

# Tuples in D

Timon Gehr

```
auto t = (1, ("2", 3.0));  
auto (x, (y, z)) = t;
```

# Why is this talk?

## The State of D 2018 Survey

**NOTE:** The survey is closed. Thanks to everyone who participated!

---

Strange things are afoot at the D Language Foundation. Odd noises and varicolored lights have been reported emanating from the cellar into the wee hours of the morning. Foundation members have been sighted, stumbling dazed and bleary-eyed in and out of the front door, arms full of mysterious black boxes. Neighbors whisper, and rumor has it that the spawn of so much secretive activity is only one arcane ritual away from seeing the light of day.

How right they are! For the past few weeks, the initiate Sebastian Wilzbach has devoted his energies to studying the Book of Modern Arcana in preparation for the ritual known as the State of D 2018 Survey. With feedback from those already steeped in the Dark arts, he has been refining the incantations of the ritual so that they prove most effective. Now, at long last, his preparations are complete and the ritual has been unleashed upon the world!

<https://dlang.org/blog/2018/02/28/the-state-of-d-2018-survey/>

# Why is this talk?

What language features do you miss?

285 out of 540 people answered this question

1	tuples	143 / 50%
2	named arguments	131 / 46%
3	string interpolation	87 / 31%
4	in-place struct initialization	81 / 28%
5	multiple alias this	80 / 28%

<https://rawgit.com/wilzbach/state-of-d/master/report.html>

# Why is this talk?

What language features do you miss?

285 out of 540 people answered this question

1	tuples	143 / 50%
2	named arguments	131 / 46%
3	string interpolation	87 / 31%
4	in-place struct initialization	81 / 28%
5	multiple alias this	80 / 28%

<https://rawgit.com/wilzbach/state-of-d/master/report.html>

So far: We have named arguments partially implemented (thanks Dennis et al!) and interpolation is about to land as well (thanks Adam et al!).

# Why do people miss tuples?<sup>1</sup>

- ▶ Allows ad-hoc packing and unpacking of related data.
- ▶ Good synergy with ranges and generic algorithms.

---

<sup>1</sup>Keep in mind: Only about 25% of respondents, 143/540, actively miss tuples.

# Bearophile's Example: Huffman Encoding (2012)

---

```
1 void main() {
2     auto s = "this is an example for huffman encoding"d;
3     foreach (t; encode(s)) {
4         auto c = t[0], e = t[1];
5         writefln("'s': %s", c, e);
6     }
7 }
```

---

```
1 ' ': 101
2 'n': 010
3 'a': 1001
4 'e': 1100
5 'f': 1101
6 'h': 0001
7 'i': 1110
8 'm': 0010
9 'o': 0011
10 's': 0111
```

---

```
1 'g': 00000
2 'l': 00001
3 'p': 01100
4 'r': 01101
5 't': 10000
6 'u': 10001
7 'x': 11110
8 'c': 111110
9 'd': 111111
```

---

## Bearophile's Example: Huffman Encoding (2012)

---

```
1 import std.typecons;
2 import std.stdio, std.algorithm, std.container, std.array;
3
4 auto encode(dstring s) {
5     auto sf = s.dup.sort.release.group;
6     auto heap = sf.map!(t => tuple(t[1], [tuple(t[0], "")]))
7         .array.heapify!q{b < a};
8
9     while (heap.length > 1) {
10        auto lo = heap.front; heap.removeFront;
11        auto hi = heap.front; heap.removeFront;
12        foreach (ref pair; lo[1]) pair[1] = '0' ~ pair[1];
13        foreach (ref pair; hi[1]) pair[1] = '1' ~ pair[1];
14        heap.insert(tuple(lo[0] + hi[0], lo[1] ~ hi[1]));
15    }
16    return heap.front[1]
17        .schwartzSort!(t => tuple(t[1].length, t[0]));
18 }
```

# Bearophile's Example: Huffman Encoding (2012)

---

```
1
2 import std.stdio, std.algorithm, std.container, std.array;
3
4 auto encode(dstring s) {
5     auto sf = s.dup.sort.release.group;
6     auto heap = sf.map!((c, f) => (f, [(c, "")]))
7         .array.heapify!q{b < a};
8
9     while (heap.length > 1) {
10         auto (lof, loa) = heap.front; heap.removeFront;
11         auto (hif, hia) = heap.front; heap.removeFront;
12         foreach (_, ref e); loa) e = '0' ~ e;
13         foreach (_, ref e); hia) e = '1' ~ e;
14         heap.insert((lof + hif, loa ~ hia));
15     }
16     auto (f, a) = heap.front;
17     return a.schwartzSort!((c, e) => (e.length, c));
18 }
```



# Bearophile's Example: Huffman Encoding (2012)

---

```
1 void main() {
2     auto s = "this is an example for huffman encoding"d;
3     foreach ((c, e); encode(s)) {
4         writefln("'s': %s", c, e);
5     }
6 }
```

---

```
1 ' ': 101
2 'n': 010
3 'a': 1001
4 'e': 1100
5 'f': 1101
6 'h': 0001
7 'i': 1110
8 'm': 0010
9 'o': 0011
10 's': 0111
```

---

```
1 'g': 00000
2 'l': 00001
3 'p': 01100
4 'r': 01101
5 't': 10000
6 'u': 10001
7 'x': 11110
8 'c': 111110
9 'd': 111111
```

---

# Drawbacks of Tuples

- ▶ Positional representation.
  - ▶ Can confuse order of elements.
- ▶ Lack of abstraction.
  - ▶ When adding new members, need to update all uses.

⇒ It is often best to reduce use of tuples in library function interfaces.

⇒ Ideally: Packing and unpacking should be close to each other.

## Robert's example (DConf'23)

---

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

---

## Similarly bad code

---

```
1 void recordGPS(double lon, double lat) {  
2     // ...  
3 }  
4  
5 void main() {  
6     double lat;  
7     double lon;  
8     recordGPS(lat, lon);  
9 }
```

---

## Similarly bad code

---

```
1 void recordGPS(double lon, double lat) {  
2     // ...  
3 }  
4  
5 void main() {  
6     double lat;  
7     double lon;  
8     recordGPS(lat, lon);  
9 }
```

---

The same design methods and solutions apply.  
I am happy that we have multiple function arguments.

