A Spoonful of Syntactic Sugar
aka
Understanding High Level Properties of D

by Walter Bright
Dlang.org
December 2022
https://twitter.com/WalterBright

Syntactic Sugar Is

- Constructs that enable expression of higher level ideas rather than low level operations
- Most proposed new features for D are syntactic sugar
- Meant to save programmer time, make code more readable, make code less buggy, automatically take care of boilerplate

But There's a Problem

- Most language specifications describe what the sugar does from a high level perspective
- If you're like me, I am generally unable to understand things from a high level description
- I gotta know what the underlying machinery is
- Here we're going to peek at the man behind the curtain

The Point to Understanding Sugar

- Make informed tradeoffs when using it
- Understand its limitations
- Build solid mental model
- Understand how it works with scope and ref

C Syntax

Is famous for having almost no syntactic sugar. Its machinery is like that of a steam locomotive, being mostly on the outside where you can see it work.

(train nerd alert: which of course is why I love steam locomotives, they are so fascinating!)

C Has

- Values
- Pointers
- Functions
- Structs

And that's about it

ref

```
struct S { int a; }

int get(ref S s) {
      return s.a;
}
```

```
int get(S* s) {
    return (*s).a;
}
```

But Are They Really The Same?

Let's test it with some compiler spelunking features.

Printing the AST Sent To The Backend With --b

```
dmd -c test.d --b
```

```
............codegen..._D4test3getFKSQm1SZi()............
 1: BCretexp
        * (& * s(0) +  0LL );

............codegen..._D4test3getFPSQm1SZi()............
 1: BCretexp
        * (& * s(0) +  0LL );
```

A Fuller Display With --f

```
dmd -c test.d --b --f

............codegen..._D4test3getFKSQm1SZi()............
 1: BCretexp
el:0x205c320 cnt=0 cs=0 * TYint 0x205c2c0
 el:0x205c2c0 cnt=0 cs=0 + TY* 0x205c200 0x205c260
  el:0x205c200 cnt=0 cs=0 & TY* 0x205c1a0
   el:0x205c1a0 cnt=0 cs=0 * 4 TYstruct 0x205c140
    el:0x205c140 cnt=0 cs=0 var TY*  s
  el:0x205c260 cnt=0 cs=0 const TYulong 0LL

............codegen..._D4test3getFPSQm1SZi()............
 1: BCretexp
el:0x205c200 cnt=0 cs=0 * TYint 0x205c1a0
 el:0x205c1a0 cnt=0 cs=0 + TY* 0x205c2c0 0x205c260
  el:0x205c2c0 cnt=0 cs=0 & TY* 0x205c320
   el:0x205c320 cnt=0 cs=0 * 4 TYstruct 0x205c140
    el:0x205c140 cnt=0 cs=0 var TY*  s
  el:0x205c260 cnt=0 cs=0 const TYulong 0LL
```

Going Further With The Generated Assembly With -vasm

```
dmd -c test.d -vasm

_D4test3getFKSQm1SZi:
0000:   8B 07               mov     EAX,[RDI]
0002:   C3                  ret

_D4test3getFPSQm1SZi:
0000:   8B 07               mov     EAX,[RDI]
0002:   C3                  ret
```

Dynamic Arrays

```
int[ ] a;
```

a.ptr = pointer to first element in array
a.length = number of elements in array

```
int add(int[ ] a)
{
    return a[0] + a[1];
}
```

```
int add(int* ptr, size_t length)
{
    return *p + *(p + 1);
}
```

out Parameters

```
struct S { int a; }

void set(ref S s, out int r) {
    r = s.a;
}
```

```
void set(S* s, int* r) {
    *r = 0;
    *r = (*s).a;
}
```

Struct Member Functions

```
struct S {
    int a;
    int set(int i) { a = i; }
}

void vol() {
    S s;
    s.set(3);
}
```

```
struct S {
    int a;
    int set(S* this, int i)
    { (*this).a = i; }
}

void vol() {
    S s;
    set(&s, 3);
}
```

Class Member Functions

```
class C {
    int a;
    int set(int i) { a = i; }
}

void vol() {
    C c = new C;
    c.set(i);
}
```

```
class C {
    void** __vptr;
    int a;
    int set(C* this, int i) { (*this).a = i; }
}

void vol() {
    C* this = new C;
    (*((*this).__vptr[1]))(this, 3);
}
```

The V Table

__vtbl = [&classinfo, &set]

Nested Function

```
int org(int i, int j) {
    int nested() { return ++i + j; }
    return nested();
}
```

```
int org(int i, int j) {
    int nested(int* p) { return ++p[0] + p[1]; }
    return nested(&i);
}
```

