`extern constinit const int foo?`

Dude, I just wanted a constant.

Martin Hořeňovský

PEX

`extern constinit const int foo?`

Dude, I just wanted a constant.

Martin Hořeňovský

PEX

ABOUT THIS TALK

We will talk about

We will talk about

- history of declaring constants in C++

We will talk about

- history of declaring constants in C++
- set of guidelines on declaring constants in C++20

We will talk about

- history of declaring constants in C++
- set of guidelines on declaring constants in C++20
- why we can't just have a damn constant

C++98

Back in C++98 we had 3 options for constants

Back in C++98 we had 3 options for constants

- a macro

Back in C++98 we had 3 options for constants

- a macro
- value in an anonymous enum

Back in C++98 we had 3 options for constants

- a macro
- value in an anonymous enum
- compile-time initialized const variable

MACROS

```
#define MY_PROJECT_THE_ANSWER 42
```

Macros have lot of issues

MACROS

```
#define MY_PROJECT_THE_ANSWER 42
```

Macros have lot of issues

- they are a separate language from C++

MACROS

```
#define MY_PROJECT_THE_ANSWER 42
```

Macros have lot of issues

- they are a separate language from C++
- they have to be namespaced in name

MACROS

```
#define MY_PROJECT_THE_ANSWER 42
```

Macros have lot of issues

- they are a separate language from C++
- they have to be namespaced in name
- don't have linkage

```cpp
#define MAX_VALUE 42

void foo(int arg) {
    auto const& clamped = std::max(arg, MAX_VALUE);
    // at this point clamped is dangling
}
```

ANONYMOUS ENUMS

```
namespace my_project {
    enum { kSomeConstant = 42 };
};
```

ANONYMOUS ENUMS

```
namespace my_project {
    enum { kSomeConstant = 42 };
};
```

Anonymous enums follow name resolution rules

ANONYMOUS ENUMS

```
namespace my_project {
    enum { kSomeConstant = 42 };
};
```

Anonymous enums follow name resolution rules

They still don't have linkage

```cpp
enum { kSomeConstant = 42 };

void foo(int arg) {
    auto const& clamped = std::max(arg, kSomeConstant);
    // at this point clamped is dangling
}
```

CONSTANT VARIABLES

CONSTANT VARIABLES

Constant variables actually have linkage!

CONSTANT VARIABLES

Constant variables actually have linkage!

```
const int kSomeConstant = 42;

void foo(int arg) {
    auto const& clamped = std::max(arg, kSomeConstant);
    // clamped is fine!
}
```

CONSTANT VARIABLES

Constant variables actually have linkage!

```
const int kSomeConstant = 42;

void foo(int arg) {
    auto const& clamped = std::max(arg, kSomeConstant);
    // clamped is fine!
}
```

Pictured: possible UB

Constant variables can stop being compile-time constants easily.

Constant variables can stop being compile-time constants easily.

```
1 // Still const, but not compile-time.
2 const int kSomeConstant = RuntimeFunction();
3
4 void f() {
5     int arr[kSomeConstant]; // This is now silently a VLA
6 }
```

Constant variables can stop being compile-time constants easily.

```
1 // Still const, but not compile-time.
2 const int kSomeConstant = RuntimeFunction();
3
4 void f() {
5     int arr[kSomeConstant]; // This is now silently a VLA
6 }
```

Not great, huh?

Not great, huh?

Let's see what C++11 gives us.

C++11

C++11 added the `constexpr` keyword

C++11 added the `constexpr` keyword

This allows us to turn complex-ish types into constants

C++11 added the `constexpr` keyword

This allows us to turn complex-ish types into constants

But also changes best practices for integral constants

`constexpr` vars must be initialized at compile time

`constexpr` vars must be initialized at compile time

```
constexpr int kSomeConstant = RuntimeFunction(); // Compile error
```

`constexpr` vars must be initialized at compile time

```
constexpr int kSomeConstant = RuntimeFunction(); // Compile error
```

But `constexpr` functions can be used at compile time

`constexpr` vars must be initialized at compile time

```
constexpr int kSomeConstant = RuntimeFunction(); // Compile error
```

But `constexpr` functions can be used at compile time

```
constexpr size_t pow2(size_t sz) {
  return sz * 2;
}

constexpr size_t kSomeConstant = pow2(3);
```

It is hilariously easy to invoke UB using `constexpr`

It is hilariously easy to invoke UB using `constexpr`

```cpp
// some header
#pragma once

constexpr int kSomeConstant = 42;

inline int limit(int arg) {
    return std::max(arg, kSomeConstant);
}
```

It is hilariously easy to invoke UB using `constexpr`

```cpp
// some header
#pragma once

constexpr int kSomeConstant = 42;

inline int limit(int arg) {
    return std::max(arg, kSomeConstant);
}
```

using the header in two source files is ODR violation

It is hilariously easy to invoke UB using `constexpr`

```cpp
// some header
#pragma once

constexpr int kSomeConstant = 42;

inline int limit(int arg) {
    return std::max(arg, kSomeConstant);
}
```

using the header in two source files is ODR violation

ODR violations are IF;NDR (UB)

The "correct" way of using the constant:

The "correct" way of using the constant:

```
inline int limit(int arg) {
    constexpr int kSomeConstant = 42;
    return std::max(arg, kSomeConstant);
}
```

What if you **do** want to use the constant in multiple functions?