# What to use tuples for?

⇒ Use tuples for plumbing.

E.g.:

- ▶ Creating ad-hoc groupings when calling into generic code.
- ▶ When type deduction is useful.
- ▶ For very localized data structures within a function or in small scripts.
- ▶ Returning multiple values that are not directly related.  
⇒ Rule of thumb: If it can be in a function argument list, it can be in a tuple.

# Type Tuple AliasSeq

---

```
1 alias AliasSeq(T...)=T; // or 'import std.meta: AliasSeq;'  
2 void main(){  
3     AliasSeq!(int, int, int) s = AliasSeq!(1, 2, 3);  
4  
5     auto a = [s]; // automatically expands  
6     assert(a == [1, 2, 3]);  
7  
8     auto b = [AliasSeq!(s, s)]; // cannot be nested  
9     assert(b == [1, 2, 3, 1, 2, 3]);  
10  
11     auto p = &s; // error: cannot take address  
12  
13     auto foo() => s; // error: cannot return sequence  
14  
15     int sum = s[0]+s[1]+s[2]; // separate variable per component  
16  
17     auto t = s; // generates another sequence of variables  
18 }
```

# TypeTuple AliasSeq

- ▶ AliasSeq can contain arbitrary aliases
- ▶ If all are types, can use like a type, to declare multiple variables
- ▶ Metaprogramming tool



## Why `AliasSeq` does not cut it

`AliasSeq!(T1,...,Tn)` is not a tuple type.

This is because it is not a type. E.g., `AliasSeq` cannot do something like this:

---

```
1 foreach((x,y);[(1,2),(2,4),(-1,2)]){
2     writeln(x," ",y);
3 }
```

---

Or this:

---

```
1 auto (x, y) = table.query!(
2     (ref row) => (row.a+row.b, row.c-row.d)
3 )(rowId);
```

---

## N.B.: Fun with `AliasSeq`

---

```
1 import std.meta:AliasSeq;
2 import std.stdio:writeln;
3
4 void main(){
5     int i = 0;
6     int[3] a = i++;
7     writeln(a);
8     int j = 0;
9     AliasSeq!(int, int, int) x = j++;
10    writeln(x);
11    int u = 1, v = 2;
12    AliasSeq!(u, v) = AliasSeq!(v, u);
13    writeln(u, " ",v);
14 }
```

---

## N.B.: Fun with AliasSeq

---

```
1 import std.meta:AliasSeq;
2 import std.stdio:writeln;
3
4 void main(){
5     int i = 0;
6     int[3] a = i++;
7     writeln(a); // [0, 0, 0]
8     int j = 0;
9     AliasSeq!(int, int, int) x = j++;
10    writeln(x); // 012
11    int u = 1, v = 2;
12    AliasSeq!(u, v) = AliasSeq!(v, u);
13    writeln(u, " ",v); // 2 2
14 }
```

---

# What is a tuple?

- ▶ In general: Product type.
  - ▶ Aggregate data type.
  - ▶ Can stick in multiple values.
  - ▶ Can get back out what we stuck in.
  - ▶ Is no more than the sum [sic] of its parts.

# What is a tuple?

- ▶ In general: Product type.
  - ▶ Aggregate data type.
  - ▶ Can stick in multiple values.
  - ▶ Can get back out what we stuck in.
  - ▶ Is no more than the sum [sic] of its parts.
- ▶ Sounds a bit like `struct`.

# What is a tuple?

- ▶ In general: Product type.
  - ▶ Aggregate data type.
  - ▶ Can stick in multiple values.
  - ▶ Can get back out what we stuck in.
  - ▶ Is no more than the sum [sic] of its parts.
- ▶ Sounds a bit like `struct`.
- ▶ `std.typecons.tuple?`

# What is a tuple?

- ▶ In general: Product type.
  - ▶ Aggregate data type.
  - ▶ Can stick in multiple values.
  - ▶ Can get back out what we stuck in.
  - ▶ Is no more than the sum [sic] of its parts.
- ▶ Sounds a bit like `struct`.
- ▶ `std.typecons.tuple?`

Simplified:

---

```
1 struct Tuple(T...){
2   T expand;
3   alias T this;
4 }
5 auto tuple(T...)(T args) => Tuple!T(args);
```

---

`struct Tuple(T...)` is a good place to start

If tuples are a special kind of `struct`, then:

- ▶ Tuple features become more broadly useful.
- ▶ Existing generic code can already handle tuples.
- ▶ Calling convention is already designed.



# Tuples: What is missing?

Sticking things in.

---

```
1 auto foo(inout(int)* w){  
2   auto t = tuple(w, w); // error  
3   return t;  
4 }
```

---

# Tuples: What is missing?

Sticking things in.

---

```
1 auto foo(inout(int)* w){  
2   auto t = tuple(w, w); // error  
3   return t;  
4 }
```

---

Bad state of affairs, but probably not in scope of a tuple design to fix.

# Tuples: What is missing?

Sticking things in.

---

```
1 auto foo(inout(int)* w){
2   auto t = tuple(w, w); // error
3   return t;
4 }
```

---

Bad state of affairs, but probably not in scope of a tuple design to fix.  
Long-term, consider adding some sort of parametric polymorphism.

# Tuples: What is missing?

Getting things back out.

---

```
1 int* foo(return scope int* a){  
2   scope int* b = ...;  
3   auto t = tuple(a, b); // error  
4   return t[0];  
5 }
```

---

# Tuples: What is missing?

Getting things back out.

---

```
1 int* foo(return scope int* a){
2   scope int* b = ...;
3   auto t = tuple(a, b); // error
4   return t[0];
5 }
```

---

DIP1000 should probably be improved.

In general, interaction with tuples and function calls is a good benchmark for type system features.

## N.B.: AliasSeq

---

```
1 auto foo(inout(int)* w){
2   auto t = AliasSeq!(w, w); // ok
3   return t; // error
4 }
```

---

```
1 int* foo(return scope int* a){
2   scope int* b = ...;
3   auto t = AliasSeq!(a, b); // ok
4   return t[0]; // ok
5 }
```

---

# Tuples: What is missing?

Unpacking.

