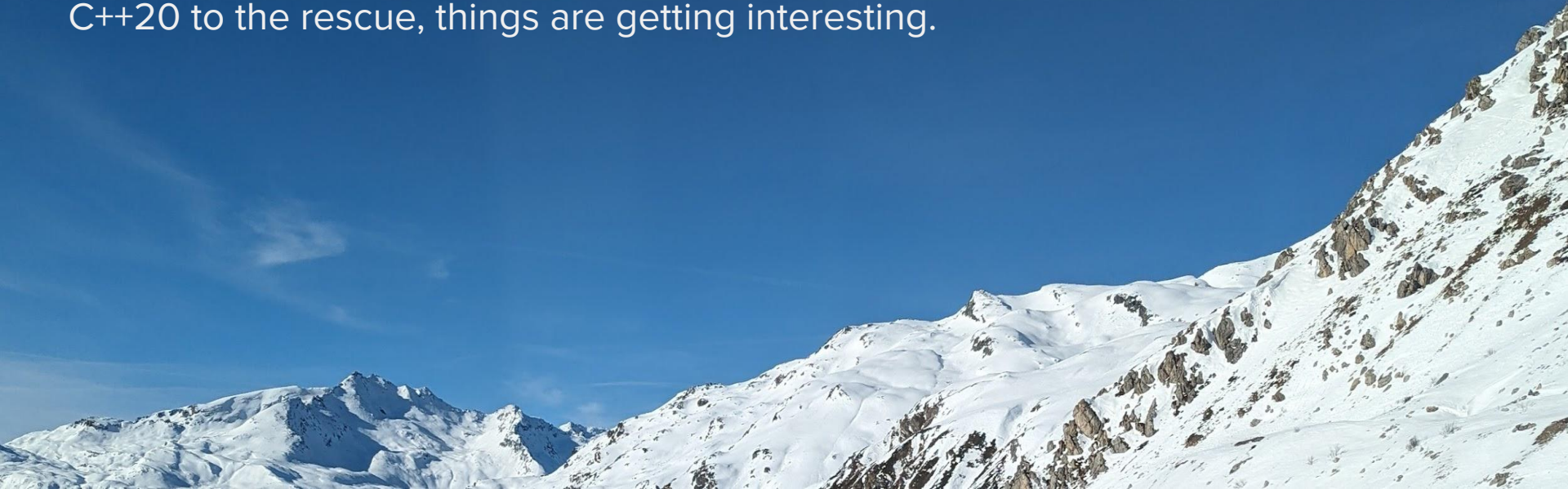
# constexpr + constinit + consteval

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr - but it was still lame.

C++17 had better constexpr - but it was still lame.

C++20 to the rescue, things are getting interesting.



# constexpr + constinit + consteval

C++11 introduced constexpr - but it was pretty lame.

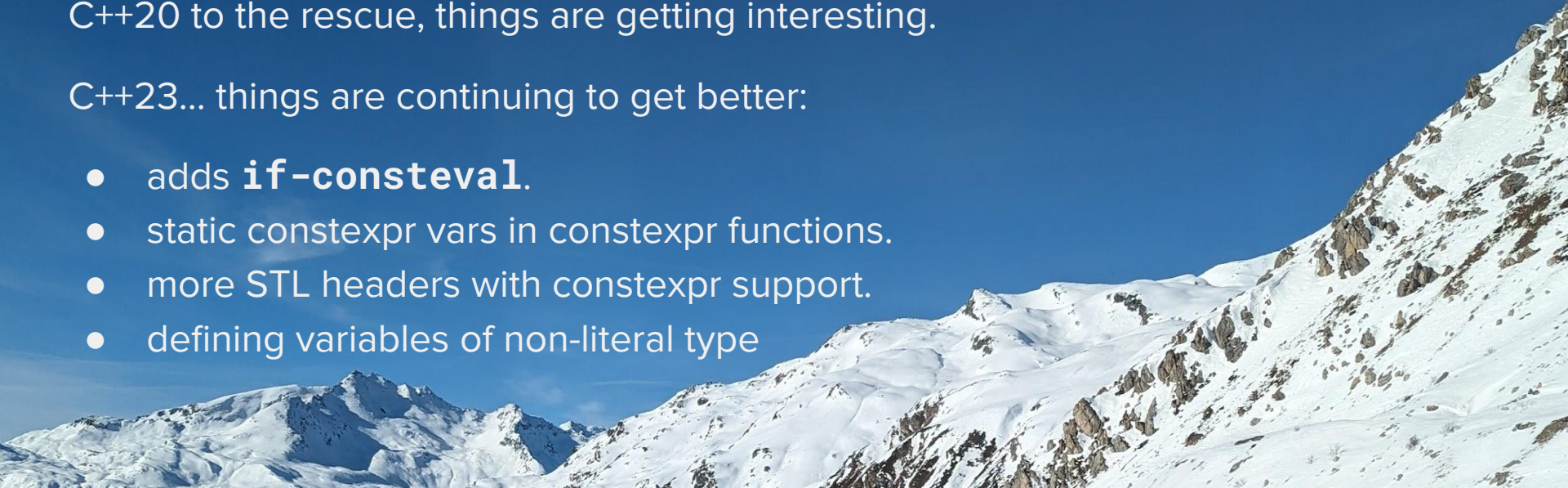
C++14 had better constexpr - but it was still lame.

C++17 had better constexpr - but it was still lame.

C++20 to the rescue, things are getting interesting.

C++23... things are continuing to get better:

- adds **if-consteval**.
- static constexpr vars in constexpr functions.
- more STL headers with constexpr support.
- defining variables of non-literal type



# constexpr + constinit + consteval

C++11 introduced constexpr - but it was pretty lame.

C++14 had better constexpr - but it was still lame.

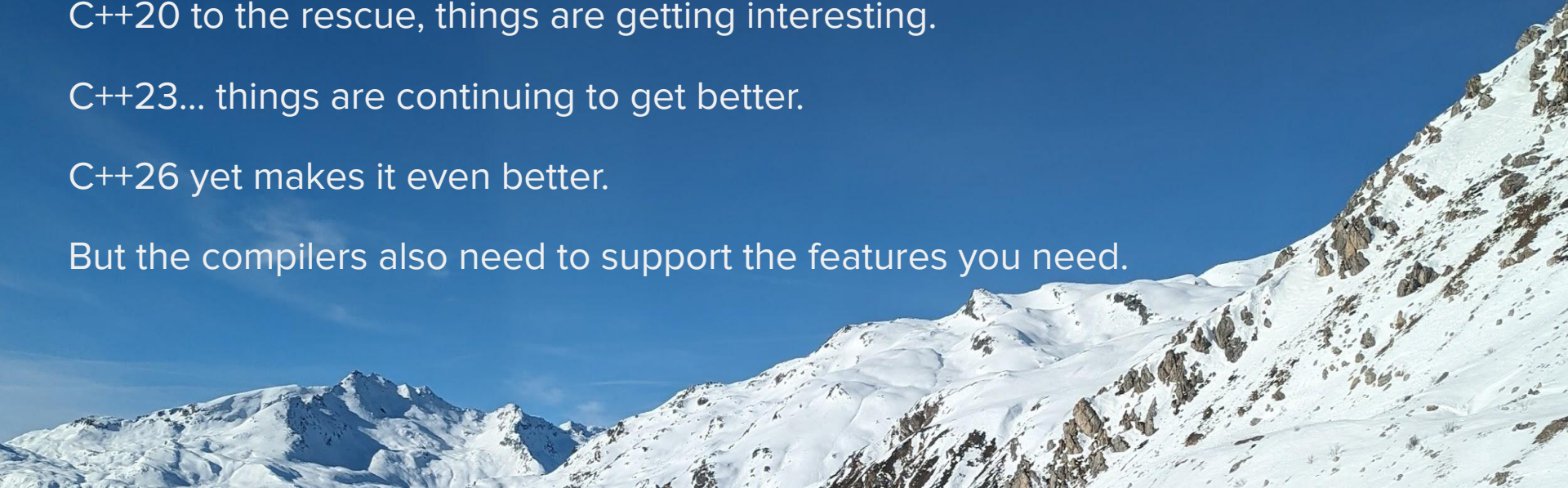
C++17 had better constexpr - but it was still lame.

C++20 to the rescue, things are getting interesting.

C++23... things are continuing to get better.

C++26 yet makes it even better.

But the compilers also need to support the features you need.



# constexpr + constinit + consteval

C++11 introduced constexpr - but it was pretty lame.

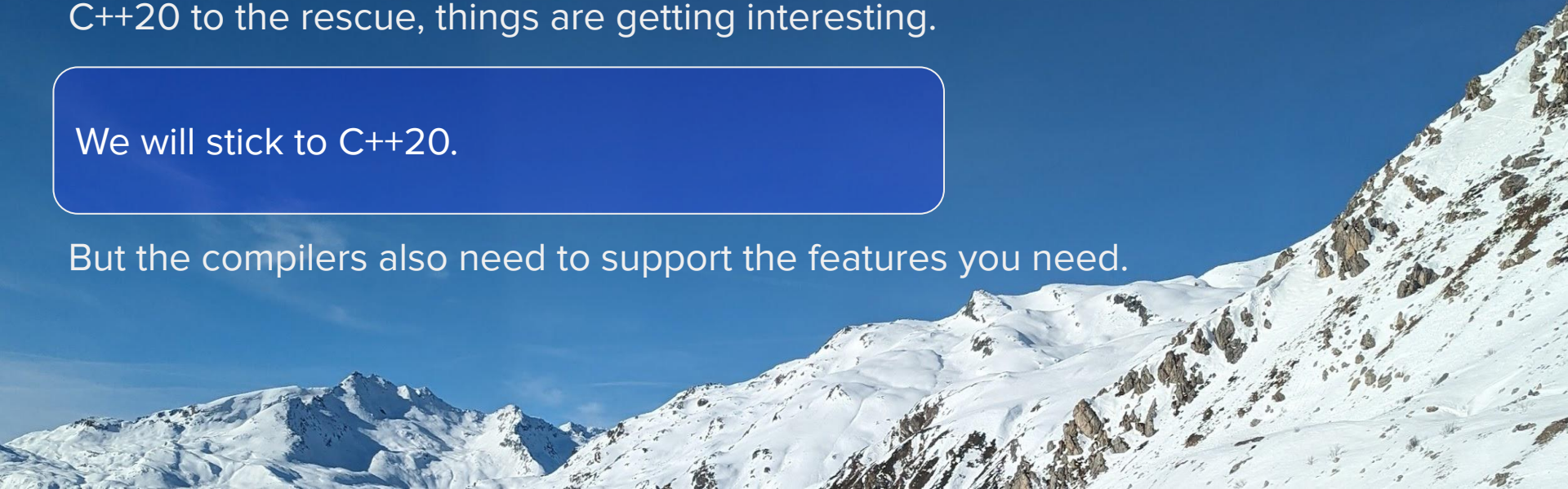
C++14 had better constexpr - but it was still lame.

C++17 had better constexpr - but it was still lame.

C++20 to the rescue, things are getting interesting.

We will stick to C++20.

But the compilers also need to support the features you need.



# The basics

A **constexpr** initialization is a const, at compile time.

In technical terms: A literal type, that is immediately initialized through a const expressions.



# The basics

A **constexpr** initialization is a const, at compile time.

In technical terms: A literal type, that is immediately initialized through a const expressions.

A **constexpr** function is one that **may** be executed at compile time - it is complicated...

In technical terms... do we have that much time?



# The basics

A **constexpr** initialization is a const, at compile time.

In technical terms: A literal type, that is immediately initialized through a const expressions.

A **constexpr** function is one that **may** be executed at compile time - it is complicated...

In technical terms... do we have that much time?

A **constexpr** function is a function that **must** be handled by the compiler. It always produces something usable for a **constexpr**.

Maybe just write the result?

Side Effect: Since the compiler handled it. We know it worked unless there was a compile error.



# The basics

A **constexpr** initialization is a const, at compile time.

In technical terms: A literal type, that is immediately initialized through a const expressions.

A **constexpr** function is one that **may** be executed at compile time - it is complicated...

In technical terms... do we have that much time?

A **constexpr** function is a function that **must** be handled by the compiler. It always produces something usable for a **constexpr**.

Maybe just write the result?

Side Effect: Since the compiler handled it. We know it worked unless there was a compile error.

A **constexpr** initialization **guarantees** static initialization.



