

Harnessing constexpr: A Path to Safer C++

Mikhail Svetkin
Meeting C++, 2025

About me

Mikhail Svetkin

Principal Software Engineer at reMarkable

Areas:

- Architecture
- Internal frameworks and libraries
- Build systems



Agenda

- Is C++ Safe?
- Common Pitfalls in C++
- Traditional Solutions
- `constexpr`
 - Evolution from C++11 to C++26
 - Case studies
 - Limitations
 - How to write a test?
- Summary

What are we going to learn today?

- How easy it is to make mistakes in C++
- How to try to protect ourselves from common pitfalls
- How `constexpr` could help

Is C++ a safe language?

Is C++ Safe? - News

- [NSA - has recommended adoption of memory-safe programming languages](#)
- [CISA - recommends transitioning to memory-safe languages such as Rust](#)
- [Google security by design](#)
- [Microsoft Azure not allowed to write new code in C++](#)

Is C++ Safe? - News



[The existential threat
against C++ and
where to go from
here - Helge Penne -
NDC TechTown 2024](#)



Is C++ Safe? - News

- Do not use memory unsafe languages in new products
- Use memory safe language
- **R**ewrite in memory safe language

What is wrong with them?

We all know that C++ is the best!

We all know that C++ is the best!
Right?

Is C++ Safe? - News

- Do not use **memory unsafe languages** in new products
- Use **memory safe language**
- **Rewrite in memory safe language**

What is a memory safe language?

Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers. ([wiki/Memory_safety](#)*)

* [Memory safety without runtime checks or garbage collection - Dhurjati, Dinakar; Kowshik, Sumant; Adve, Vikram; Lattner, Chris \(1 January 2003\)](#)

What is a memory safe language?

[Open Source Security Foundation](#) - A memory safe by default language prevents (by default) common memory safety vulnerabilities, including:

- Access errors (invalid read/write of a pointer)
- Uninitialized variables (variable that has not been assigned a value is used)
- Memory leak (memory usage is not tracked or is tracked incorrectly)
- Race conditions
- Undefined behavior

Is C++ is a memory safe language?

Is C++ is a memory safe language?

No

Common pitfalls in C++

- <https://cwe.mitre.org/top25/>

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12
12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9
24	CWE-400	Uncontrolled Resource Consumption	3.23	0	+13
25	CWE-306	Missing Authentication for Critical Function	2.73	5	-5

Common pitfalls in C++

- <https://cwe.mitre.org/top25/>

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12
12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9
24	CWE-400	Uncontrolled Resource Consumption	3.23	0	+13
25	CWE-306	Missing Authentication for Critical Function	2.73	5	-5

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{"failed to parse"};
    }

    return result;
}
```

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

Common pitfalls in C++

```
TEST_CASE("valid-input") {  
    const auto r = fromBlob<int>(std::string_view{"10"});  
    REQUIRE(r.has_value());  
    REQUIRE(*r == 10);  
}
```

```
TEST_CASE("invalid-input") {  
    const auto r = fromBlob<int>(std::string_view{"ups"});  
    REQUIRE(r.has_value() == false);  
    REQUIRE(r.error().ends_with("error: 22"));  
}
```

Common pitfalls in C++

Test #1: from-blob:valid-input	Passed
Test #2: from-blob:invalid-input	Passed
Test #3: production	Failed

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```


Common pitfalls in C++ - Use after free

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

Common pitfalls in C++

```
template<typename T>
std::expected<T, std::errc> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{ec};
    }

    return result;
}
```

Common pitfalls in C++

Test #1: from-blob:valid-input	Passed
Test #2: from-blob:invalid-input	Passed
Test #3: production	Passed

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]);
    if (*version > 2)
    {
        ...
    }

    ...
    return header;
}
```

Common pitfalls in C++

Test #1: parse-header:version-1	Passed
Test #2: parse-header:version-2	Passed
Test #3: production	Failed

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]);
    if (*version > 2)
    {
        ...
    }

    ...
    return header;
}
```

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]);
    if (*version > 2)
    {
        ...
    }

    ...
    return header;
}
```


Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]); // input[1] = "<!,>"
    if (*version > 2) // version = 22
    {
        ...
    }

    ...
    return header;
}
```

Common pitfalls in C++ - Dereference without check

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]); // input[1] = "<!,>"
    if (*version > 2) // version = 22
    {
        ...
    }

    ...
    return header;
}
```