---

```
1 import std.typecons: tuple;  
2 auto t = tuple(1, tuple(2, 3));  
3 auto (x, (y, z)) = t;
```

---

# Tuples: What is missing?

Unpacking.

---

```
1 import std.typecons: tuple;
2 auto t = tuple(1, tuple(2, 3));
3 auto (x, (y, z)) = t;
```

---

Now, have to do:

---

```
1 import std.typecons: tuple;
2 auto t = tuple(1, tuple(2, 3));
3 auto x = t[0], y = t[1][0], z=t[1][0];
```

---

- ▶ Annoying. Manual labor that language should handle.
- ▶ People often opt to not declare the new variables.
- ▶ Makes tuple code unnecessarily hard to read.



# Tuples: What is missing?

Unpacking.

---

```
1 import std.typecons: tuple;
2 auto t = tuple(1, tuple(2, 3));
3 auto (x, (y, z)) = t;
```

---

Now, have to do:

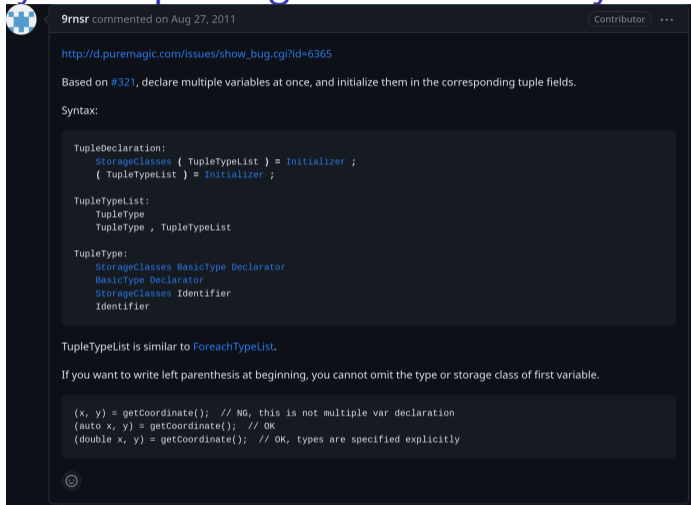
---

```
1 import std.typecons: tuple;
2 auto t = tuple(1, tuple(2, 3));
3 auto x = t[0], y = t[1][0], z=t[1][1];
```

---

- ▶ Annoying. Manual labor that language should handle.
- ▶ People often opt to not declare the new variables.
- ▶ Makes tuple code unnecessarily hard to read.

# Why no unpacking? A bit of History



9rnsr commented on Aug 27, 2011

[http://d.puremagic.com/issues/show\\_bug.cgi?id=6365](http://d.puremagic.com/issues/show_bug.cgi?id=6365)

Based on #321, declare multiple variables at once, and initialize them in the corresponding tuple fields.

Syntax:

```
TupleDeclaration:
  StorageClasses ( TupleTypeList ) = Initializer ;
  ( TupleTypeList ) = Initializer ;

TupleTypeList:
  TupleType
  TupleType , TupleTypeList

TupleType:
  StorageClasses BasicType Declarator
  BasicType Declarator
  StorageClasses Identifier
  Identifier
```

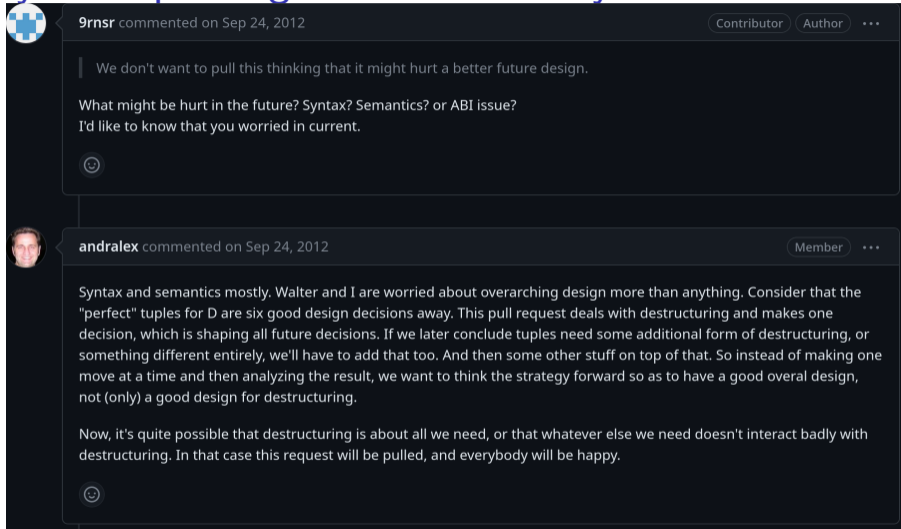
TupleTypeList is similar to [ForeachTypeList](#).


If you want to write left parenthesis at beginning, you cannot omit the type or storage class of first variable.

```
(x, y) = getCoordinate(); // NG, this is not multiple var declaration
(auto x, y) = getCoordinate(); // OK
(double x, y) = getCoordinate(); // OK, types are specified explicitly
```

<https://github.com/dlang/dmd/pull/341>


# Why no unpacking? A bit of History


A screenshot of two GitHub comments on a pull request. The first comment is from user '9rnsr' and the second is from 'andrax'. Both comments discuss concerns about future design decisions related to destructuring in D programming language.

 **9rnsr** commented on Sep 24, 2012 Contributor Author ...

We don't want to pull this thinking that it might hurt a better future design.


What might be hurt in the future? Syntax? Semantics? or ABI issue?  
I'd like to know that you worried in current.



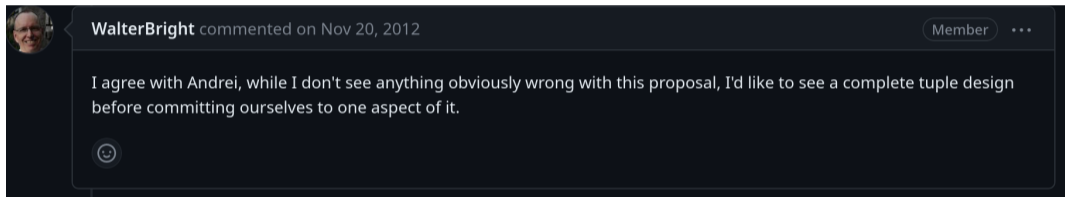
 **andrax** commented on Sep 24, 2012 Member ...