# A constexpr function requires

- The return value and the parameters must be a LiteralType (until C++23)



# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType



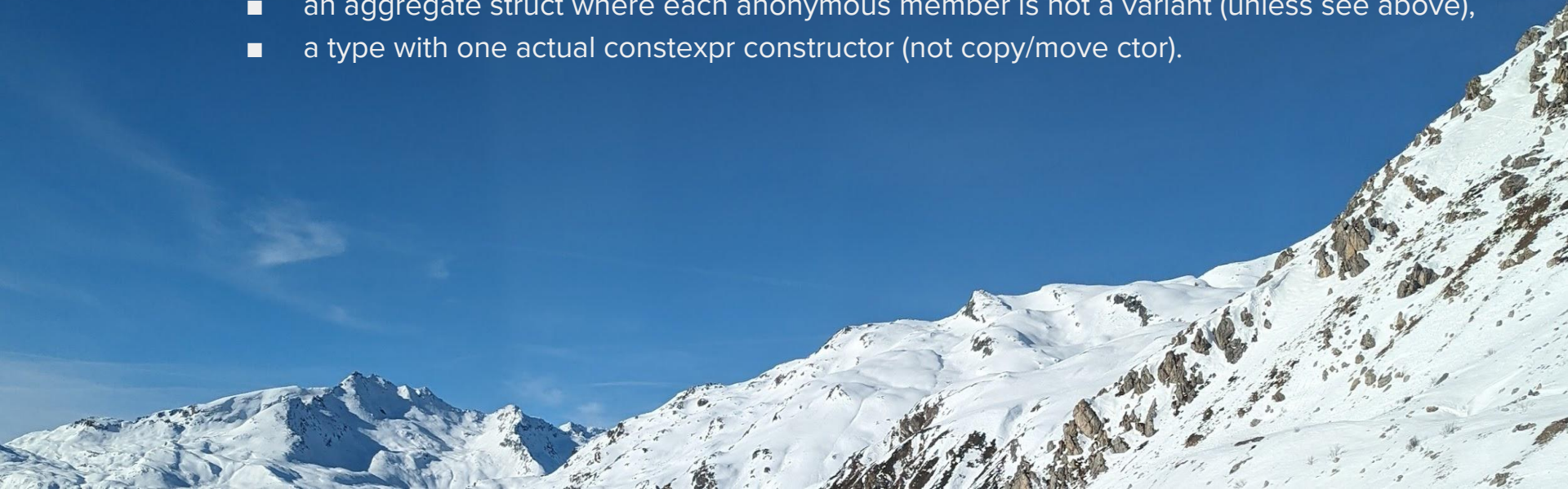
# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial ( < C++20 ) constexpr ( >= C++20 ) destructor



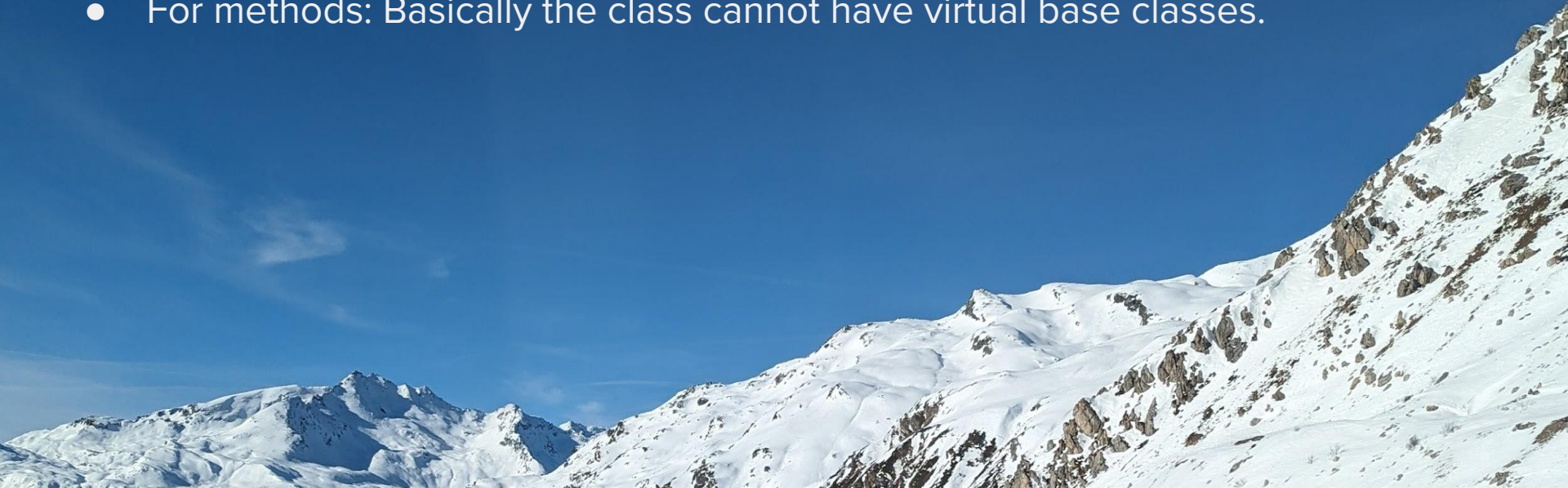
# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial constexpr destructor (\*), AND that is one of:
    - a lambda type,
    - an aggregate union without variant members (unless there is a non-volatile LiteralType),
    - an aggregate struct where each anonymous member is not a variant (unless see above),
    - a type with one actual constexpr constructor (not copy/move ctor).



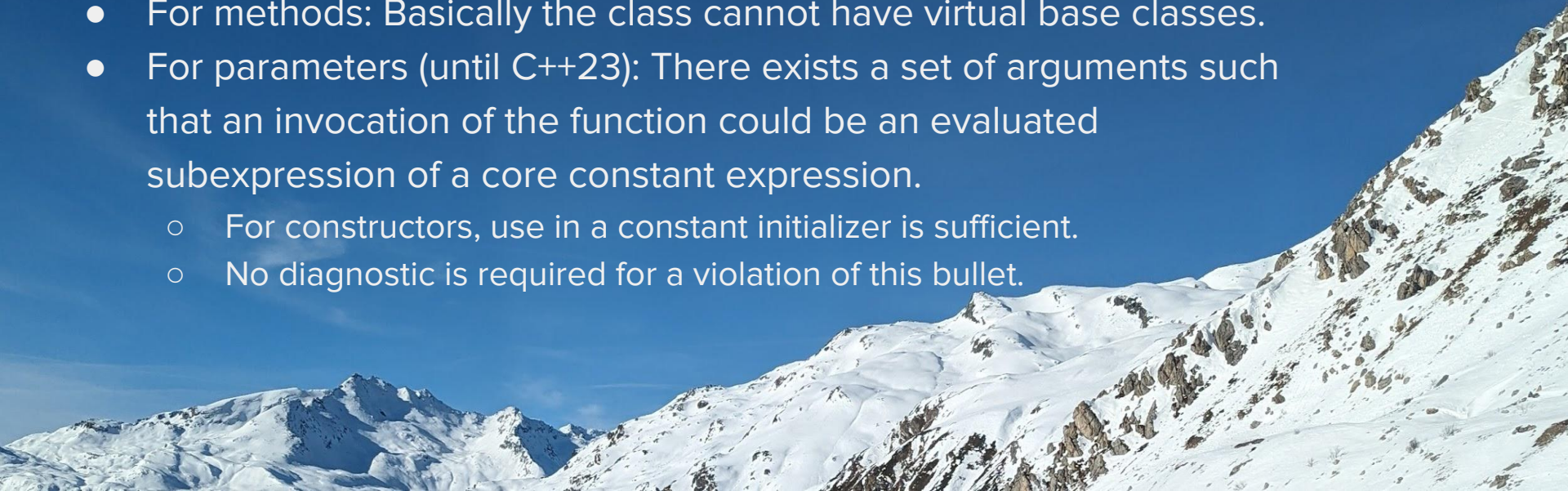
# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial constexpr destructor (\*), AND that is one of:
    - a lambda type, an aggregate (\*), OR
    - a type with an actual constexpr constructor (\*).
- For methods: Basically the class cannot have virtual base classes.



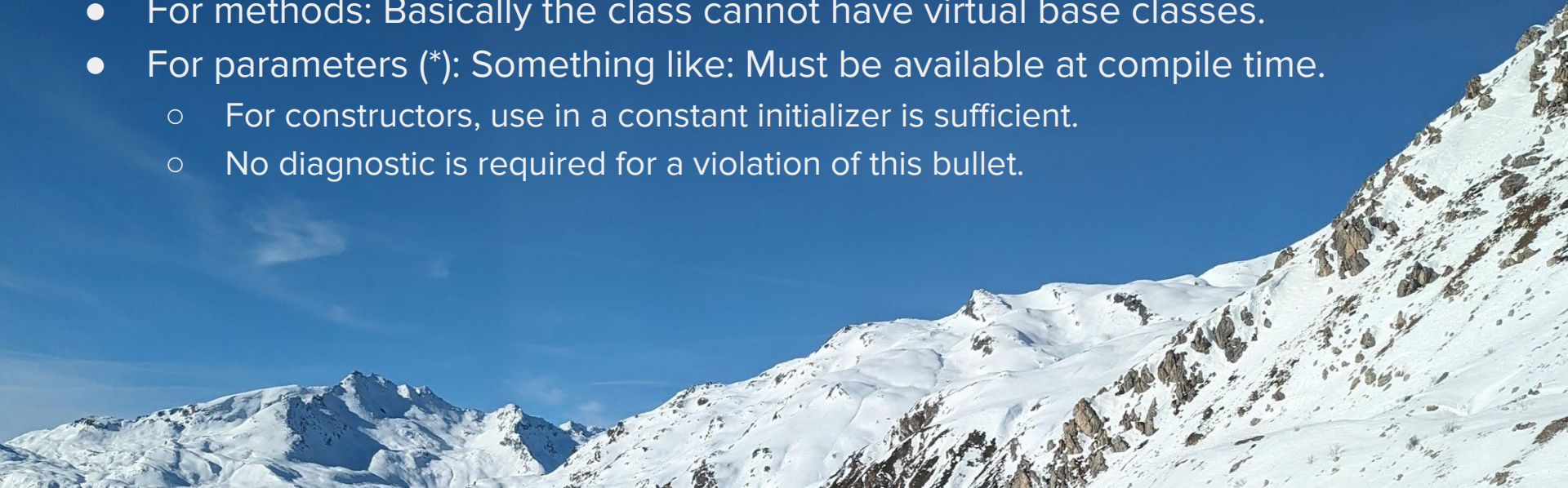
# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial constexpr destructor (\*), AND that is one of:
    - a lambda type, an aggregate (\*), OR
    - a type with an actual constexpr constructor (\*).
- For methods: Basically the class cannot have virtual base classes.
- For parameters (until C++23): There exists a set of arguments such that an invocation of the function could be an evaluated subexpression of a core constant expression.
  - For constructors, use in a constant initializer is sufficient.
  - No diagnostic is required for a violation of this bullet.



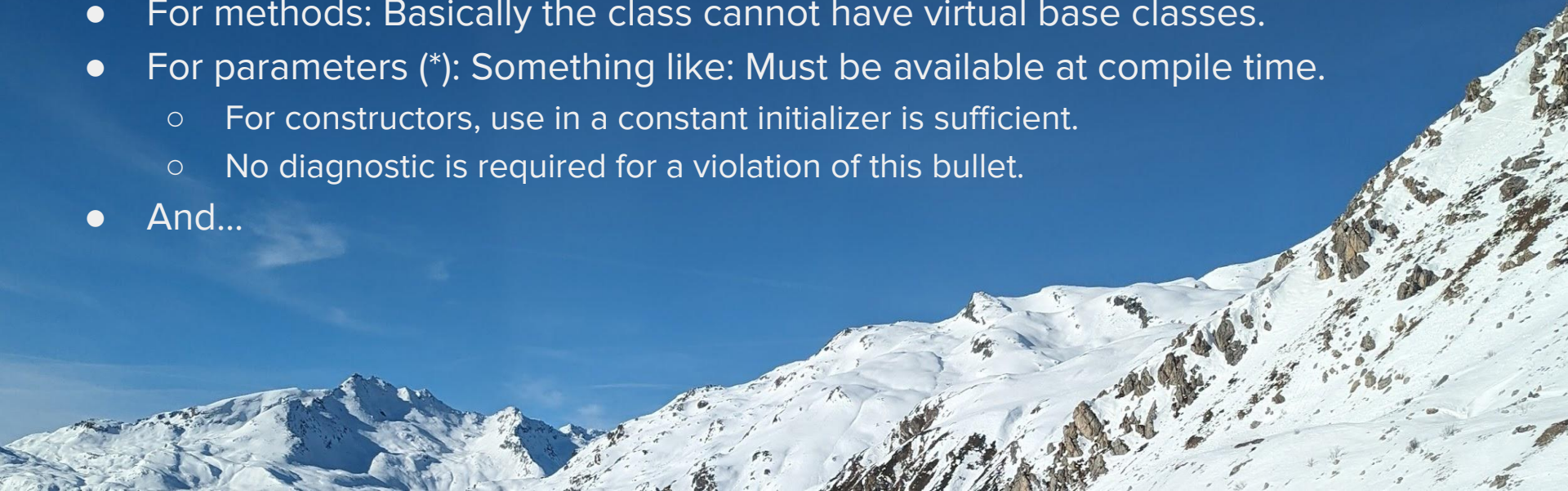
# A constexpr function requires

- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial constexpr destructor (\*), AND that is one of:
    - a lambda type, an aggregate (\*), OR
    - a type with an actual constexpr constructor (\*).
- For methods: Basically the class cannot have virtual base classes.
- For parameters (\*): Something like: Must be available at compile time.
  - For constructors, use in a constant initializer is sufficient.
  - No diagnostic is required for a violation of this bullet.



