Debugging constexpr evaluation

Artemy Pestretsov
Head of C and C++ Ecosystem @ JetBrains

JETBRAINS

Question

Debugging constexpr evaluation

Agenda

1. Problem statement
2. Examples / demos
3. Solution
4. Under the hood
5. Closing

`constexpr` has grown up

`constexpr` has grown up

Standard	What you can do
C++11	Single return statement

`constexpr` has grown up

Standard	What you can do
C++11	Single return statement
C++14	Loops, variables, branches

`constexpr` has grown up

Standard	What you can do
C++11	Single return statement
C++14	Loops, variables, branches
C++17	`if constexpr`, lambdas

`constexpr` has grown up

Standard	What you can do
C++11	Single return statement
C++14	Loops, variables, branches
C++17	`if constexpr`, lambdas
C++20	`new/delete`, virtual, vector

`constexpr` has grown up

Standard	What you can do
C++11	Single return statement
C++14	Loops, variables, branches
C++17	`if constexpr`, lambdas
C++20	`new/delete`, virtual, vector
C++23	`unique_ptr`, more STL

`constexpr` has grown up

Standard	What you can do	Debugging complexity
C++11	Single return statement	🟢
C++14	Loops, variables, branches	
C++17	`if constexpr`, lambdas	
C++20	`new/delete`, virtual, vector	
C++23	`unique_ptr`, more STL	

`constexpr` has grown up

Standard	What you can do	Debugging complexity
C++11	Single return statement	🟢
C++14	Loops, variables, branches	🟡
C++17	`if constexpr`, lambdas	
C++20	`new/delete`, virtual, vector	
C++23	`unique_ptr`, more STL	

`constexpr` has grown up

Standard	What you can do	Debugging complexity
C++11	Single return statement	🟢
C++14	Loops, variables, branches	🟡
C++17	`if constexpr`, lambdas	🟡
C++20	`new/delete`, virtual, vector	
C++23	`unique_ptr`, more STL	

`constexpr` has grown up

Standard	What you can do	Debugging complexity
C++11	Single return statement	🟢
C++14	Loops, variables, branches	🟡
C++17	`if constexpr`, lambdas	🟡
C++20	`new/delete`, virtual, vector	🔴
C++23	`unique_ptr`, more STL	

`constexpr` has grown up

Standard	What you can do	Debugging complexity
C++11	Single return statement	🟢
C++14	Loops, variables, branches	🟡
C++17	`if constexpr`, lambdas	🟡
C++20	`new`/`delete`, virtual, vector	🔴
C++23	`unique_ptr`, more STL	🔴

`constexpr` has grown up

Standard	What you can do	Debugging complexity
…	…	…
C++17	`if constexpr`, lambdas	🟡
C++20	`new/delete`, virtual, vector	🔴
C++23	`unique_ptr`, more STL	🔴

`constexpr` has grown up

Standard	What you can do	Debugging complexity
…	…	…
C++17	`if constexpr`, lambdas	🟡
C++20	`new/delete`, virtual, vector	🔴
C++23	`unique_ptr`, more STL	🔴
C++26	`exceptions, reflection, placement new`	

`constexpr` has grown up

Standard	What you can do	Debugging complexity
…	…	…
C++17	`if constexpr`, lambdas	🟡
C++20	`new/delete`, virtual, vector	🔴
C++23	`unique_ptr`, more STL	🔴
C++26	`exceptions, reflection, placement new`	💀

constexpr has grown up

```
// C++11
constexpr int square(int x) { return x * x; }
```

constexpr has grown up

```
// C++11: You could debug this in your head
constexpr int square(int x) { return x * x; }
```

constexpr has grown up

```
// C++26
template <info... Members>
using Object = typename ObjectStorage<Members...>::type;

template <class T, auto... Values>
inline constexpr T object_value = T{Values...};

consteval info make_object(std::vector<Field> fields) {
   canonicalize_fields(fields);
   std::vector<info> members;
   for (const auto& field : fields) {
       auto member = std::meta::data_member_spec(
           field.declaration.type, {.name = field.declaration.name});
       members.push_back(std::meta::reflect_constant(member));
   }
   auto generated_type = std::meta::substitute(^^Object, members);
   std::vector<info> arguments{generated_type};
   for (const auto& field : fields)
```

constexpr has grown up

```
// C++26
template <info... Members>
using Object = typename ObjectStorage<Members...>::type;

template <class T, auto... Values>
inline constexpr T object_value = T{V

consteval info make_object(std::vecto
   canonicalize_fields(fields);
   std::vector<info> members;
   for (const auto& field : fields) {
       auto member = std::meta::data_member_spec(
           field.declaration.type, {.name = field.declaration.name});
       members.push_back(std::meta::reflect_constant(member));
   }
   auto generated_type = std::meta::substitute(^^Object, members);
   std::vector<info> arguments{generated_type};
   for (const auto& field : fields)
```

How do you debug `constexpr` now?