Syntax and semantics mostly. Walter and I are worried about overarching design more than anything. Consider that the "perfect" tuples for D are six good design decisions away. This pull request deals with destructuring and makes one decision, which is shaping all future decisions. If we later conclude tuples need some additional form of destructuring, or something different entirely, we'll have to add that too. And then some other stuff on top of that. So instead of making one move at a time and then analyzing the result, we want to think the strategy forward so as to have a good overall design, not (only) a good design for destructuring.

Now, it's quite possible that destructuring is about all we need, or that whatever else we need doesn't interact badly with destructuring. In that case this request will be pulled, and everybody will be happy.



# Why no unpacking? A bit of History



<https://github.com/dlang/dmd/pull/341>

>11 years later we still do not have  
unpacking.

It would be good to have unpacking.

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>



## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.digitalmars-d@puremagic.com>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)
- ▶ tuple<1, 2, 3>

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)
- ▶ tuple<1, 2, 3>
- ▶ (:1, 2, 3)

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)
- ▶ tuple<1, 2, 3>
- ▶ (:1, 2, 3)
- ▶ (|1, 2, 3)

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)
- ▶ tuple<1, 2, 3>
- ▶ (:1, 2, 3)
- ▶ (|1, 2, 3)
- ▶ ... (too lazy to show Unicode suggestions)

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>



## Some more context.

In 2012:

- ▶ Worry: tuple literal syntax.<sup>23</sup> (1, 2, 3) clashes with comma operator and (e) is a parenthesized expression.

Suggestions included:

- ▶ {1, 2, 3}
- ▶ <1, 2, 3>
- ▶ [1, 2, 3]
- ▶ |1, 2, 3|
- ▶ (|1, 2, 3|)
- ▶ tuple<1, 2, 3>
- ▶ (:1, 2, 3)
- ▶ (|1, 2, 3)
- ▶ ... (too lazy to show Unicode suggestions)
- ▶ This is an obstacle according to the “full tuple design” doctrine.

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

# Some more context.

## Change Log: 2.079.0

previous version: [2.078.3](#) – next version: [2.079.1](#)

### Download D 2.079.0

released Mar 01, 2018

#### Compiler changes

1. [Argument mismatch errors have been improved](#)
2. [The deprecation period of using the result of comma expression has ended](#)
3. [Function parameters with default values are now allowed after variadic template parameters](#)
4. [The `delete` keyword has been deprecated.](#)
5. [The deprecation period of the `-dylib` flag on OSX has ended. Use `-shared`](#)

---

<sup>2</sup>[https://forum.dlang.org/thread/k3ns2a\\$1ndc\\$1@digitalmars.com](https://forum.dlang.org/thread/k3ns2a$1ndc$1@digitalmars.com)

<sup>3</sup><https://forum.dlang.org/thread/mailman.372.1364547485.4724.>

# A voice of reason?

Re: DIP19: Remove comma operator from D and provision better syntactic support for tuples

September 23, 2012

**bearophile**



Posted in reply to  
[Andrei Alexandrescu](#)

This is a complex topic, and in this post I am not able to discuss everything that needs to be discussed. So I will discuss only part of the story.

First: tuples are important enough. I think they should be built-in in a modern language, but maybe having them as half-built-in will be enough in D. Currently in D we have (deprecated) built-in complex numbers that I use only once in a while, and half-usable library defined tuples that I use all the time.

Second: removing comma operator from D has some advantages unrelated to tuple syntax. Even disallowing bad looking C-like code that uses commas is an improvement by itself (but maybe it's not a big enough improvement...).

Third: replacing the packing syntax `tuple(x,y)` with `(x,y)` is nice and maybe even expected in a partially functional language as D, but that's not going to improve D usability a lot. What I am asking for is different: I'd like D tuples to support handy unpacking syntax:

- 1) In function signatures;
- 2) In foreach;
- 3) At assignment points;
- 4) In switch cases.

Note: in the examples below I have used the `tuple(x,y)` syntax for simplicity, feel free to replace it with a shorter `(x,y)` syntax if you want.

-----

<https://forum.dlang.org/post/baebmmuynrgtakzfmxso@forum.dlang.org>

## A closer look at Kenji Hara's proposal

---

```
1 auto (x, y) = tuple(1, "123");  
2 (int x, y) = tuple(1, 123);
```

---

## A closer look at Kenji Hara's proposal

---

```
1 auto (x, y) = tuple(1, "123");  
2 (int x, y) = tuple(1, 123);
```

---

- ▶ Looks a bit like it should mean: declare new `x`, assign to existing `y`.

## A closer look at Kenji Hara's proposal

---

```
1 auto (x, y) = tuple(1, "123");  
2 (int x, y) = tuple(1, 123);
```

---

▶ Looks a bit like it should mean: declare new `x`, assign to existing `y`.

Otherwise: IMNSHO: Slightly generalize it and ship it.

---

```
1 auto (x, (y, z)) = tuple(1, tuple("2", 3.0));
```

---

Re: DIP19: Remove comma operator from D and provision better syntactic support for tuples

September 23, 2012

**Andrei  
Alexandrescu**



Posted in reply to  
[Timon Gehr](#)

On 9/23/12 6:42 PM, Timon Gehr wrote:

That is because it does not base the discussion on the right limitations of built-in tuples:

```
auto (a,b) = (1,"3");  
(auto a, string b) = (1, "3");
```

I meant to mention that but forgot. The interesting thing about this is that, if we decide it's the main issue with today's tuples, we pull Kenji's patch and close the case.

<https://forum.dlang.org/post/baebmmuynrgtakzfmxso@forum.dlang.org>

It is the main issue with today's tuples.



