D Templates: A Tutorial

Philippe Sigaud

2012, January 3rd

Introduction

Templates are a central feature of D, giving you powerful compile-time code generation abilities that'll make your code cleaner, more flexible and even more efficient. They are used everywhere in Phobos — D standard library — and therefore any D user should know about them. But, based on C++ templates as they are, D templates can be a bit daunting at first. The D Programming Language website's documentation is a good start, though its description of templates is spread among many different files and (as it's a language reference) its material doesn't so much *teach* you how to use templates as *show* you their syntax and semantics.

This document aims to be a kind of tutorial on D templates, to show the beginning D coder what can be achieved with them. When I was using C++, I remember *never* using templates for more than *containers-of-T* stuff, and considered Boost-level[1] metaprogramming the kind of code I could never understand, never mind produce. Well, D's sane syntax for templates and nifty features such as `static if`, `alias` or tuples cured me of that impression. I hope this document will help you, too.

What's in This Document

The first part deals with the very basics: how to declare and instantiate a template, the standard 'building blocks' you'll use in almost all your templates, along with function, struct and class templates. Throughout the text, examples will present applications of these concepts.

The second part is about more advanced topics a D template user will probably use, but not on a daily basis, like template constraints, mixin templates or operator overloading.

[1] The Boost C++ library collection makes heavy use of templates.

The third part presents other metaprogramming tools: string mixins, compile-time function evaluation, and __traits. These are seen from a template-y point of view: how they can interact with templates and what you can build with them in conjunction with templates.

The fourth part presents more developed examples of what can be done with templates, based on real needs I had at some time and that could be fulfilled with templates.

Finally, an appendix on the ubiquitous is expression and another giving resources and further reading advice complete this document.

Conventions

To make this document more easily readable, I'll use standard coding books conventions, by highlighting parts of the text. Mainly, in this doc:

- D keywords will be marked like this: `int`, `static if`, `__traits` (these will be colored or not, depending on what was used to generate the document).
- Symbols and names used in code samples and cited in the text will be written like this: `myFunc`, `flatten`.
- internal links will be like this.
- external links will be like this.
- Syntax-highlighted code samples are shown like this:

```d
/**
 * This is a doc comment.
 */
module intro;

import std.stdio;

void main()
{
    int times = 10;
    // This is a comment
    foreach(i; 0..times)
        writeln("Hello, Word!");
}
```

Numbered lines will be used only when necessary.

I will sometimes make a little explanatory detour, discussing a small piece of info too small to be in its own section but of interest to the reader nonetheless. These will be marked so:

> **Semi-Literate Programming.** Most code samples presented in this document will compile with a reasonably recent D compiler. In the `utils` directory, there is a small D script called `codesamples.d` that extracts code samples from the mardkdown files.
>
> Samples with a `module name;` declaration will be extracted, a file called `name.d` will be created and compiled (possibly with a stub `main()` if none exists). The previous code sample creates a file called `intro.d` and so on. The compilation results are put in a file called `result.txt`. Samples that depend on other samples just import them (yeah for D modularity, no need for specific mark-up to weave code together). Samples with a name ending in `_error` will *not* compile, as expected: they are there to show errors, mistakes and gotchas. Anonymous samples are not extracted: I use them to show small snippets not intended to stand by themselves, or just to show D-ish pseudocode.
>
> All in all, you can see this entire document as a gigantic D package, describing hundreds of small modules.

Finally, some sections in this doc are not finished yet. The sections I consider unfinished will contain this:

> **Unfinished.** Hey, now I've added a Thanks section. But as long as I'm adding new parts in the document, new appendices, and new section, this intro will not be finished.

I probably forgot some 'unfinished' tags, do not hesitate to tell me so.

How to Get This Document

This document is just a markdown file hosted on Github. Don't hesitate to fork it or (even better for me) to make pull requests! For those of you reading this on paper, the address is:

https://github.com/PhilippeSigaud/D-templates-tutorial

Thanks

As soon as I publicly released this document, D community members gave me help, suggestions, corrections, and code samples. This is cool to see a D network emerge and people participating in common projects. The following people helped me:

Craig Dillabaugh, Andrej Mitrovic, Justin Whear, Zachary Lund, Jacob Carlborg, Timon Gehr, Simen Kjaeras, Andrei Alexandrescu, Bjorn Lietz-Spendig.

Thanks guys!

Basics

Why Templates?

