Simple @safe D

Dr. Robert Schadek

DConf 2023

Simple @safe D

How to make enemies quickly

Dr. Robert Schadek

DConf 2023

The Problem 1/2

```d
class C { // Boilerplate free code
  int theI;

  nothrow C fun(return ref int i) return scope @safe pure {
    i = this.theI;
    return this;
  }
}

@safe unittest {
  C c = new C();
  int i;
  C c2 = c.fun(i);
}
```

The Problem 2/2

- Just look at all the syntax
- DIP1035 -- `@system` Variables ...

The Problem 2/2

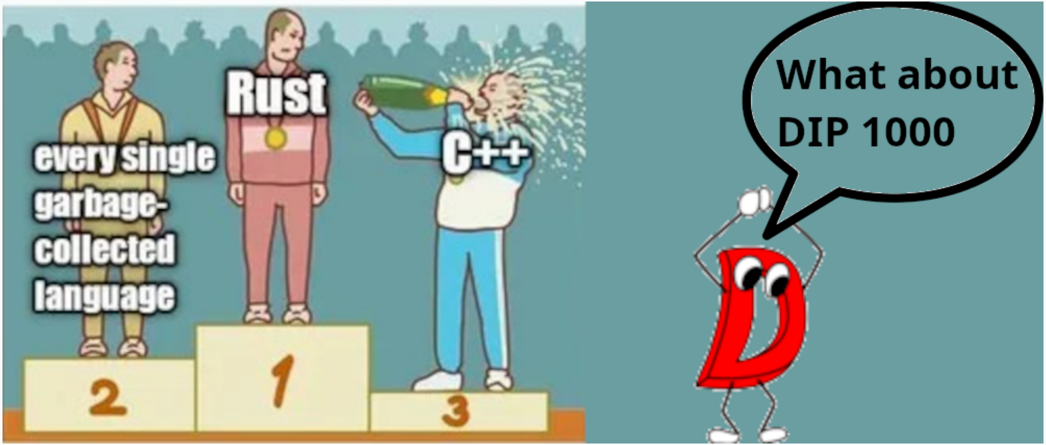
- Just look at all the syntax
- DIP1035 -- `@system` Variables ...

- Thinking that D is the C/C++ successor ... it is not, that is rust
- Thinking `@safe` languages are the new thing ... they are not. Most languages are safe already, python, JS, java

Memory safety in C++77

C++
C++
C++
C++
every single garbage-collected language
Rust
C++
2
1
3

Memory safety in C++77



Hopefully, there are still some people in the room with me at this point

DIP1000

```d
void thorin(scope int*);
void gloin(int*);
int* balin(scope int* q, int* r) {
  /* error, q escapes to global gp
  gp = q; */
  gp = r; // ok

  // ok, q does not escape thorin()
  thorin(q);
  thorin(r); // ok

  /* error, gloin() escapes q
  gloin(q); */
  gloin(r); // ok that gloin() escapes r

  /* error, cannot return 'scope' q
  return q; */
  return r; // ok
}
```

```d
  // Will not even parse
  int a, c;
  int* b = balin(&a, &a);

  // The GC will check the owns
  // for us
  int* c = new int;
  int* d = balin(c, c);
}
```

The Solution

- instead of adding things ... lets remove things

The Solution

- instead of adding things … lets remove things

- old school `@safe`
- No unary `&` --- remove this from the grammar in `@safe`
- No `return` by `ref`
- No slicing of static arrays

The Consequences

- No need for DIP1000, DIP1021, and DIP 1035
- No user defined `@safe` container that behave like in-builds
- No Manual Memory Management (MMM) in `@safe` code
- Clear definition of `@property`
- etc...

Consequences and Remedies

Passing data down

```
@safe:

void main() {
  int v = 10;
  child(&v);
}

void child(scope int* i) {
}
```

Passing data down

```
@safe:

void main() {
  int v = 10;
  child(&v);
}

void child(scope int* i) {
}
```

```
@safe:

void main() {
  int v = 10;
  child(v);
}

void child(ref int i) {
}
```

Passing data up

```
@safe:

void main() {
  int v;
  child(&v);
}

void child(scope int* i) {
  *i = 10;
}
```

Passing data up

```
@safe:

void main() {
  int v;
  child(&v);
}

void child(scope int* i) {
  *i = 10;
}
```

```
@safe:

void main() {
  int v;
  child(v);
}

void child(out int i) {
  i = 10;
}
```