## Back to Bearophile's example (2012)

---

```
1
2 import std.stdio, std.algorithm, std.container, std.array;
3
4 auto encode(dstring s) {
5     auto sf = s.dup.sort.release.group;
6     auto heap = sf.map!((c, f) => (f, [(c, "")]))
7         .array.heapify!q{b < a};
8
9     while (heap.length > 1) {
10         auto (lof, loa) = heap.front; heap.removeFront;
11         auto (hif, hia) = heap.front; heap.removeFront;
12         foreach (_, ref e); loa) e = '0' ~ e;
13         foreach (_, ref e); hia) e = '1' ~ e;
14         heap.insert((lof + hif, loa ~ hia));
15     }
16     auto (f, a) = heap.front;
17     return a.schwartzSort!((c, e) => (e.length, c));
18 }
```

## Bearophile's example with only unpacking

---

```
1 import std.typecons;
2 import std.stdio, std.algorithm, std.container, std.array;
3
4 auto encode(dstring s) {
5     auto sf = s.dup.sort.release.group;
6     auto heap = sf.map!((c, f) => tuple(f, [tuple(c, "")]))
7         .array.heapify!q{b < a};
8
9     while (heap.length > 1) {
10         auto (lof, loa) = heap.front; heap.removeFront;
11         auto (hif, hia) = heap.front; heap.removeFront;
12         foreach (_, ref e); loa) e = '0' ~ e;
13         foreach (_, ref e); hia) e = '1' ~ e;
14         heap.insert(tuple(lof + hif, loa ~ hia));
15     }
16     auto (f, a) = heap.front;
17     return a.schwartzSort!((c, e) => tuple(e.length, c));
18 }
```

## Bearophile's example now

---

```
1 import std.typecons;
2 import std.stdio, std.algorithm, std.container, std.array;
3
4 auto encode(dstring s) {
5     auto sf = s.dup.sort.release.group;
6     auto heap = sf.map!(t => tuple(t[1], [tuple(t[0], "")]))
7         .array.heapify!q{b < a};
8
9     while (heap.length > 1) {
10        auto lo = heap.front; heap.removeFront;
11        auto hi = heap.front; heap.removeFront;
12        foreach (ref pair; lo[1]) pair[1] = '0' ~ pair[1];
13        foreach (ref pair; hi[1]) pair[1] = '1' ~ pair[1];
14        heap.insert(tuple(lo[0] + hi[0], lo[1] ~ hi[1]));
15    }
16    return heap.front[1]
17        .schwartzSort!(t => tuple(t[1].length, t[0]));
18 }
```

## Slipped past Walter and Andrei

---

```
1 auto a = [tuple(1, "2"), tuple(3, "4"), tuple(4, "5")];
2 foreach(x, y; a) {
3     writeln(x, " ", y);
4 }
```

---

## Slipped past Walter and Andrei

---

```
1 auto a = [tuple(1, "2"), tuple(3, "4"), tuple(4, "5")];
2 foreach(x, y; a) {
3     writeln(x, " ", y);
4 }
```

---

```
1 0 Tuple!(int, string)(1, "2")
2 1 Tuple!(int, string)(3, "4")
3 2 Tuple!(int, string)(4, "5")
```

---

# Slipped past Walter and Andrei

---

```
1 auto a = [tuple(1, "2"), tuple(3, "4"), tuple(4, "5")];
2 foreach(x, y; a.map!(x => x)) {
3   writeln(x, " ", y);
4 }
```

---

```
1 1 2
2 3 4
3 4 5
```

---

# WIP tuple implementation

https:

[//github.com/tgehr/DIPs/blob/tuple-syntax/DIPs/DIP1xxx-tg.md](https://github.com/tgehr/DIPs/blob/tuple-syntax/DIPs/DIP1xxx-tg.md)

<https://github.com/tgehr/dmd/tree/tuple-syntax>

DIP and implementation still need updates. Unpacking should be viable soon.

## Tuples: What is the hold-up?

- ▶ Benefit of `static foreach`: very small design space.
- ▶ In contrast, for “full tuple design”: many moving parts. More political.
- ▶ Has to fit into existing language.
- ▶ Unfortunately, different syntax preferences exist.
  - ▶ Standard would be `(1, 2, 3)`.
    - ▶ Can be used. Comma operator has been removed.
    - ▶ More tricky to integrate with function parameter lists. (More later.)



## Tuples: What is the hold-up?

- ▶ Benefit of `static foreach`: very small design space.
- ▶ In contrast, for “full tuple design”: many moving parts. More political.
- ▶ Has to fit into existing language.
- ▶ Unfortunately, different syntax preferences exist.
  - ▶ Standard would be `(1, 2, 3)`.
    - ▶ Can be used. Comma operator has been removed.
    - ▶ More tricky to integrate with function parameter lists. (More later.)
  - ▶ DIP32 proposed `{1, 2, 3}`, to avoid comma operator clash.
    - ▶ Clashes with delegates instead.
    - ▶ `{a, b} => a+b` syntax (matches single tuple)

## Tuples: What is the hold-up?

- ▶ Benefit of `static foreach`: very small design space.
- ▶ In contrast, for “full tuple design”: many moving parts. More political.
- ▶ Has to fit into existing language.
- ▶ Unfortunately, different syntax preferences exist.
  - ▶ Standard would be `(1, 2, 3)`.
    - ▶ Can be used. Comma operator has been removed.
    - ▶ More tricky to integrate with function parameter lists. (More later.)
  - ▶ DIP32 proposed `{1, 2, 3}`, to avoid comma operator clash.
    - ▶ Clashes with delegates instead.
    - ▶ `{a, b} => a+b` syntax (matches single tuple)
  - ▶ Some people want to reuse `[1, 2, 3]`.
    - ▶ “Unify tuples and arrays” camp.

## Tuples: What is the hold-up?

- ▶ Benefit of `static foreach`: very small design space.
- ▶ In contrast, for “full tuple design”: many moving parts. More political.
- ▶ Has to fit into existing language.
- ▶ Unfortunately, different syntax preferences exist.
  - ▶ Standard would be `(1, 2, 3)`.
    - ▶ Can be used. Comma operator has been removed.
    - ▶ More tricky to integrate with function parameter lists. (More later.)
  - ▶ DIP32 proposed `{1, 2, 3}`, to avoid comma operator clash.
    - ▶ Clashes with delegates instead.
    - ▶ `{a, b} => a+b` syntax (matches single tuple)
  - ▶ Some people want to reuse `[1, 2, 3]`.
    - ▶ “Unify tuples and arrays” camp.
- ▶ Deciding on a tuple literal syntax is a blocker for unpacking even if we never add tuple literals.

