Stack memory is AWESOME
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

```bash
$ 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

<table>
<thead>
<tr>
<th>Time</th>
<th>Session Title</th>
<th>Speaker</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:45</td>
<td>Lunch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:30</td>
<td>Stack Memory is Awesome!</td>
<td>Dennis Korpel</td>
<td>[Show Details]</td>
</tr>
<tr>
<td>14:30</td>
<td>Simple @safe D</td>
<td>Robert Schadek</td>
<td>[Hide Details]</td>
</tr>
</tbody>
</table>

**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
string getPlaque()
{
    return "eternal star";
}

immutable int maxCoins = 999;

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

```
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
```
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

```cpp
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;
}
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>int x = 0</td>
<td>factorial(0)</td>
</tr>
<tr>
<td>1001</td>
<td>int result = ...</td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>framePtr = 1016</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>int x = 1</td>
<td>factorial(1)</td>
</tr>
<tr>
<td>1016</td>
<td>int result = ...</td>
<td></td>
</tr>
<tr>
<td>1020</td>
<td>framePtr = 1028</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>int x = 2</td>
<td>factorial(2)</td>
</tr>
<tr>
<td>1028</td>
<td>int result = ...</td>
<td></td>
</tr>
<tr>
<td>1032</td>
<td>framePtr = 1040</td>
<td></td>
</tr>
<tr>
<td>1036</td>
<td>int x = 3</td>
<td>factorial(3)</td>
</tr>
<tr>
<td>1040</td>
<td>int result = ...</td>
<td></td>
</tr>
<tr>
<td>1044</td>
<td>framePtr = 1048</td>
<td></td>
</tr>
<tr>
<td>1048</td>
<td>int result = ...</td>
<td></td>
</tr>
</tbody>
</table>
```

```
framePtr = 1052
 ...
 ...
```
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

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

```assembly
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 esp, 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)

```java
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!
void main()
{
    string[] words = "BitDW BitFS BitS".split();
    foreach(word; words)
    {
        writeln(word);
    }
}
No memory allocation

```cpp
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

```plaintext
immutable int[] primes = [2, 3, 5, 7, 11, 13];
immutable creditsText = "Created by Dennis";
immutable imgIcon = import("icon.bin");
```

Heap: large / compressed files

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

Please don’t use global mutable memory
Dynamic data

Heap
- Fragmentation
- Complex
- Wastes bytes on alignment and metadata

Stack
- Non-deterministic
- Full of indirections
- Performs syscalls

Tightly packed
- Simple
- No overhead

Predictable
- Cache friendly
- Instant allocation and de-allocation
Limitations of

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

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

// Error: returning `a[]` escapes a reference to local variable `a`
Using stack memory more
Example in D

```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

```c
import core.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)

"The specified string is copied before this function returns"
DIP1000
(Not the best name)
**DIP1000**

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

```c
int* outside;

void monkeyCage() @safe
{
    int star;
    scope int* sp = &star;
    outside = sp;  // Error: sp escapes scope of monkeyCage
}
```
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: \( \text{return } v \)
- Parameter assignment: \( f(v) \)
- Array literal assignment: \( [v] \)
Has pointers?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>int*</td>
<td>int</td>
</tr>
<tr>
<td>int[]</td>
<td>int[4]</td>
</tr>
<tr>
<td>class C</td>
<td>struct {int x;}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p[0 .. 1]</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>s.x</code></td>
<td><code>s</code></td>
</tr>
<tr>
<td><code>s.b ? s.x : p</code></td>
<td><code>s, p</code></td>
</tr>
</tbody>
</table>

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

int* p;
S s;
```
Lifetime?

• Lexical order of scope variables

```c
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

<table>
<thead>
<tr>
<th>Care</th>
<th>Don't care</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Expressions</td>
</tr>
<tr>
<td>Has it pointers?</td>
<td>Exact type</td>
</tr>
<tr>
<td>Assignments</td>
<td>Control flow</td>
</tr>
</tbody>
</table>
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()
return scope

- Lifetime in between global and scope
- The ‘inout’ of lifetime: scope in, scope out
- non-scope in, non-scope out

```c
int* identity(@safe return scope int* x)
{
    return x;
}
```
return ref