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1])
    if (!version.has_value()) {
        return std::unexpected{...};
    }

    if (*version > 2) { ... }

    ...
    return header;
}
```

Common pitfalls in C++

Test #1: parse-header:version-1	Passed
Test #2: parse-header:version-2	Passed
Test #3: parse-header:invalid-version.....	Passed
Test #4: production	Failed

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1])
    if (!version.has_value()) {
        return std::unexpected{...};
    }

    if (*version > 2) { ... }

    ...
    return header;
}
```

Common pitfalls in C++

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1])
    if (!version.has_value()) {
        return std::unexpected{...};
    }

    if (*version > 2) { ... }

    ...
    return header;
}
```

Common pitfalls in C++ - Out of bound read

```
std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1])
    if (!version.has_value()) {
        return std::unexpected{...};
    }

    if (*version > 2) { ... }

    ...
    return header;
}
```

Common pitfalls in C++

```
bool Settings::load(const File &file)
{
    ...
}
```


Common pitfalls in C++

```
50:  bool Settings::load(const File &file)
    {
        ...
550: }
```

Common pitfalls in C++

```
50:  bool Settings::load(const File &file)
    {
        ...

90:      char *someBuffer = (char *)malloc(MaxBufferBytes);

        ...

550:  }
```

Common pitfalls in C++

```
50:  bool Settings::load(const File &file)
    {
        ...

90:      char *someBuffer = (char *)malloc(MaxBufferBytes);

        ...

546:      free(someBuffer);

        ...

550:  }
```

Common pitfalls in C++

```
50:  bool Settings::load(const File &file)
    {
        ...

90:      char *someBuffer = (char *)malloc(MaxBufferBytes);

        ...

250:     if (...) return false;

        ...

546:     free(someBuffer);

        ...

550: }
```

Common pitfalls in C++

```
50:  bool Settings::load(const File &file)
    {
        ...

90:  char *someBuffer = (char *)malloc(MaxBufferBytes);

        ...

250:  if (...) return false;

        ...

546:  free(someBuffer);

        ...

550: }
```

Common pitfalls in C++ - Memory leak

```
50:  bool Settings::load(const File &file)
    {
        ...

90:  char *someBuffer = (char *)malloc(MaxBufferBytes);

        ...

250:  if (...) return false;

        ...

546:  free(someBuffer);

        ...

550:  }
```

Common pitfalls in C++ - Reports

- Microsoft ~70% of common vulnerabilities are due to memory safety
- Google's Project Zero team 67% percent of zero-day vulnerabilities in 2021 were memory corruption

How to prevent such issues?

Traditional Solutions

- Static analyzers - detects issues **before execution**
- Sanitizers - detects issues **during execution**

Static analyzers - Overview

- Mostly detects **semantic** issues
- Might generate some **false positives**
- Usually **limited** to a translation unit
- Limited to **predefined** rules
- Does not detect **concurrency** issues
- Increase **build time***

Static analyzers - Tools

3rd-party tools:

- cppcheck
- clang-tidy
- SonarQube
- ...

Builtin:

- gcc -fanalyzer
- clang --analyze
- msvc /analyze

Static analyzers - CMake Integration

```
# CMakeLists.txt
set(CMAKE_CXX_CPPCHECK cppcheck --enable=all)
set(CMAKE_CXX_CLANG_TIDY clang-tidy
    -checks=*,clang-analyzer-*,bugprone-*,cppcoreguidelines-*
)

target_compile_options(<target>
    PRIVATE
        $$<CXX_COMPILER_ID:MSVC>:/analyze>
        $$<CXX_COMPILER_ID:GNU>:-fanalyzer>
        $$<CXX_COMPILER_ID:Clang>:--analyze>
)
```

Static analyzers - Case studies

Test #1: use-after-free	Passed
Test #2: dereference	Passed
Test #3: out-of-bound	Passed
Test #4: memory-leak	Failed

Sanitizers - Overview

- **0** false positives
- You need to **compile** in a special mode and **run**
 - You need a good test coverage
 - Not available on all platforms (Bare Metal, Embedded*, Windows*)
- Runtime overhead
 - performance from **2x** to **10x**
 - memory usage **2x**
- Increase build time and binary size
- Sometimes hard to read reports*

Sanitizers - Tools

- [AddressSanitizer \(ASan\)](#)
 - detects addressability issues
- [LeakSanitizer \(LSan\)](#)
 - detects memory leaks
- [ThreadSanitizer \(TSan\)](#)
 - detects data races and deadlocks
- [MemorySanitizer \(MSan\)](#)
 - detects use of uninitialized memory
- [UndefinedBehaviorSanitizer \(UBSan\)](#)
 - detects undefined behavior

Sanitizers - CMake Integration

