Simple @safe D

Dr. Robert Schadek

DConf 2023

The Problem 1/2

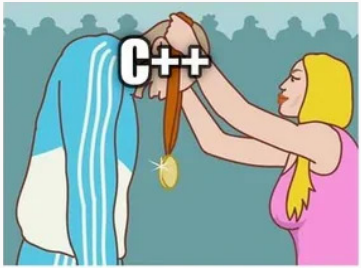
```
1  class C { // Boilerplate free code
2    int theI;
3
4    nothrow C fun(return ref int i) return scope @safe pure {
5      i = this.theI;
6      return this;
7    }
8  }
9
10 @safe unittest {
11   C c = new C();
12   int i;
13   C c2 = c.fun(i);
14 }
```

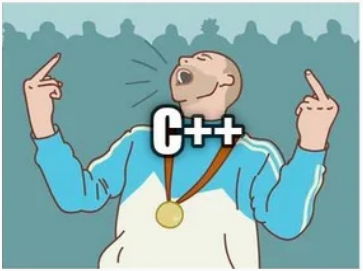
The Problem 2/2

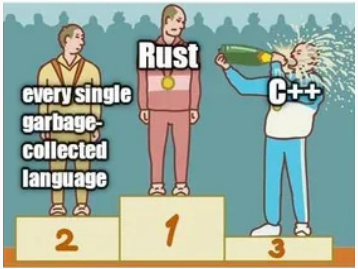
- Just from the syntax DIP1000
- DIP1021 -- Argument Ownership and Function Calls
- DIP1035 -- system@ Variables ... are dead on arrival

- 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

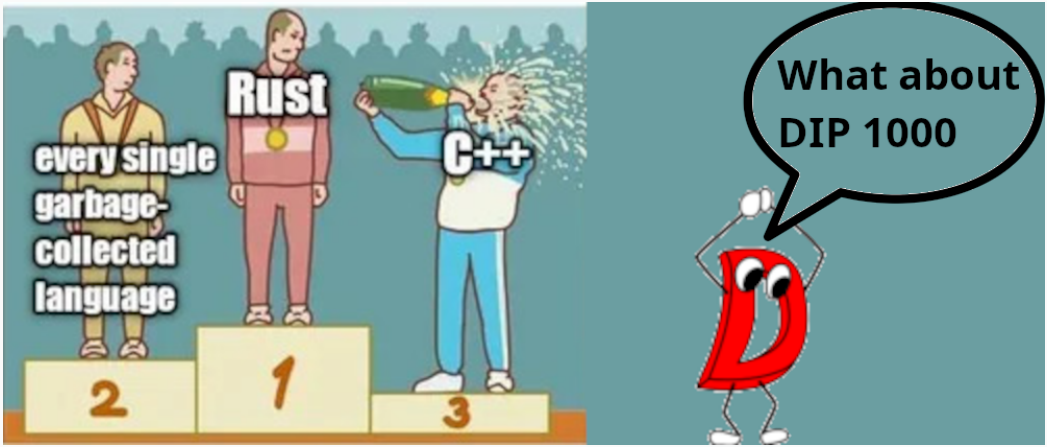
Memory safety in C++77







Memory safety in C++77



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

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...

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  }
```

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

```
@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

```
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

```
// @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, ref 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

```
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;
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 }
```

```
18 void fun(out Nullable!int into) {
19   Array a;
20   a.get(2, into);
21 }
22
23 void main() {
24   Nullable!int a;
25   fun(a);
26 }
```

returning ref

```d
import std.typecons : Nullable;

@safe:

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

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

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

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

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

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

`@property` what do you even get?

```
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) {
2      int i = h - l;
3      foreach(it; l .. h) {
4          i += (it * 1337) % 15;
5      }
6      return i;
7  }
8
9  struct A {
10     int[100] a;
11 }
```

```
13 A fun() {
14     A a;
15     foreach(i; 0 .. 100) {
16         a.a[i] = uniform(0, 100);
17     }
18     return a;
19 }
20
21 int main() {
22     A  a = fun();
23     return a.a[5] % 100 == 0;
24 }
```


much tool use, too little tool application.
served.
de 33

Continuations

assertS

```
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 $?

asserts

```
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

```
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

```
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 less terrible

```
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; FieldNameTuple!(IndexOfParameter)) {
13     static foreach(arg; args) {{
14       alias MT = typeof(__traits(getMember, IndexOfParameter, mem));
15       alias MTUP = unpack!MT;
16       static if(is(MTUP == typeof(arg))) {
17         __traits(getMember, params, mem) = arg;
18       }
19     }}
20   }
```

Nested Functions

```d
import std.array : appender;
import std.conv : to;

string toString(int[] arr) {
  auto app = appender!string();

  void toString(int a) {
    app.put(to!string(a));
  }

  foreach(idx, it; arr) {
    if(idx > 0) {
      app.put(", ");
    }
    toString(it);
  }

  return app.data;
}
```

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

Nested Imports

```
1  import std.array : appender;
2  import std.conv : to;
3
4  string toString(int[] arr) {
5    auto app = appender!string();
6
7    void toString(int a) {
8      app.put(to!string(a));
9    }
10
11   foreach(idx, it; arr) {
12     if(idx > 0) {
13       app.put(", ");
14     }
15     toString(it);
16   }
17
18   return app.data;
19 }
```

- 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
- 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
- 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

```
(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));
}
```