Delegate

```
int delegate() dg;
dg.ptr = context pointer
dg.funcptr = function pointer
```

Struct Delegate

```
struct S {
    int a;
    int set(int i) { a = i; }
}

void bym() {
    S s;
    int delegate(int) dgs = &s.set;
}
```

dgs.ptr set to &s
dgs.funcptr set to &set

Class Delegate

```
class C {
    int a;
    int set(int i) { a = i; }
}

void bym() {
    C c = new C;
    int delegate(int) dgc = &c.set;
}
```

dgc.ptr set to c
dgc.funcptr set to c.__vptr[1]

Nested Function Delegate

```
int org(int i, int j) {
    int nested(int z) { return ++i + j + z; }
    int delegate(int) dgn = &nested;
    return dgn();
}
```

dgn.ptr set to &i
dgn.funcptr set to &nested

dgs, dgc, and dgn are all of the same type! Once formed, they are indistinguishable from each other.

Lazy Function Parameters

```
int fid(lazy int x) { return x + 1; }

int fum(int i) { return fid(++i); }
```

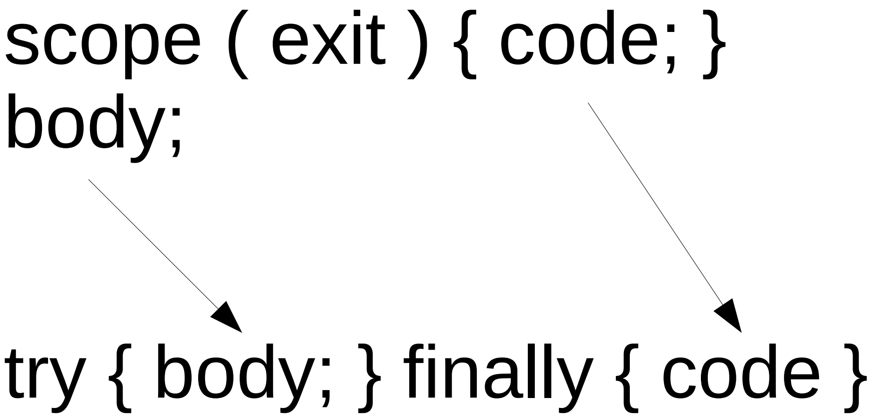
```
int fid(int delegate dg) { return dg() + 1; }

int fum(int i) { return fid(() => ++i); }
```

Scope Statement

- Scope (exit)
- Scope (failure)
- Scope (success)

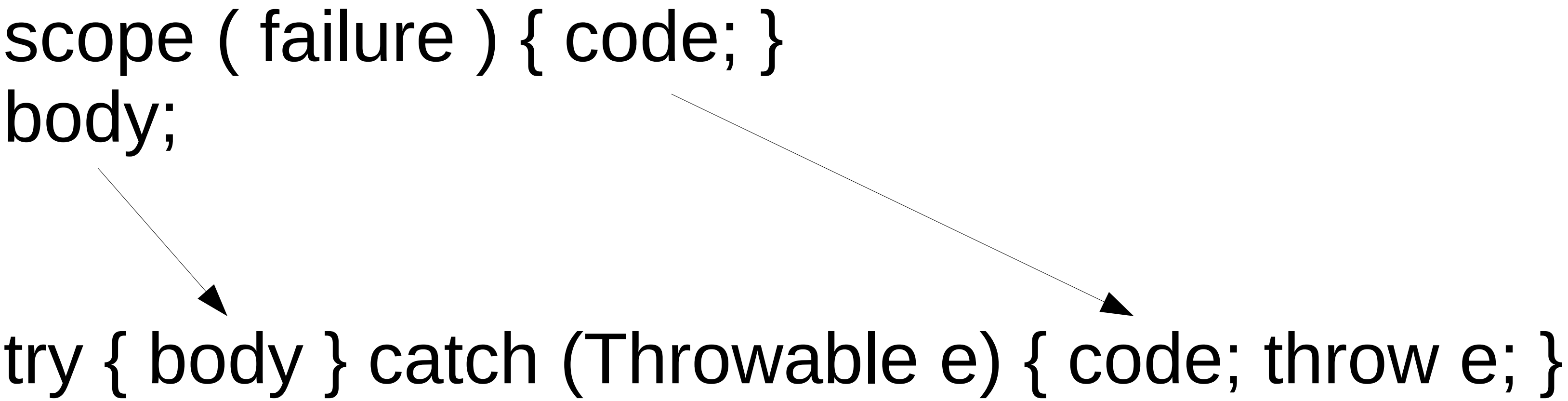
Scope Exit



```
scope ( exit ) { code; }
body;

try { body; } finally { code }
```