"Just" don't use references

"Just" don't use references

```
// some header
#pragma once

constexpr int kSomeConstant = 42;

inline int my_max(int lhs, int rhs) {
    return lhs < rhs? rhs : lhs;
}

inline int limit(int arg) {
    return my_max(arg, kSomeConstant);
}
```

Or launder your constants through templates

Or launder your constants through templates

```
template <typename = void>
struct helper { static const size_t constant_holder; };
template <typename T>
const size_t helper<T>::constant_holder = 42;

// Convenience name
// Must be static for local linkage.
static constexpr const size_t& kSomeConstant
    = helper<>::constant_holder;
```

C++14 expanded `constexpr` functions, but is irrelevant for constants

C++14 expanded `constexpr` functions, but is irrelevant for constants

Let's talk about C++17

C++17

C++17 added `inline` variables

C++17 added `inline` variables

`inline` variables are unified by the compiler like `inline` functions are

`inline constexpr` variables solve the ODR issue

`inline constexpr` variables solve the ODR issue

```cpp
// some header
#pragma once

inline constexpr int kSomeConstant = 42;

inline int limit(int arg) {
    return std::max(arg, kSomeConstant);
}
```

`inline constexpr` variables solve the ODR issue

```cpp
// some header
#pragma once

inline constexpr int kSomeConstant = 42;

inline int limit(int arg) {
    return std::max(arg, kSomeConstant);
}
```

`kSomeConstant` has the same address in all TUs

Finally, let's talk C++20

C++20

C++20 added `constinit` variables

C++20 added `constinit` variables

`constinit` variables **must** be "static initialized"

C++20 added `constinit` variables

`constinit` variables **must** be "static initialized"

i.e. they must be initialized during compilation

Unlike `constexpr`, `constinit` **does not** imply `const`

Unlike `constexpr`, `constinit` **does not** imply `const`

```
constexpr int sqr(int i) {
    return i * i;
}

constinit int squared = sqr(2);

int main() {
    assert(squared == 4);
    squared = sqr(3);
    assert(squared == 9);
}
```

`constinit` **does not** imply "usable at compile time"

`constinit` **does not** imply "usable at compile time"

```
constinit int a = 1;
constexpr int b = a; // error: a is not usable in constexpr context
```

but you can make it so:

but you can make it so:

```
constinit const int a = 1; // const int is old-style constant
constexpr int b = a;       // fine
```

So what does `constinit` do?

`constinit` prevents *static initialization order fiasco*

`constinit` prevents *static initialization order fiasco*

`constinit` is an optimization hint for the compiler

```
extern thread_local int x1;

int f1() {
    return x1;
}

extern thread_local constinit int x2;

int f2() {
    return x2;
}
```

```
extern thread_local int x1;

int f1() {
    return x1;
}

extern thread_local constinit int x2;

int f2() {
    return x2;
}
```

```
f1():
        push    rax
        cmp     qword ptr [rip + thread-local initialization ro
        je      .LBB0_2
        call    thread-local initialization routine for x1@PLT
.LBB0_2:
        mov     rax, qword ptr [rip + x1@GOTTPOFF]
        mov     eax, dword ptr fs:[rax]
        pop     rcx
        ret

f2():
        mov     rax, qword ptr [rip + x2@GOTTPOFF]
        mov     eax, dword ptr fs:[rax]
        ret
```

```
extern thread_local int x1;

int f1() {
    return x1;
}

extern thread_local constinit int x2;

int f2() {
    return x2;
}
```

```
f1():
        push    rax
        cmp     qword ptr [rip + thread-local initialization ro
        je      .LBB0_2
        call    thread-local initialization routine for x1@PLT
.LBB0_2:
        mov     rax, qword ptr [rip + x1@GOTTPOFF]
        mov     eax, dword ptr fs:[rax]
        pop     rcx
        ret

f2():
        mov     rax, qword ptr [rip + x2@GOTTPOFF]
        mov     eax, dword ptr fs:[rax]
        ret
```

That's all theory we need to cover

That's all theory we need to cover

Let's talk about actually declaring the constants.