# A constexpr function requires

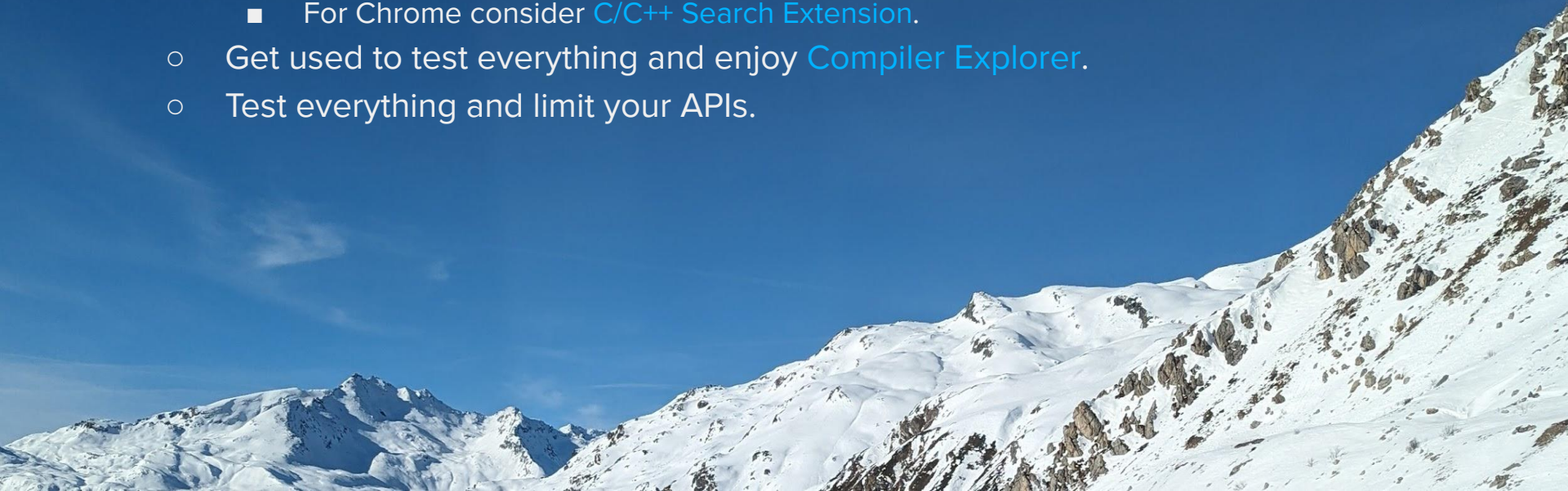
- The return value and the parameters must be a LiteralType (\*).
  - LiteralType: void, scalar, reference, array-of-LiteralType, OR
  - an object with trivial constexpr destructor (\*), AND that is one of:
    - a lambda type, an aggregate (\*), OR
    - a type with an actual constexpr constructor (\*).
- For methods: Basically the class cannot have virtual base classes.
- For parameters (\*): Something like: Must be available at compile time.
  - For constructors, use in a constant initializer is sufficient.
  - No diagnostic is required for a violation of this bullet.
- And...



# Side Note



- C++ ~~sucks~~ is awesome but, we are getting too complicated.
  - C++20: 1834 Pages, 6.7 MB (as a PDF).
  - Ensure you always have the standard at hand.
  - Have fast indexed access to the standard, thanks for [cppreference.com](https://cppreference.com)!
    - If you use VSCE consider [Cpp Reference by Guyutongxue](#).
    - For Chrome consider [C/C++ Search Extension](#).
  - Get used to test everything and enjoy [Compiler Explorer](#).
  - Test everything and limit your APIs.



# Side Note

Test everything and limit your APIs.

A photograph of a snow-covered mountain range under a clear blue sky. The mountains are rugged and covered in a thick layer of snow, with some rocky outcrops visible. The sky is a deep, clear blue with a few wispy clouds near the horizon.

# What to do? Silly stuff...

```
#include <string_view>
```

```
constexpr std::string_view text = "constant";
```



# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time

int foo() { volatile int result = 7; return result; } // volatile prevent constexpr

int main() {
    static constexpr int x = twice(8);
    // static constexpr int y = foo(); Error
    static constexpr int z = triple(4);
    int three = triple(z); // Does not accept `x`: not const
    return x + y + twice(foo()); // Maybe=> Cannot use triple.
}
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time

int foo() { volatile int result = 7; return result; } // volatile prevent constexpr

int main() {
    static constexpr int x = twice(8);
    // static constexpr int y = foo(); Error
    static constexpr int z = triple(4);
    int three = triple(z); // Does not accept `x`: not const
    return x + y + twice(foo()); // Maybe=> Cannot use triple.
}
```

```
Compile with GCC -O1
foo():
    mov    DWORD PTR [rsp-4], 7
    mov    eax, DWORD PTR [rsp-4]
    ret
main:
    mov    DWORD PTR [rsp-4], 7
    mov    eax, DWORD PTR [rsp-4]
    lea   eax, [rax+rax+28]
    ret
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time

int foo() { volatile int result = 7; return result; } // volatile prevent constexpr

int main() {
    static constexpr int x = twice(8);
    // static constexpr int y = foo(); Error
    static constexpr int z = triple(4);
    int three = triple(z); // Does not accept `x`: not const
    return x + y + twice(foo()); // Maybe=> Cannot use triple
}
```

Compile with GCC -O1

foo():

```
mov  DWORD PTR [rsp-4], 7
mov  eax, DWORD PTR [rsp-4]
ret
```

main:

```
mov  DWORD PTR [rsp-4], 7
mov  eax, DWORD PTR [rsp-4]
lea  eax, [rax+rax+28]
ret
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time
```

```
constexpr int foo() { /*volatile*/ int result = 7; return result; }
```

```
int main() {
    static constexpr int x = twice(8);

    static constexpr int z = triple(4);

    static constexpr auto r = x + z + twice(foo());
    return r;
}
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time
```

```
constexpr int foo() { /*volatile*/ int result = 7; return result; }
```

```
int main() {
    static constexpr int x = twice(8);

    static constexpr int z = triple(4);

    static constexpr auto r = x + z + twice(foo());
    return r;
}
```

Compile with GCC -O0

```
main:
    push    rbp
    mov     rbp, rsp
    mov     eax, 42
    pop     rbp
    ret
```

Compile with GCC -O1

```
main:
    mov     eax, 42
    ret
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // Maybe
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time

constexpr int foo() { /*volatile*/ int result = 7; return result; }

int main() {
    static constexpr int x = twice(8);

    static constexpr int z = triple(4);

    return [&]() constexpr { return x + z + twice(foo()); }();
}
```

# What to do? Silly stuff...



```
constexpr int twice(int v) { return 2 * v; } // `v` must be compile-time
constexpr int triple(int v) { return 3 * v; } // `v` must be compile-time
```

```
constexpr int foo() { /*volatile*/ int result = 7; return result; }
```

```
int main() {
    static constexpr int x = twice(8);

    static constexpr int z = triple(4);

    return [&]() constexpr { return x + z + twice(foo()); }();
}
```

# Verify constexpr works?

How do we know that "maybe" translates into compile-time?

- Short: we don't know.
- Compiling with `-O0` could be an indicator.
- An optimizing compiler may result in the same code reductions.



# A hash function



```
uint64_t GetSimpleHash(std::string_view data) {
    constexpr uint64_t kArbitrary = 5'008'709'998'333'326'415ULL, kPrimeNum10k = 104'729ULL;
    std::size_t len = data.length();
    uint64_t result = kArbitrary + len, add = 0;
    const char* str = data.data(), *end = data.end() - 3;
    while (str < end) {
        std::memcpy(&add, str, 4);
        result = result * 6'571 ^ ((17 * add + (add >> 16U)) ^ (result >> 32U));
        str += 4;
    }
    switch (data.length() % 4) {
        case 3: add = 0; std::memcpy(&add, str, 3); return result * 193 + kPrimeNum10k * add;
        case 2: add = 0; std::memcpy(&add, str, 2); return result * 193 + kPrimeNum10k * add;
        case 1: add = 0; std::memcpy(&add, str, 1); return result * 193 + kPrimeNum10k * add;
        default: break;
    }
    return result;
}
```

# A hash function



```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    constexpr uint64_t kArbitrary = 5'008'709'998'333'326'415ULL, kPrimeNum10k = 104'729ULL;
    std::size_t len = data.length();
    uint64_t result = kArbitrary + len, add = 0;
    const char* str = data.data(), *end = data.end() - 3;
    while (str < end) {
        std::memcpy(&add, str, 4);
        result = result * 6'571 ^ ((17 * add + (add >> 16U)) ^ (result >> 32U));
        str += 4;
    }
    switch (data.length() % 4) {
        case 3: add = 0; std::memcpy(&add, str, 3); return result * 193 + kPrimeNum10k * add;
        case 2: add = 0; std::memcpy(&add, str, 2); return result * 193 + kPrimeNum10k * add;
        case 1: add = 0; std::memcpy(&add, str, 1); return result * 193 + kPrimeNum10k * add;
        default: break;
    }
    return result;
}
```

# A hash function



```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    constexpr uint64_t kArbitrary = 5'008'709'998'333'326'415ULL, kPrimeNum10k = 104'729ULL;
    std::size_t len = data.length();
    uint64_t result = kArbitrary + len, add = 0;
    const char* str = data.data(), *end = data.end() - 3;
    while (str < end) {
        std::memcpy(&add, str, 4);
        result = result * 6'571 ^ ((17 * add + (add >> 16U)) ^ (result >> 32U));
        str += 4;
    }
    switch (data.length() % 4) {
        case 3: add = 0; std::memcpy(&add, str, 3); return result * 193 + kPrimeNum10k * add;
        case 2: add = 0; std::memcpy(&add, str, 2); return result * 193 + kPrimeNum10k * add;
        case 1: add = 0; std::memcpy(&add, str, 1); return result * 193 + kPrimeNum10k * add;
        default: break;
    }
    return result;
}
```

ERROR: non-constexpr function 'memcpy'

# A hash function



```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {  
    constexpr uint64_t kArbitrary = 5'008'709'998'333'326'415ULL, kPrimeNum10k = 104'729ULL;  
    std::size_t len = data.length(), pos = 0;  
    uint64_t result = kArbitrary + len;  
    while (len >= 4) {  
        uint64_t add = data[pos++];  
        add += data[pos++] << 8U;  
        add += data[pos++] << 16U;  
        add += data[pos++] << 24U;  
        result = result * 6'571 ^ ((17 * add + (add >> 16U)) ^ (result >> 32U));  
        len -= 4;  
    }  
    uint64_t add = 0;  
    switch (len) {  
        case 3: add += data[pos + 2] << 16U;  
        case 2: add += data[pos + 1] << 8U;  
        case 1: add += data[pos]; result * 193 + kPrimeNum10k * add;  
        default: return result;  
    }  
}
```

The **constexpr** variant must produce a compile-time constant.

If it compiles, we know it DID work.

# A hash function