```
# CMakeLists.txt
target_compile_options(<target>
  PRIVATE
    $<$<CXX_COMPILER_ID:MSVC>:/fsanitize=address>
    $<$<CXX_COMPILER_ID:GNU,Clang>:-fsanitize=address>
)

target_link_options(<target>
  PRIVATE
    $<$<CXX_COMPILER_ID:MSVC>:/fsanitize=address>
    $<$<CXX_COMPILER_ID:GNU,Clang>:-fsanitize=address>
)
```


Sanitizers - Case study

Test #1: use-after-free	Failed
Test #2: memory-leak	Failed
Test #3: parse-header:version-1	Passed
Test #4: parse-header:version-2	Passed
Test #5: parse-header-dereference:invalid-version	Failed
Test #6: parse-header-out-of-bound:invalid-input	Failed

Traditional Solutions - Summary

- Static analyzers - are good for checking semantics
- Sanitizers - actually detects errors
- Increase build time*
- Runtime overhead
- Not enabled by default
- Requires configuration

What else can we do?

constexpr ALL THE
THINGS!





BEN DEANE / bdeane@blizzard.com / [@ben_deane](https://twitter.com/ben_deane)

JASON TURNER / jason@emptycrate.com / [@lefticus](https://twitter.com/lefticus)

CPPCON / MONDAY 25TH SEPTEMBER 2017

8 years later

constexpr ~~all the things~~
most of the things

constexpr - C++11

Language features:

- `constexpr` - specifies that the value of a variable or function can appear in `constant expressions`
- `constant expressions` [5.19] is a core constant expression unless:
 - ...
 - **an operation that would have undefined behavior**
 - ...

Library features:

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    std::cout << 55 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib47 = fibonacci(47);  
    std::cout << fib47 << std::endl;  
}
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib47 = fibonacci(47);  
    std::cout << fib47 << std::endl;  
}
```

main.cpp:11 **error**: constexpr variable 'fib47' must be initialized by a constant expression

```
constexpr auto fib47 = fibonacci(47);
```

constexpr - C++11

```
constexpr int fibonacci(int n, int a = 0, b = 1) {  
    return n == 0 ? a : fibonacci(n - 1, b, a + b);  
}
```

```
int main() {  
    constexpr int fib47 = fibonacci(47);  
    std::cout << fib47 << std::endl;  
}
```

main.cpp:11 **error**: constexpr variable 'fib47' must be initialized by a constant expression

```
constexpr auto fib47 = fibonacci(47);
```

main.cpp:6 **note**: value **2971215073** is outside the range of representable values of type 'int'

constexpr - C++14

Language features:

- More complex expressions allowed:
`if/else, for, while, do-while`
- Mutable variables: local variables declared inside constexpr functions

Library features:

- `<complex>`
- `<chrono>`
- `<utility>`
- `std::array*`
- `std::tuple*`

constexpr - C++14

```
constexpr int fibonacci(int n) {  
    if (n ≤ 1) return n;  
  
    int a = 0, b = 1;  
    for (int i = 2; i ≤ n; ++i) {  
        int temp = a + b;  
        a = b;  
        b = temp;  
    }  
    return b;  
}  
  
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++14

```
constexpr int fibonacci(int n) {  
    if (n ≤ 1) return n;  
  
    int a = 0, b = 1;  
    for (int i = 2; i ≤ n; ++i) {  
        int temp = a + b;  
        a = b;  
        b = temp;  
    }  
    return b;  
}  
  
int main() {  
    constexpr int fib10 = fibonacci(10);  
    std::cout << fib10 << std::endl;  
}
```

constexpr - C++14

```
constexpr int fibonacci(int n) {  
    if (n ≤ 1) return n;  
  
    int a = 0, b = 1;  
    for (int i = 2; i ≤ n; ++i) {  
        int temp = a + b;  
        a = b;  
        b = temp;  
    }  
    return b;  
}  
  
int main() {  
    std::cout << 55 << std::endl;  
}
```

constexpr - C++17

Language features:

- `constexpr` lambda
- `if constexpr`

Library features:

- `std::string_view`
- `std::char_traits`
- `std::chrono::duration*`, `time_point*`
- `std::atomic<T>::is_always_lock_free`
- `std::addressof`
- `std::reverse_iterator`
- `std::move_iterator`
- `std::array::(c|r)begin`, `(c|r)end`

constexpr - C++17

```
int main() {  
    auto max = [] (int a, int b) {  
        if (a > b) {  
            return a;  
        }  
  
        return b;  
    };  
  
    constexpr auto m = max(100, 50);  
    std::cout << m << std::endl;  
}
```

constexpr - C++17

```
int main() {  
    auto max = [] (int a, int b) constexpr {  
        if (a > b) {  
            return a;  
        }  
  
        return b;  
    };  
  
    constexpr auto m = max(100, 50);  
    std::cout << m << std::endl;  
}
```

constexpr - C++17

