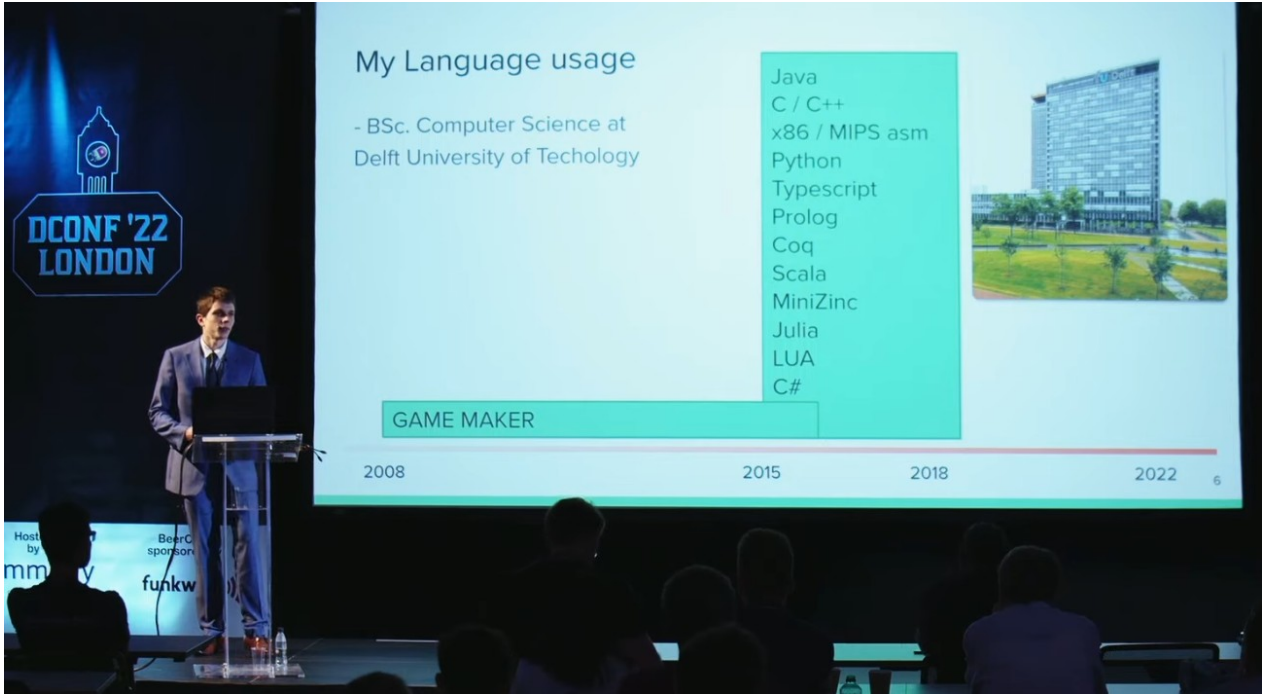
Stack memory

Dennis Korpel

DConf '23

Last year



DConf '22: The Jack of all Trades -- Dennis Korpel
youtu.be/f9RzegZmnUc

Coming up

- Different types of memory:
  🌐Global, 🥞 Stack, 🏔️ Heap
- What makes stack memory so great
- How DIP1000 makes it memory safe
- Problems and future work of DIP1000

Why memory speed matters

- My desktop has 32 GB RAM
- Only 192 KiB is fast
- 1 ns vs 100 ns
- Performance often memory-bound

```
> lscpu

…

Caches (sum of all):
  L1d:                    192 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     3 MiB (6 instances)
  L3:                     32 MiB (1 instance)
```

Why memory safety matters

- Memory corruption bugs are
  - common
  - hard to debug
  - wreaking havoc
  - expensive

⚠ **Warning**: this might all be moot

11:45 Lunch

13:30 Stack Memory is Awesome!

by Dennis Korpel [Show Details]

14:30 Simple @safe D

by Robert Schadek [Hide Details]

Audience: All
Duration: 45 minutes

DIP1000 adds quite a bit of syntax to the language and makes D look a lot less beautiful, in my opinion. Instead of trying to add things to the language, why not take a look at things that need to be removed to achieve the same level of memory safety? This talk shows how to remove three things from the language to make it memory-safe and still live with the consequences.

Different types of memory

- 🌐 Global
- 🥞 Stack
- 🏔️ Heap

Global memory

D source #1

```d
string getPlaque()
{
    return "eternal star";
}

immutable int maxCoins = 999;

struct Player
{
    static const short lives = 4;
}
```

ldc 1.33.0 (Editor #1)

ldc 1.33.0 -betterC -O1

Output... Filter... Libraries Overrides

```
immutable(char)[] example.getPlaque():
        lea     rdx, [rip + .L.str]
        mov     eax, 12
        ret

.L.str:
        .asciz  "eternal star"

immutable(int) example.maxCoins:
        .long   999

const(short) example.Player.lives:
        .short  4
```

d.godbolt.org

🌐 Global memory

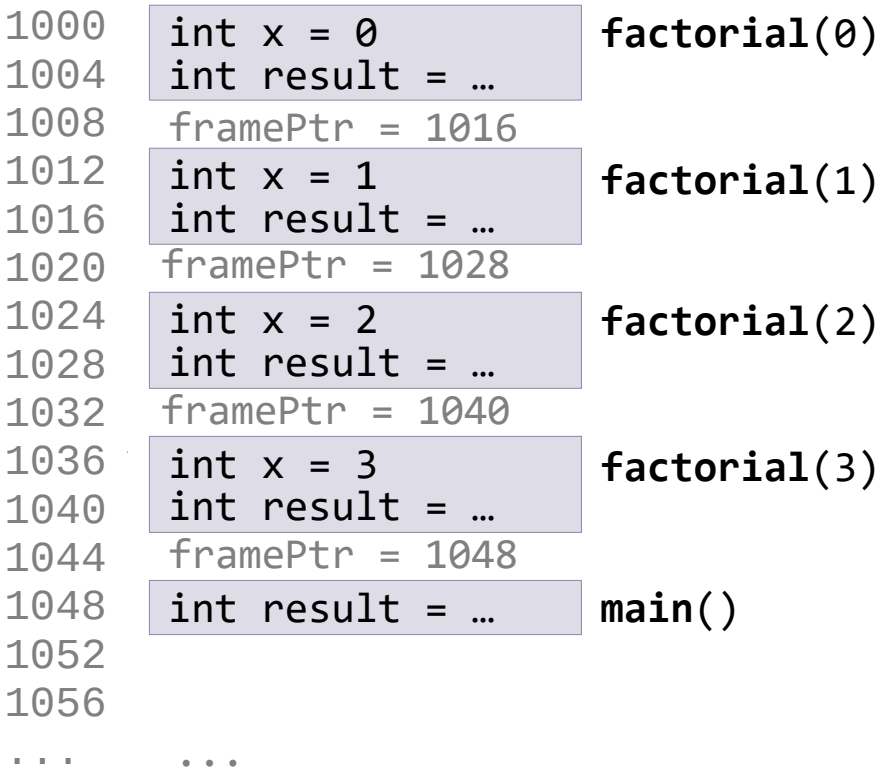
- Must all be known upfront
- Stored uncompressed in .exe
- OS loads it into RAM when program starts
- OS unloads it when program exits

Stack memory

- Function local variables
- Not global because of recursion
- OS initializes a region

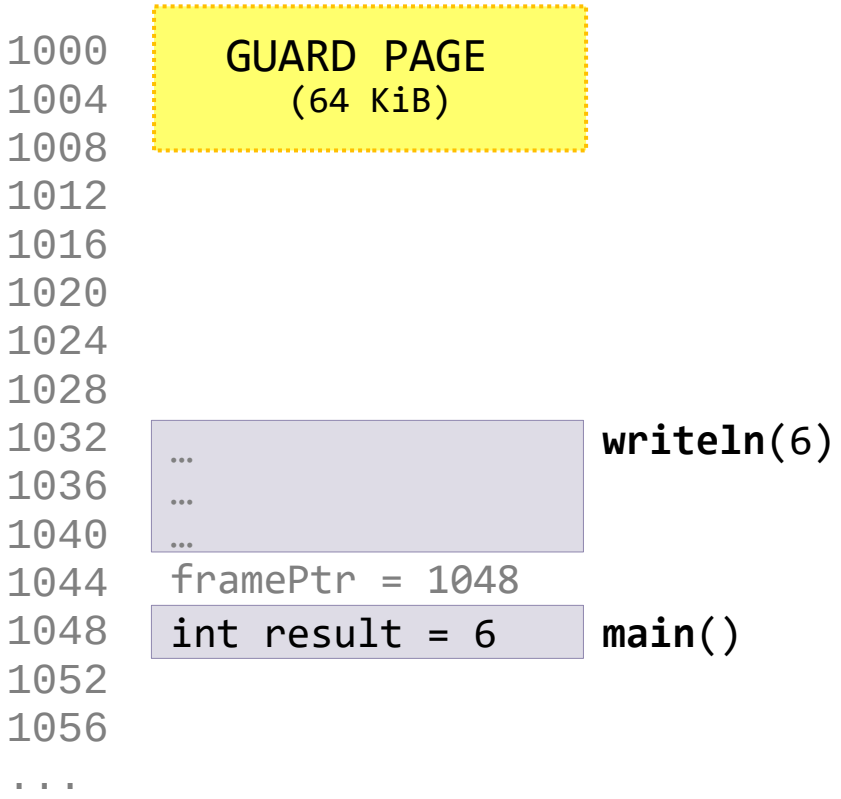
```
Default size:
1 MB on Windows
8 MB on Linux
```

Stack memory




```
import std;

void main()
{
    int result = factorial(3);
    writeln(result);
}

int factorial(int x)
{
    if (x == 0)
        return 1;
    int result = x * factorial(x - 1);
    return result;
}
```

Stack memory




```d
import std;

void main()
{
    int result = factorial(3);
    writeln(result);
}

int factorial(int x)
{
    if (x == 0)
        return 1;
    int result = x * factorial(x - 1);
    return result;
}
```

Stack memory

D source #1

```d
int factorial(int x)
{
    if (x == 0)
        return 1;
    int result = x * factorial(x - 1);
    return result;
}
```

ldc 1.33.0 (Editor #1)

ldc 1.33.0 -betterC

Output... Filter... Libraries Overrides

```
int example.factorial(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], edi
        cmp     dword ptr [rbp - 4], 0
        jne     .LBB0_2
        mov     eax, 1
        add     rsp, 16
        pop     rbp
        ret
.LBB0_2:
        mov     eax, dword ptr [rbp - 4]
        mov     dword ptr [rbp - 12], eax
```

...