```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    constexpr uint64_t kArbitrary = 5'008'709'998'333'326'415ULL, kPrimeNum10k = 104'729ULL;
    std::size_t len = data.length(), pos = 0;
    uint64_t result = kArbitrary + len;
    while (len >= 4) {
        uint64_t add = data[pos++];
        add += data[pos++] << 8U;
        add += data[pos++] << 16U;
        add += data[pos++] << 24U;
        result = result * 6'571 ^ ((17 * add + (add >> 16U)) ^ (result >> 32U));
        len -= 4;
    }
    uint64_t add = 0;
    switch (len) {
        case 3: add += data[pos + 2] << 16U;
        case 2: add += data[pos + 1] << 8U;
        case 1: add += data[pos]; result * 193 + kPrimeNum10k * add;
        default: return result;
    }
}
```

The **constexpr** variant may produce a constant.

# A hash function



```
namespace compile_time {
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

namespace run_time {
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}
```

Why not provide two versions? And let the developers use the correct version.

# A hash function



```
namespace compile_time {  
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}  
  
namespace run_time {  
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}
```

Why not provide two versions? And let the developers use the correct version. And **be assured** they will always update the code whenever more compile-time evaluation is possible :-)

# A hash function



```
namespace compile_time {
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

namespace run_time {
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    if (std::is_constant_evaluated()) {
        return compile_time::GetSimpleHash(data);
    } else {
        return run_time::GetSimpleHash(data);
    }
}
```

# A hash function



```
namespace compile_time {
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

namespace run_time {
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    if (std::is_constant_evaluated()) {
        return compile_time::GetSimpleHash(data);
    } else {
        return run_time::GetSimpleHash(data);
    }
}
```

You may not use `std::is_constant_evaluated` in a `constexpr if` statement. The answer would be always true.

# A hash function



```
namespace compile_time {
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

namespace run_time {
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    if (std::is_constant_evaluated()) {
        return compile_time::GetSimpleHash(data);
    } else {
        return run_time::GetSimpleHash(data);
    }
}
```

You may not use `std::is_constant_evaluated` in a `constexpr if` statement. The answer would be **always true**. Logically **IF** you execute it at compile time, then it is executed at compile time.

# A hash function



```
namespace compile_time {  
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}
```

You **cannot** use **constexpr** here. Because below **may not** use **constexpr-if**.

```
namespace run_time {  
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}
```

```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {  
    if (std::is_constant_evaluated()) {  
        return compile_time::GetSimpleHash(data);  
    } else {  
        return run_time::GetSimpleHash(data);  
    }  
}
```

You **may not** use **std::is\_constant\_evaluated** in a **constexpr if statement**. The answer would be **always true**. Logically **IF** you execute it at compile time, then it is executed at compile time.

# A hash function: C++20



```
namespace compile_time {  
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}
```

You **cannot** use **constexpr** in C++20.

```
namespace run_time {  
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }  
}
```

```
inline constexpr uint64_t GetSimpleHash(std::string_view data) {  
    if (std::is_constant_evaluated()) {  
        return compile_time::GetSimpleHash(data);  
    } else {  
        return run_time::GetSimpleHash(data);  
    }  
}
```

It is possible to detect compile time evaluation.

But functions called at compile time must be **constexpr** functions.

# A hash function: C++23



```
namespace compile_time {
    inline constexpr uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

namespace run_time {
    inline uint64_t GetSimpleHash(std::string_view data) { /* ... */ }
}

inline constexpr uint64_t GetSimpleHash(std::string_view data) {
    if constexpr {
        return compile_time::GetSimpleHash(data);
    } else {
        return run_time::GetSimpleHash(data);
    }
}
```

C++23 has **if-constexpr**

Compile time functions can now be marked **constexpr**.

[Clang >= 17; Not in GCC 14]

# Objects



# Objects

Objects can be constructed at compile time.

Some restrictions apply.



# Object Initialization



```
#include <iostream>

struct Test {
    constexpr Test() = default;
    ~Test() {
        --v;
        std::cout << v << "\n";
    }
    static inline int v = 42;
};

int main() {
    static constexpr Test test;

    return Test::v;
}
```

Returns: 42

Prints: 41

This works because:

- The constructor allows constant init.
  - It is **defaulted**: it does not need **constexpr**.
  - Write it out anyway to document intent.
- The destructor can be non-constant.
- The variable has static storage.
- The variable is declared **constexpr**, it **cannot** be declared **constexpr**.

# Object Initialization



```
#include <iostream>

struct Test {
    constexpr Test() = default;
    constexpr ~Test() {
        // --v;
        // std::cout << v << "\n";
    }
    int v = 42;
};

int main() {
    constexpr Test test;

    return test.v;
}
```

Returns 42

This works because:

- The constructor allows constant init.
  - It is **defaulted**: it does not need **constexpr**.
  - Write it out anyway to document intent.
- The destructor is **constexpr**.
- The variable does not need **static** storage.

# Object Initialization



```
#include <iostream>

struct Test {
    constexpr Test() = default;
    constexpr ~Test() = default;

    int v = 42;
};

int main() {
    static constexpr Test test1;
    constexpr Test test2;
    return test::v;
}
```

Returns 42

This works because:

- The constructor allows constant init.
  - It is **defaulted**: it does not need **constexpr**.
  - Write it out anyway to document intent.
- The destructor is **constexpr**.
  - It **cannot** be declared **constexpr**.
  - It is **defaulted**: it does not need **constexpr**.
  - Write it out anyway to document intent.
- Variables can be:
  - **static constexpr**
  - **constexpr**

**This talk is about constexpr.**



No more constinit & consteval after this

~~constexpr~~

~~constinit~~

# Container

Let's store some stuff...



# Container

Containers need space for the objects.

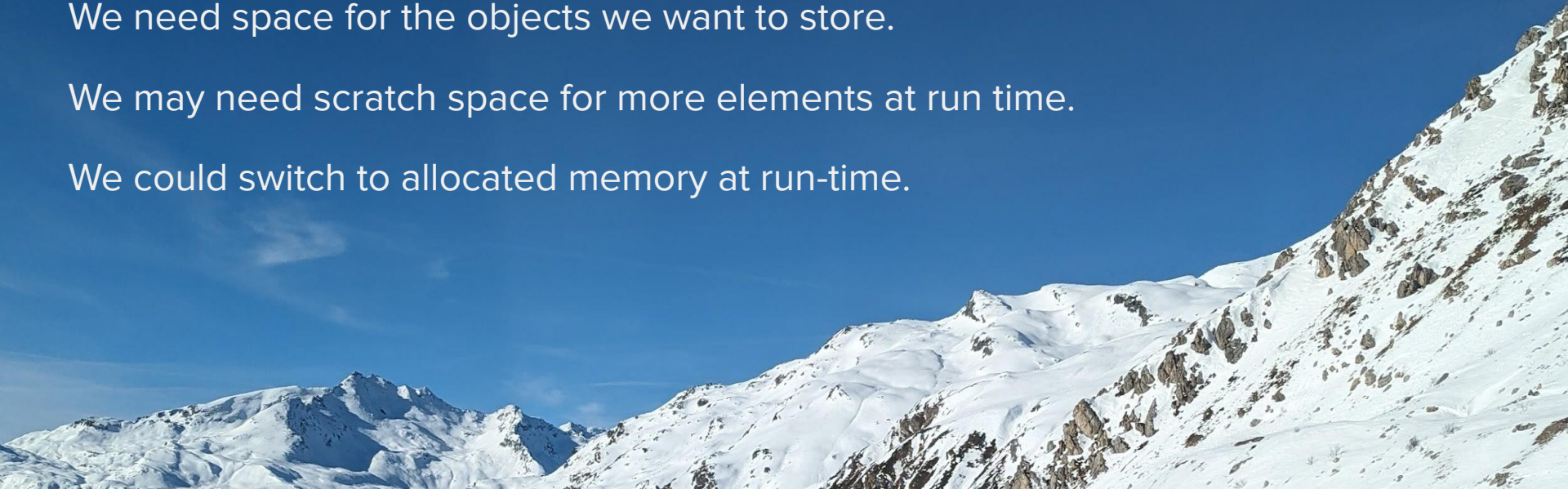
C++20 does not allow static variables in constexpr functions.

C++20 does however allow constexpr construction and destruction.

We need space for the objects we want to store.

We may need scratch space for more elements at run time.

We could switch to allocated memory at run-time.



# Container

Containers need space for the objects.

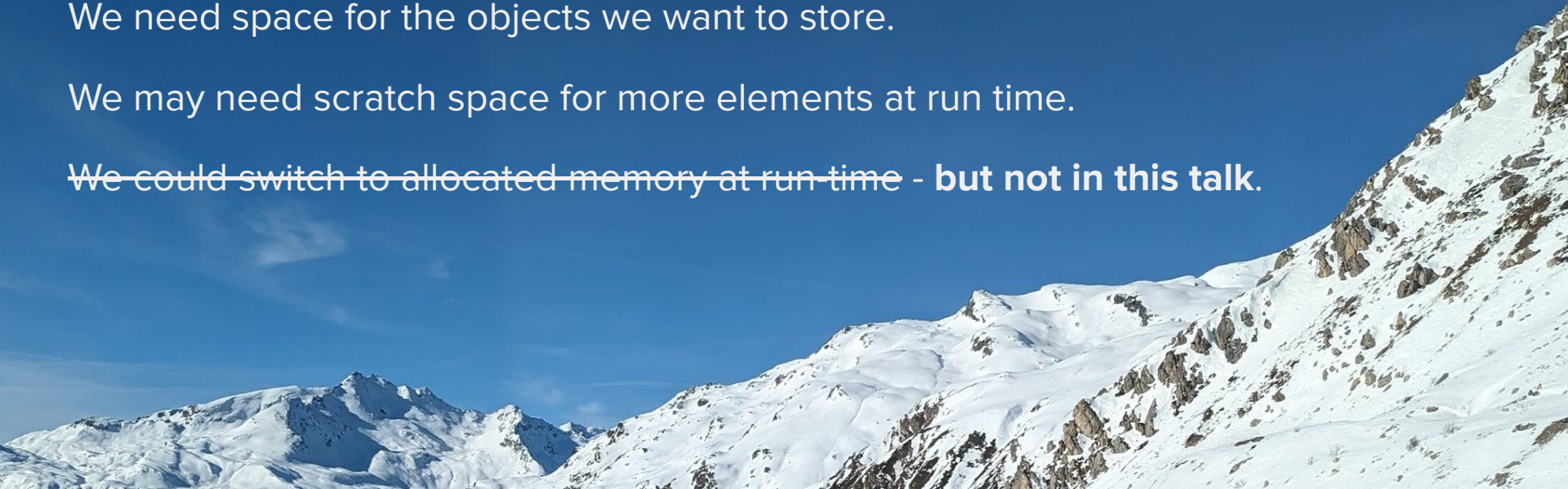
C++20 does not allow static variables in constexpr functions.

C++20 does however allow constexpr construction and destruction.

We need space for the objects we want to store.

We may need scratch space for more elements at run time.

~~We could switch to allocated memory at run time~~ - **but not in this talk.**



# std::array

```
namespace std {  
  
template <class T, size_t N>  
struct array {  
    using value_type = T;  
    // ...  
    // implicit constructor  
    // implicit destructor  
    // ...  
}  
  
} // namespace std
```

- A vector with constant space reservation.
- Implicit con-/destructor allow constexpr.
- Initialization is annoying.

# std::array



```
#include <array>
#include <string_view>
```

```
constexpr std::array<std::string_view, 20>
```

```
lines{
    "Bakerloo",           "Central",
    "Circle",            "District",
    "Hammersmith & City", "Jubilee",
    "Metropolitan",      "Northern",
    "Piccadilly",        "Victoria",
    "Waterloo & City",
};
```

```
int main() {
    return lines.size();
}
```

- A vector with constant space reservation.
- Implicit con-/destructor allow constexpr.
- `std::string` type supports constexpr.
  - But it largely does not work.
- `std::string_view` works perfectly fine.
- You can oversize a `std::array`.
  - Additional elements get default initialized.
- You cannot ask for less space!
  - That would be a compile time error.
- Initialization is annoying.

# std::array

```
namespace std {  
  
template <class T, size_t N>  
struct array {  
    using value_type = T;  
    // ...  
    // implicit constructor  
    // implicit destructor  
    // ...  
}  
  
template <class T, size_t N>  
constexpr array<remove_cv_t<T>, N>  
to_array(T (&a)[N]);  
  
template <class T, size_t N>  
constexpr array<remove_cv_t<T>, N>  
to_array(T (&&a)[N]);  
  
} // namespace std
```

- A vector with constant space reservation.
- Implicit con-/destructor allow constexpr.
- `std::string` type supports constexpr.
  - But it largely does not work.
- `std::string_view` works perfectly fine.
- You can oversize a `std::array`.
  - Additional elements get default initialized.
- You cannot ask for less space!
  - That would be a compile time error.
- Initialization is annoying - before C++20.

# std::array