Here you are, reading a book-size document on D templates. But why should you be interested in templates? Let say you have this wonderful tree struct:

```
module basicTree1;

struct Tree {
    int value;
    Tree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

Which is used like this:

```
module usingBasicTree1;

import basicTree1;

void main()
{
    auto tree = Tree(0, [Tree(1), Tree(2, [Tree(3), Tree(4), Tree(5)])]);
    assert(!tree.isLeaf);
    assert(tree.size() == 6);
}
```

All is well and good, this is a nice basic `int`-holding n-ary tree. But what if, after some time, you need a `float`-holding one? No problem, this can be coded easily, with a bit of copy-pasting. First, we change `Tree` to `IntTree` and then create a `FloatTree` node:

```
module basicTree2;

struct IntTree {
    int value;
    IntTree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}

struct FloatTree {
    float value;
    FloatTree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

But that's a lot of code duplication: the only changes are the types of `value` and `children`, which become a `float` and `FloatTree` instead of an `int` and `IntTree`. What a waste! And what if we need another tree, for example holding functions (a tree of callbacks, say)? There must be a better way.

Let's observe the previous code. What we need here is a way to produce code by generating different tree types, injecting the type for `value` as user-defined

input. It's a bit like a function: pushing parameters and getting a result. Let's imagine some code with a placeholder, let's call it `Type`, to represent the type of `value`.

```
struct Tree {
    Type value;
    Tree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

But here `Type` is a symbol introduced without warning in the struct scope. If you try to compile that code, the compiler will rightfully complain and ask you 'where does `Type` come from?'. With functions, parameters are introduced in a parameter list(!) that will push them into the following scope, the function body. That's what we want here. We should tell the compiler that `Type` is our placeholder, to be provided latter on. Let's continue with our imaginary syntax here:

```
module genericTree;

struct Tree(Type) {
    Type value;
    Tree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

See how I introduced `Type` in a (one-element) parameter list, right after `Tree`? Ideally, this is what a struct definition abstracted on the type of `value` should look like, right? A sort of... recipe to be used to generate the `Tree` we need.

Look no further, the previous definition is standard D for a struct template! You can compile it and use it to your heart content (see the next chapters to do that).

That's the essence of templates: writing some code, seeing way to abstract it on certain items (types, symbols, numbers, the possibility are numerous) and defining such a generic recipe, for further perusal.

What is a Template?

In the next chapters, you'll see how to define function, struct and class templates. The nifty syntax demonstrated just before is a special case for these constructs. It's a simplified version of the full template declaration syntax, which we will see in the next section.

But before that, I'd like to introduce what a template *really* is, because this definition is the most fundamental of the whole document. As I said, a template is a way to define a blueprint to generate some code, be it a class definition, a function or... what? What could be the most abstract unit of code?

Let us say you have a wonderful piece of code, full of function definitions, structs and their methods, new symbols, and so on. This piece of code offer a few entry points for further parameterization: some types, some symbols could be abstracted away and put into a template parameter list.

```
// Could be parameterized on Type, as seen before
struct Tree { ... }

// Could be parameterized on the Tree type also
// And maybe on f?
Tree mapOnTree(Tree input, Tree delegate(Tree) f) { ... }

void printTree(Tree input) { ... }

// What about defining a different TreeArray for all possible Trees?
alias TreeArray = Tree[];
```

But where is the basic unit to hold this code? Well, a code block of course, or a *scope*. Ideally, we would like a way to group all the previous declaration into one unit, with the same parameter list:

```
// scope?
(Type)
```

```
{
    struct Tree { /* using Type here */}
    Tree mapOnTree(Tree input, Tree delegate(Tree) f) { /* here also */ }
    void printTree(Tree input) { /* the same */ }
    alias TreeArray = Tree[]; // You get it
}
```

Since we will need to 'call' it to produce some code (a bit like you'd call a function), this code block needs a name. And then, we just need to tell the compiler: 'here, this is a blueprint'. The D keyword for that is (you got it) `template`:

```
template MyDefs(Type)
{
    struct Tree {...}
    Tree mapOnTree(Tree input, Tree delegate(Tree) f) {...}
    void printTree(Tree input) {...}
    alias TreeArray = Tree[];
}
```

Here you are. This is what a template is, at its core: a named, parameterized, code block, ready to be instantiated just for you.