DECLARATION GUIDELINES

DEFINE A CONSTANT IN A HEADER

DEFINE A CONSTANT IN A HEADER

Use `inline constexpr`

DEFINE A CONSTANT IN A HEADER

Use `inline constexpr`

```
inline constexpr std::string_view kTargetName = "abc";
```

ENSURE COMP-TIME INITIALIZATION

ENSURE COMP-TIME INITIALIZATION

Use `constinit`

ENSURE COMP-TIME INITIALIZATION

Use `constinit`

```
extern constinit std::shared_ptr<int> a_ptr;
```

HAVE COMPILE-TIME LOOKUP TABLE IN FUNCTION

HAVE COMPILE-TIME LOOKUP TABLE IN FUNCTION

Use `static constexpr`.

`static constexpr` can optimize better than plain `constexpr` and never optimizes worse

`static constexpr` can optimize better than plain `constexpr` and never optimizes worse

```cpp
#include <string_view>

std::string_view static_constexpr(int idx) {
  static constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}

std::string_view plain_constexpr(int idx) {
  constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}
```

`static constexpr` can optimize better than plain `constexpr` and never optimizes worse

```cpp
#include <string_view>

std::string_view static_constexpr(int idx) {
  static constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}

std::string_view plain_constexpr(int idx) {
  constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}
```

```
1  static_constexpr(int):
2          movsx   rdi, edi
3          sal     rdi, 4
4          mov     rax, QWORD PTR static_constexpr_table
5          mov     rdx, QWORD PTR static_constexpr_table
6          ret
7
8  plain_constexpr(int):
9          mov     QWORD PTR [rsp-64], OFFSET FLAT:.LC0
10         movsx   rdi, edi
11         mov     QWORD PTR [rsp-56], 1
12         sal     rdi, 4
13         mov     QWORD PTR [rsp-48], OFFSET FLAT:.LC1
14         mov     QWORD PTR [rsp-40], 1
15         mov     QWORD PTR [rsp-32], OFFSET FLAT:.LC0
16         mov     QWORD PTR [rsp-24], 1
17         mov     QWORD PTR [rsp-16], OFFSET FLAT:.LC2
18         mov     rdx, QWORD PTR [rsp-64+rdi]
19         mov     QWORD PTR [rsp-72], 1
20         mov     rax, QWORD PTR [rsp-72+rdi]
21         ret
```

`static constexpr` can optimize better than plain `constexpr` and never optimizes worse

```cpp
#include <string_view>

std::string_view static_constexpr(int idx) {
  static constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}

std::string_view plain_constexpr(int idx) {
  constexpr std::string_view array[] = {
    "a", "l", "a", "z"
  };
  return array[idx];
}
```

```
static_constexpr(int):
        movsx   rdi, edi
        sal     rdi, 4
        mov     rax, QWORD PTR static_constexpr_table
        mov     rdx, QWORD PTR static_constexpr_table
        ret

plain_constexpr(int):
        mov     QWORD PTR [rsp-64], OFFSET FLAT:.LC0
        movsx   rdi, edi
        mov     QWORD PTR [rsp-56], 1
        sal     rdi, 4
        mov     QWORD PTR [rsp-48], OFFSET FLAT:.LC1
        mov     QWORD PTR [rsp-40], 1
        mov     QWORD PTR [rsp-32], OFFSET FLAT:.LC0
        mov     QWORD PTR [rsp-24], 1
        mov     QWORD PTR [rsp-16], OFFSET FLAT:.LC2
        mov     rdx, QWORD PTR [rsp-64+rdi]
        mov     QWORD PTR [rsp-72], 1
        mov     rax, QWORD PTR [rsp-72+rdi]
        ret
```

PROCESS CONSTANTS IN SINGLE TU

`constexpr` variables in headers are processed in every TU that includes the header.

`constexpr` variables in headers are processed in every TU that includes the header.

This can become expensive, e.g. with large arrays

But what if we still want to process the data during compilation, just in 1 TU?

header

```
extern    // external linkage
constinit // initialized at compile time
const     // constant after initialization
T foo;
```

header

```
extern    // external linkage
constinit // initialized at compile time
const     // constant after initialization
T foo;
```

1 cpp file

```
constexpr      // usable at compile time
T foo = .... ; // initializing declaration
```

This triggers separate bugs in MSVC and Clang

https://developercommunity.visualstudio.com/t/clexe-rejects-initializing-constexpr-de/10911733

https://github.com/llvm/llvm-project/issues/140203

MSVC does not support `constinit` declaration and `constexpr` initializating declaration.

MSVC does not support `constinit` declaration and `constexpr` initializating declaration.

Clang issues a spurious warning

Final fun fact: array constants have to be `constexpr`.

Final fun fact: array constants have to be `constexpr`.

Otherwise dereference (`arr[1]`) is an lvalue-to-rvalue conversion, which is not allowed in constexpr context.

CONCLUSION

In C++, not even constants are easy.

In C++, not even constants are easy.

It took until C++17 to make `constexpr` vars safe.

In C++, not even constants are easy.

It took until C++17 to make `constexpr` vars safe.

Since C++17, few guidelines cover 99% of the cases.

THE END

Questions?