```
int main() {  
    constexpr auto max = [] (int a, int b) {  
        if (a > b) {  
            return a;  
        }  
  
        return b;  
    };  
  
    constexpr auto m = max(100, 50);  
    std::cout << m << std::endl;  
}
```


constexpr - C++17

```
int main() {  
    constexpr auto max = [] (int a, int b) {  
        if (a > b) {  
            return a;  
        }  
  
        return b;  
    };  
  
    constexpr auto m = max(100, 50);  
    std::cout << m << std::endl;  
}
```

constexpr - C++17

```
int main() {  
    constexpr auto max = [] (int a, int b) {  
        if (a > b) {  
            return a;  
        }  
  
        return b;  
    };  
  
    std::cout << 100 << std::endl;  
}
```

constexpr - C++17

```
template<typename T, typename ... Ts>
constexpr int sum(T t, Ts ... ts) {
    if constexpr (sizeof...(Ts) == 0) {
        return t;
    } else {
        return t + sum(ts...);
    }
}

int main() {
    constexpr int value = sum(1, 2, 3, 4, 5);
    std::cout << value << std::endl;
}
```

constexpr - C++17

```
template<typename T, typename ... Ts>
constexpr int sum(T t, Ts ... ts) {
    if constexpr (sizeof...(Ts) == 0) {
        return t;
    } else {
        return t + sum(ts...);
    }
}

int main() {
    constexpr int value = sum(1, 2, 3, 4, 5);
    std::cout << value << std::endl;
}
```

constexpr - C++17

```
template<typename T, typename ... Ts>
constexpr int sum(T t, Ts ... ts) {
    if constexpr (sizeof...(Ts) == 0) {
        return t;
    } else {
        return t + sum(ts...);
    }
}
```

```
int main() {
    std::cout << 15 << std::endl;
}
```

constexpr - C++20

Language features:

- `constexpr`
- `constinit`
- `new/delete`
- `try-catch`
- `virtual` functions
- changing the active member of a `union`

Library features:

- `std::vector`
- `std::string`
- `std::optional`
- `std::variant`
- `std::allocator`
- `std::swap`
- `std::source_location`
- `std::ranges`
- `std::invoke`
- `std::is_constant_evaluated`
- `<algorithm/complex*/numeric>`

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    constexpr auto result = sort(4, 3, 2, 1);  
    return result[0];  
}
```

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    constexpr auto result = sort(4, 3, 2, 1);  
    return result[0];  
}
```

main.cpp:9 **error:** 'result' is not a constant expression because it refers to a result of 'operator new'

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    constexpr auto result = sort(4, 3, 2, 1);  
    return result[0];  
}
```

main.cpp:9 **error**: 'result' is not a constant expression because it refers to a result of 'operator new'

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    return []() consteval {  
        auto result = sort(4, 3, 2, 1);  
        return result[0];  
    }();  
}
```

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    return []() consteval {  
        auto result = sort(4, 3, 2, 1);  
        return result[0];  
    }();  
}
```

constexpr - C++20

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    return 1;  
}
```

constexpr - C++23

Language features:

- Permitting `static constexpr` variables in constexpr functions
- `if constexpr`
- Non-literal variables (and labels and `gotos`) in constexpr
- Relaxing some constexpr restrictions

Library features:

- `std::unique_ptr`
- `std::bitset`
- `std::to_char<int>`
- `std::from_chars<int>`
- `std::type_info::operator=()`
- `<cstdlib>*`
- `<cmath>*`

constexpr - C++23

```
constexpr auto sort(std::integral auto ... values) {  
    std::vector v{values...};  
    std::ranges::sort(v);  
    return v;  
}
```

```
int main() {  
    constexpr auto result = sort(4, 3, 2, 1); // error: is not a constant expression...  
    return result[0];  
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```


constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    constexpr auto result = sort<[] { return std::vector{4,3,2,1}; }>();
    return result[0];
}
```

constexpr - C++23

```
template<auto Builder>
constexpr auto sort() {
    static constexpr auto s = [] {
        constexpr auto size = [] { return Builder().size(); }();
        const auto v = Builder();

        std::array<typename decltype(v)::value_type, size> result{};
        std::copy(v.begin(), v.end(), result.begin());
        std::ranges::sort(result);
        return result;
    }();

    return std::span{s};
}

int main() {
    return 1;
}
```

constexpr - C++23



[Understanding The
constexpr 2-Step -
Jason Turner - C++
on Sea 2024](#)



constexpr - C++26

Language features:

- cast from `void*`
- placement new
- structured bindings and references to constexpr variables
- `constexpr` exceptions
- user-generated `static_assert` messages

Library features:

- `std::stable_sort`
- `std::atomic`
- `std::inplace_vector`
- `std::bad_alloc/cast`
- `<cmath>*`
- `<complex>*`
- All containers and adaptors

constexpr - C++26

All containers and adaptors:

- [P3372 constexpr containers and adaptors](#)
- [Hana Dusíková - libc++ missing constexpr](#)



constexpr - C++26