Container

```
void fun2(const(int)[] arr) {
}

void fun3() {
  const(int[]) arr = [1,2,3];
  fun2(arr);
}
```

returning ref

```
// @safe:

struct Array {
  int[10] arr;

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

void main() {
  int* a = &fun();
}

ref int fun() {
  Array a;
  return a[2];
}
```

returning ref

```
// @safe:

struct Array {
  int[10] arr;

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

void main() {
  int* a = &fun();
}

ref int fun() {
  Array a;
  return a[2];
}
```

```
@safe:

struct Array {
  @safe:
  int[10] arr;

  void get(size_t i, out int into) {
    into = this.arr[i];
  }
}

void fun(out int into) {
  Array a;
  a.get(2, into);
}

void main() {
  int a;
  fun(a);
}
```

returning ref

```
import std.typecons : Nullable;

@safe:

struct Array {
  @safe:
  int[10] arr;

  void get(size_t i
      , out Nullable!int into)
  {
    if(i < this.arr.length) {
      into = this.arr[i];
    }
  }
}
```

returning ref

```
import std.typecons : Nullable;

@safe:

struct Array {
  @safe:
  int[10] arr;

  void get(size_t i
      , out Nullable!int into)
  {
    if(i < this.arr.length) {
      into = this.arr[i];
    }
  }
}
```

```
void fun(out Nullable!int into) {
  Array a;
  a.get(2, into);
}

void main() {
  Nullable!int a;
  fun(a);
}
```

@property what do you even get?

```
struct S {
  int b;
  @property ref a() {
    return b;
  }
}

void main() {
  S s;
  auto ptr = &s.a;
}
```

not being smart

```
int uniform(int l, int h) {
    int i = h - l;
    foreach(it; l .. h) {
        i += (it * 1337) % 15;
    }
    return i;
}

struct A {
    int[100] a;
}

A fun() {
    A a;
    foreach(i; 0 .. 100) {
        a.a[i] = uniform(0, 100);
    }
    return a;
}

int main() {
    A  a = fun();
    return a.a[5] % 100 == 0;
}
```

much tool use, too little tool application.

Continuations

assertS

```
class Assert {
  int a = 1000;
  invariant {
    assert(a != 0);
  }

  int fun()
    in {
      assert(a != 0);
    }
    out(ret) {
      assert(ret != 0);
    }
    body {
      assert(false);
    }
}
```

```
int main() {
  Assert a = new Assert();
  assert(a.fun() != 0);
  return 0;
}
```

- `dmd -release -run asserttest.d`
- echo $?

assertS

```
class Assert {
  int a = 1000;
  invariant {
    assert(a != 0);
  }

  int fun()
    in {
      assert(a != 0);
    }
    out(ret) {
      assert(ret != 0);
    }
    body {
      assert(false);
    }
}
```

```
int main() {
  Assert a = new Assert();
  assert(a.fun() != 0);
  return 0;
}
```

- `dmd -release -run asserttest.d`
- echo $?
- No assert, in/out contrast, or invariant

Template Constraints

```d
ptrdiff_t indexOf(Range)(Range s, dchar c, CaseSensitive cs = Yes.caseSensitive)
if (isInputRange!Range && isSomeChar!(ElementType!Range) && !isSomeString!Range)
{
    return _indexOf(s, c, cs);
}

/// Ditto
ptrdiff_t indexOf(C)(scope const(C)[] s, dchar c
    , CaseSensitive cs = Yes.caseSensitive)
if (isSomeChar!C)
{
    return _indexOf(s, c, cs);
}

/// Ditto
```

Template Constraints continued

```d
ptrdiff_t indexOf(Range)(Range s, dchar c, size_t startIdx
    , CaseSensitive cs = Yes.caseSensitive)
if (isInputRange!Range && isSomeChar!(ElementType!Range) && !isSomeString!Range)
{
    return _indexOf(s, c, startIdx, cs);
}

/// Ditto
ptrdiff_t indexOf(C)(scope const(C)[] s, dchar c, size_t startIdx
    , CaseSensitive cs = Yes.caseSensitive)
if (isSomeChar!C)
{
    return _indexOf(s, c, startIdx, cs);
}
```

Template Constraints continued

```
private ptrdiff_t _indexOf(Range)(Range s, dchar c
    , CaseSensitive cs = Yes.caseSensitive)
if (isInputRange!Range && isSomeChar!(ElementType!Range))
{
  // impl here
}

private ptrdiff_t _indexOf(Range)(Range s, dchar c, size_t startIdx
    , CaseSensitive cs = Yes.caseSensitive)
if (isInputRange!Range && isSomeChar!(ElementType!Range))
{
  // impl here
}
```

