Simple @safe D

Dr. Robert Schadek
DConf 2023

Simple @safe D

How to make enemies quickly

Dr. Robert Schadek

DConf 2023

```
1  class C { // Boilerplate free code
2    int theI;
3
4    nothrow C fun(return ref int i) return scope @safe pure {
5      i = this.theI;
6      return this;
7    }
8  }
9
10 @safe unittest {
11   C c = new C();
12   int i;
13   C c2 = c.fun(i);
14 }
```
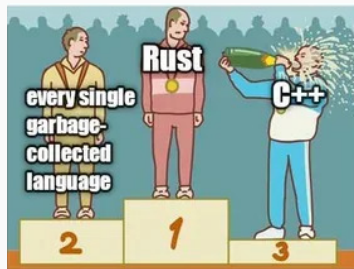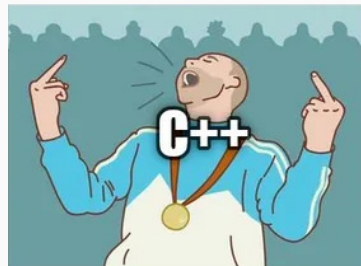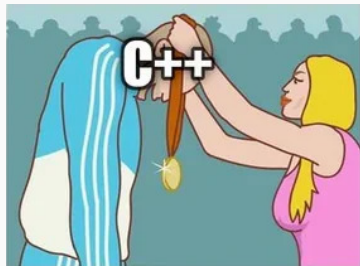
The Problem 2/2

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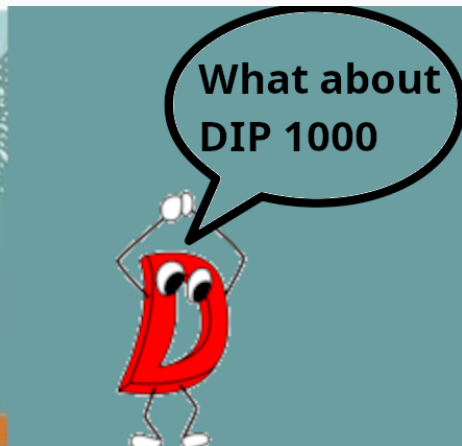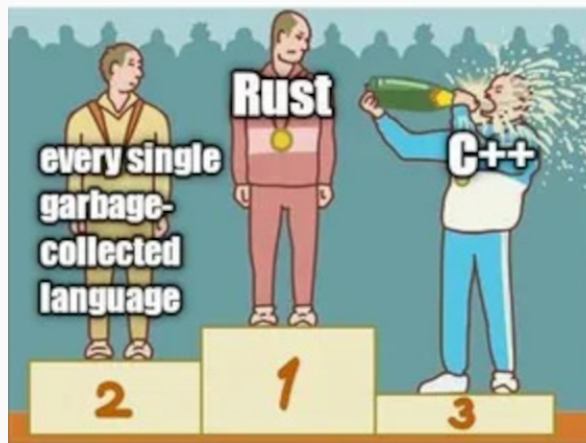
- Just look at all the syntax
- DIP1035 -- `@system` Variables ...

- 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

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

DIP1000

```
1  void thorin(scope int*);
2  void gloin(int*);
3  int* balin(scope int* q, int* r) {
4    /* error, q escapes to global gp
5    gp = q; */
6    gp = r; // ok
7
8    // ok, q does not escape thorin()
9    thorin(q);
10   thorin(r); // ok
11
12   /* error, gloin() escapes q
13   gloin(q); */
14   gloin(r); // ok that gloin() escapes r
15
16   /* error, cannot return 'scope' q
17   return q; */
18   return r; // ok
19 }
```

```
1    // Will not even parse
2    int a, c;
3    int* b = balin(&a, &a);
4
5    // The GC will check the owns
6    // for us
7    int* c = new int;
8    int* d = balin(c, c);
9  }
```

5

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

```
1  @safe:
2
3  void main() {
4    int v = 10;
5    child(&v);
6  }
7
8  void child(scope int* i) {
9  }
```

Passing data down

```
1  @safe:
2
3  void main() {
4    int v = 10;
5    child(&v);
6  }
7
8  void child(scope int* i) {
9  }
```

```
1  @safe:
2
3  void main() {
4    int v = 10;
5    child(v);
6  }
7
8  void child(ref int i) {
9  }
```

Passing data up

```
1  @safe:
2
3  void main() {
4    int v;
5    child(&v);
6  }
7
8  void child(scope int* i) {
9    *i = 10;
10 }
```

Passing data up

```
1  @safe:
2
3  void main () {
4    int v;
5    child (&v);
6  }
7
8  void child (scope int* i) {
9    *i = 10;
10 }
```

```
1  @safe:
2
3  void main () {
4    int v;
5    child (v);
6  }
7
8  void child (out int i) {
9    i = 10;
10 }
```