 Heap memory

- Dynamically allocated at run time
- OS provides base functions
- libc: `malloc(size)` `free(ptr)`

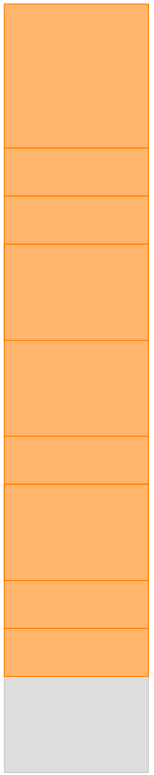
 Heap memory

- In D, used through Garbage Collector (GC)

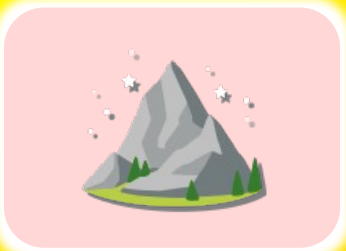
```d
void main()
{
    Object o = new Object();  // new operator
    int[] ms = [10, 12, 16];  // array literal
    ms.length = 4;            // set array length
    ms ~= 10;                 // concatenation
}
```

 Heap memory

- Algorithm to manage blocks
- More complex than stack
- Doesn’t just shrink/grow from one end



Q: Which one is the best?



A: No memory allocation!

No memory allocation

```
void main()
{
    string[] words = "BitDW BitFS BitS".split();
    foreach(word; words)
    {
        writeln(word);
    }
}
```

Pointless to create an array in the first place

No memory allocation

```
void main()
{
    auto words = "BitDW BitFS BitS".splitter();
    foreach(word; words)
    {
        writeln(word);
    }
}
```

Can lazily iterate over elements

Static data

🌐 Global: as long as it fits

```
immutable int[] primes = [2, 3, 5, 7, 11, 13];

immutable creditsText = “Created by Dennis”;

immutable imgIcon = import("icon.bin");
```

 Heap: large / compressed files

```
import std.file : read;
void main()
{
    ubyte[] data = cast(ubyte[]) read("img.png");
}
```

Dynamic data

Please

don’t

don’t

don’t

don’t

DON’T

use  global mutable memory

Dynamic data

Heap

Stack

Fragmentation

Non-deterministic

Complex

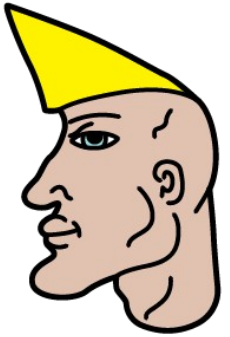
Full of indirections

Wastes bytes on alignment and metadata

Performs syscalls

Tightly packed

Predictable



Simple

Cache friendly

No overhead

Instant allocation and de-allocation

Limitations of 

- Limited size
- Static size*
- Limited lifetime (cannot return stack memory)

```
Default size:
1 MB on Windows
8 MB on Linux
```

```d
int[] getSlice()
{
    int[3] a = [10, 20, 30];
    return a[];
}

// Error: returning `a[]` escapes a reference to local variable `a`
```

*unless you use alloca() (not recommended)

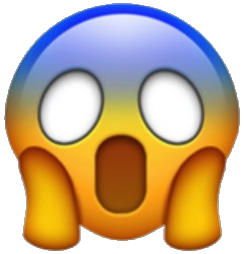

Using stack memory more

Example in D

```
import std.format : format;

void drawText(int x, int y, const char[] msg);

void drawNameTag(string name)
{
    drawText(10, 10, format("name: %s"));
}
```

Returns GC string!

At 60 fps, creates **172 Kb** garbage / minute



Example in C

```
import core.stdc.stdio : snprintf;

void drawText(int x, int y, const char* msg);

void drawNameTag(const char* name)
{
    char[128] buf = void;
    snprintf(buf.ptr, buf.length, "name: %s", name);
    drawText(10, 10, buf.ptr);
}
```

Faster, but uglier code
And is it memory safe?

Example in C

- Documentation (if you're lucky)

```
void glfwWindowHintString ( int          hint,
                            const char * value
                          )
```

This function sets hints for the next call to **glfwCreateWindow**. The hints, once set, retain their val
glfwDefaultWindowHints, or until the library is terminated.

Only string type hints can be set with this function. Integer value hints are set with **glfwWindowHir**

This function does not check whether the specified hint values are valid. If you set hints to invalid v
call to **glfwCreateWindow**.

Some hints are platform specific. These may be set on any platform but they will only affect their sp
them. Setting these hints requires no platform specific headers or functions.

Parameters

`[in]` **hint** The **window hint** to set.

`[in]` **value** The new value of the window hint.

Errors

Possible errors include **GLFW_NOT_INITIALIZED** and **GLFW_INVALID_ENUM**.

Pointer lifetime

The specified string is copied before this function returns.

"The specified string is copied before this function returns"


DIP1000
(Not the best name)


STACK
DON'T CRACK
64

DIP1000

- **`scope`** storage class:
  variable holds value that may not escape current { block }
- Address of local now allowed, becomes scope value

optional →

```
int* outside;

void monkeyCage() @safe
{
    int star;
    scope int* sp = &star;
    outside = sp; // Error: sp escapes scope of monkeyCage
}
```

DIP1000

For every “assignment” of “variables” which “have pointers”

va = v

va may not have a longer “lifetime” than **v**

Assignment?

- Assignment expression: `va = v`
- Return statement: `return v`
- Parameter assignment: `f(v)`
- Array literal assignment: `[v]`

Has pointers?

Yes	No
`int*`	`int`
`int[]`	`int[4]`
`class C`	`struct {int x;}`

- **struct** / static array: depends on child types
- **const**, **immutable**, **shared** don't matter

Variables?

Expression	Variable
`p[0 .. 1]`	`p`
`s.x`	`s`
`s.b ? s.x : p`	`s, p`

```
struct S
{
    int* x;
    bool b;
}

int* p;
S s;
```

Lifetime?

- Lexical order of scope variables

```
void main()
{
    scope int* s0;
    scope int* s1;
    s1 = s0; // ok
    s0 = s1; // error

}

// Error: scope variable `s1` assigned to `s0` with
longer lifetime
```

- Matters because of destructors

In short

Care	Don't care
Variables	Expressions
Has it pointers?	Exact type
Assignments	Control flow

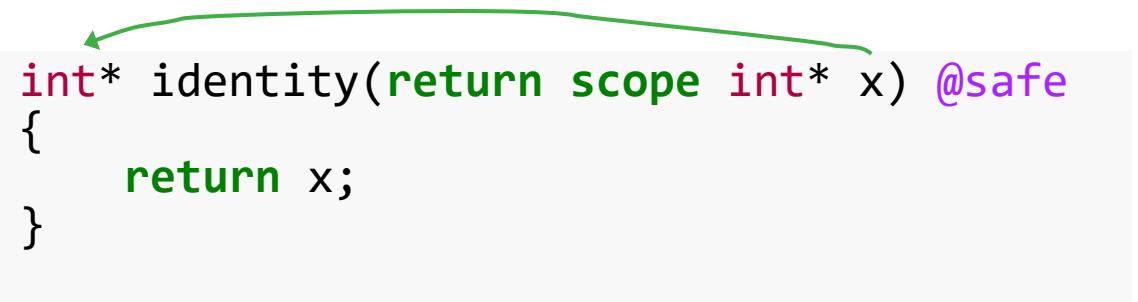
Pseudo code implementation

```
checkAssignment(e0, e1):

va = expToVariable(e0)

if !hasPointers(va)
    return

foreach v in escapeByValue(e1):
    if !hasPointers(v):
        continue
    if va.lifetime > v.lifetime:
        function.setUnsafe()
```

Actual source file in dmd repository:
compiler/src/dmd/escape.d : checkAssignEscape

return scope

- Lifetime in between global and `scope`
- The '`inout`' of lifetime: `scope` in, `scope` out
- non-scope in, non-scope out



return ref

- D has **ref** parameters, passed by pointer
- Not pointer types

```
void f(scope ref int x);
```

- scope is ‘built-in’

```
int* globalPtr;
void f(ref int x) @safe
{
    globalPtr = &x; // Error
    int* p = &x; // p inferred scope
}
```

return ref

- You can return a **return ref** parameter

```
int* addressOf(return ref int x) @safe
{
    return &x;
}
```

- local variable in, scope out

```
int global;
void main() @safe
{
    int local;
    int* g = addressOf(global); // non-scope
    int* l = addressOf(local); // scope
}
```

Parameter storage classes

Action	Storage class	Terminology
`return &p`	`return ref`	escape by reference
`return p`	`return scope`	escape by value
`return *p`	`scope`	no escaping

Parameter storage classes

Static array	Dynamic array	Storage class	Terminology
`return a[]`	`return &a`	`return ref`	escape by reference
`return a[0]`	`return a[]`	`return scope`	escape by value
`return *a[0]`	`return a[0]`	`scope`	no escaping

Subtle differences between static/dynamic array operations!