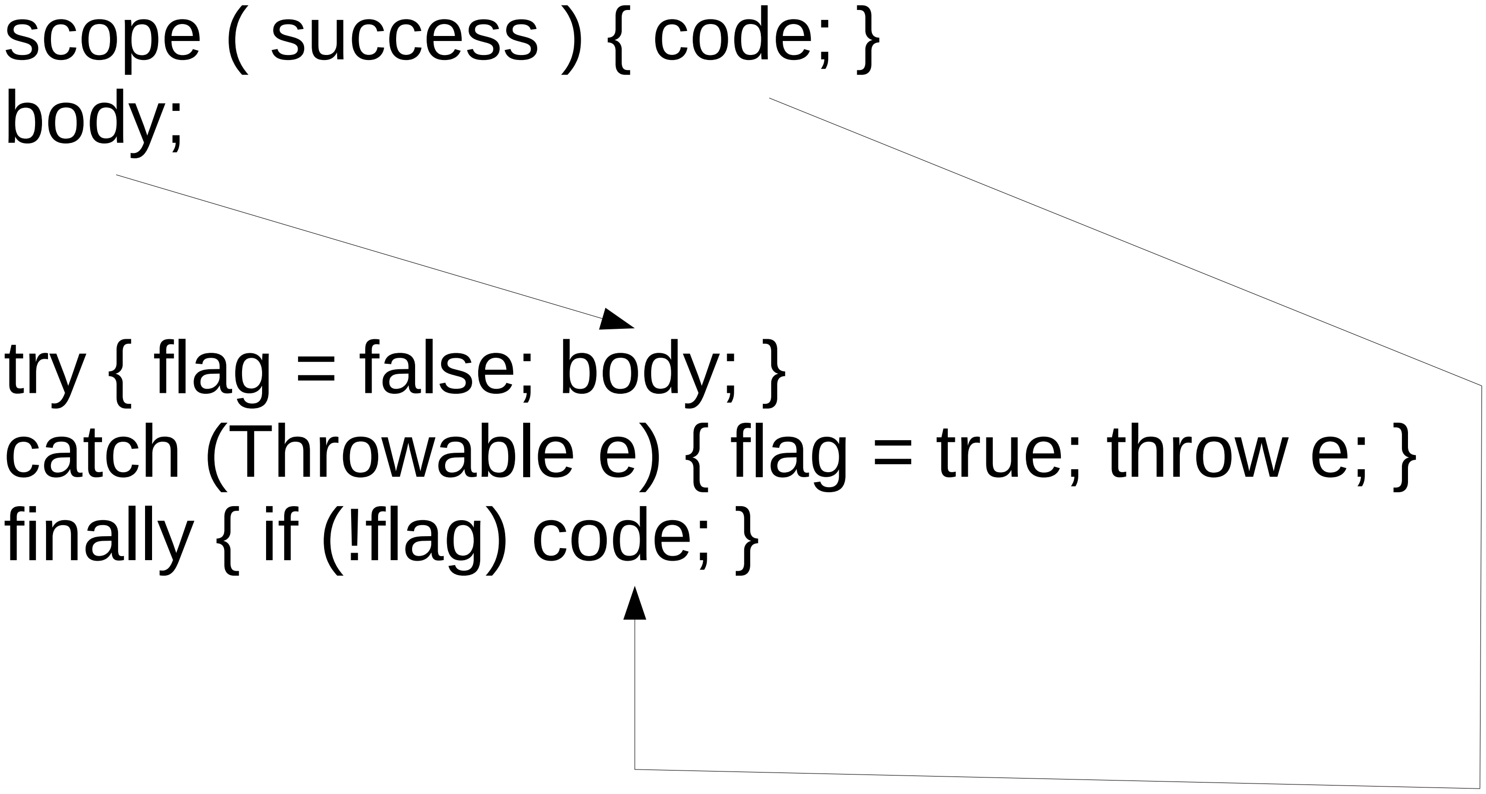
Scope Failure



```
scope ( failure ) { code; }
body;
```

```
try { body } catch (Throwable e) { code; throw e; }
```