## N.B. Unify tuples and arrays?

Summary: There are people that want `(int, int)` to be the same as `int[2]`.

- ▶ Does not really work in D. Tuples should slice by value.
  - ▶ Static arrays are already built to interoperate with dynamic arrays.
- 

```
1 int[2] a = [1, 2];
2 int[] b = a[];
3 ---
4 (int, int) a = [1, 2];
5 auto b = a[]; // ???
6 ---
7 (int, double) a = [1, 2.0];
8 auto b = a[]; // ???
9 ---
10 auto a = [1, 2]; // ???
11 auto b = [1, 2.0]; // ???
```

---

- ▶ Dynamic arrays are reference types. Array literals are dynamic arrays.
- ▶ Heterogeneous dynamic arrays don't seem to make all that much sense.

## The case for `(1, 2, 3)`

- ▶ It is the most widespread tuple syntax.
- ▶ It no longer clashes with the comma operator.
- ▶ Even if it were possible to unify tuples and arrays: good to have different syntax for homogeneous and possibly heterogeneous lists.
- ▶ Matches function argument list notation.  
(E.g., `tuple(1, 2, 3)` is similar to `(1, 2, 3)`).
- ▶ By committing to `(1, 2, 3)`, we can pull most basic unpacking.

## Challenges with (1, 2, 3): Single-element tuple

► Single-element array is [e], but (e) is a parenthesized expression.

⇒ Single-element tuple written as (e,).

---

```
1 auto () = ();
2 auto (a,) = (1,);
3 auto (a, b) = (1, 2);
4 auto (a, b, c) = (1, 2, 3);
```

---

```
1 auto () = tuple();
2 auto (a,) = tuple(1,); // or tuple(1)
3 auto (a, b) = tuple(1, 2);
4 auto (a, b, c) = tuple(1, 2, 3);
```

---

## Challenges with (1, 2, 3): Unpacking in argument lists

---

```
1 auto r = [(1, 2), (3, 4)].map!((a, b) => a + b);
2 assert(equal(r, [3, 7]));
```

---

```
1 void foo(int a, int b){}
2
3     foo(1, 2); // ok
4 auto t = (1, 2);
5 foo(t); // error
```

---

Issue: By default this does not work. Cannot call (a, b) with e.g. (1, 2):

## Unpacking in argument lists: Manual approach

---

```
1 auto r = [(1, 2), (3, 4)].map!(( (a, b)) => a + b );
2 assert(equal(r, [3, 7]));
```

---

```
1 void foo((int a, int b)){
2
3     foo(1, 2); // error
4     foo((1, 2)); // ok
5 auto t = (1, 2);
6 foo(t); // ok
```

---

Issue: What is the type of `foo`? Hard to make work with Phobos tuple.  
Other issue: `(( .. ))` is ugly and unsatisfactory.



## Unpacking in argument lists: Limited expansion

---

```
1 auto r = [(1, 2), (3, 4)].map!((a, b) => a + b);
2 assert(equal(r, [3, 7]));
```

---

```
1 void foo(int a, int b){}
2
3     foo(1, 2); // ok
4     foo((1, 2)); // ok
5 auto t = (1, 2);
6     foo(t); // ok
```

---

Issue: How to call `void foo(typeof((1, 2)) t) as foo((1, 2))?`

## Original design: `alias this`

---

```
1 struct Tuple(T...){
2     T expand;
3     alias expand this;
4 }
5 auto tuple(T...)(T args) => Tuple!T(args);
6
7 int foo(int a, int b) => a + b;
8 int bar(Tuple!(int, int) t) => t[0] + t[1];
9
10 foo(1, 2); // ok
11 foo(tuple(1, 2)); // ok
12
13
14 bar(1, 2); // error
15 bar(tuple(1, 2)); // ok
16 // can expand tuple via 'alias this' if needed
```

---

Issue: Walter does not like `alias this`.

## N.B.: alias this Design breaks Phobos due to Language Wart

---

```
1 enum bool hasLvalueElements(R) = isInputRange!R
2   && is(typeof(isLvalue(lvalueOf!R.front)))
3   && (!isBidirectionalRange!R
4     || is(typeof(isLvalue(lvalueOf!R.back))))
5   && (!isRandomAccessRange!R
6     || is(typeof(isLvalue(lvalueOf!R[0]))));
7
8 private void isLvalue(T)(T) if(0);
9 private void isLvalue(T)(ref T) if(1);
10 // ---
11 auto r = iota(3).map!tuple; // [(0,), (1,), (2,)]
12 static assert(!hasLvalueElements!(typeof(r)));
13
14 static assert(!is(typeof(isLvalue(r.front))));
15 static assert(is(typeof(isLvalue(r.front.expand))));
```

---

## N.B.: `alias this` Design breaks Phobos due to Language Wart

---

```
1 enum bool hasLvalueElements(R) = isInputRange!R
2   && is(typeof(isLvalue(lvalueOf!R.front)))
3   && (!isBidirectionalRange!R
4     || is(typeof(isLvalue(lvalueOf!R.back))))
5   && (!isRandomAccessRange!R
6     || is(typeof(isLvalue(lvalueOf!R[0]))));
7
8 private void isLvalue(T)(T) if(0);
9 private void isLvalue(T)(ref T) if(1);
10 // ---
11 auto r = iota(3).map!tuple; // [(0,), (1,), (2,)]
12 static assert(!hasLvalueElements!(typeof(r)));
13
14 static assert(!is(typeof(isLvalue(r.front))));
15 static assert(is(typeof(isLvalue(r.front.expand))));
```

---

Summary: Walter is probably right.