Member functions

```
struct S
{
    int x;

    int f()
    {
        return x;
    }
}
```

Member functions

```
struct S
{
    int x;

    int f()
    {
        return this.x;
    }
}
```

Member functions have hidden this parameter

Member functions

```
struct S
{
    int x;
}

int f(ref S this_)
{
    return this_.x;
}
```

Member functions

```
struct S
{
    int x;
}

int f(const ref S this_)
{
    return this_.x;
}
```

Member functions

```
struct S
{
    int x;

    int f() const
    {
        return this.x;
    }
}
```

Modifiers for `**this**` parameter outside parameter list

Member functions

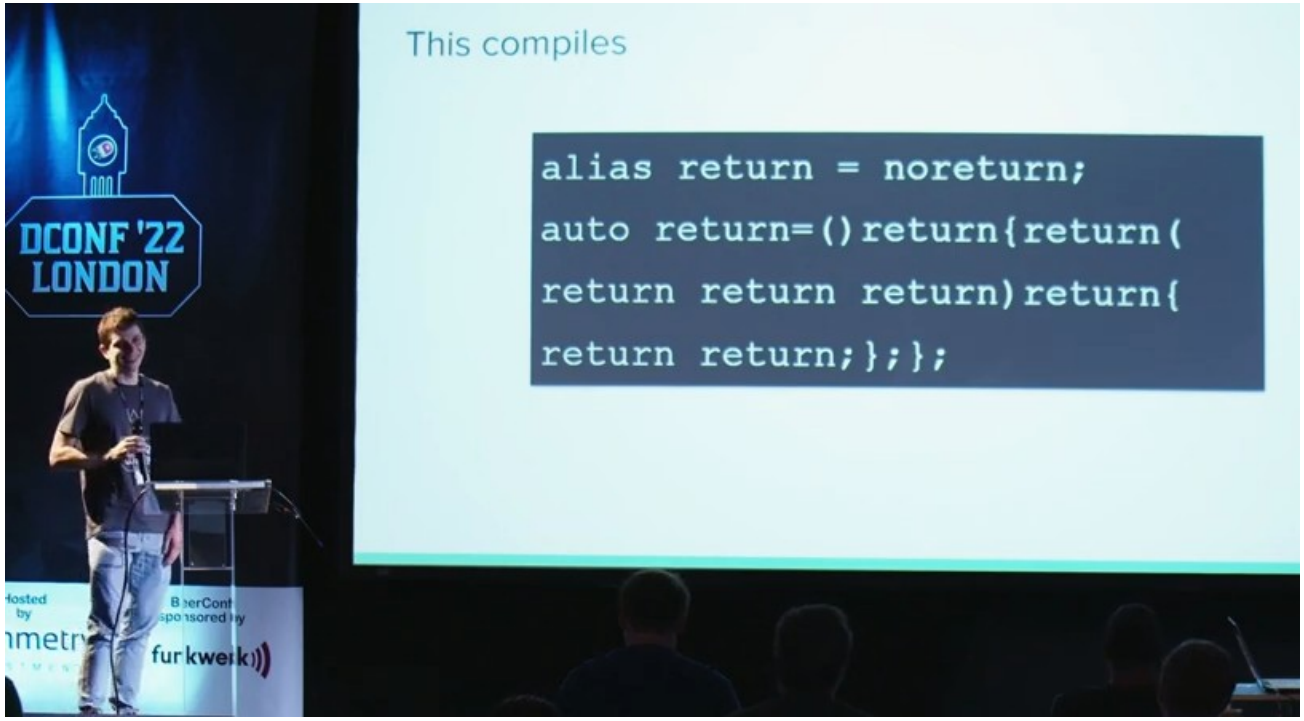
```
struct S
{
    int x;
}

int* f(return ref S this_)
{
    return &this_.x;
}
```

Member functions

```
struct S
{
    int x;

    int* f() return
    {
        return &this.x;
    }
}
```

Same applies to **return**

Looks silly



DConf '22 Lightning Talks
www.youtu.be/GOKIH7AQJR0

Example in C

```
import core.stdc.stdio : snprintf;

void drawText(int x, int y, const char* msg);

void drawNameTag(const char* name)
{
    char[128] buf = void;
    snprintf(buf.ptr, buf.length, "name: %s", name);
    drawText(10, 10, buf.ptr);
}
```

Can we improve this?

Easy stack memory

```
struct StackString
{
    char[128] buffer;  = void; doesn't work here (as of dmd 2.105)
    size_t length;
    char[] toSlice()
    {
        return this.buffer[0 .. this.length];
    }
    alias toSlice this;
}

StackString concat(string l, string r)
{
    StackString s = void;
    s.length = l.length + r.length;
    s.buffer[0 .. l.length] = l[];
    s.buffer[l.length .. s.length] = r[];
    return s;
}
```

Easy stack memory

```
StackString concat(string l, string r);

void drawText(int x, int y, const scope char[] msg);

void drawNameTag(string name)
{
    drawText(10, 10, concat("name: ", name));
}
```

Success! ...But is it `@safe`?

Making it @safe

```
struct StackString
{
    char[128] buffer;
    size_t length;
    char[] toSlice() @safe
    {
        return this.buffer[0 .. this.length];
    }
    alias toSlice this;
}
```

```
Error: returning `this.buffer[0..this.length]`
       escapes a reference to parameter `this`
```

Making it @safe

```
struct StackString
{
    char[128] buffer;
    size_t length;
    char[] toSlice() @safe return
    {
        return this.buffer[0 .. this.length];
    }
    alias toSlice this;
}
```

```
Error: returning `this.buffer[0..this.length]`
       escapes a reference to parameter `this`
       perhaps annotate the function with `return`
```

Inference

```
struct StackString
{
    char[128] buffer;
    size_t length;
    auto toSlice()
    {
        return this.buffer[0 .. this.length];
    }
    alias toSlice this;
}
```

- In auto-return, nested, or template functions
- `scope`, `return scope`, `return ref` are inferred
- Just like `@nogc nothrow pure @safe`

Improvements

- Use malloc for larger sizes, free in destructor
- `std.internal.cstring : tempCString`
- `dmd.common.string : SmallBuffer`


Gotchas

scope transitivity

- **scope** is a variable storage class, not a type constructor
- Only applies to first indirection of variable's type

```
int* f() @safe
{
    scope int* x;
    scope int** y = &x; // Error: can't take address of scope
    return *y; // allowed: dereferencing y removes scope
}
```

classes

- In a class member function, **this** is not **ref**
- Can’t store scope values in class
- You can safely stack allocate a class with **scope**

```
class Chuckya {}

void main() @safe @nogc
{
    scope Chuckya c = new Chuckya();
}
```

classes

- class constructors / member functions are **scope** in practice

  ...but not annotated as such

```
class Chuckya
{
    float x, y, z;
    this(float x, float y; float z) @safe scope
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

return ref and **scope**

ref + return scope **return ref + scope**

```
struct StringArray
{
    private string[] arr;

    ref string opIndex(size_t i) scope return
    {
        return this.arr[i];
    }

    string[] opIndex() return scope
    {
        return this.arr[0 .. $];
    }
}
```

Order matters!

return ref and **scope**

ref + return scope **return ref + scope**

```
struct Array(T)
{
    private T[] arr;