• D has ref parameters, passed by pointer
• Not pointer types
• scope is ‘built-in’

```c
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

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

- **local variable in, scope out**

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

<table>
<thead>
<tr>
<th>Action</th>
<th>Storage class</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>return &amp;p</td>
<td>return ref</td>
<td>escape by reference</td>
</tr>
<tr>
<td>return p</td>
<td>return scope</td>
<td>escape by value</td>
</tr>
<tr>
<td>return *p</td>
<td>scope</td>
<td>no escaping</td>
</tr>
</tbody>
</table>
Parameter storage classes

<table>
<thead>
<tr>
<th>Static array</th>
<th>Dynamic array</th>
<th>Storage class</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>return a[]</td>
<td>return &amp;a</td>
<td>return ref</td>
<td>escape by reference</td>
</tr>
<tr>
<td>return a[0]</td>
<td>return a[]</td>
<td>return scope</td>
<td>escape by value</td>
</tr>
<tr>
<td>return *a[0]</td>
<td>return a[0]</td>
<td>scope</td>
<td>no escaping</td>
</tr>
</tbody>
</table>

Subtle differences between static/dynamic array operations!
Member functions

```c
struct S {
    int x;

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

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

Member functions have hidden this parameter
Member functions

```c
struct S {
    int x;
}

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

```c++
struct S {
    int x;
};

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

```cpp
struct S {
    int x;

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

Modifiers for `this` parameter outside parameter list
Member functions

```c
struct S {
    int x;
};

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

```c
struct S {
    int x;

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

Same applies to `return`
Looks silly

DConf '22 Lightning Talks
www.youtube.be/GOKI7AQJR0
Example in C

```c
import core.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

```d
struct StackString
{
    char[128] buffer;
    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

```cpp
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

```c
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

```cpp
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`
**scope transitivity**

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

```c
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

```plaintext
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;
    }
}
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 .. $];
    }
}
return ref and scope

\[ \text{ref \ + \ return \ scope} \]
\[ \text{return \ ref \ + \ scope} \]

```cpp
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
Sat Aug 26 2023 13:43:58 UTC
"You want to go forward, what do you do? You put it in D." -- Barack Obama

## Summary: dip1000

146 issues found.

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

- DIP25 Introduced return ref

www.dlang.org/dips/25
DIP 1000

“Superseded”

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

return scope: first support #5972

MartinNowak merged 14 commits into dlang:scope from WalterBright:return-scope on Nov 1, 2016

WalterBright commented on Jul 25, 2016

Works much like return ref.

Fix https://issues.dlang.org/show_bug.cgi?id=5270
Fix https://issues.dlang.org/show_bug.cgi?id=6993
Fix https://issues.dlang.org/show_bug.cgi?id=14238
Fix https://issues.dlang.org/show_bug.cgi?id=15544
Fix https://issues.dlang.org/show_bug.cgi?id=15996

www.github.com/dlang/dmd/pull/5972
• Breaking change

```c
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

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

```cpp
auto drawText(/*scope*/ string txt) {
}
pragma(msg, drawText.mangleof);

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

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

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

rainers commented on Jun 6, 2017

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

clang-bot commented on Jun 6, 2017

<table>
<thead>
<tr>
<th>Fix</th>
<th>Bugzilla</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>17432</td>
<td>[DIP1000] scope delegates change type, but not mangling</td>
</tr>
</tbody>
</table>
Extend Return Scope Semantics

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

https://github.com/dlang/dmd/pull/8504
void assign(ref scope int* target, return scope int* source) @safe {
    target = source;
}

void main() @safe {
    int x;
    int* y;
    assign(y, &x); // allowed
}
Common to assign to `this` parameter

```c
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

```c
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

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

Compiles with -preview=dip1000

2019
Mathias Lang @The D Language Foundation Is there a plan to enable DIP1000 by default?

Mathias Lang Specifically, a timeline

Walter: spec needs to be finished

Atila: we need to turn on warnings for DIP1000 violations
One day, writing @safe pure -dip1000 code, memory corruption, the compiler wrongly stack allocated an array literal