```
#include <array>
#include <string_view>

int main() {
    constexpr auto lines =
        std::to_array<std::string_view>({
            "Bakerloo",
            "Central",
            "Circle",
            "District",
            "Hammersmith & City",
            "Jubilee",
            "Metropolitan",
            "Northern",
            "Piccadilly",
            "Victoria",
            "Waterloo & City",
        });
    return lines.size();
}
```

- A vector with constant space reservation.
- Implicit con-/destructor allow constexpr.
- `std::string` type supports constexpr.
  - But it largely does not work.
- `std::string_view` works perfectly fine.
- You can oversize a `std::array`.
  - Additional elements get default initialized.
- You cannot ask for less space!
  - That would be a compile time error.
- Initialization is annoying - before C++20.
- The example code compiles to...

# std::array



```
#include <array>
#include <string_view>

int main() {
    constexpr auto lines =
        std::to_array<std::string_view>({
            "Bakerloo",
            "Central",
            "Circle",
            "District",
            "Hammersmith & City",
            "Jubilee",
            "Metropolitan",
            "Northern",
            "Piccadilly",
            "Victoria",
            "Waterloo & City",
        });
    return lines.size();
}
```

- A vector with constant space reservation.
- Implicit con-/destructor allow constexpr.
- `std::string` type supports constexpr.
  - But it largely does not work.
- `std::string_view` works perfectly fine.
- You can oversize a `std::array`.
  - Additional elements get default initialized.
- You cannot ask for less space!
  - That would be a compile time error.
- Initialization is annoying - before C++20.
- The example code compiles to:

```
mov     eax, 11
ret
```

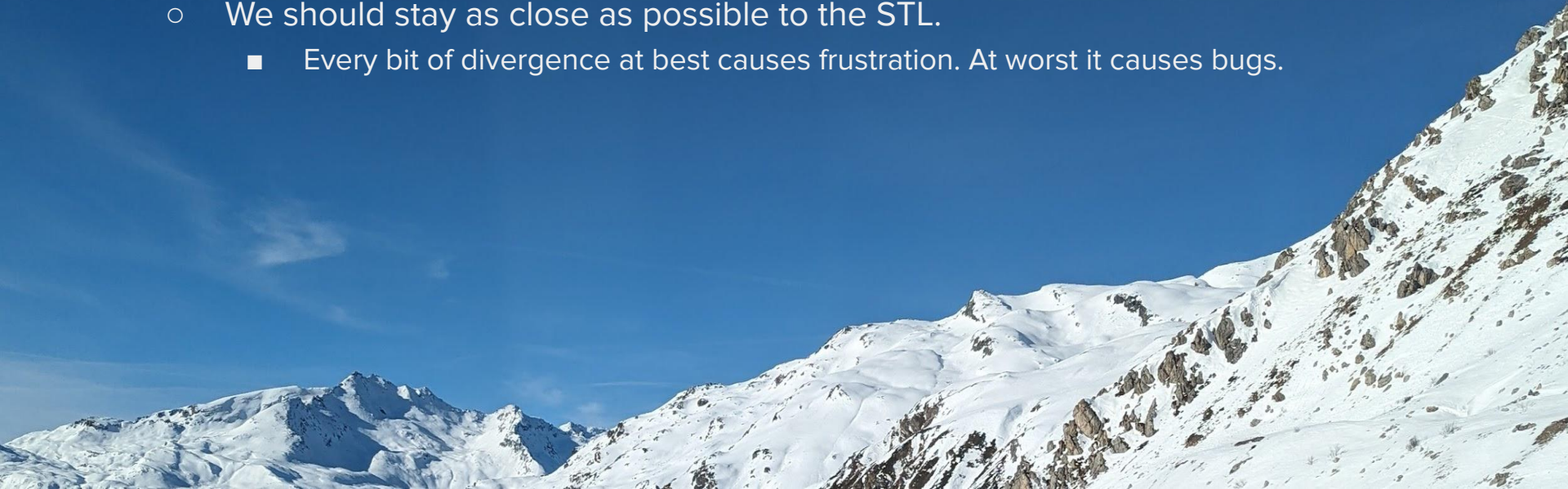
# std::array

- Type `std::array` is a bit inflexible.
- We must provide the correct number of elements.
  - Unless we want additional default initialized elements.
  - At least if not enough elements are asked for there is a compile time error.
  - Even better, since C++20 there is `std::to_array`.



# static constexpr uninitialized memory

- In order to combine compile time and run-time:
  - We will need some scratch space.
  - We will want uninitialized memory.
  - We might need to suppress destructors for some use cases.
    - We probably need some fancy templates.
  - We should stay as close as possible to the STL.
    - Every bit of divergence at best causes frustration. At worst it causes bugs.



# Scratch space



```
struct None final {};  
  
template <typename T>  
union Data {  
    constexpr Data() = default;  
  
    None none = {};  
    T data;  
};  
  
int main() {  
    Data d;  
    return d.data;  
}
```

Unions: abused & mysterious.

Well one member must be initialized.

The compiler knows which member is used.

Many compilers implement, as a non-standard language extension, the ability to read inactive members of a union. UBSan may disagree!

Type None should be secretly stashed away.

# Scratch space

```
template <typename T>
union Data {
    constexpr Data() noexcept : none{} {}

    constexpr Data(const Data&) noexcept = default;
    constexpr Data& operator=(const Data&) noexcept = default;

    constexpr Data(Data&&) noexcept = default;
    constexpr Data& operator=(Data&&) noexcept = default;

    constexpr ~Data() noexcept = default;

    None none;
    T data;
};
```

One member must be initialized.

Behold the [rule of 5/3/0](#).

- If T does not have it, then the union won't either - unless it is provided.

Trivial destructors are fine...

But non trivial destructors would prevent constexpr, so they need to be prevented.

# Scratch space

```
template <typename T>
union Data {
    constexpr Data() noexcept : none{} {}

    constexpr Data(const Data&) noexcept = default;
    constexpr Data& operator=(const Data&) noexcept = default;

    constexpr Data(Data&&) noexcept = default;
    constexpr Data& operator=(Data&&) noexcept = default;

    constexpr ~Data() noexcept
    requires(std::is_trivially_destructible_v<T>) = default;
    constexpr ~Data() noexcept
    requires(!std::is_trivially_destructible_v<T>) {};

    None none;
    T data;
};
```

One member must be initialized.

Behold the rule of 5/3/0.

- If T does not have it, then the union won't either - unless it is provided.

Trivial destructors are fine.

Non trivial destructors would prevent constexpr, so they need to be prevented.

# Practical: mbo Library



[helly25.com/mbo](https://helly25.com/mbo) | [github.com/helly25/mbo](https://github.com/helly25/mbo) | License: Apache 2.0

Used in production:

- We have a complex schema that is expressed in DataLog relations.
- We generate a lot of code.
- Since code is generated, as much as possible should be handled by compiler.
- Replacing all `std::vector/set/map` instances took about a week of work.
- The affected code contributes to roughly 33% of run time.
- Overall we saved about 1% run time.
- The change helped speed up service start-up time.

# mbo::container



## LimitedVector

- Like `std::array` and `std::vector` combined.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
  
- One sentinel element.
- Direct implementation.

# mbo::container



## LimitedVector

- Like `std::array` and `std::vector` combined.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
- One sentinel element.
- Direct implementation.

## LimitedSet

- Like `std::set` but with index access.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
- At least one element.
- Based on `LimitedOrdered`.

# mbo::container



## LimitedVector

- Like `std::array` and `std::vector` combined.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
  
- One sentinel element.
- Direct implementation.

## LimitedSet

- Like `std::set` but with index access.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
  
- At least one element.
- Based on `LimitedOrdered`.

## LimitedMap

- Like `std::map` but with index access.
- No allocators.
- Max reservation at compile time.
- Uninitialized memory.
- Optional destructor suppression.
  
- At least one element.
- Based on `LimitedOrdered`.

# LimitedVector



```
template<typename T,  
        std::size_t Capacity>  
requires(LimitedVectorValid<T>)  
class LimitedVector final {  
private:  
    struct None final {};  
    union Data { /* ... */ };  
  
public:  
    // ...  
  
private:  
    std::size_t size_{0};  
    Data values_[Capacity + 1];  
};
```

Needs a sentinel - mostly for ASAN.

Using `std::array` would have been better, but ASAN does not like it.

# LimitedVector



```
template<typename T,  
        auto CapacityOrOptions>  
requires(LimitedVectorValid<T>)  
class LimitedVector final {  
private:  
    struct None final {};  
    union Data { /* ... */ };  
  
public:  
    // ...  
  
private:  
    static constexpr std::size_t capacity...  
    std::size_t size_{0};  
    Data values_[capacity + 1];  
};
```

Needs a sentinel - mostly for ASAN.

Using `std::array` would have been better, but ASAN does not like it.

Size alone is not enough.

Adding boolean template arguments per option is bad as it prevents staying compatible with STL.

We can construct a helper type that allows both.

Using `auto` we can accept both a size and the helper type.

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <auto CapacityOrOptions>
requires (
    std::convertible_to<decltype(CapacityOrOptions), std::size_t> ||
    std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                Options>)
struct LimitedVector {
    static constexpr std::size_t capacity = 42; // FIX
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the **auto** argument type is valid.

Note that:

- CapacityOrOptions is a value not a type.
- The value may be const or some such.
- Use **remove\_cvref\_t** not decay\_t.

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <auto CapacityOrOptions>
requires (
    std::convertible_to<decltype(CapacityOrOptions), std::size_t> ||
    std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                Options>)
struct LimitedVector {
    static constexpr std::size_t capacity =
        std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                    Options>
        ? CapacityOrOptions.capacity
        : CapacityOrOptions;
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the auto argument type is valid.

Initialize the **capacity** member from either.

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <auto CapacityOrOptions>
requires (
    std::convertible_to<decltype(CapacityOrOptions), std::size_t> ||
    std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                Options>)
struct LimitedVector {
    static constexpr std::size_t capacity =
        std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                    Options>
        ? CapacityOrOptions.capacity
        : CapacityOrOptions;
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the auto argument type is valid.

Initialize the **capacity** member from either.

**Compile Error**

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <auto CapacityOrOptions>
requires (
    std::convertible_to<decltype(CapacityOrOptions), std::size_t> ||
    std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                Options>)
struct LimitedVector {
    static constexpr std::size_t capacity =
        std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                    Options>
        ? CapacityOrOptions.capacity
        : CapacityOrOptions;
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the auto argument type is valid.

Initialize the **capacity** member from either.

Using the ternary operator does of course not work. We do not know the type and the compiler gets one good side and either something that does not have a member capacity or something that cannot be converted.

**Compile Error**

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <auto CapacityOrOptions>
requires (
    std::convertible_to<decltype(CapacityOrOptions), std::size_t> ||
    std::same_as<std::remove_cvref_t<decltype(CapacityOrOptions)>,
                Options>)
struct LimitedVector {
    static constexpr Options options = Options(CapacityOrOptions);
    static constexpr std::size_t capacity = options.capacity;
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the auto argument type is valid.

Create an Options member from the template arg.

Initialize the capacity from the options.

Ensure the conversion is possible. That includes relying on the copy constructor as well as using helper functions.

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <typename T>
concept IsCapacityOrOptions =
    std::convertible_to<T, std::size_t> ||
    std::same_as<std::remove_cvref_t<T>, Options>;

template <auto CapacityOrOptions>
requires (IsCapacityOrOptions<decltype(CapacityOrOptions)>)
struct LimitedVector {
    static constexpr Options options = Options(CapacityOrOptions);
    static constexpr std::size_t capacity = options.capacity;
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Initialize the capacity from the options.

Use a concept for ~~prosperity~~ readability.

- The concept cannot replace auto.
- We still need a requires line.

We are overly strict here as we require the exact Options type. That is done to limit the API surface.

# Mixed auto template argument



```
#include <concepts>

struct Options { std::size_t capacity = 0; bool option = false; };

template <typename T>
concept IsCapacityOrOptions =
    std::convertible_to<T, std::size_t> ||
    std::same_as<std::remove_cvref_t<T>, Options>;

template <auto CapacityOrOptions>
requires (IsCapacityOrOptions<decltype(CapacityOrOptions)>)
struct LimitedVector {
    static constexpr Options options{CapacityOrOptions};
    static constexpr std::size_t capacity{options.capacity};
};

int main() {
    return LimitedVector<25>::capacity
        + LimitedVector<Options{.capacity = 17}>::capacity;
}
```

Use a type to hold the capacity and the options.

Require that the auto argument type is valid.

Create an Options member from the template arg.

Initialize the capacity from the options.

Use a concept for ~~prosperity~~ readability.

Note: the members are **static constexpr**.

# LimitedVector

```
const T& operator[](int i) const noexcept {  
    assert(0 <= i && i < size);  
    return values_[i].data;  
}
```

```
T& push_back(T&& val) noexcept {  
    assert(size < capacity);  
    auto& ref{values_[size_++]};  
    new(&ref.data) T(std::forward<T>(val));  
    return ref.data;  
}
```

Access needs bounds checking.

# LimitedVector

```
const T& operator[](int i) const noexcept {  
    assert(0 <= i && i < size);  
    return values_[i].data;  
}
```

There is no constexpr?