    ref T opIndex(size_t i)
    {
        return this.arr[i];
    }

    T[] opIndex()
    {
        return this.arr[0 .. $];
    }
}
```

Let the compiler infer


Troubles

DLang's Issue Tracking System – Issue List

Home | New | Browse | Search | Search [?] | Reports | My Requests | Preferences | Help | Log out dkorpel@live.nl

Sat Aug 26 2023 13:43:58 UTC

"You want to go forward, what do you do? You put it in D." -- Barack Obama

Hide Search Description

Summary: dip1000

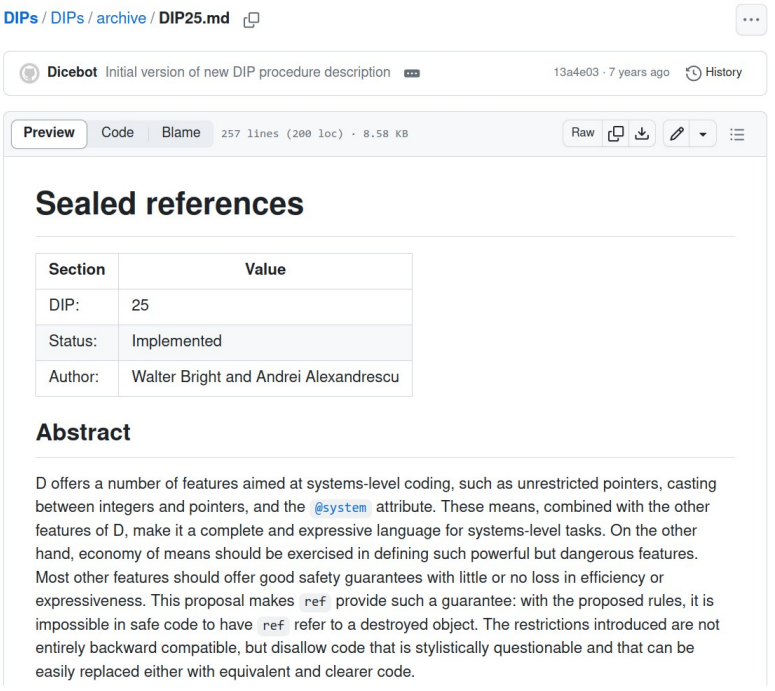
146 issues found.

ID ▼	Product	Comp	Assignee				Changed
24105	D	dmd	nobody@puremagic.com	RESO	FIXE	Dip1000 C variadics not marked as scope should not accept scope arguments	Thu 16:10
24062	D	dmd	nobody@puremagic.com	NEW	---	DIP1000 Provide reason why destructor was not scope when calling member function	2023-07-30
23985	D	dmd	nobody@puremagic.com	NEW	---	[dip1000] return scope fails to infer after assignment	2023-06-11
23941	D	dmd	nobody@puremagic.com	RESO	WONT	[DIP1000] Overloading by scope should be allowed	2023-06-26
23933	D	dmd	nobody@puremagic.com	RESO	INVA	auto return type disables DIP1000 scope check	2023-05-24
23891	D	dmd	nobody@puremagic.com	NEW	---	[DIP1000] unnamed delegates ignore lifetimes	2023-05-05
23813	D	dmd	nobody@puremagic.com	REOP	---	DIP1000 can introduce memory corruption in @safe function with typesafe variadics	2023-03-29
23751	D	dmd	nobody@puremagic.com	NEW	---	Returning by ref from opApply fools DIP1000	2023-02-28
23749	D	phobos	nobody@puremagic.com	RESO	WORK	Can't writeln a static array of strings with -preview=dip1000	2023-02-27

History

2015

- DIP25 Introduced **return ref**



DIPs / DIPs / archive / DIP25.md

Dicebot Initial version of new DIP procedure description — 13a4e03 · 7 years ago — History

Preview | Code | Blame — 257 lines (200 loc) · 8.58 KB — Raw

Sealed references

Section	Value
DIP:	25
Status:	Implemented
Author:	Walter Bright and Andrei Alexandrescu

Abstract

D offers a number of features aimed at systems-level coding, such as unrestricted pointers, casting between integers and pointers, and the `@system` attribute. These means, combined with the other features of D, make it a complete and expressive language for systems-level tasks. On the other hand, economy of means should be exercised in defining such powerful but dangerous features. Most other features should offer good safety guarantees with little or no loss in efficiency or expressiveness. This proposal makes `ref` provide such a guarantee: with the proposed rules, it is impossible in safe code to have `ref` refer to a destroyed object. The restrictions introduced are not entirely backward compatible, but disallow code that is stylistically questionable and that can be easily replaced either with equivalent and clearer code.

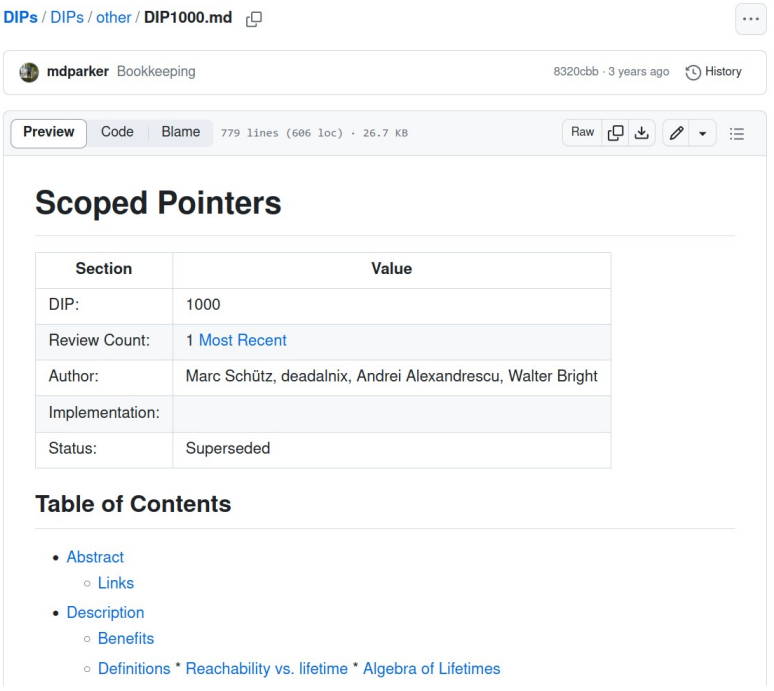
www.dlang.org/dips/25

DIP 1000

2016

“Superseded”

```
scope int* foo(); // outdated now
```



DIPs / DIPs / other / DIP1000.md

mdparker Bookkeeping 8320cbb · 3 years ago History

Preview Code Blame 779 lines (606 loc) · 26.7 KB Raw

Scoped Pointers

Section	Value
DIP:	1000
Review Count:	1 Most Recent
Author:	Marc Schütz, deadalnix, Andrei Alexandrescu, Walter Bright
Implementation:	
Status:	Superseded

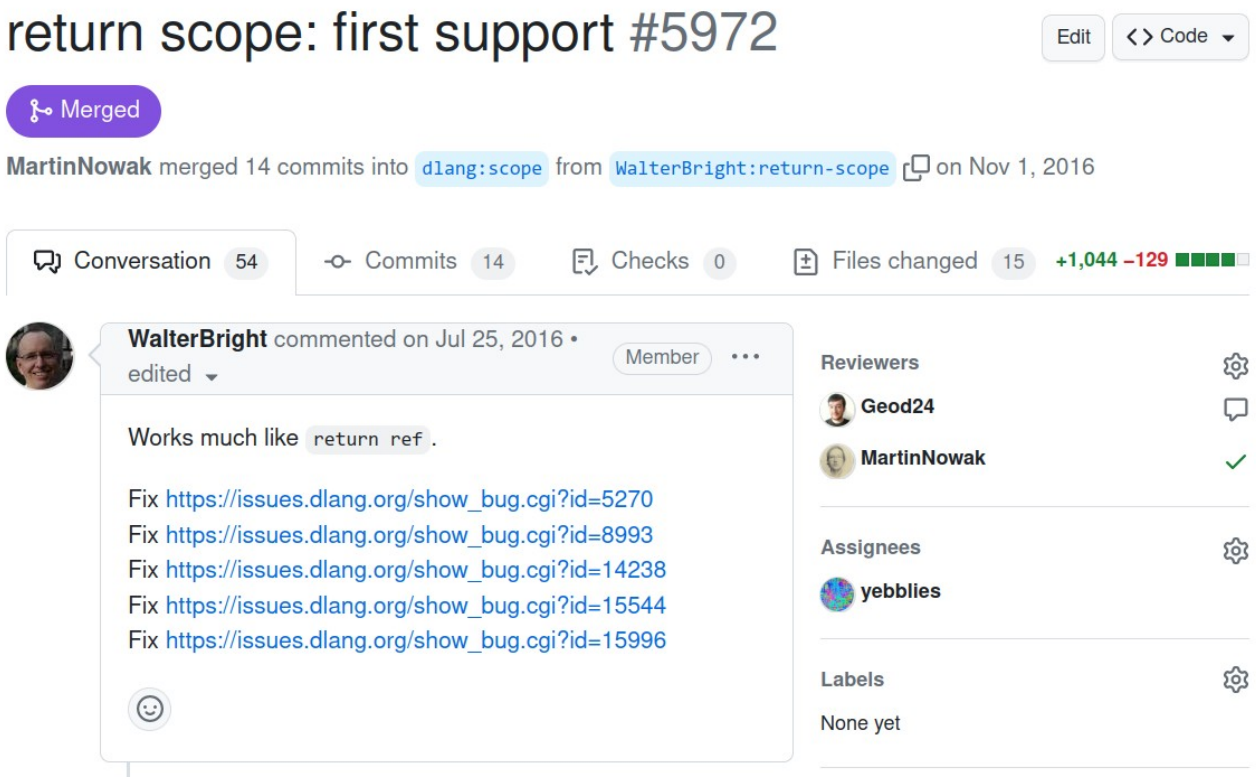
Table of Contents

- Abstract
  - Links
- Description
  - Benefits
  - Definitions * Reachability vs. lifetime * Algebra of Lifetimes

www.dlang.org/dips/1000

Implementation

2016



www.github.com/dlang/dmd/pull/5972

DIP 1000 **2016**

- Breaking change

```
void main() @safe
{
    int l;
    int* p = &l; // Error:
    // cannot take address of local `l` in `@safe` function `main`