```
static_assert(false, std::format("The answer is {}. ", 42));
```

error: call to non-`'constexpr'` function `'std::string std::format'`

C++29 - `constexpr` `std::format` [P3391R1](#) - 🙌

constexpr - C++26

```
static_assert(false, std::format("The answer is {}. ", 42)); // with P3391R1
```

constexpr - C++26

```
static_assert(false, std::format("The answer is {}. ", 42)); // with P3391R1
```

```
error: static assertion failed: The answer is 42.
```

constexpr - C++26

```
constexpr auto parse(std::string_view input) {  
    if (input.empty()) {  
        throw std::format("invalid input = {}", input);  
    }  
  
    return 10;  
}  
  
int main() {  
    constexpr auto r = parse("ups");  
    return r;  
}
```

constexpr - C++26

```
constexpr auto parse(std::string_view input) {  
    if (input.empty()) {  
        throw std::format("invalid input = {}", input);  
    }  
  
    return 10;  
}  
  
int main() {  
    constexpr auto r = parse("ups"); // compile time error: invalid input = 'ups'  
    return r;  
}
```

Let's sprinkle some `constexpr`

constexpr - Case study - Use after free

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

constexpr - Case study - Use after free

```
template<typename T>
constexpr std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_, ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```


constexpr - Case study - Use after free

```
TEST_CASE("valid-input") {  
    const auto r = fromBlob<int>(std::string_view{"10"});  
    REQUIRE(r.has_value());  
    REQUIRE(*r == 10);  
}
```

```
TEST_CASE("invalid-input") {  
    const auto r = fromBlob<int>(std::string_view{"ups"});  
    REQUIRE(r.has_value() == false);  
    REQUIRE(r.error().ends_with("error: 22"));  
}
```

constexpr - Case study - Use after free

```
TEST_CASE("valid-input") {  
    constexpr auto r = fromBlob<int>(std::string_view{"10"});  
    REQUIRE(r.has_value());  
    REQUIRE(*r == 10);  
}  
  
TEST_CASE("invalid-input") {  
    constexpr auto r = fromBlob<int>(std::string_view{"ups"});  
    REQUIRE(r.has_value() == false);  
    REQUIRE(r.error().ends_with("error: 22"));  
}
```

constexpr - Case study - Use after free

```
TEST_CASE("valid-input") {  
    constexpr auto r = fromBlob<int>(std::string_view{"10"});  
    STATIC_REQUIRE(r.has_value());  
    STATIC_REQUIRE(*r == 10);  
}  
  
TEST_CASE("invalid-input") {  
    constexpr auto r = fromBlob<int>(std::string_view{"ups"});  
    STATIC_REQUIRE(r.has_value() == false);  
    STATIC_REQUIRE(r.error().ends_with("error: 22"));  
}
```

constexpr - Case study - Use after free

Test #1: from-blob:valid-input Failed

Test #2: from-blob:invalid-input Failed

error: call to non-constexpr function 'std::string std::format'

constexpr - Case study - Use after free

Test #1: from-blob:valid-input	Failed
Test #2: from-blob:invalid-input	Failed

error: call to non-‘constexpr’ function ‘std::string std::format’

- C++29 - `constexpr std::format` [P3391R1](#) - 🙌

constexpr - Case study - Use after free

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            std::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

constexpr - Case study - Use after free

```
template<typename T>
std::expected<T, std::string_view> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_, ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{
            stdx::format("failed to parse, error: {}", static_cast<int>(ec))};
    }

    return result;
}
```

constexpr - Case study - Use after free

```
Test #1: from-blob:valid-input ..... Passed
Test #2: from-blob:invalid-input ..... Failed
```


constexpr - Case study - Use after free

Test #1: from-blob:valid-input Passed

Test #2: from-blob:invalid-input Failed

test.cpp:41:22: **error:** constexpr variable 'r' must be initialized by a constant expression

```
    constexpr auto r = fromBlob<int>(std::string_view{"ups"});
```

test.cpp:41:22: **note:** pointer to subobject of temporary is not a constant expression

test.cpp:28:12: **note:** temporary created here

```
    return std::unexpected{std::string{"failed to parse"}};
```