Container

```
1 void fun2(const(int)[] arr) {
2 }
3
4 void fun3() {
5   const(int[]) arr = [1,2,3];
6   fun2(arr);
7 }
```

returning ref

```
1  // @safe:
2
3  struct Array {
4    int[10] arr;
5
6    ref int opIndex(size_t i) {
7      return this.arr[i];
8    }
9  }
10
11 void main() {
12   int* a = &fun();
13 }
14
15 ref int fun() {
16   Array a;
17   return a[2];
18 }
```

returning ref

```
1  // @safe:
2
3  struct Array {
4    int[10] arr;
5
6    ref int opIndex(size_t i) {
7      return this.arr[i];
8    }
9  }
10
11 void main() {
12   int* a = &fun();
13 }
14
15 ref int fun() {
16   Array a;
17   return a[2];
18 }
```

```
1  @safe:
2
3  struct Array {
4    @safe:
5    int[10] arr;
6
7    void get(size_t i, ref int into) {
8      into = this.arr[i];
9    }
10 }
11
12 void fun(out int into) {
13   Array a;
14   a.get(2, into);
15 }
16
17 void main() {
18   int a;
19   fun(a);
```

```
1  import std.typecons : Nullable;
2
3  @safe:
4
5  struct Array {
6    @safe:
7    int[10] arr;
8
9    void get(size_t i
10       , ref Nullable!int into)
11   {
12     if(i < this.arr.length) {
13       into = this.arr[i];
14     }
15   }
16 }
```

returning ref

```
1  import std.typecons : Nullable;      18  void fun(out Nullable!int into) {
2                                        19    Array a;
3  @safe:                               20    a.get(2, into);
4                                        21  }
5  struct Array {                       22
6    @safe:                             23  void main() {
7    int[10] arr;                       24    Nullable!int a;
8                                        25    fun(a);
9    void get(size_t i                  26  }
10        , ref Nullable!int into)
11    {
12      if(i < this.arr.length) {
13        into = this.arr[i];
14      }
15    }
16  }
```

12

```
1  struct S {
2    int b;
3    @property ref a() {
4      return b;
5    }
6  }
7
8  void main() {
9    S s;
10   auto ptr = &s.a;
11 }
```

not being smart

```
1  int uniform(int l, int h) {      13  A fun() {
2      int i = h - l;               14      A a;
3      foreach(it; l .. h) {        15      foreach(i; 0 .. 100) {
4          i += (it * 1337) % 15;   16          a.a[i] = uniform(0, 100);
5      }                            17      }
6      return i;                    18      return a;
7  }                                19  }
8                                   20
9  struct A {                       21  int main() {
10     int[100] a;                  22      A  a = fun();
11 }                                23      return a.a[5] % 100 == 0;
                                     24  }
```

...much tool use, too little tool application.

...served.

...de 33
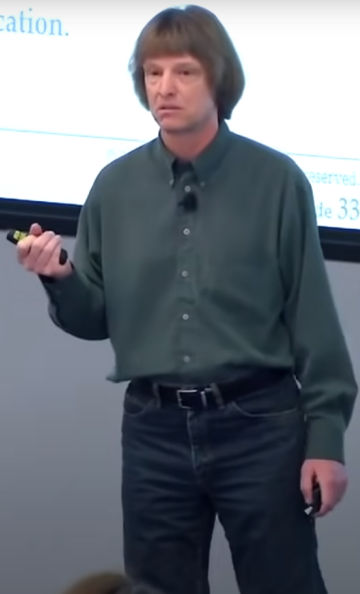
Continuations

```
1   class Assert {
2     int a = 1000;
3     invariant {
4       assert(a != 0);
5     }
6
7     int fun()
8       in {
9         assert(a != 0);
10      }
11      out(ret) {
12        assert(ret != 0);
13      }
14      body {
15        assert(false);
16      }
17  }
```

```
19  int main() {
20    Assert a = new Assert();
21    assert(a.fun() != 0);
22    return 0;
23  }
```

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

16

```
1   class Assert {
2     int a = 1000;
3     invariant {
4       assert(a != 0);
5     }
6
7     int fun()
8       in {
9         assert(a != 0);
10      }
11      out(ret) {
12        assert(ret != 0);
13      }
14      body {
15        assert(false);
16      }
17    }
```

```
19  int main() {
20    Assert a = new Assert();
21    assert(a.fun() != 0);
22    return 0;
23  }
```

- dmd -release -run asserttest.d

- echo $?