    int[4] arr;
    int[] s = arr[]; // No 'scope', no error!
}
```

```
-transition=safe
-dip1000
-preview=dip1000
```

Is the switch ready for programmers?

Linking issues

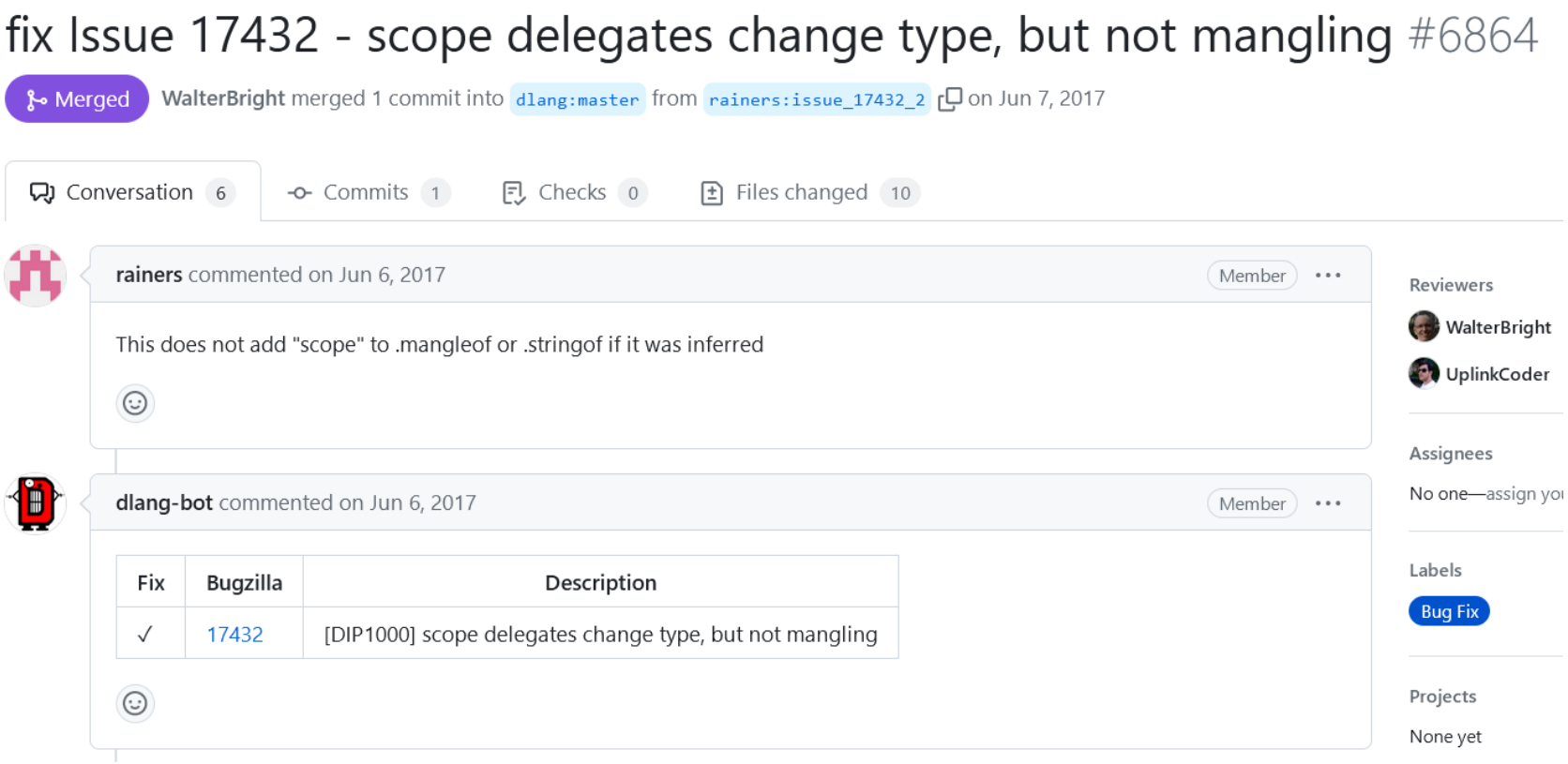
2017

- Phobos is pre-compiled
- scope is part of mangle

```d
auto drawText(/*scope*/ string txt)
{
}

pragma(msg, drawText.mangleof);

// with dip1000:
// _D3app8drawTextFNaNbNiNfMAyaZv
// without dip1000:
// _D3app8drawTextFNaNbNiNfAyaZv
```

Linking issues

2017



fix Issue 17432 - scope delegates change type, but not mangling #6864

Merged **WalterBright** merged 1 commit into `dlang:master` from `rainers:issue_17432_2` on Jun 7, 2017

Conversation 6 | Commits 1 | Checks 0 | Files changed 10

rainers commented on Jun 6, 2017 — Member

This does not add "scope" to .mangleof or .stringof if it was inferred

dlang-bot commented on Jun 6, 2017 — Member

Fix	Bugzilla	Description
✓	17432	[DIP1000] scope delegates change type, but not mangling

Reviewers
WalterBright
UplinkCoder

Assignees
No one—assign yo

Labels
Bug Fix

Projects
None yet

Extend Return Scope Semantics **2018**

```
//  ┌──────────────<──────────────────┐
int* identity(return scope int* x) @safe
{
    return x;
}

void main() @safe
{
    int x;
    //    ┌─────<─────┐
    int* y = identity(&x);

}
```

https://github.com/dlang/dmd/pull/8504

Extend Return Scope Semantics **2018**

```
//                                ┌──────────────<──────────────┐
void assign(ref scope int* target, return scope int* source) @safe
{
    target = source;
}

void main() @safe {
    int x;
    int* y;
    //      ┌─<─┐
    assign(y, &x); // allowed
}
```

Extend Return Scope Semantics **2018**

- Common to assign to `**this**` parameter

```
struct S
{
    int* x;
    this(int* x)
    {
        this.x = x;
    }

    void opAssign(int* x)
    {
        this.x = x;
    }
}
```

Extend Return Scope Semantics **2018**

- Common to assign to `**this**` parameter

```
struct S
{
    int* x;
    this(return scope int* x)
    {
        this.x = x;
    }

    void opAssign(return scope int* x)
    {
        this.x = x;
    }
}
```

Extend Return Scope Semantics **2018**

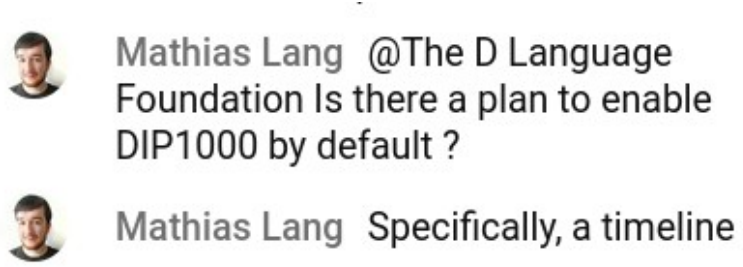
- Walter only person in the world understanding dip1000
- Other contributors begging for documentation

Phobos **2019**

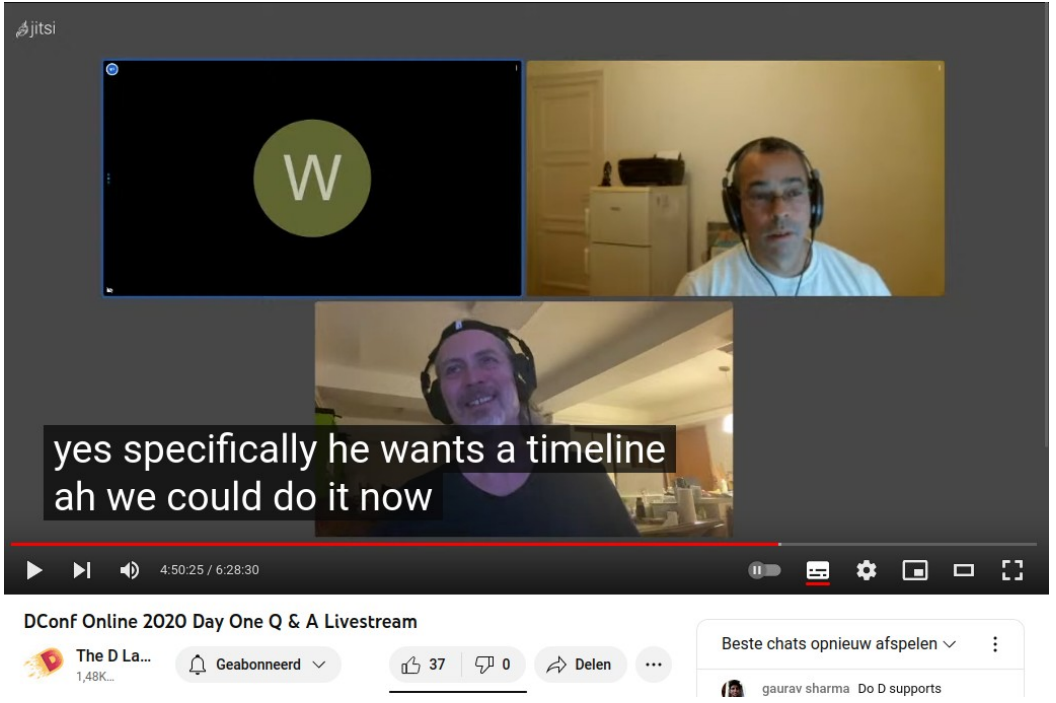
Compiles with `-preview=dip1000`

DConf Online 2020

2020



Walter: spec needs to be finished
Atila: we need to turn on warnings for DIP1000 violations



My involvement

2021

One day,

writing `@safe pure -dip1000` code,

memory corruption,

the compiler wrongly stack allocated an array literal

```
void f(char[] x) pure; // x must be scope

char[] g(char[] x) pure; // x mUsT Be SCoPe

g(['a', 'b']) // okay to stack allocate
```

My involvement 2021

- dip1000 + pure is a DEADLY COMBO
  https://forum.dlang.org/thread/jnkdcngzytgtobihzggj@forum.dlang.org
- Down the rabbit hole
- DIP1000: The return of 'Extend Return Scope Semantics'
  https://forum.dlang.org/thread/zzovywgswjmwneqwbdnm@forum.dlang.org
- DIP1000: 'return scope' ambiguity and why you can't make opIndex work
  https://forum.dlang.org/post/nbbtdbgifaurxoknyeuu@forum.dlang.org

My involvement

2022

- Made DIP1000 errors consistent
- Deprecation warnings for DIP1000 now enabled

lifetime violations	default	-preview=dip1000
`@safe`	warn	error
`auto`	warn if called from `@safe`	infer `@system`
`@system`	allowed	allowed

https://github.com/dlang/dmd/pull/14102

My involvement

2023

- Deprecation warnings now disabled

make new safety checks warnings when using default feature setting #15411

Merged **dkorpel** merged 1 commit into `dlang:master` from `WalterBright:safeObsolete` on Jul 27

Conversation 7 | Commits 1 | Checks 40 | Files changed 5

WalterBright commented on Jul 14 — Member

Consider this error:

```
../.dub/packages/vibe-core/1.22.6/vibe-core/source/vibe/core/log.d(426,19): Deprecation: scope variable `text`
```

which occurs when compiling buildkite/dlang-tour/core. It's been failing for quite a while now, and doesn't get fixed. There are pages of variations of this error. A simple repro of the error:

```
@safe:
void foo(int* p);
void bar(scope int* abc) {    foo(abc); }
```

so yes, it is an error according to dip1000. But this message is occurring by default. What we did is break everyone's code that was working, and working when in good faith they added `scope` and it remained working, but now it fails to compile. The fact that this is long time broken in dlang-tour/core shows why we need to fix this.

Reviewers: dkorpel

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

https://github.com/dlang/dmd/pull/15411


Past issues

Bad implementation