Template Declarations

Here is the syntax for a template declaration:

```
template templateName(list, of, parameters)
{
  // Some syntactically correct declarations here
  // The arguments are accessible inside the template scope.
}
```

`templateName` is your usual D identifier and the list of parameters is a comma-separated list of zero or more template parameters. These can be:

Types (`identifier`) An `identifier` alone by itself is considered a type name. The common D style is to use identifiers beginning with a capital letter (`Range`, `Rest`), as for any user-defined types. Many D templates use the C++ tradition of one-capital-letter names for types, starting from `T` (`U`, `V`, ...). Do not feel constrained by this, use what makes your templates most easy to understand.

Aliases (`alias identifier`) You declare them with `alias identifier`. They will capture not types but *symbols*: variable names, class *names*, even other template names. They will also accept many compile-time literals: strings, arrays, function literals, ... Mostly, if you need a widely-accepting template, use an `alias` parameter. Note that they will *not* accept built-in types as arguments, however, since `int` is not a valid identifier in D (it's a keyword).

Literal values (`typeName identifier`) They are all declared like this: `typeName identifier`. Literal values can be integral values (`int`, `ulong`, ...), `enum`-based, strings, `chars`, floating-point values or boolean values. Any expression that can be evaluated at compile-time is OK. For example : `int depth` or `string name`.

Template parameters tuples (`identifier...`) The syntax is `identifier...` (yes, three dots) and the tuple must be the last parameter of a template. Template parameters tuples will capture under one identifier *an entire list of template parameters* (types, names, literals, ...). These tuples will store any template argument you will throw at them. If no argument is passed, you will just get an empty, zero-length, tuple. Really, as they can deal with types as well as symbols, these tuples are a bit of a mongrel type but they are wonderfully powerful and easy to use, as you will see in section Tuples.

Of those, types and aliases are the most common, while floating point values are fairly rare: their use as arguments for compile-time calculations have been superseded by D's Compile-Time Function Evaluation, aka CTFE. You'll see different uses of these parameters in this document.

Note that pointers, arrays, objects (instantiated classes), structs or functions are not part of this list. But as I said, alias parameters allow you to capture and use array, class, function or struct *names* and then access their capacities.

> **Aliases, Symbols and Names.** There is big difference between built-in types like `int` or `double[3]` and user-defined types. A user-defined type, say a class called `MyClass`, is a type name. So, it's *both* a type (the class `MyClass`, accepted by type templates arguments) and a name, a symbol (`MyClass`, accepted by `alias` template parameters). On the other hand, `int`, being a D keyword is not a symbol nor a name. It's just a type. You cannot pass it to an alias template parameter.

The template body can contain any standard D declarations: variable, function, class, interface, other templates, alias declarations,... The only exception I can think of is declaring a `module`, as this is done at the top-level scope.

Syntax and Semantics. Code inside a `template` declaration only has to be syntactically correct D code (that is: code that looks like D code). The semantics are not checked until instantiation. That means you can code happily, writing templates upon templates and the compiler won't bat an eye if you do not exercise your templates by instantiating them.

Inside the template body, the parameters are all accessible as placeholders for the future arguments. Also, the template's own name refers to its current instantiation when the code is generated. This is mostly used in struct templates and class templates.

Here are some template declaration examples:

```
1  module declaration;
2
3  template ArrayOf(T) // T is a type
4  {
5      alias ArrayType   = T[];
6      alias ElementType = T;
7  }
8
9  template Transformer(From, To) // From and To are types, too
10 {
11     To transform(From from)
12     {
13         import std.conv;
14         return to!(To)(from);
15     }
16
17     class Modificator
18     {
19         From f;
20         To t;
21         this(From f) { /*...*/ }
22     }
23 }
24
25 template nameOf(alias a)
26 {
27     enum string name = a.stringof; // enum: manifest constant
28                                    // determined at compile-time.
29                                    // See below.
30 }
31
32 template ComplicatedOne(T, string s, alias a, bool b, int i)
```

```
33 { /* some code using T, s, a, b and i */ }
34
35 template Minimalist() {} // Zero-parameter template declaration.
36
37 template OneOrMore(FirstType, Rest...) // Rest is a tuple.
38 { /*...*/ }
39
40 template ZeroOrMore(Types...) // Types is a tuple.
41 { /*...*/ }
42
43 template Multiple(T)     { /*...*/ } // One arg version.
44 template Multiple(T,U)   { /*...*/ } // Two args,
45 template Multiple(T,U,V) { /*...*/ } // and three.
```

The full syntax for template declarations is slightly more complex, I'll introduce more of it in the next sections. You'll see for example type restrictions in section Templates Specializations, default values in section Default Values, instantiation constraints in Template Constraints, and more on tuples in section tuples.