Template Constraints C++

```cpp
template <unsigned N>
struct Fibonacci {
  enum {
    value = Fibonacci<N-1>::value
      + Fibonacci<N-2>::value
  };
};

template <>
struct Fibonacci<1> {
  enum {
    value = 1
  };
};

template <>
struct Fibonacci<0> {
  enum {
    value = 0
  };
};

int main() {
  int f = Fibonacci<10>::value;
  return f;
}
```

Template Constraints less terrible

```d
struct IndexOfParameter {
  Nullable!size_t startIdx;
  Nullable!CaseSensitive cs;
}

ptrdiff_t saneIndexOf(Range)(Range s, dchar c
    , IndexOfParameter idp = IndexOfParameter.init)
{
  alias ECT = ElementEncodingType!(Range);
  static assert(isSomeChar!(ECT), Range.stringof
      , " must consists of some kind of Character not "
      , ECT.stringof);

  //
  // jump depending on types and passed parameters
  //
}
```

Template Constraints less terrible

```
template unpack(T) {
  static if(is(T : Nullable!F, F)) {
    alias unpack = F;
  } else {
    alias unpack = T;
  }
}

ptrdiff_t saneIndexOf2(Range, Needle, T...)(Range r, Needle n, T args)
{
  IndexOfParameter params;
  static foreach(mem; __traits(allMembers, IndexOfParameter)) {{
    alias MT = typeof(__traits(getMember, IndexOfParameter, mem));
    alias MTUP = unpack!MT;
    static foreach(arg; args) {{
      static if(is(MTUP == typeof(arg))) {
        __traits(getMember, params, mem) = arg;
      }
    }}
  }}
```

Nested Functions

```d
string toString(int[] arr) {
  auto app = appender!string();
  size_t idx;

  void toString(int a) {
    if(idx > 0) {
      app.put(", ");
    }
    app.put(to!string(a));
  }

  foreach(it; arr) {
    toString(it);
    ++idx;
  }

  return app.data;
}
```

Nested Functions

```d
string toString(int[] arr) {
  auto app = appender!string();
  size_t idx;

  void toString(int a) {
    if(idx > 0) {
      app.put(", ");
    }
    app.put(to!string(a));
  }

  foreach(it; arr) {
    toString(it);
    ++idx;
  }

  return app.data;
}
```

- especially bad if the use parent function parameters
- pull out and make private

Nested Imports

```
string toString(int a) {
  import std.conv : to;

  return to!string(a);
}

//
// MANY MANY LINES OF CODE
//

string toString(double a) {
  import std.conv : to;

  return to!string(a);
}
```

Nested Imports

```
string toString(int a) {
  import std.conv : to;

  return to!string(a);
}

//
// MANY MANY LINES OF CODE
//

string toString(double a) {
  import std.conv : to;

  return to!string(a);
}
```

- refactoring gets a lot harder, because you never include all used symbols

Conclusions

Conclusion

- `scope`, `ref`, `return` are good things

Conclusion

- `scope`, `ref`, `return` are good things
- but not in `@safe` code

Conclusion

- `scope`, `ref`, `return` are good things
- but not in `@safe` code
- `@safe` code should be simple and safe

Conclusion

- `scope`, `ref`, `return` are good things
- but not in `@safe` code
- `@safe` code should be simple and safe
- why not use it in `@trusted`

Conclusion

- `scope`, `ref`, `return` are good things
- but not in `@safe` code
- `@safe` code should be simple and safe
- why not use it in `@trusted`
- you ain't gonna need it

Conclusion

- `scope`, `ref`, `return` are good things
- but not in `@safe` code
- `@safe` code should be simple and safe
- why not use it in `@trusted`
- you ain't gonna need it
- simple is better than complicated

The END

Appendix

Please don't add

Tuple

```
(double,double) gps() {
  double lon;
  double lat;

  return (lon,lat);
}

void main() {
  double (lat,lon) = gps();
}
```

Tuple

```
(double,double) gps() {
  double lon;
  double lat;

  return (lon,lat);
}

void main() {
  double (lat,lon) = gps();
}
```

```
import std.typecons : Tuple, tuple;
import std.math : isClose;

Tuple!(double,double) gps() {
  double lon = 1.0;
  double lat = 2.0;

  return tuple(lon,lat);
}

void main() {
  Tuple!(double,double) c = gps();
  double lon = c[1];
  double lat = c[0];

  assert(isClose(lon, 1.0));
  assert(isClose(lat, 2.0));
}
```