constexpr - Case study - Dereference without check

```
template<typename T>
constexpr std::expected<T, std::errc> fromBlob(std::span<const char> input, int base = 10)
{
    T result;
    auto [_ , ec] = std::from_chars(input.data(), input.data() + input.size(), result, base);

    if (ec != std::errc()) {
        return std::unexpected{ec};
    }

    return result;
}
```

constexpr - Case study - Dereference without check

```
constexpr std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]);
    if (*version > 2)
    {
        ...
    }

    ...
    return header;
}
```

constexpr - Case study - Dereference without check

```
TEST_CASE("invalid-version") {  
    constexpr std::array input = {  
        std::string_view{"0011"},  
        std::string_view{"<!.>"},  
    };  
  
    constexpr auto h = parseHeader(input);  
    ...  
}
```

constexpr - Case study - Dereference without check

Test #1: parse-header:version-1	Passed
Test #2: parse-header:version-2	Passed
Test #3: parse-header:invalid-version.....	Failed

constexpr - Case study - Dereference without check

```
Test #1: parse-header:version-1 ..... Passed
Test #2: parse-header:version-2 ..... Passed
Test #3: parse-header:invalid-version..... Failed
```

```
test.cpp:41:22: error: constexpr variable 'h' must be initialized by a constant
expression
```

```
    constexpr auto h = parseHeader(input);
```

```
test.cpp:41:22: error: accessing std::expected<...>::value instead of initialized
std::expected<...>::error
```

constexpr - Case study - Out of bound read

```
constexpr std::expected<Header, std::string> parseHeader(std::span<std::string_view> input)
{
    ...

    const auto version = fromBlob<uint32_t>(input[1]);
    if (version > 2) {
        ...
    }

    ...
    return header;
}
```

constexpr - Case study - Out of bound read

```
TEST_CASE("invalid-input") {  
    constexpr std::array input = {  
        std::string_view{"0011"},  
    };  
  
    constexpr auto h = parseHeader(input);  
    ...  
}
```


constexpr - Case study - Out of bound read

Test #1: parse-header:invalid-version.....	Passed
Test #2: parse-header:invalid-input.....	Failed

constexpr - Case study - Out of bound read

Test #1: parse-header:invalid-version..... Passed

Test #2: parse-header:invalid-input..... Failed

test.cpp:41:22: **error:** constexpr variable 'h' must be initialized by a constant expression

```
constexpr auto h = parseHeader(input);
```

test.cpp:41:22: in 'constexpr' expansion of 'input.std::span<const std::basic_string_view<char> >::operator[](1)'

test.cpp:41:22: **error:** array subscript value '1' is outside the bound

constexpr - Case study - Memory leak

```
bool Settings::load(const File &file)
{
    ...

    char *someBuffer = (char *)malloc(MaxBufferBytes);

    ...

    if (...) return false;

    ...

    free(someBuffer);

    ...
}
```

constexpr - Case study - Memory leak

```
constexpr bool Settings::load(const File &file)
{
    ...

    char *someBuffer = (char *)malloc(MaxBufferBytes);

    ...

    if (...) return false;

    ...

    free(someBuffer);

    ...
}
```

constexpr - Case study - Memory leak

Test #1: settings-load:valid-input **Failed**

error: call to non-`'constexpr'` function `'File::read'`

error: call to non-`'constexpr'` function `'malloc'`

error: call to non-`'constexpr'` function `'free'`

constexpr - Case study - Memory leak

```
constexpr bool Settings::load(const File &file)
{
    ...

    char *someBuffer = (char *)malloc(MaxBufferBytes);

    ...

    if (...) return false;

    ...

    free(someBuffer);

    ...
}
```

constexpr - Case study - Memory leak

```
constexpr bool Settings::load(const IFile &file)
{
    ...

    char *someBuffer = (char *)malloc(MaxBufferBytes);

    ...

    if (...) return false;

    ...

    free(someBuffer);

    ...
}
```

constexpr - Case study - Memory leak

```
constexpr bool Settings::load(const IFile &file)
{
    ...

    char *someBuffer = (char *)malloc(MaxBufferBytes);

    ...

    if (...) return false;

    ...

    free(someBuffer);

    ...
}
```


constexpr - Case study - Memory leak

```
constexpr bool Settings::load(const IFile &file)
{
    ...

    auto someBuffer = new char[MaxBufferBytes];

    ...

    if (...) return false;

    ...

    delete someBuffer;

    ...
}
```

constexpr - Case study - Memory leak

Test #1: settings-load:valid-input	Passed
Test #2: settings-load:new-condition	Failed