```
T& push_back(T&& val) noexcept {  
    assert(size < capacity);  
    auto& ref{values_[size++]};  
    new(&ref.data) T(std::forward<T>(val));  
    return ref.data;  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept {  
    ABSL_LOG_IF(FATAL, 0 <= i && i < size);  
    return values_[i].data;  
}
```

```
T& push_back(T&& val) noexcept {  
    ABSL_LOG_IF(FATAL, size < capacity);  
    auto& ref{values_[size++]};  
    new(&ref.data) T(std::forward<T>(val));  
    return ref.data;  
}
```

Access needs bounds checking.

Using assert prevents constexpr.

Abseil logging allows constexpr.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept {  
    ABSL_LOG_IF(FATAL, 0 <= i && i < size);  
    return values_[i].data;  
}
```

```
T& push_back(T&& val) noexcept {  
    ABSL_LOG_IF(FATAL, size < capacity);  
    auto& ref{values_[size++]};  
    new(&ref.data) T(std::forward<T>(val));  
    return ref.data;  
}
```

Access needs bounds checking.

Using assert prevents constexpr.

Abseil logging allows constexpr:

We need run-time to produce **readable** failure messages.

We need compile-time to fail with **good enough** error messages.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept {  
    if (i < 0 || i >= size) {  
        throw std::runtime_error(__FILE__);  
    }  
    return values_[i].data;  
}
```

```
T& push_back(T&& val) noexcept {  
    if (size >= capacity) {  
        throw std::runtime_error(__FILE__);  
    }  
    auto& ref{values_[size++]};  
    new(&ref.data) T(std::forward<T>(val));  
    return ref.data;  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

But exceptions require to drop **`noexcept`**.

# LimitedVector



```
std::string error_message(  
    std::string_view file,  
    uint64_t line,  
    std::string_view msg) {  
    return std::format("{}:{} : {}", file, line, msg)  
}
```

Not constexpr.

```
constexpr const T& operator[](int i) const {  
    if (i < 0 || i >= size) {  
        throw std::runtime_error(error_message(  
            __FILE__, __LINE__, "Bad Index."));  
    }  
    return values[i].data;  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

The nice thing about using exceptions:

The error message can be constructed at run-time using a non `constexpr` function.

# LimitedVector - With Dragons Macros



```
static constexpr bool kRequireThrows = false;

#define MBO_REQUIRE(condition, message) \
    if constexpr (!kRequireThrows) { \
        ABSL_LOG_IF(FATAL, !(condition)) << message; \
    } else if (condition) { /* GOOD */ \
    } else throw std::runtime_error( \
        __FILE__ ":" __LINE__ " : " #condition ":" message)

constexpr const T&
operator[](int i) const noexcept(!kRequireThrows) {
    MBO_REQUIRE(i >= 0 && i < size, "Bad Index.");
    return values[i].data;
}
```

Access needs bounds checking.

Using assert prevents constexpr.

Abseil logging allows constexpr.

Exceptions work in constexpr functions.

Switching from crashing to exceptions:

- requires macros (pretty much),
- but enables conditional noexcept.

The Compiler Explorer link demonstrates the error handling in various compilers using only C++17.

# LimitedVector - With Dragons Macros



```
static constexpr bool kRequireThrows = false;

#define MBO_REQUIRE(condition, message) ...

constexpr const T&
operator[](int i) const noexcept(!kRequireThrows) {
    MBO_REQUIRE(i >= 0 && i < size, "Bad Index.");
    return values[i].data;
}

T& push_back(T&& val) noexcept(!kRequireThrows) {
    MBO_REQUIRE(size < capacity, "Called at capacity.");
    auto& ref{values_[size_++]};
    new(&ref.data) T(std::forward<T>(val));
    return ref.data;
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Switching from crashing to exceptions:

- Requires macros
- But enables conditional `noexcept`

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept {
    MBO_REQUIRE(0 <= i && i < size, "Bad Index.");
    return values_[i].data;
}

constexpr T& push_back(T&& val) noexcept {
    MBO_REQUIRE(size < capacity, "Called at capacity.");
    auto& ref{values_[size++]};
    std::construct_at(&ref.data, std::forward<T>(val));
    return ref.data;
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept {  
    MBO_REQUIRE(0 <= i && i < size, "Bad Index.");  
    return values_[i].data;  
}
```

```
constexpr T& push_back(T&& val) noexcept {  
    MBO_REQUIRE(size < capacity, "Called at capacity.");  
    auto& ref{values_[size++]};  
    std::construct_at(&ref.data, std::forward<T>(val));  
    return ref.data;  
}
```

```
constexpr T& push_back(const T& val) noexcept {/*_*/}
```

```
constexpr void pop_back() noexcept {  
    MBO_REQUIRE(size_ > 0, "Called at capacity.");  
    std::destroy_at(&values_[--size_].data);  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use **`std::destroy_at`**.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept;  
constexpr T& push_back(T&& val) noexcept;  
constexpr T& push_back(const T& val) noexcept;  
constexpr void pop_back() noexcept;
```

```
constexpr std::size_t size() const noexcept {  
    return size_;  
}
```

```
constexpr bool empty() const noexcept {  
    return size_ == 0;  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept;  
constexpr T& push_back(T&& val) noexcept;  
constexpr T& push_back(const T& val) noexcept;  
constexpr void pop_back() noexcept;  
constexpr std::size_t size() const noexcept;  
constexpr bool empty() const noexcept;
```

```
constexpr void clear() noexcept {  
    while (!empty()) { pop_back(); }  
}
```

```
constexpr void resize(std::size_t new_size) noexcept {  
    while (new_size < size()) { pop_back(); }  
    while (new_size > size()) { push_back({}); }  
}
```

```
constexpr void reserve(std::size_t size) noexcept {  
    MBO_REQUIRE(size <= Capacity, "Bad size.");  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

**Follow the given interface and build on top.**

# LimitedVector



```
constexpr const T& operator[](int i) const noexcept;  
constexpr T& push_back(T&& val) noexcept;  
constexpr T& push_back(const T& val) noexcept;  
constexpr void pop_back() noexcept;  
constexpr std::size_t size() const noexcept;  
constexpr bool empty() const noexcept;  
constexpr void clear() noexcept;  
constexpr void resize(std::size_t new_size) noexcept;  
constexpr void reserve(std::size_t size) noexcept;
```

```
constexpr LimitedVector() noexcept = default;
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

**One actual `constexpr` constructor!**

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor)) {
    clear();
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

**The destructor is problematic.**

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor)) {
    clear();
}
```

Does **not** call `clear`.

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

The destructor is problematic.

# LimitedVector



```
constexpr LimitedVector() noexcept = default;
```

```
constexpr ~LimitedVector() noexcept  
requires(Options::Has(Flag::kEmptyDestructor))  
= default;
```

There is no constexpr?

```
~LimitedVector() noexcept  
requires(!Options::Has(Flag::kEmptyDestructor)) {  
    clear();  
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

Options and destructor suppression!

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

constexpr ~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
         std::is_trivially_destructible_v<T>) {
    clear();
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

**Options and destructor suppression!**

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

constexpr ~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
         std::is_trivially_destructible_v<T>) {
    clear();
}
```

Requires trivial destructor?

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

Options and destructor suppression!

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

constexpr ~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
        std::is_trivially_destructible_v<T>) {
    clear();
}

~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
        !std::is_trivially_destructible_v<T>) {
    clear();
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

**Options and destructor suppression!**

# LimitedVector



```
constexpr LimitedVector() noexcept = default;

constexpr ~LimitedVector() noexcept
requires(Options::Has(Flag::kEmptyDestructor))
= default;

constexpr ~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
std::is_trivially_destructible_v<T>) {
    clear();
}

~LimitedVector() noexcept
requires(!Options::Has(Flag::kEmptyDestructor) &&
!std::is_trivially_destructible_v<T>) {
    clear();
}
```

Access needs bounds checking.

Using `assert` prevents `constexpr`.

Abseil logging allows `constexpr`.

Exceptions work in `constexpr` functions.

Replace `new` with `std::construct_at`.

Placement `new` is `constexpr` in C++2c.

We also need to use `std::destroy_at`.

Follow the given interface and build on top.

One actual `constexpr` constructor!

**Options and destructor suppression!**

# LimitedVector

Including options and additional concepts... some 1k lines later: LimitedVector is:

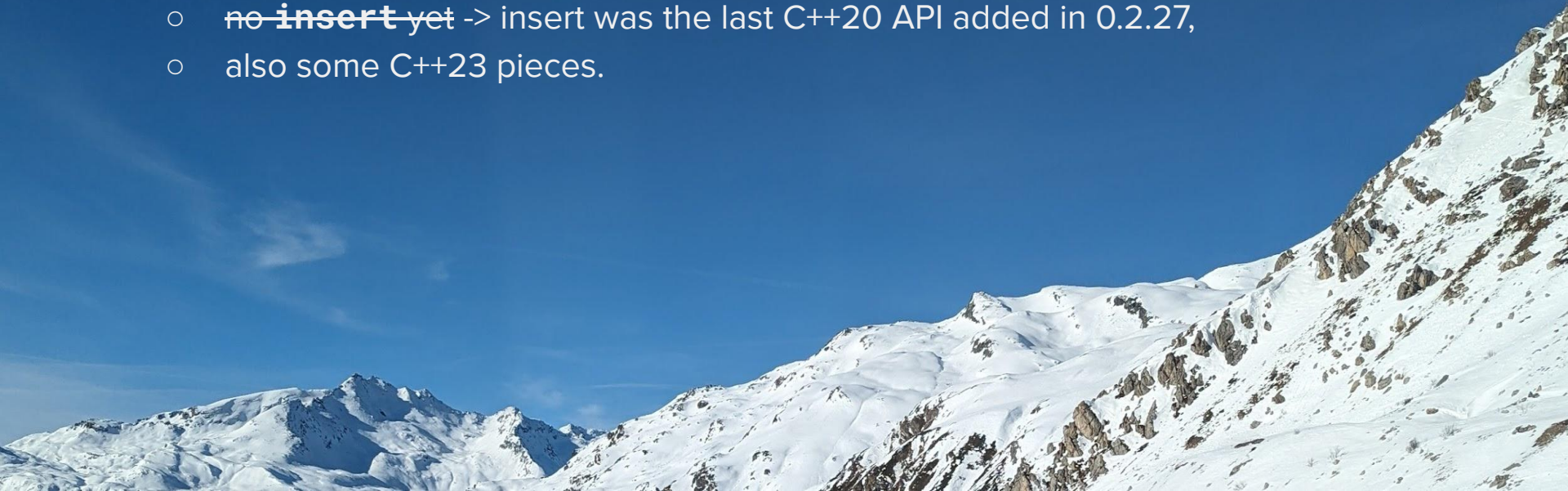
- constexpr safe,
- works at compile-time and at run-time,
- shares the C++20 interface.