```c
void f(char[] x) pure; // x must be scope
char[] g(char[] x) pure; // x must be scope
char[] g(['a', 'b']); // okay to stack allocate
```
My involvement

2021

- dip1000 + pure is a DEADLY COMBO
  https://forum.dlang.org/thread/jnkdcngzytobihzggj@forum.dlang.org

- Down the rabbit hole

- DIP1000: The return of 'Extend Return Scope Semantics'
  https://forum.dlang.org/thread/zzovyywsgwjmweqwbdnm@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

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

<table>
<thead>
<tr>
<th>lifetime violations</th>
<th>default</th>
<th>-preview=dip1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>@safe</td>
<td>warn</td>
<td>error</td>
</tr>
<tr>
<td>auto</td>
<td>warn if called from @safe</td>
<td>infer @system</td>
</tr>
<tr>
<td>@system</td>
<td>allowed</td>
<td>allowed</td>
</tr>
</tbody>
</table>

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

- Deprecation warnings now disabled

https://github.com/dlang/dmd/pull/15411
Past issues
Bad implementation

Global variable incremented in constructor
Bad implementation

• Parameters are created later

```java
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

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

    // 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 temporary's 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 cd = this.loc.line = v.loc.line)
        return cd < 0;
    if (this.loc.line == v.loc.line)
        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

```
2180   arg = arg.optimize(WANTValue, p.isReference());

2182   /* Determine if this parameter is the "first reference" parameter through
       which
       * later "return" arguments can be stored.
       */
2183   if (l == 0 && this && p.isReference() && p.type &&
2184      (tf.next && tf.next.ty == TVoid || IsCtorCall))
2185     [Type tb = t.type.baseElement();
2186      if (tb.isMutable() && tb.hasPointers())
2187        firstArg = arg;
2188    }
```

Callee

```
643   const bool vaIsFirstRef = va && va.isReference() && va.isReference();
644   if (log && vaIsRef) printf("va is ref \"%s\"", va.toChars());
645
656   /* Determine if va is the first parameter, through which other 'return' parameters
      * can be assigned.
      */
657   if (va && va != va.op->getParent2();
658   if (p == fd && fd.type && fd.type.isTypeFunction())
659     [Type tf = fd.type.isTypeFunction();
660      if ((tf.next() || (tf.nextOf(fd) ? tf.ty == TVoid && fd.isCtorDeclaration())))
661      return false;
662    else if (va == fd.value()) 'this' of a non-static member function is considered to
663      be the first parameter
664      return true;
665    else if (fd.this && fd.parameters && fd.parameters.length && (*fd.parameters)[0]
666      == va) // va is first parameter
667      return true;
668
669    return false;
670  }
```

87
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

```cpp
int global;
int* escaped;
void qux() @safe{
    int stack=1337;
    // ...
}
```

adr 10/26/2022 1:20 PM

```
put it in bugzilla maybe the fix will be if(code == that) error("nice try timon");
```

```cpp
void main() @safe{
    qux();
    import std.stdio;
    version(THRASH_STACK) writeln("thraashing stack");
    writeln("*escaped");
}
```

Should have noticed this earlier, DIP1000 has exactly the same issues as `inout`, the lacking expressiveness directly leads to unsoundness in exactly the same way. (edited)
Scope inference

• Start of function analysis: parameters are \textit{maybeScope}
• Take the address / assign it to non-scope: not \textit{maybeScope}
• Return (reference to) the variable: infer \textit{return ref / scope}
• End of function analysis: turn \textit{maybeScope} into \textit{scope}
Scope inference

• Killed by assignment to temporaries

```cpp
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

```c
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
Nested functions

- Accessing outer variables

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

Fix 22977 - can escape scope pointer returned by nested function #14236

dkorpel commented on Jun 21, 2022 • edited by PetarKirov

Blocked by:

- Add missing `return scope` to `std_file` phobos#8481
- Mark unitests for `vec_range.should` @system atilaneves/autotest#69
- Remove `scope` from `opindex` libmir/mir-algorithm#464
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
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
i receive:  you receive:

Your code painfully refactored to remove ostensible aliasing

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)