Usual approaches

Usual approaches

1. Remove `constexpr`, debug at runtime

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*
2. Add `std::cout`

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*
2. Add `std::cout` -- *`not allowed in constexpr`*

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*
2. Add `std::cout` -- *not allowed in constexpr*
3.

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*
2. Add `std::cout` -- *not allowed in constexpr*

3. Stare harder

Usual approaches

1. Remove `constexpr`, debug at runtime -- *behavior might differ*
2. Add `std::cout` -- *not allowed in constexpr*

3. Stare harder

4. **Spend tokens $$$**

If only we could properly debug it…

Demo

Under the hood

Under the hood

How does a compiler evaluate `constexpr`?

CTFE

CTFE

Sources

CTFE

Sources → Parse

CTFE

Sources → Parse → AST (Abstract Syntax Tree)

CTFE

Sources → Parse → AST → Interpret

CTFE

Sources → Parse → AST → Interpret → Result

CTFE

Sources → Parse → **AST** → Interpret → Result

Under the hood

Under the hood: naive approach

1. Set breakpoints

Under the hood: naive approach

1. Set breakpoints
2. Run *AST-interpreter* until breakpoint hit

Under the hood: naive approach

1. Set breakpoints
2. Run *AST-interpreter* until breakpoint hit
3. Inspect state

Under the hood: naive approach

1. Set breakpoints
2. Run *AST-interpreter* until breakpoint hit
3. Inspect state
4. Continue

Under the hood: naive approach

1. Set breakpoints
2. Run *AST-interpreter* until breakpoint hit
3. Inspect state
4. Continue
5. State accumulates, mutations persist

Under the hood: naive approach

1. Set breakpoints
2. Run *AST-interpreter* until breakpoint hit
3. Inspect state
4. Continue
5. State accumulates, mutations persist

Our approach?

Instead of maintaining state,
re-evaluate from scratch every time

Under the hood: our approach

Under the hood: our approach

User action	Command list

Under the hood: our approach

User action	Command list
Start Debugging	[]

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto] ⇒ execute([StepInto])

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto]

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto] ⇒ execute([...,...])

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto]
Step Over	[StepInto, StepInto, StepOver]

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto]
Step Over	[StepInto, StepInto, StepOver] ⇒ ...

Q: how to implement step back?

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto]
Step Over	[StepInto, StepInto, StepOver]

Under the hood: our approach

User action	**Command list**
Start Debugging	[]
Step Into	[StepInto]
Step Into	[StepInto, StepInto]
Step Over	[StepInto, StepInto, StepOver]
Step Back	[StepInto, StepInto] ← just pop [StepOver] and re-execute

Why this works?

- constexpr evaluation is deterministic

Why this works?

- constexpr evaluation is deterministic
- No I/O, no global state

Why this works?

- constexpr evaluation is deterministic
- No I/O, no global state
- Re-evaluation is fast (N-steps per evaluation)

Mental model

Like a pure function -- you can call it again and get the same answer

Limitations

Not supported yet

- Breakpoints*

Limitations

Not supported yet

- Breakpoints* – on CR, not in production

Key takeaways

Key takeaways

1. `constexpr` **is now debuggable**

Key takeaways

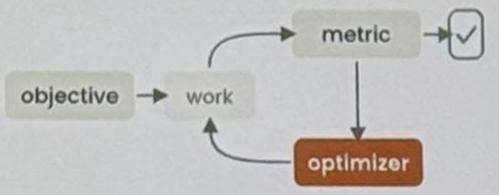
1. `constexpr` **is now debuggable**
2. It's an AST interpreter, not magic

Key takeaways

1. `constexpr` **is now debuggable**
2. It's an AST interpreter, not magic
3. Simplicity is good!

Bonus

The optimizer



- Most primitive form: "try harder."
  - Same session, another step. Works more often than it should.
- More structured: **read the failure**, work out which change would address *that*.
  - **Basic debugging**, on purpose: a debugger, print statements—find where expected and actual part ways. The step comes from evidence.
- Wider: **"try this five different ways** and take the best aspects of each."
- Most important: make the step durable. It has to land in something the **next session reads**—a skill file, a test, a hook, CLAUDE.md, the code.

We’ve got you covered

Bundled skills

clion-debugger Bundled CLion constexpr_debugger_demos
Debugger-first runtime root-cause analysis for native and multi-language code in CLion (C, C++, Rust, Pyth..

clion-embedded-freertos Bundled
Reads the live FreeRTOS kernel objects of a paused embedded debug session in CLion: every task with its s

clion-embedded-hardfault Bundled
Root-cause analysis of ARM Cortex-M faults on a bare-metal MCU target in CLion. Use when a debug sessi.

clion-machine-state Bundled
Reads the machine state of a paused native debug session in CLion: CPU registers, SVD-decoded periphera

Thank you!

artemy.pestretsov@jetbrains.com

linkedin.com/in/pestretsov