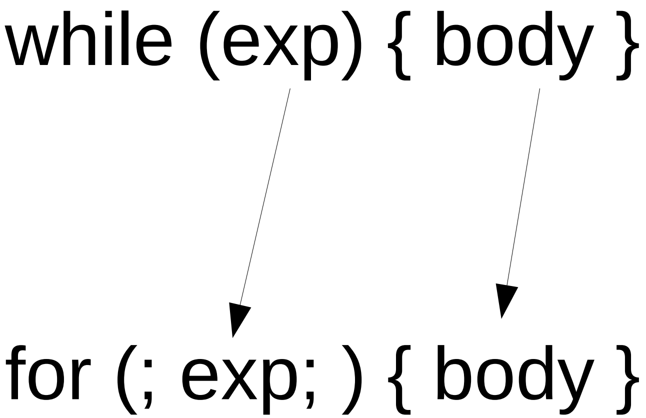
Scope Success



```
scope ( success ) { code; }
body;
```

```
try { flag = false; body; }
catch (Throwable e) { flag = true; throw e; }
finally { if (!flag) code; }
```

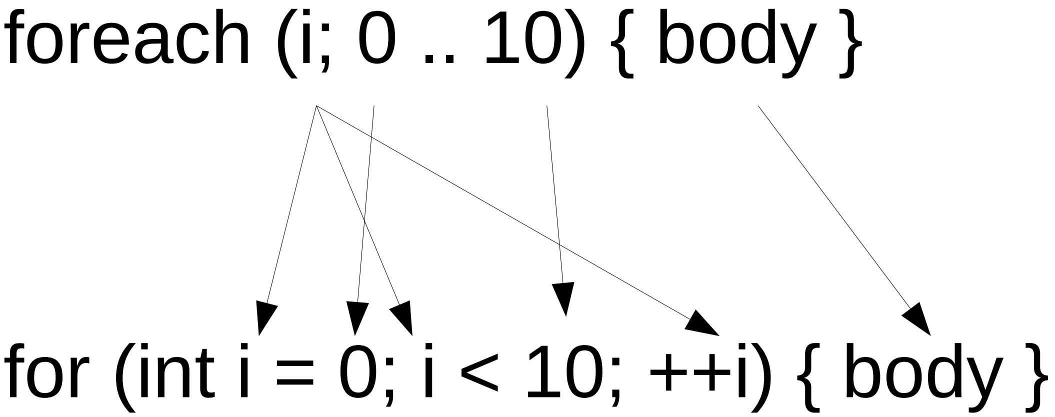
While Loop



while (exp) { body }

for (; exp;) { body }

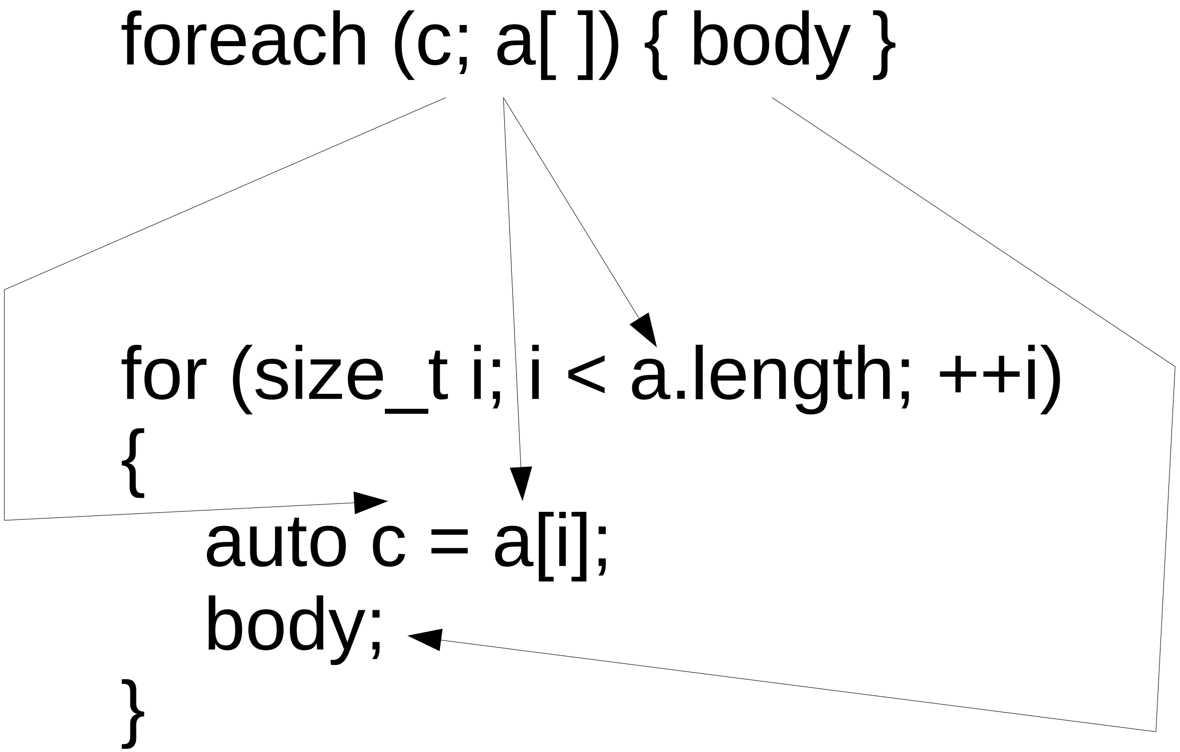
Foreach



```
foreach (i; 0 .. 10) { body }
```

```
for (int i = 0; i < 10; ++i) { body }
```