# LimitedVector

Including options and additional concepts... some 1k lines later: LimitedVector is:

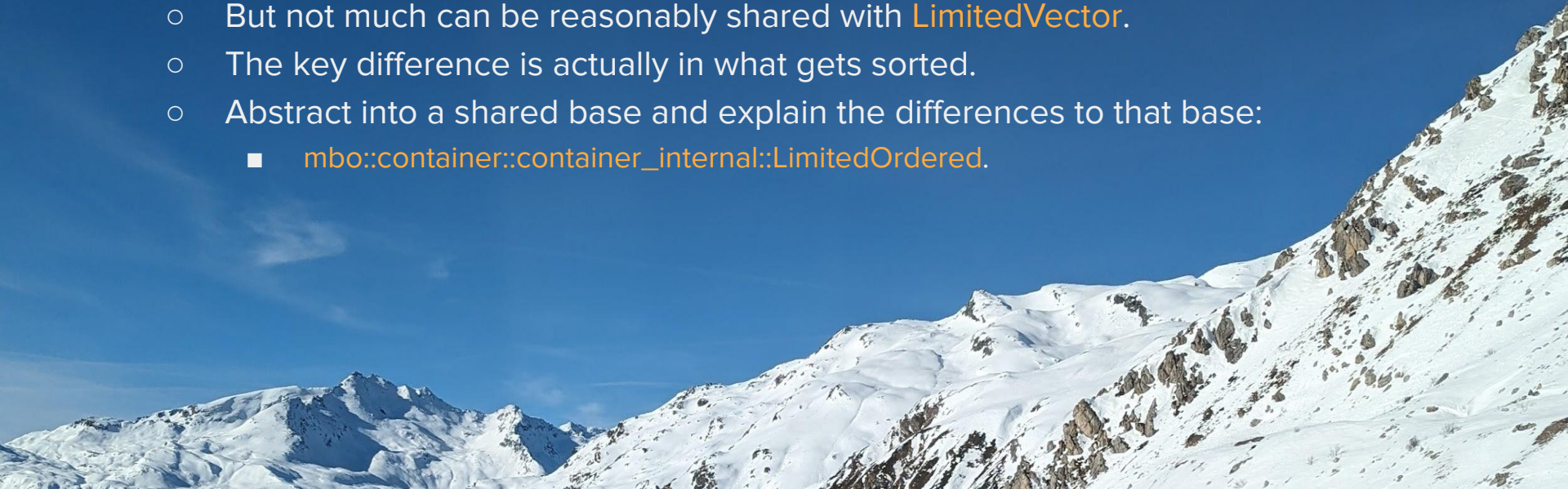
- constexpr safe,
- works at compile-time and at run-time,
- shares the C++20 interface,
  - ~~no insert~~ yet -> insert was the last C++20 API added in 0.2.27,
  - also some C++23 pieces.



# Next Up

`LimitedMap` and `LimitedSet`.

- Much of the same.
- Both types need sorting.
- Most of the code is the same between the two.
  - But not much can be reasonably shared with `LimitedVector`.
  - The key difference is actually in what gets sorted.
  - Abstract into a shared base and explain the differences to that base:
    - `mbo::container::container_internal::LimitedOrdered`.



# LimitedOrdered



```
template<
    typename Key,
    typename Mapped,
    typename Value,
    auto options,
    typename Compare = std::less<Key>>
requires(LimitedOrderedValid<Key, Mapped, Value>)
class LimitedOrdered;
```

LimitedOrdered is internal:

So do not worry about compliance.

# LimitedOrdered



```
template<std::forward_iterator It>
requires std::convertible_to<
    mbo::types::ForwardIteratorValueType<It>, Value>
constexpr LimitedOrdered(
    It first, It last, const Compare& comp = Compare()
) noexcept
: key_comp_(comp) {
while (first < last) {
    emplace(*first++);
}
}
```

A constructor... from iterators.

# LimitedOrdered



```
template<std::forward_iterator It>
requires std::convertible_to<
    mbo::types::ForwardIteratorValueType<It>, Value>
constexpr LimitedOrdered(
    It first, It last, const Compare& comp = Compare()
) noexcept(!kRequireThrows)
: key_comp_(comp) {
    if constexpr (Has(Flag::kRequireSortedInput)) {
        MBO_REQUIRE(std::is_sorted(first, last, key_comp_),
            "Input not sorted.");
        while (first < last) {
            std::construct_at(&values_[size_++].data, *first);
            ++first;
        }
    } else {
        while (first < last) {
            emplace(*first++);
        }
    }
}
```

A constructor... from iterators.

We have options... that allow to control optimizations.

# LimitedOrdered



...

A constructor... from iterators.

More constructors, destructors,  
iterators and much more...

# LimitedOrdered



```
constexpr iterator lower_bound(const Key& key) {  
    return std::lower_bound(begin(), end(), key, val_comp_);  
}
```

```
constexpr const_iterator lower_bound(const Key& key) const {  
    return std::lower_bound(begin(), end(), key, val_comp_);  
}
```

```
constexpr iterator upper_bound(const Key& key) {  
    return std::upper_bound(begin(), end(), key, val_comp_);  
}
```

```
constexpr const_iterator upper_bound(const Key& key) const {  
    return std::upper_bound(begin(), end(), key, val_comp_);  
}
```

A constructor... from iterators.

More constructors, destructors, iterators and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!

# LimitedOrdered



```
constexpr iterator lower_bound(const Key& key) {
    return std::lower_bound(begin(), end(), key, val_comp_);
}

constexpr const_iterator lower_bound(const Key& key) const {
    return std::lower_bound(begin(), end(), key, val_comp_);
}

constexpr iterator upper_bound(const Key& key) {
    return std::upper_bound(begin(), end(), key, val_comp_);
}

constexpr const_iterator upper_bound(const Key& key) const {
    return std::upper_bound(begin(), end(), key, val_comp_);
}

// Consider: __attribute__((always_inline))
```

A constructor... from iterators.

More constructors, destructors, iterators and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!

# LimitedOrdered



```
static constexpr size_type npos =
    static_cast<size_type>(-1);

constexpr std::size_t index_of(
    const Key& key) const {
    const_iterator it = lower_bound(key);
    return it == end() ||
        key_comp_(key, GetKey(*it))
            ? npos
            : it - begin();
}
```

A constructor... from iterators.

More constructors, destructors, iterators  
and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!
- But we can do better since we are contiguous!

# LimitedOrdered



```
static constexpr size_type npos =
    static_cast<size_type>(-1);

constexpr std::size_t index_of(
    const Key& key) const
requires(
    !kOptimizeIndexOf || (
        kOptimizeIndexOf &&
        !kCustomIndexOfBeyondUnroll &&
        Capacity > kUnrollMaxCapacity));
```

A constructor... from iterators.

More constructors, destructors, iterators  
and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!
- But we can do better since we are contiguous!
- Again we have options.

# LimitedOrdered



```
static constexpr size_type npos =
    static_cast<size_type>(-1);

constexpr std::size_t index_of(
    const Key& key) const
requires(
    !kOptimizeIndexOf || (
        kOptimizeIndexOf &&
        !kCustomIndexOfBeyondUnroll &&
        Capacity > kUnrollMaxCapacity));

// Here be dragons:
// The unrolled/optimized variants.
```

A constructor... from iterators.

More constructors, destructors, iterators  
and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!
- But we can do better since we are contiguous!
- Again we have options.

# LimitedOrdered



```
static constexpr size_type npos =
    static_cast<size_type>(-1);

constexpr std::size_t index_of(
    const Key& key) const
requires(
    !kOptimizeIndexOf || (
        kOptimizeIndexOf &&
        !kCustomIndexOfBeyondUnroll &&
        Capacity > kUnrollMaxCapacity));

// Here be dragons:
// The unrolled/optimized variants.

// Per unroll step ~20 loc.
// 32 unroll steps.
// Around 80 loc per function/variant.
```

A constructor... from iterators.

More constructors, destructors, iterators  
and much more...

Comparison:

- Given compare, we can call lower\_bound and upper\_bound.
- Using those two we can implement everything else!
- But we can do better since we are contiguous!
- Again we have options.

# LimitedOrdered::insert

```
constexpr iterator insert(  
    const_iterator pos, T&& value);
```

```
constexpr iterator insert(  
    const_iterator pos, size_type count, const T& value);
```

```
constexpr iterator insert(  
    const_iterator pos, const T& value);
```

```
template<typename InputIt>  
constexpr iterator insert(  
    const_iterator pos, InputIt first, InputIt last)
```

```
constexpr iterator insert(  
    const_iterator pos, std::initializer_list<T> list)
```

- Requires a position.
- Requires movement of elements.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(
    const_iterator pos, U&& value);

constexpr iterator insert(
    const_iterator pos, size_type count, const T& value);

constexpr iterator insert(
    const_iterator pos, const T& value);

template<typename InputIt>
requires(std::constructible_from<
    T, decltype(*std::declval<InputIt>())>)>
constexpr iterator insert(
    const_iterator pos, InputIt first, InputIt last)

template<std::constructible_from<T> U>
constexpr iterator insert(
    const_iterator pos, std::initializer_list<U> list)
```

- Requires a position.
- Requires movement of elements.
- We can apply **more constraints**.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(const_iterator pos, U&& value) {
    MBO_REQUIRE(size_ < Capacity, "Called at capacity.");
    MBO_REQUIRE(begin() <= pos && pos <= end(), "Bad Index.");
    const iterator dst = const_cast<iterator>(pos);
    std::move_backward(dst, end(), end() + 1);
    *dst, move(value);
    return dst;
}
```

- Requires a position.
- Requires movement of elements.
- We can apply more templates.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(const_iterator pos, U&& value) {
    MBO_REQUIRE(size_ < Capacity, "Called at capacity.");
    MBO_REQUIRE(begin() <= pos && pos <= end(), "Bad Index.");
    const iterator dst = const_cast<iterator>(pos);
    std::move_backward(dst, end(), end() + 1);
    *dst, move(value);
    return dst;
}
```

Fails for non trivial types.

- Requires a position.
- Requires movement of elements.
- We can apply more templates.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(const_iterator pos, U&& value) {
    MBO_REQUIRE(size_ < Capacity, "Called at capacity.");
    MBO_REQUIRE(begin() <= pos && pos <= end(), "Bad Index.");
    const iterator dst = const_cast<iterator>(pos);
    std::move_backward(dst, end(), end() + 1);
    std::construct_at(&*dst, std::forward<U>(value));
    return dst;
}
```

- Requires a position.
- Requires movement of elements.
- We can apply more templates.

The move is just fine. But if we use non-trivial types - which we can do at run-time - then we must correctly construct those types.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(const_iterator pos, U&& value) {
    MBO_REQUIRE(size_ < Capacity, "Called at capacity.");
    MBO_REQUIRE(begin() <= pos && pos <= end(), "Bad Index.");
    const iterator dst = const_cast<iterator>(pos);
    std::move_backward(dst, end(), end() + 1);
    std::construct_at(&*dst, std::forward<U>(value));
    return dst;
}
```

ERROR: ... pointer to different array...

- Requires a position.
- Requires movement of elements.
- We can apply more templates.

# LimitedOrdered::insert

```
template<std::constructible_from<T> U>
constexpr iterator insert(const_iterator pos, U&& value) {
    MBO_REQUIRE(size_ < Capacity, "Called at capacity.");
    MBO_REQUIRE(begin() <= pos && pos <= end(), "Bad Index.");
    const iterator dst = const_cast<iterator>(pos);
    move_backward(dst, 1);
    std::construct_at(&*dst, std::forward<U>(value));
    return dst;
}

constexpr std::size_t move_backward(
    iterator dst, std::size_t count) {
    std::size_t pos = size_;
    while (dst < &values_[pos].data) {
        --pos;
        std::construct_at(&values_[pos + count].data,
            std::move(values_[pos].data));
    }
    size_ += count;
    return pos;
}
```

- Requires a position.
- Requires movement of elements.
- We can apply more templates.
- Pointer ops are problematic.
- Some STL funcs cannot be used. Even when they are theoretically constexpr.
- **const\_cast** works fine.
- **reinterpret\_cast** is no go!
- **std::bit\_cast** may work.

# LimitedOrdered / Map / Set

LimitedOrdered: ~1k LOC

LimitedSet: ~300 LOC

LimitedMap: ~400 LOC



# LimitedSet: An Example