## New design: opArgs

---

```
1 struct Tuple(T...){
2     T expand;
3     alias opArgs=expand;
4 }
5 auto tuple(T...)(T args) => Tuple!T(args);
6
7 int foo(int a, int b) => a + b;
8 int fun(Tuple!(int, int) t) => t.expand[0] + t.expand[1];
9
10 foo(1, 2); // ok
11 foo(tuple(1, 2)); // ok
12
13
14 fun(1, 2); // ok
15 fun(tuple(1, 2)); // ok
16 // foo and fun are the same
```

---

Issue: Need a way to disable expansion.

## New design: opArgs (cont.)

---

```
1 struct Tuple(T...){
2     T expand;
3     alias opArgs=expand;
4 }
5 auto tuple(T...)(T args) => Tuple!T(args);
6
7 int foo(int a, int b) => a + b;
8 int bar(Tuple!(int, int) t,) => t.expand[0] + t.expand[1];
9
10 foo(1, 2); // ok
11 foo(tuple(1, 2)); // ok
12 foo(tuple(1, 2),); // error
13
14 bar(1, 2); // error
15 bar(tuple(1, 2)); // error
16 bar(tuple(1, 2),); // ok
```

---

Issue: Generic code may still be caught off-guard by the expansion. (Minor?)

## opArgs with tuple literal/type syntax

---

```
1 int foo(int a, int b) => a + b;
2 int fun((int a, int b)) => a + b; // _same_ signature as foo
3 int bar((int a, int b),) => a + b; // no expansion
4
5 foo(1, 2);      fun(1, 2);      // ok
6 foo((1, 2));   fun((1, 2));     // ok
7 foo(((1, 2))); fun(((1, 2)));  // ok
8 foo((1, 2),); fun((1, 2),);    // error
9
10 bar(1, 2);     // error
11 bar((1, 2));  // error
12 bar((1, 2),); // ok
13 bar(((1, 2),)); // ok
14 bar((((1, 2),))); // ok
```

---

Elegant unification of multiple arguments and a single tuple argument.

## Issue: IFTI

---

```
1 void foo(T...)(T args) => (args,);
2 void bar(T)(T arg) => arg;
3
4 void main(){
5     foo((1, 2)); // will be foo!(int, int)
6     foo(1, 2);  // this is foo!(int, int) already
7
8     bar((1, 2)); // ok, bar!((int, int))
9     bar(1, 2);  // now bar!((int, int)) works
10    // Issue: Potential change in behavior for overloading.
11 }
```

---

- ▶ I do not have a full design for this at the moment.
- ▶ I am not sure how severe the breakage is.
- ▶ ⇒ Editions?



## Issue: Unintuitive “expansion”

---

```
1 void main(){
2     auto t = (1, 2);
3     writeln(t); // 12
4     writeln(t,); // (1, 2)
5 }
```

---

# IFTI example: Identity function

---

```
1 T id(T)(T arg) => arg;
2
3 void main(){
4     assert(id(1) == 1);
5
6     assert(id() == ());
7     assert(id(1,) == (1,));
8     assert(id(1, 2) == (1, 2));
9     assert(id(1, 2, 3) == (1, 2, 3));
10 }
```

---

## IFTI example: Identity function without `opArgs`

---

```
1 T id(T)(T arg) => arg;
2
3 void main(){
4     assert(id((1)) == (1)); // or just id(1) == 1
5
6     assert(id(()) == ());
7     assert(id((1,)) == (1,));
8     assert(id((1, 2)) == (1, 2));
9     assert(id((1, 2, 3)) == (1, 2, 3));
10 }
```

---

## N.B.: Trailing comma breakage

---

```
1 T id(T)(T arg,) => arg;
2
3 void main(){
4     assert(id(1,) == 1);
5     assert(id(1) == 1); // works now, error with proposal
6 }
```

---

In general: Trailing `,` are already valid and completely ignored, so the usage for the one-element tuple case may change the behavior of existing code with the `opArgs` proposal. (Though I do not expect a lot of code like this out in the wild.)

Recall there is also the ugly `((a, b)) => a + b` alternative to unpack in parameter lists. Also, DIP32 suggests to use `{}` for tuples and to unpack in lambdas like `{a, b} => a + b`.

## Alternative design

More conservative design, but closer to `alias this`: `opArgs` only influences call-site and is used iff a single tuple argument is passed to a function that expects more than one argument.

## Another case for tuple literal syntax

---

```
1 void main(){
2     import std.stdio, std.typecons, std.algorithm;
3     auto t = tuple(
4         [1,2,3].map!(x=>2*x),
5         [1,2,3].map!(x=>3*x),
6     );
7     writeln([t, t]);
8 }
```

---

What I want it to print:

---

```
1 [[2, 4, 6], [3, 6, 9]], ([2, 4, 6], [3, 6, 9])]
```

---

## Another case for tuple literal syntax

---

```
1 void main(){
2     import std.stdio, std.typecons, std.algorithm;
3     auto t = tuple(
4         [1,2,3].map!(x=>2*x),
5         [1,2,3].map!(x=>3*x),
6     );
7     writeln([t, t]);
8 }
```

---

What I want it to print:

---

```
1 [[2, 4, 6], [3, 6, 9]], ([2, 4, 6], [3, 6, 9])]
```

---

What you might expect it to print:

---

```
1 [Tuple!(MapResult!(__lambda1, int []), MapResult!(__lambda2, int
  []))([2, 4, 6], [3, 6, 9]), Tuple!(MapResult!(__lambda1, int
  []), MapResult!(__lambda2, int []))([2, 4, 6], [3, 6, 9])]
```

---

## Another case for tuple literal syntax (or at least fix Phobos)

---

```
1 void main(){
2     import std.stdio, std.typecons, std.algorithm;
3     auto t = tuple(
4         [1,2,3].map!(x=>2*x),
5         [1,2,3].map!(x=>3*x),
6     );
7     writeln([t, t]);
8 }
```

---

What I want it to print:

---

```
1 [[2, 4, 6], [3, 6, 9]], ([2, 4, 6], [3, 6, 9])]
```

---

What it actually prints:

---

```
1 [Tuple!(MapResult!(__lambda1, int []), MapResult!(__lambda2, int
  []))(const(MapResult!(__lambda1, int []))([1, 2, 3]), const(
  MapResult!(__lambda2, int []))([1, 2, 3])), Tuple!(MapResult
  !(__lambda1, int []), MapResult!(__lambda2, int []))(const(
  MapResult!(__lambda1, int []))([1, 2, 3]), const(MapResult!(
```



## Second try.

What I want it to print:

---

```
1 [[([2, 4, 6], [3, 6, 9]), ([2, 4, 6], [3, 6, 9])]
```

---

What it actually prints:

---

```
1 [Tuple!(MapResult!(__lambda1, int []), MapResult!(__lambda2, int
  []))(const(MapResult!(__lambda1, int []))([1, 2, 3]), const(
  MapResult!(__lambda2, int []))([1, 2, 3])), Tuple!(MapResult
  !(__lambda1, int []), MapResult!(__lambda2, int []))(const(
  MapResult!(__lambda1, int []))([1, 2, 3]), const(MapResult!(
  __lambda2, int []))([1, 2, 3]))]
```

---

## Second try.

What I want it to print:

---

```
1 [[([2, 4, 6], [3, 6, 9]), ([2, 4, 6], [3, 6, 9])]
```

---

What it actually prints:

---

```
1 [Tuple!(MapResult!(__lambda1, int []), MapResult!(__lambda2, int
  []))(const(MapResult!(__lambda1, int []))([1, 2, 3]), const(
  MapResult!(__lambda2, int []))([1, 2, 3])), Tuple!(MapResult
  !(__lambda1, int []), MapResult!(__lambda2, int []))(const(
  MapResult!(__lambda1, int []))([1, 2, 3]), const(MapResult!(
  __lambda2, int []))([1, 2, 3]))]
```

---

N.B.: This is only one consequence of a misguided “const correctness” push.

# Tuple type syntax

---

```
1  () t0 = ();  
2  (int,) t1 = (1,);  
3  (int, int) t2 = (1, 2);
```

---

Pro: Intuitive, convenient.

Con: Looks like tuple of types.

---

```
1  unit t0 = ();  
2  int^^1 t1 = (1,);  
3  (int*int) t2 = (1, 2);
```

---

Pro: Principled

Con: Unintuitive

---

```
1  Tuple!() t0 = ();  
2  Tuple!(int) t1 = (1);  
3  Tuple!(int, int) t2 = (1, 2);
```

---

Pro: Unambiguous, works already.

Con: Verbose, may clash with Phobos tuple.

## Issue: Empty tuple vs unit type

---

```
1 alias A = ();  
2  
3 auto t = A; // ok?  
4 A t = (); // ok?  
5  
6 void foo(alias T)(){ T x; }  
7 void bar(alias a)(){ auto t = a; }  
8  
9 foo!((()))(); // ok?  
10 bar!((()))(); // ok?
```

---

Possible solution: Keep as expression, turn into type if needed.

Issue: This is magic.

## Issue: Template type arguments

---

```
1 void foo(T)(T arg){}
2
3 foo!(int, int)(1, 2); // error, too many template args
4 foo!((int, int))(1, 2); // ok with opArgs
5 foo!((int, int))(1, 2); // ok with opArgs
6
7 foo((1, 2)); // ok, foo!((int, int))
8 foo(1, 2); // ok with opArgs, foo!((int, int))
```

---

I.e., `opArgs` may lead to “default” tuple type syntax being unintuitive, as it would not apply at type level.

# Move semantics

---

```
1 import std.stdio, std.typecons, std.algorithm;
2
3 struct Vector(T){
4     T[] payload;
5     this(this){ writeln("copying"); payload=payload.dup; }
6     ~this(){ writeln("destroying"); }
7     void push(T arg){ payload~=arg; }
8     T opIndex(size_t i){ return payload[i]; }
9 }
```

---

# Move semantics

---

```
1 Tuple!(Vector!int, Vector!int) foo(){
2     Vector!int a, b;
3     foreach(i;0..10){
4         a.push(i);
5         b.push(2*i);
6     }
7     return tuple(a, b);
8 }
9 void main(){
10     auto ab = foo();
11     auto a = ab[0], b = ab[1];
12     // will be: auto (a, b) = foo();
13     // ...
14 }
```

---

# Move semantics

What we want:

---

```
1 (Vector<int>, Vector<int>) foo(){
2     Vector<int> a, b;
3     foreach(i;0..10){
4         a.push(i);
5         b.push(2*i);
6     }
7     return (a, b);
8 }
9 void main(){
10     auto (a, b) = foo();
11     // ...
12 }
```

---

---

```
1 destroying
2 destroying
```

---



# Move semantics

What we get:

---

```
1 copying
2 copying
3 copying
4 copying
5 copying
6 copying
7 destroying
8 destroying
9 destroying
10 destroying
11 destroying
12 destroying
13 destroying
14 destroying
15 destroying
16 destroying
```

---

- ▶ Not a pure language issue.
- ▶ May be partially fixable in Phobos.
- ▶ However, unpacking use case important to keep in mind for move semantics push.
- ▶ There should be a way to unpack a value into multiple new values while avoiding both copies and destructors.
- ▶ Needs cooperation from the unpacked type to bypass its own destructor.
- ▶ Suggestion: Pull unpacking first, fix move semantics later.

## Named tuple fields / records ?

---

```
1 void foo(int a, int b){}
2
3 foo(b: 1, a: 2); // foo(2, 1)
4
5 auto t = (b: 1, a: 2);
6 foo(t); // foo(2, 1) ?
7 writeln(t.a + t.b); // 3
```

---

# State of Implementation

<https://github.com/tgehr/dmd/tree/tuple-syntax>

## Feature

Unpacking declaration

Unpacking in foreach

Unpacking in parameter list

Unpacking assignment

opArgs

Tuple expression

Tuple type

Named tuples

## State

Almost done (need to fix `immutable (int,) x = (2,);`)

Works, still need to handle `ref`

Started

Not implemented

Basic support

Forwards to `std.typecons.tuple`, no magic so far

Forwards to `std.typecons.Tuple`

Not designed yet

I might propose only unpacking at first, the remaining ideas hopefully sufficiently address the “full tuple design” doctrine.