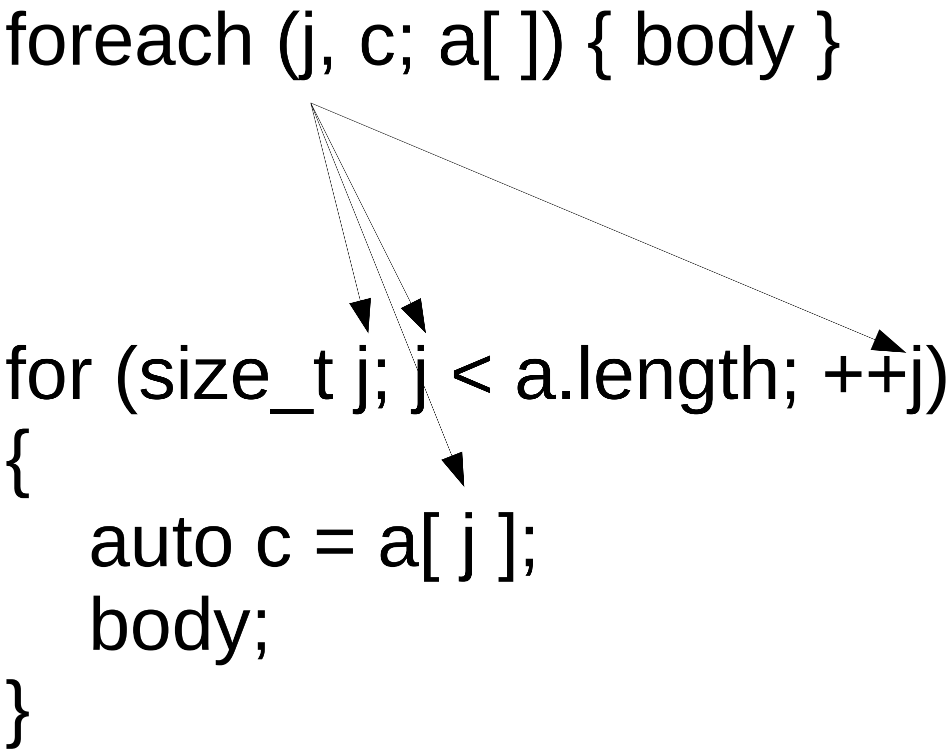
Foreach over Array



```
foreach (c; a[ ]) { body }
```

```
for (size_t i; i < a.length; ++i)
{
    auto c = a[i];
    body;
}
```

Foreach over Array with Index



```
foreach (j, c; a[ ]) { body }
```

```
for (size_t j; j < a.length; ++j)
{
    auto c = a[ j ];
    body;
}
```

Foreach with Ref to Array

```
foreach (ref c; a[]) { body }
```

```
for (size_t i; i < a.length; ++i)
{
    ref c = a[i];
    body;
}
```

Foreach over Range

```
foreach (e; range) { body }
```

```
for (auto r = range; !r.empty; r.popFront)
{
    auto e = r.front;
    body;
}
```

Syntactic Sugar Is Useful When

- It is heavily used
- Additional semantics can be used
  - Such as safety
- Less prone to errors
- Easier to refactor
- Easier to encapsulate

The Rest Belongs In Library

- Library code can be improved without needing to make expensive and disruptive changes to the compiler
- Users can create their own sugar in the library
  - For example, the earlier bitfield library implementation

The Sugar High

- Language becomes too big and unmanageable
- Less can be more
- The gold is in picking the right amount

Future Sugar

- Sum types
- Pattern matching
- Tuples

A Spoonful of Syntactic Sugar
aka
Understanding High Level Properties of D

by Walter Bright
Dlang.org
December 2022
https://twitter.com/WalterBright