```
2415 +     /*********************************
2416 +      * Determine if `this` has a lifetime that lasts past
2417 +      * the destruction of `v`
2418 +      * Params:
2419 +      *  v = variable to test against
2420 +      * Returns:
2421 +      *  true if it does
2422 +      */
2423 +     final bool enclosesLifetimeOf(VarDeclaration v) const pure
2424 +     {
2425 +         return sequenceNumber < v.sequenceNumber;
2426 +     }
2427   }
```

```
__gshared uint nextSequenceNumber;

class VarDeclaration : Declaration
{
    this(...)
    {
        sequenceNumber = ++nextSequenceNumber;
    }
}
```

Global variable incremented in constructor

Bad implementation

- Parameters are created later

```
final bool enclosesLifetimeOf(VarDeclaration v) const pure
{
    return sequenceNumber < v.sequenceNumber;
    // FIXME: VarDeclaration's for parameters are created in semantic3, so
    //        they will have a greater sequence number than local variables.
    //        Hence reverse the result for mixed comparisons.
    const exp = this.isParameter() == v.isParameter();

    return (sequenceNumber < v.sequenceNumber) == exp;
}
```

Bad implementation

```
final bool enclosesLifetimeOf(VarDeclaration v) const pure
{
    // VarDeclaration's with these STC's need special treatment
    enum special = STC.temp | STC.foreach_;

    // Sequence numbers work when there are no special VarDeclaration's involved
    if (!((this.storage_class | v.storage_class) & special))
    {
        // FIXME: VarDeclaration's for parameters are created in semantic3, so
        //        they will have a greater sequence number than local variables.
        //        Hence reverse the result for mixed comparisons.
        const exp = this.isParameter() == v.isParameter();

        return (this.sequenceNumber < v.sequenceNumber) == exp;
    }

    // Assume that semantic produces temporaries according to their lifetime
    // (It won't create a temporary before the actual content)
    if ((this.storage_class & special) && (v.storage_class & special))
        return this.sequenceNumber < v.sequenceNumber;

    // Fall back to lexical order
    assert(this.loc != Loc.initial);
    assert(v.loc != Loc.initial);

    if (auto ld = this.loc.linnum - v.loc.linnum)
        return ld < 0;
    if (this.loc.linnum != v.loc.linnum)
        return this.loc.linnum < v.loc.linnum;

    if (auto cd = this.loc.charnum - v.loc.charnum)
        return cd < 0;
    if (this.loc.charnum != v.loc.charnum)
        return this.loc.charnum < v.loc.charnum;
```

Became this mess

Fixed now by incrementing `sequenceNumber` later

Code duplication

Caller

30 compiler/src/dmd/expressionsem.d

```
@@ -2011,11 +2011,18 @@ private bool functionParameters(const ref Loc loc, Scope* sc,
2011                 return errorInout(wildmatch);
2012         }
2013
2014 -       Expression firstArg = ((tf.next && tf.next.ty == Tvoid || isCtorCall) &&
2015 -                              tthis &&
2016 -                              tthis.isMutable() && tthis.toBasetype().ty == Tstruct &&
2017 -                              tthis.hasPointers())
2018 -                             ? ethis : null;
```

```
2180                 arg = arg.optimize(WANTvalue, p.isReference());
2181 -
2182 -               /* Determine if this parameter is the "first reference" parameter through
                     which
2183 -                * later "return" arguments can be stored.
2184 -                */
2185 -               if (i == 0 && !tthis && p.isReference() && p.type &&
2186 -                   (tf.next && tf.next.ty == Tvoid || isCtorCall))
2187 -               {
2188 -                   Type tb = p.type.baseElemOf();
2189 -                   if (tb.isMutable() && tb.hasPointers())
2190 -                   {
2191 -                       firstArg = arg;
2192 -                   }
2193 -               }
2194             }
```

Callee

```
@@ -653,30 +693,23 @@ bool checkAssignEscape(Scope* sc, Expression e, bool gag, bool byRef)
653         const bool vaIsRef = va && va.isParameter() && va.isReference();
654         if (log && vaIsRef) printf("va is ref `%s`\n", va.toChars());
655
656 -       /* Determine if va is the first parameter, through which other 'return' parameters
657 -        * can be assigned.
658 -        * This works the same as returning the value via a return statement.
659 -        * Although va is marked as `ref`, it is not regarded as returning by `ref`.
660 -        * https://dlang.org.spec/function.html#return-ref-parameters
661 -        */
662 -       bool isFirstRef()
663         {
664 -           if (!vaIsRef)
665 -               return false;
666 -           Dsymbol p = va.toParent2();
667 -           if (p == fd && fd.type && fd.type.isTypeFunction())
668             {
669 -               TypeFunction tf = fd.type.isTypeFunction();
670 -               if (!tf.nextOf() || (tf.nextOf().ty != Tvoid && !fd.isCtorDeclaration()))
671 -                   return false;
672 -               if (va == fd.vthis) // `this` of a non-static member function is considered to
                     be the first parameter
673 -                   return true;
674 -               if (!fd.vthis && fd.parameters && fd.parameters.length && (*fd.parameters)[0]
                     == va) // va is first parameter
675 -                   return true;
676             }
677 -           return false;
678         }
679 -       const bool vaIsFirstRef = isFirstRef();
680         if (log && vaIsFirstRef) printf("va is first ref `%s`\n", va.toChars());
```

Code duplication

- Caller / callee
- **this** parameter / regular parameters
- escape by value / escape by reference
- assign expression / return statement / function call

Overfitted bug fixes

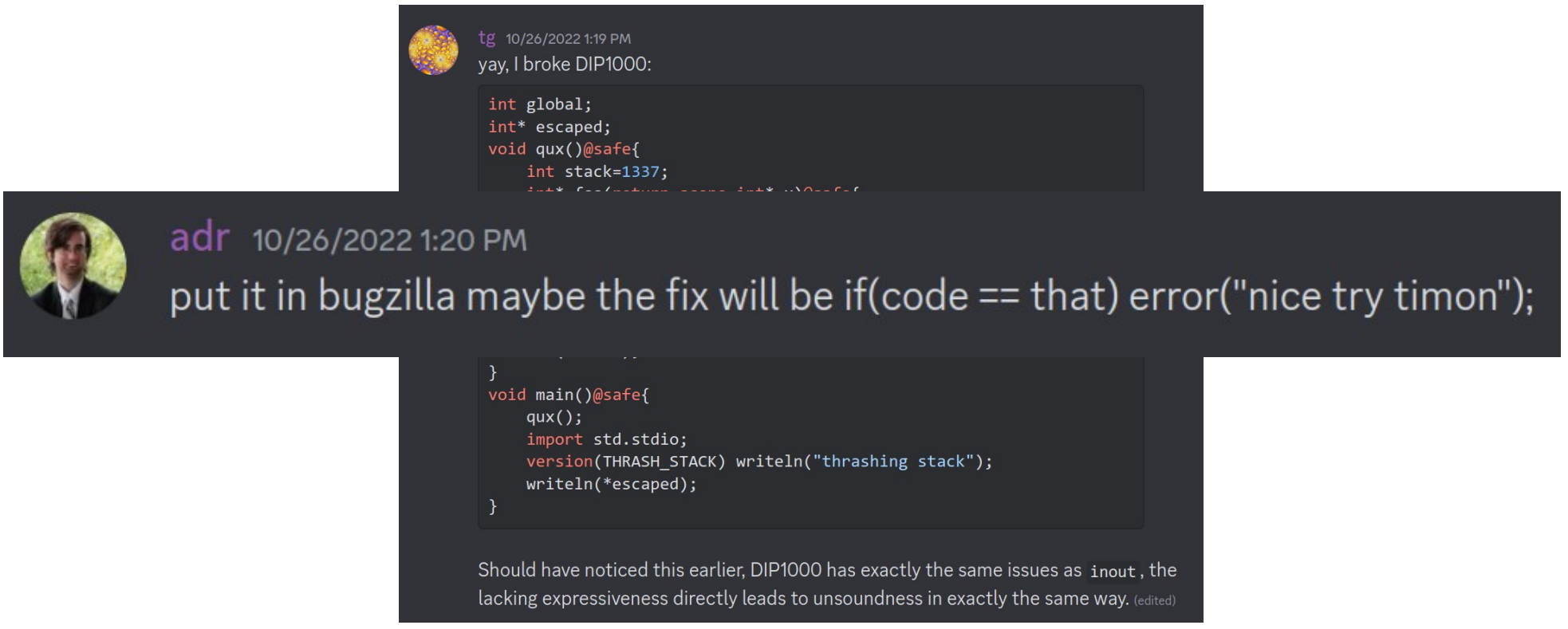
- Someone files Bugzilla issue
- Pull Request: fixes only the issue’s code snippet
- Code review: what about other cases?
- Walter: separate issue

Overfitted bug fixes

- **return ref scope** ambiguity
- Even compiler was confused
- Walter: but it’s fixed now
- Me: no it’s not

https://github.com/dlang/dmd/pull/13357
https://github.com/dlang/dmd/pull/13677
https://github.com/dlang/dmd/pull/13691
https://github.com/dlang/dmd/pull/13693
https://github.com/dlang/dmd/pull/13802
...

Overfitted bug fixes




Current issues

Scope inference

- Start of function analysis: parameters are *maybeScope*
- Take the address / assign it to non-scope: not *maybeScope*
- Return (reference to) the variable: infer **return ref / scope**
- End of function analysis: turn *maybeScope* into **scope**

Scope inference

- Killed by assignment to temporaries