constexpr - Case study - Memory leak

Test #1: settings-load:valid-input Passed

Test #2: settings-load:new-condition Failed

test.cpp:90:22: **error:** is not a constant expression because allocated storage has not been deallocated

```
    auto someBuffer = new char[MaxBufferBytes];
```

constexpr - Case study - Memory leak

Test #1: settings-load:valid-input Passed

Test #2: settings-load:new-condition Failed

test.cpp:90:22: **error**: is not a constant expression because allocated storage has not been deallocated

```
    auto someBuffer = new char[MaxBufferBytes];
```

test.cpp:546:22: **error**: non-array deallocation of object allocated with array allocation

```
    delete someBuffer;
```

constexpr - Case studies - Summary

- All issues have been **detected** at compile-time
- Works as **sanitizers** but at **compile-time**
- You need to have a **good test coverage**
- You need to **change/split/modify** code to be `constexpr` compatible.

constexpr - Limitations

Language features:

- IO (filesystem*, network)
- Threads
- Coroutines

Library features:

- I don't think anyone knows

constexpr - Limitations

Ecosystem:

- Tooling (Profiling, Debugging*)
- Missing `constexpr` in 3rd-party libraries
- Test frameworks integrations

* [CLion - Introducing the Constexpr Debugger](#)

How to test `constexpr` code?

constexpr - How to write a test

```
TEST_CASE("example") {  
    REQUIRE(10 == 9);  
}
```

```
$ cmake --preset host  
$ cmake --build --preset host  
$ ctest --preset host
```

Test project: demo/build/host

Start 1: test-example

1/1 Test #1: test-example*** Failed 0.01 sec

test.cpp:2: **FAILED:**

REQUIRE(10 == 9)

constexpr - How to write a test

```
TEST_CASE("example") {  
    REQUIRE(10 == 9);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    STATIC_REQUIRE(10 == 9);  
}
```



test.cpp:2: error: static assertion failed: 10 = 9

constexpr - How to write a test

```
TEST_CASE("example") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    REQUIRE(data[0] == 4);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    REQUIRE(data[0] == 4);  
}
```

test.cpp:5: **FAILED**: REQUIRE(data[0] == 4)
with expansion:
3 == 4

constexpr - How to write a test

```
TEST_CASE("example") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    STATIC_REQUIRE(data[0] == 4);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    STATIC_REQUIRE(data[0] == 4);  
}
```

```
test.cpp:2:26: error: the value of 'data' is not usable in a constant expression  
      : note: 'data' was not declared 'constexpr'
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    STATIC_REQUIRE(data[0] == 4);  
}
```


constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    STATIC_REQUIRE(data[0] == 4);  
}
```

test.cpp:3: error: no match for call to '(const std::ranges::__sort_fn) (const std::array<int, 4>&)'

constexpr - How to write a test

```
TEST_CASE("example") {  
    [] () constexpr {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        STATIC_REQUIRE(data[0] == 4);  
    }();  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    [] () consteval {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        STATIC_REQUIRE(data[0] == 4);  
    }();  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    [] () constexpr {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        STATIC_REQUIRE(data[0] == 4);  
    }();  
}
```

```
test.cpp:3:26: error: the value of 'data' is not usable in a constant expression  
      : note: 'data' was not declared 'constexpr'
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

test.cpp:7: **error:** static assertion failed: result[0] == 4

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

test.cpp:7: error: static assertion failed: result[0] == 4

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

```
test.cpp:7: error: static assertion failed: result[0] == 4  
: note: the comparison reduces to '(3 == 4)'
```


constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

```
test.cpp:7: error: static assertion failed: result[0] == 4  
: note: the comparison reduces to '(3 == 4)' - only for simple types
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

test.cpp:7: error: static assertion failed: result[0] == 4

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    STATIC_REQUIRE(result[0] == 4);  
}
```

test.cpp:7: error: static assertion failed: (3 == 4)

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, std::format("{} ≠ 4", result[0]));  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, std::format("{} ≠ 4", result[0]));  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, fmt::format(FMT_COMPILE("{} != 4"), result[0]));  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, fmt::format(FMT_COMPILE("{} != 4"), result[0]));  
}
```

- fmt v12 - support `constexpr fmt::format` and `FMT_STATIC_FORMAT`

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, fmt::format(FMT_COMPILE("{} != 4"), result[0]));  
}
```

test.cpp:7: error: static assertion failed: (3 != 4)