- No assert, in/out contrast, or invariant

Template Constraints

```
1  ptrdiff_t indexOf(Range)(Range s, dchar c, CaseSensitive cs = Yes.caseSensitive)
2  if (isInputRange!Range && isSomeChar!(ElementType!Range) && !isSomeString!Range)
3  {
4      return _indexOf(s, c, cs);
5  }
6
7  /// Ditto
8  ptrdiff_t indexOf(C)(scope const(C)[] s, dchar c
9      , CaseSensitive cs = Yes.caseSensitive)
10 if (isSomeChar!C)
11 {
12     return _indexOf(s, c, cs);
13 }
14
15 /// Ditto
```

Template Constraints continued

```
15  ptrdiff_t indexOf(Range)(Range s, dchar c, size_t startIdx
16      , CaseSensitive cs = Yes.caseSensitive)
17  if (isInputRange!Range && isSomeChar!(ElementType!Range) && !isSomeString!Range)
18  {
19      return _indexOf(s, c, startIdx, cs);
20  }
21
22  /// Ditto
23  ptrdiff_t indexOf(C)(scope const(C)[] s, dchar c, size_t startIdx
24      , CaseSensitive cs = Yes.caseSensitive)
25  if (isSomeChar!C)
26  {
27      return _indexOf(s, c, startIdx, cs);
28  }
```

Template Constraints continued

```
30   private ptrdiff_t _indexOf(Range)(Range s, dchar c
31        , CaseSensitive cs = Yes.caseSensitive)
32   if (isInputRange!Range && isSomeChar!(ElementType!Range))
33   {
34     // impl here
35   }
36
37   private ptrdiff_t _indexOf(Range)(Range s, dchar c, size_t startIdx
38        , CaseSensitive cs = Yes.caseSensitive)
39   if (isInputRange!Range && isSomeChar!(ElementType!Range))
40   {
41     // impl here
42   }
```

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
  };
};
```

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

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

Template Constraints less terrible

```d
1  struct IndexOfParameter {
2    Nullable!size_t startIdx;
3    Nullable!CaseSensitive cs;
4  }
5
6  ptrdiff_t saneIndexOf(Range)(Range s, dchar c
7      , IndexOfParameter idp = IndexOfParameter.init)
8  {
9    alias ECT = ElementEncodingType!(Range);
10   static assert(isSomeChar!(ECT), Range.stringof
11       , " must consists of some kind of Character not "
12       , ECT.stringof);
13
14   //
15   // jump depending on types and passed parameters
16   //
17  }
```

Template Constraints less terrible

```
1  template unpack(T) {
2    static if(is(T : Nullable!F, F)) {
3      alias unpack = F;
4    } else {
5      alias unpack = T;
6    }
7  }
8
9  ptrdiff_t saneIndexOf2(Range, Needle, T...)(Range r, Needle n, T args)
10 {
11   IndexOfParameter params;
12   static foreach(mem; __traits(allMembers, IndexOfParameter)) {{
13     alias MT = typeof(__traits(getMember, IndexOfParameter, mem));
14     alias MTUP = unpack!MT;
15     static foreach(arg; args) {{
16       static if(is(MTUP == typeof(arg))) {
17         __traits(getMember, params, mem) = arg;
18       }
19     }}
20   }}
```

Nested Functions

```
1  string toString(int[] arr) {
2    auto app = appender!string();
3    size_t idx;
4
5    void toString(int a) {
6      if(idx > 0) {
7        app.put(", ");
8      }
9      app.put(to!string(a));
10   }
11
12   foreach(it; arr) {
13     toString(it);
14     ++idx;
15   }
16
17   return app.data;
18 }
```

Nested Functions

```
1  string toString(int[] arr) {
2    auto app = appender!string();
3    size_t idx;
4
5    void toString(int a) {
6      if(idx > 0) {
7        app.put(", ");
8      }
9      app.put(to!string(a));
10   }
11
12   foreach(it; arr) {
13     toString(it);
14     ++idx;
15   }
16
17   return app.data;
18 }
```

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

Nested Imports

```
1  string toString(int a) {
2    import std.conv : to;
3
4    return to!string(a);
5  }
6
7  //
8  // MANY MANY LINES OF CODE
9  //
10
11 string toString(double a) {
12   import std.conv : to;
13
14   return to!string(a);
15 }
```

Nested Imports

```
1  string toString(int a) {
2    import std.conv : to;
3
4    return to!string(a);
5  }
6
7  //
8  // MANY MANY LINES OF CODE
9  //
10
11 string toString(double a) {
12   import std.conv : to;
13
14   return to!string(a);
15 }
```

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

Conclusions

- `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
- why not use it in `@trusted`

Conclusion

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

Conclusion

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

Conclusion

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

The END

Appendix

Please don't add

Tuple

```
1  (double,double) gps() {
2    double lon;
3    double lat;
4
5    return (lon,lat);
6  }
7
8  void main() {
9    double (lat,lon) = gps();
10 }
```

Tuple

```
1  (double,double) gps() {
2    double lon;
3    double lat;
4
5    return (lon,lat);
6  }
7
8  void main() {
9    double (lat,lon) = gps();
10 }
```

```
1  import std.typecons : Tuple, tuple;
2  import std.math : isClose;
3
4  Tuple!(double,double) gps() {
5    double lon = 1.0;
6    double lat = 2.0;
7
8    return tuple(lon,lat);
9  }
10
11 void main() {
12   Tuple!(double,double) c = gps();
13   double lon = c[1];
14   double lat = c[0];
15
16   assert(isClose(lon, 1.0));
17   assert(isClose(lat, 2.0));
18 }
```