```
int* f()(int* p)
{
    auto p2 = p; // p not maybeScope anymore
    return new int;
}
```

https://issues.dlang.org/show_bug.cgi?id=20674

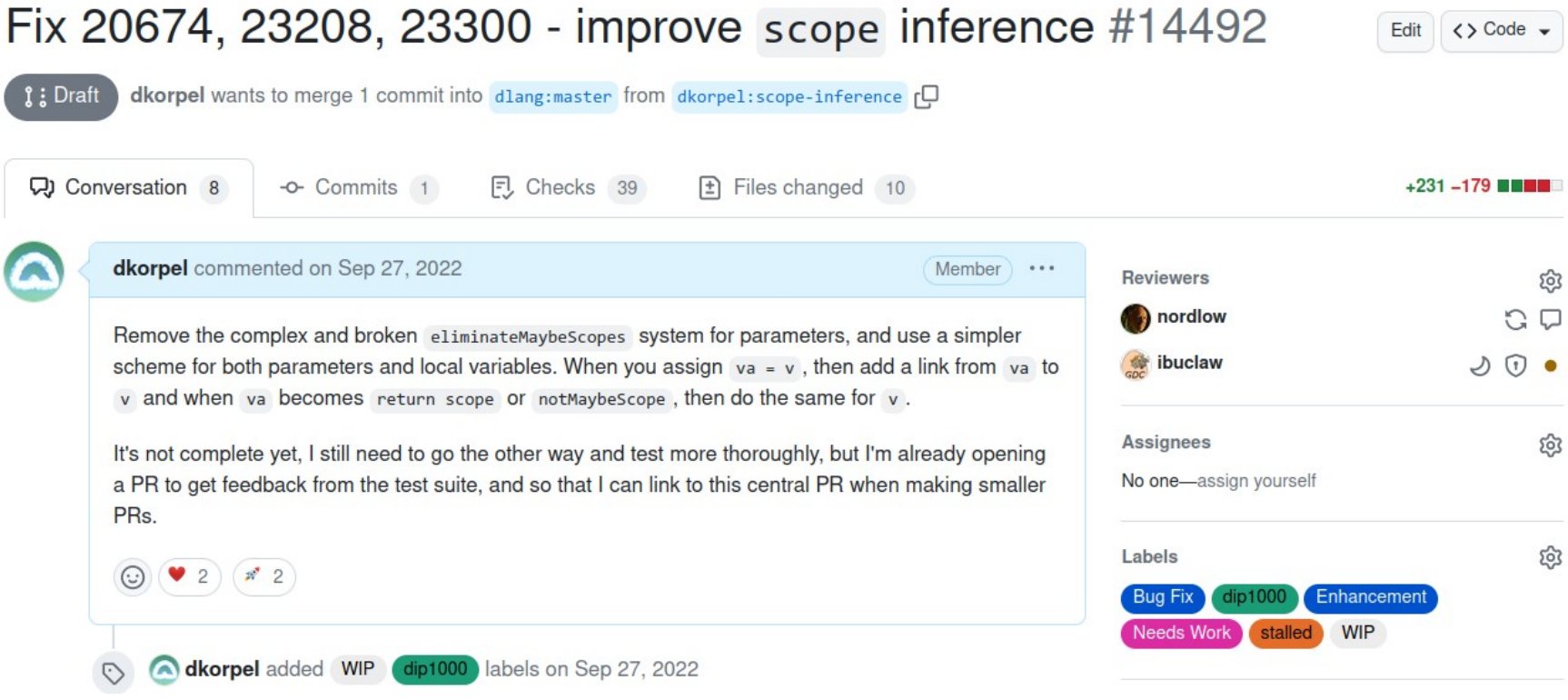
Scope inference

- Missing return scope inference