```
static constexpr mbo::container::LimitedSet<
    std::tuple<TypeId, EdgeId, TypeId>,
    mbo::container::LimitedOptions<
        ${valid_edges_len},
        mbo::container::LimitedOptionsFlag::kRequireSortedInput>{}> kValidEdges_{
% for source in sorted(schema.sources()):
% for edge in sorted(source.edges()):
% for target in sorted(edge.targets):
    {TypeId::${source}, EdgeId::${edge}, NodeId::${target}},
% endfor
% endfor
% endfor
};
```

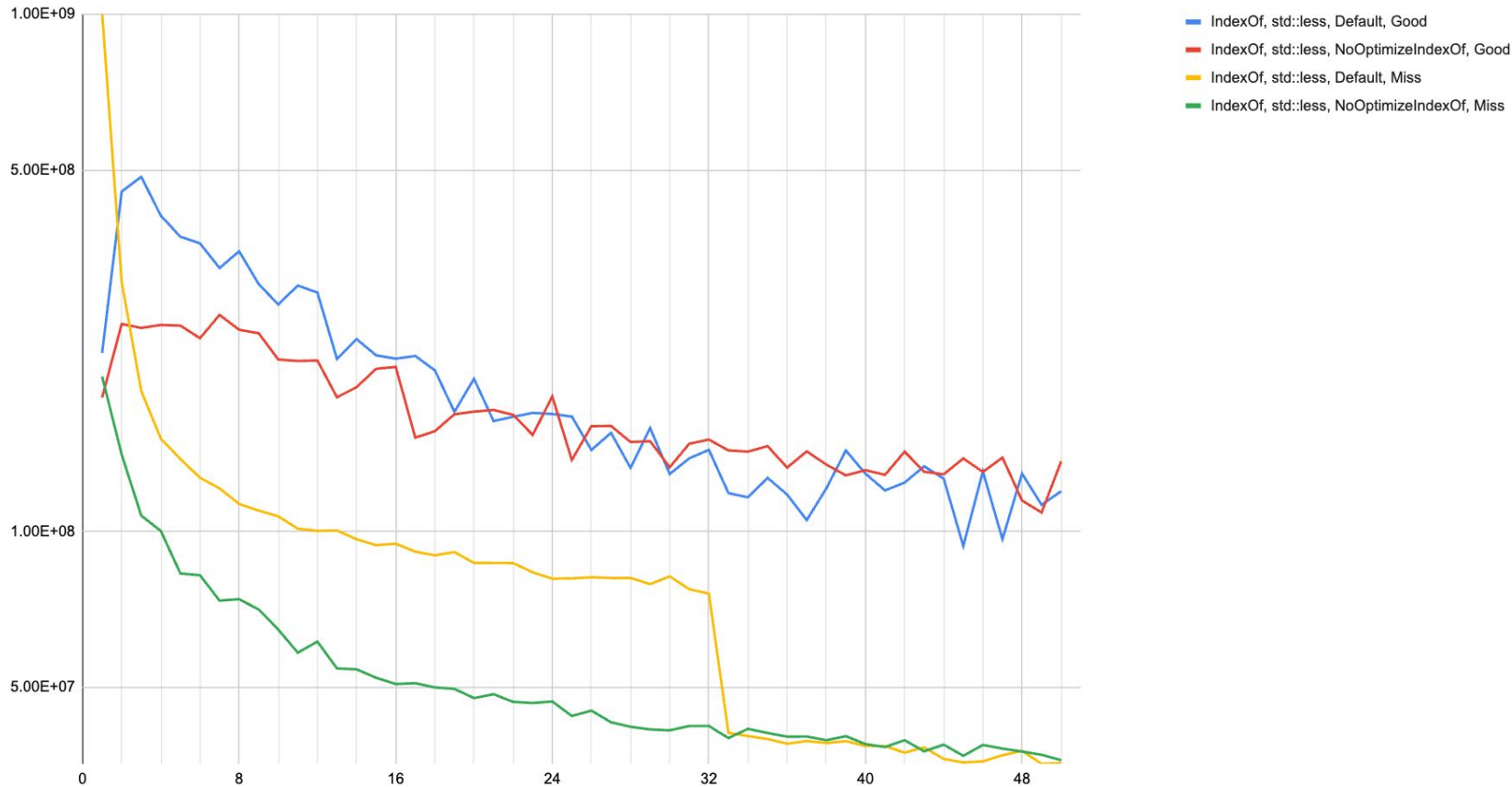
# LimitedSet: An Example

```
static constexpr mbo::container::LimitedSet<
    std::tuple<TypeId, EdgeId, TypeId>,
    mbo::container::LimitedOptions<
        ${valid_edges_len},
        mbo::container::LimitedOptionsFlag::kRequireSortedInput>{}> kValidEdges_{
% for source in sorted(schema.sources()):
% for edge in sorted(source.edges()):
% for target in sorted(edge.targets):
    {TypeId::${source}, EdgeId::${edge}, NodeId::${target}},
% endfor
% endfor
% endfor
};
```

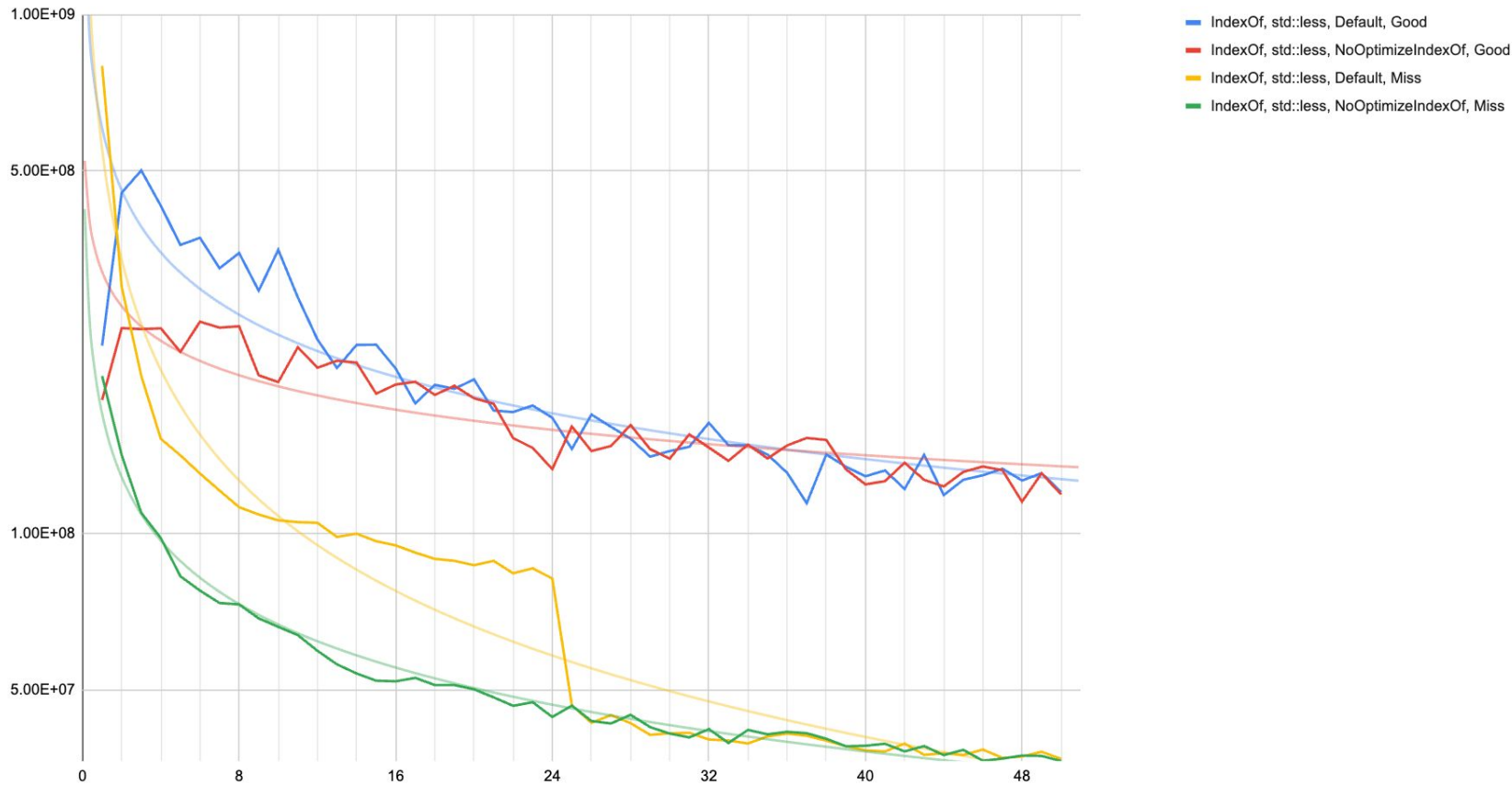
# LimitedSet: An Example

```
static constexpr mbo::container::LimitedSet<
    std::tuple<TypeId, EdgeId, TypeId>,
    mbo::container::LimitedOptions<
        ${valid_edges_len},
        mbo::container::LimitedOptionsFlag::kRequireSortedInput>{}> kValidEdges_{
    % for source in sorted(schema.sources()):
    % for edge in sorted(source.edges()):
    % for target in sorted(edge.targets()):
        {TypeId::${source}, EdgeId::${edge}, NodeId::${target}},
    % endfor
    % endfor
    % endfor
};
```

# Performance: X86-64, 32 unroll, IndexOf



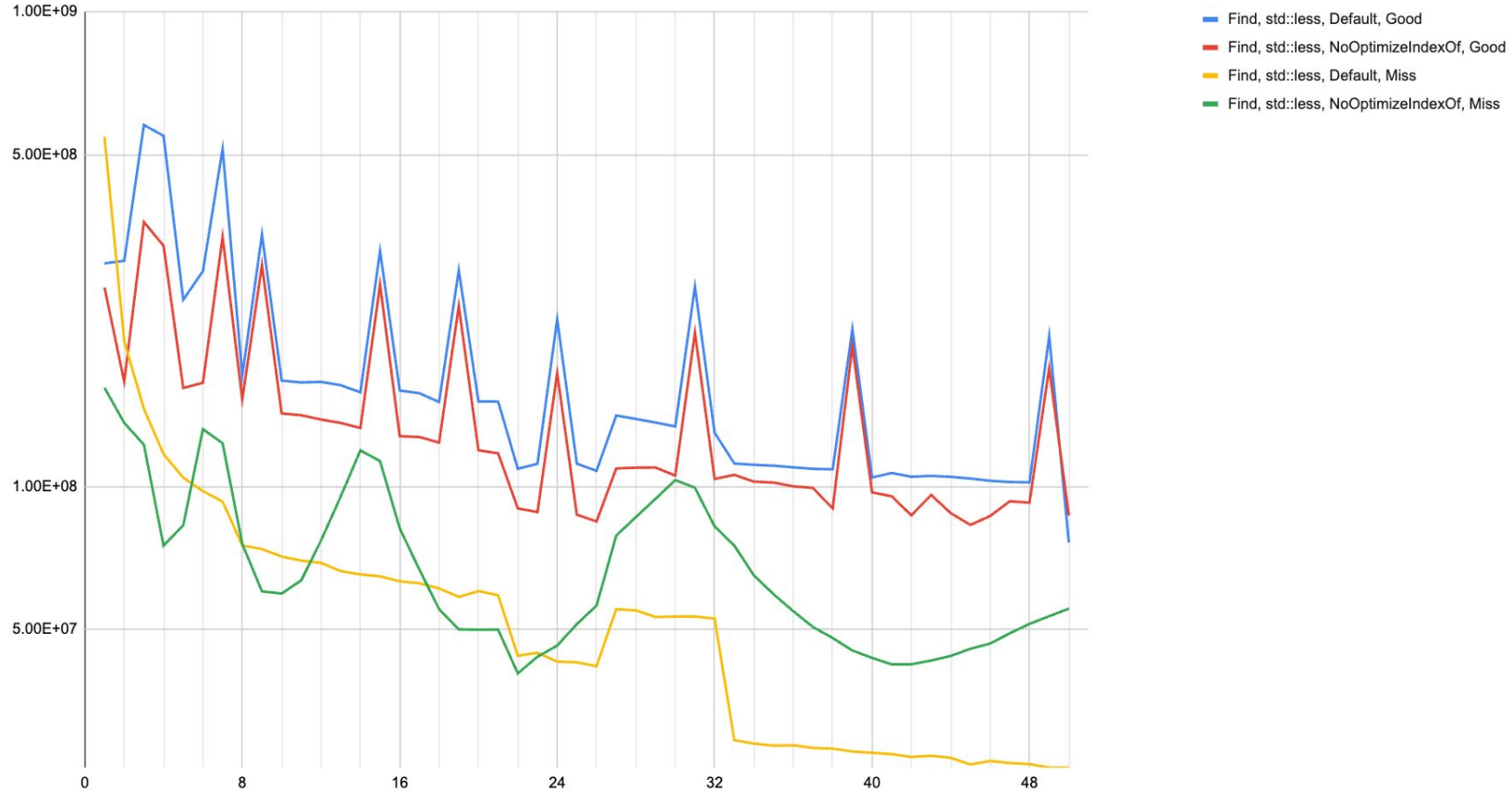
# Performance: X86-64, 24 unroll, IndexOf







# Performance: Apple Mac Studio M2 Max





# Thank you

---

[github.com/helly25/mbo](https://github.com/helly25/mbo)  
[helly25.com/](https://helly25.com/)

# Questions?



---

[github.com/helly25/mbo](https://github.com/helly25/mbo)  
[helly25.com/](https://helly25.com/)