constexpr - How to write a test

```
namespace stdx {  
  
constexpr std::string format(auto fmt, auto&&... args) {  
    std::string text;  
    fmt::format_to(std::back_inserter(text), fmt,  
                  std::forward<decltype(args)>(args) ...);  
    return text;  
};  
  
} // namespace stdx  
  
static_assert(false, stdx::format(FMT_COMPILE("The answer is {}."), 42));  
  
error: static assertion failed: The answer is 42.
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
        return data;  
    }();  
  
    static_assert(result[0] == 4, stdx::format("{} ≠ 4", result[0]));  
}
```

test.cpp:7: error: static assertion failed: (3 ≠ 4)

constexpr - How to write a test

```
TEST_CASE("example") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    REQUIRE(data[0] == 4);  
  
    // other operations  
    REQUIRE(...);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{stdx::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    }();  
  
    static_assert(result.has_value(), result.error());  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    constexpr auto result = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{std::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    }();  
  
    static_assert(result.has_value(), result.error());  
}
```

test.cpp:11: error: 'result' is non-constant condition for static assertion

constexpr - How to write a test

```
TEST_CASE("example") {  
    auto test_case = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{std::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    };  
  
    constexpr auto error = run_test_case<test_case>(); // 2-step constexpr model  
    static_assert(error.empty(), error);  
}
```

constexpr - How to write a test

```
TEST_CASE("example") {  
    auto test_case = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{stdx::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    };  
  
    constexpr auto error = run_test_case<test_case>(); // 2-step constexpr model  
    static_assert(error.empty(), error);  
}
```

test.cpp:11: **error**: static assertion failed: 3 ≠ 4

constexpr - How to write a test

```
TEST_CASE("example") {  
    auto test_case = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{stdx::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    };  
  
    constexpr auto error = run_test_case<test_case>(); // 2-step constexpr model  
    static_assert(error.empty(), error); // What if don't have C++26  
}
```


constexpr - How to write a test

```
TEST_CASE("example") {  
    auto test_case = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{std::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    };  
  
    constexpr auto error = run_test_case<test_case>(); // 2-step constexpr model  
    static_assert(std::static_verify<error.empty(), error>);  
}
```

constexpr - How to write a test

```
namespace stdx {  
  
template <bool C, static_string msg>  
concept _static_assert = C;  
  
template <bool C, static_string msg>  
concept static_verify = _static_assert<C, msg>;  
  
} // namespace stdx
```

constexpr - How to write a test

```
constexpr stdx::static_string msg = "3 ≠ 4";  
static_assert(stdx::static_verify<3 = 4, msg>);
```

<source>:1: static assertion failed

test.cpp:1: **note**: because 'stdx::static_verify<3 = 4, msg>' evaluated to false

note: because '_static_assert<false, static_string<7UL>{{"3 ≠ 4"}}>

msvc: because 'static_string<7UL>{char{51, 32, 33, 61, 32, 52, 0}}

constexpr - How to write a test

```
constexpr stdx::static_string msg = "3 ≠ 4";  
static_assert(stdx::static_verify<3 = 4, msg>); // works with c++23/20
```

<source>:1: static assertion failed

test.cpp:1: **note**: because 'stdx::static_verify<3 = 4, msg>' evaluated to false

note: because '_static_assert<false, static_string<7UL>{{"3 ≠ 4"}}>

msvc: because 'static_string<7UL>{char{51, 32, 33, 61, 32, 52, 0}}'

constexpr - How to write a test

```
TEST_CASE("example") {  
    auto test_case = []() → std::expected<void, std::string> {  
        std::array data = {6, 5, 4, 3};  
        std::ranges::sort(data);  
  
        if (data[0] ≠ 4) { return std::unexpected{std::format("{} ≠ 4", data[0])}; }  
        ...  
        return {};  
    };  
  
    constexpr auto error = run_test_case<test_case>(); // 2-step constexpr model  
    static_assert(std::static_verify<error.empty(), error>);  
}
```

constexpr - How to write a test

```
STATIC_TEST_CASE("...") {  
    std::array data = {6, 5, 4, 3};  
    std::ranges::sort(data);  
  
    REQUIRE(data[0] == 4);  
}
```

Summary

- `constexpr` is **memory safest subset** of C++
- `constexpr` partially replaces Sanitizers (lack of concurrency support)
- `constexpr` partially replaces Static analyzers
 - does not work without tests
- `constexpr` requires adaptation (sprinkle `constexpr`, refactoring)
- `constexpr` is a builtin feature which almost works out the box

constexpr - Want to learn more?

- [boost.pfr](#) / [magic_enum](#) / [fmt](#)
- [Jason Turner constexpr](#)
- [Ben Dean](#)
 - [Formatted Diagnostics with C++20](#)
 - [Intel standard library extensions](#)

Thank you!
Questions?

email: mikhail.svetkin@gmail.com  <https://t.me/msvetkin>