```
int* rsfail()(scope int* p, int* r) @safe
{
    r = p;
    return r; // should infer return scope on p
}
```

https://issues.dlang.org/show_bug.cgi?id=23208

Improve scope inference



Fix 20674, 23208, 23300 - improve `scope` inference #14492

Edit | Code

Draft **dkorpel** wants to merge 1 commit into `dlang:master` from `dkorpel:scope-inference`

Conversation 8 | Commits 1 | Checks 39 | Files changed 10 | +231 −179

dkorpel commented on Sep 27, 2022 — Member

Remove the complex and broken `eliminateMaybeScopes` system for parameters, and use a simpler scheme for both parameters and local variables. When you assign `va = v`, then add a link from `va` to `v` and when `va` becomes `return scope` or `notMaybeScope`, then do the same for `v`.

It's not complete yet, I still need to go the other way and test more thoroughly, but I'm already opening a PR to get feedback from the test suite, and so that I can link to this central PR when making smaller PRs.

❤ 2 🚀 2

dkorpel added WIP dip1000 labels on Sep 27, 2022

Reviewers
nordlow
ibuclaw

Assignees
No one—assign yourself

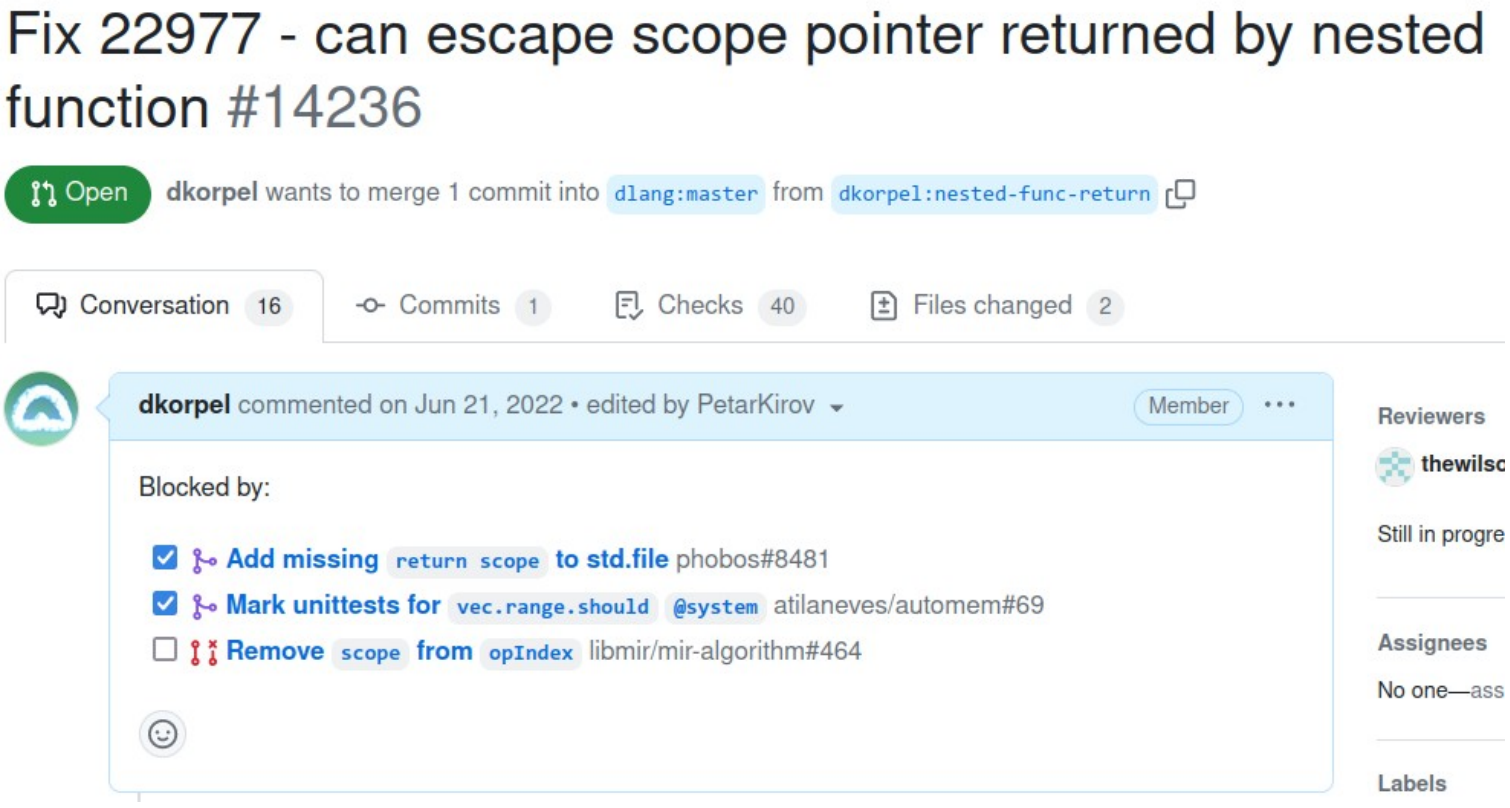
Labels
Bug Fix | dip1000 | Enhancement | Needs Work | stalled | WIP

Nested functions

- Accessing outer variables

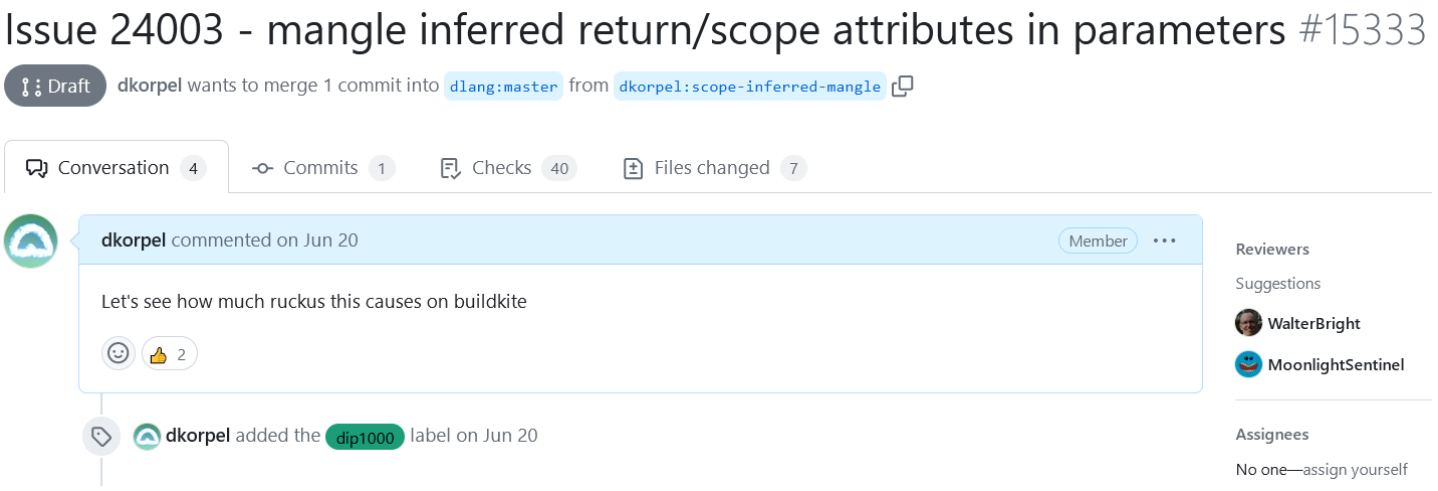
```
auto p0(scope string s) @safe
{
    string scfunc() { return s; }
    return scfunc();
}
```

Nested functions



Mangled names

- Inferred scope ignored in mangle
- Compiler internally compares types by mangle
- Solution: same scope inference without -dip1000




Limitations in design

scope is not precise

- Applies to single pointer object
- Not struct members
- Only one level of indirection

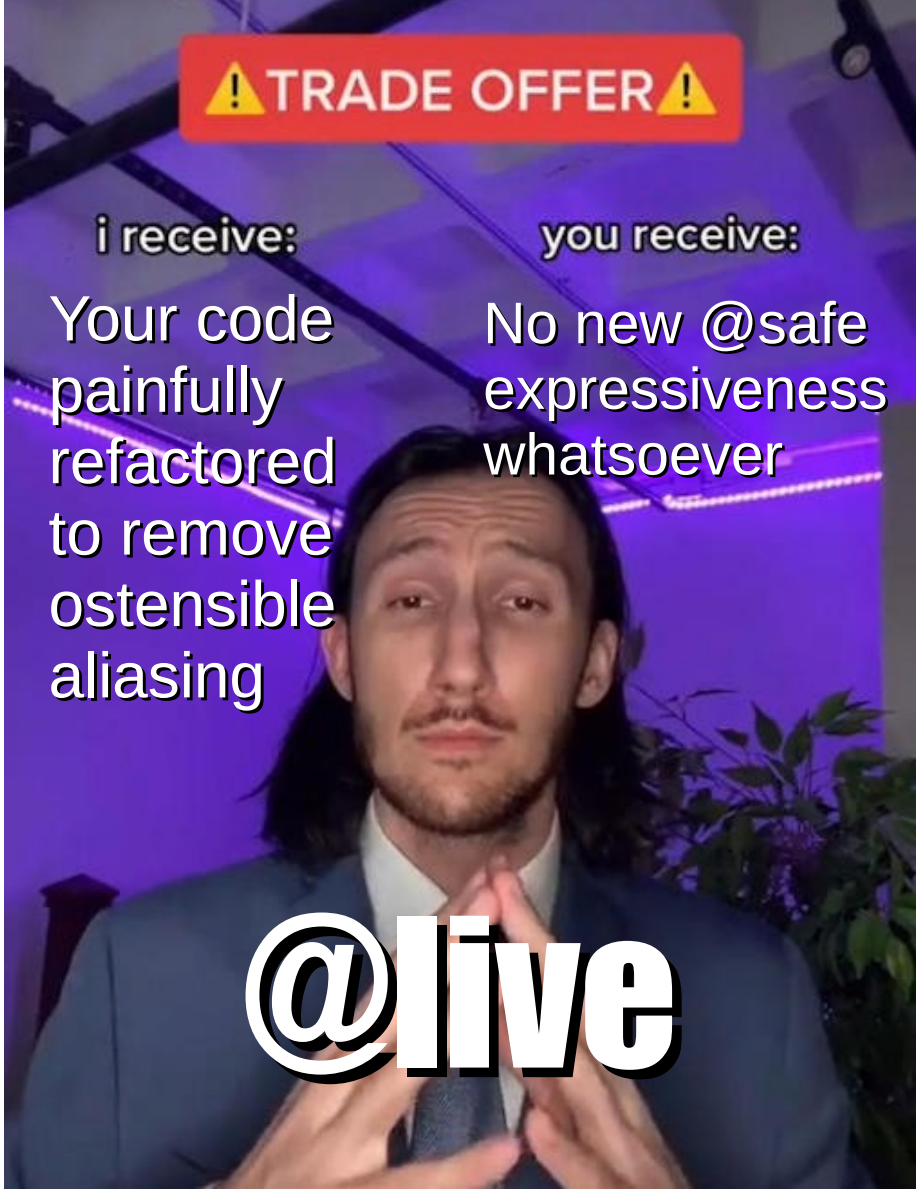
Resizing

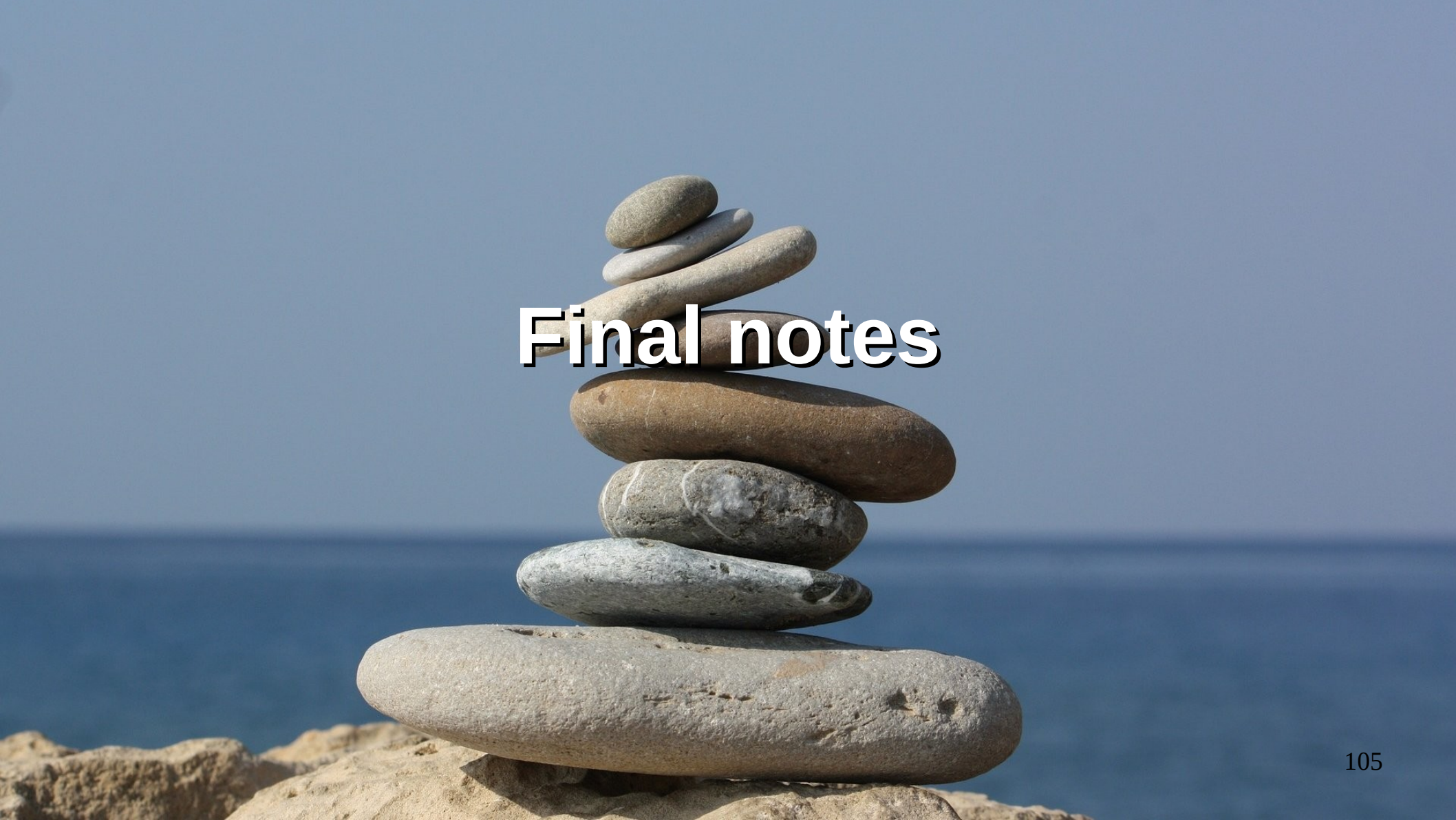
```
void main() @safe
{
    import automem.vector;

    auto vec1 = vector(1, 2, 3);
    int[] slice1 = vec1[];
    vec1.reserve(4096);
    int[] slice2 = vec1[];
    // slice 1 is dangling pointer now
}
```

https://github.com/atilaneves/automem/issues/25

`-preview=dip1021` and `@live`

- Attempt to add ownership and borrowing
- Manual `free()` / resize is still `@system`
- Don’t enable any new `@safe` / `@trusted` code


⚠TRADE OFFER⚠
i receive:
Your code painfully refactored to remove ostensible aliasing
you receive:
No new @safe expressiveness whatsoever
@live



Final notes

General lessons

- Tests and documentation good
- Code duplication bad
- Find root cause of Bugzilla issue
- Fix unstable foundation
- rejects-valid better than accepts-invalid

My verdict

- Prefer no allocation or stack allocation
- DIP1000 is a simple idea
- Complex execution
- Works best with flat data (textures, audio samples, matrices)