There is a limitation that's interesting to keep in mind: templates can be declared in almost any scope, except inside a (regular) function.

> **enum.** In the previous code, see line 27? It defines a `string` called `name` as a member of `nameOf`. The `enum` placed right before means `name` is a compile-time constant. You can see it as a kind of storage class, in the line of `immutable` or `const`, one that means the value is totally defined and fixed at runtime. You'll see numerous examples of `enum` in this document.

Instantiating a Template

Syntax

To instantiate a template, use the following syntax:

```
templateName!(list, of, arguments)
```

Note the exclamation point (`!`) before the comma-separated argument list. That's what differentiate template arguments lists from standard (funtion) argument lists. If both are present (for function templates), we will use:

```
templateName!(template, argument, list)(runtime, agument, list)
```

There is a small trick to get a shorter instantiation syntax: if the argument list contains only one argument with a length of one token, you can drop the parenthesis, like this:

```
templateName!argument
```

So, these are all valid template instantiations:

```
Template!(int)
Template!int

Template!("string arg")
Template!"string arg"

map!(foo)(range); // foo is a symbol, capture by an alias.
                  // range is a runtime argument.
map!foo(range);   // ditto

// But:
Multiple!(int, double)
// And not:
// Multiple!int, double //??
```

Templates as Template Arguments

Arguments can themselves be the result of another template instantiation. If a template returns a type upon instantiation, it's perfectly OK to use it inside another template argument list. In this document you'll regularly see Matrioshka calls like this: `firstTemp!(secondTempl!(Arguments), OtherArguments)`.

Selecting among Declarations

The compiler will have a look at the declarations (if more than one template was declared with the called name) and select the one with the correct number of arguments and the correct types to instantiate. If more than one template can be instantiated, it will complain and stop there (though, have a look on template specializations and template constraints.

Effect of Instantiation

When you instantiate a template, the global effect is that a new named scope (code block) is created at the template declaration scope. The name of this new scope is the template name with its argument list: `templateName!(args)`.

Inside this block, the parameters are now 'replaced' with the corresponding arguments (storage classes get applied, variables are initialized, ...). Here's what possible instantiations of the previous templates might look like:

```
module instantiation;
import declaration;

void main()
{
    ArrayOf!(int).ArrayType myArray;

     // From is an alias for the type double
     // To for the type int
    alias transfo = Transformer!(double,int);

    struct MyStruct { /*...*/ }

    // "MyStruct" is a identifier -> captured by alias
    auto name = nameOf!(MyStruct).name;

    alias complicatedExample =
    ComplicatedOne!( int[]   // a type
                   , "Hello" // a string literal
                   , ArrayOf // a name
                   , true    // a boolean literal
                   , 1+2     // calculated to be the integral '3'.
                   );

    alias min1 = Minimalist!(); // No argument

    // FirstType is 'int'
    // Rest is 'double,string,"abc"'
    alias oneOrMore =
          OneOrMore!( int
                    , double, string, "abc"
                    );

    // Types is a 1-element tuple: (int)
    alias zero1 = ZeroOrMore!(int);
    // Types is (int,double,string)
    alias zero2 = ZeroOrMore!(int,double,string);
    // Types is the empty tuple: ()
    alias zero3 = ZeroOrMore!();

    // Selects the one-arg version
    alias mult1 = Multiple!(int);
```

```
    // The three args version.
    alias mult2 = Multiple!(int,double,string);
    // Error! No 0-arg version
    //alias mult3 = Multiple!();
}
```

Outside the scope (that is, where you put the template instantiation in your own code), the internal declarations are accessible by fully qualifying them:

```
module internaldeclarations1;
import declaration;

// ArrayType is accessible (it's int[])
// array is a completly standard dynamic array of ints.
ArrayOf!(int).ArrayType array;
ArrayOf!(int).ElementType element; // the same, element is an int.

void main()
{
    // the transform function is accessible. Instantiated like this,
    // it's a function from double to string.
    auto s = Transformer!(double,string).transform(3.14159);
    assert(is(typeof(s) == string)); // s is a string
}
```

Obviously, using templates like this, with their full name, is a pain. The nifty D `alias` declaration is your friend:

```
module internaldeclarations2;
import declaration;

alias DtoS = Transformer!(double, string);

void main()
{
    auto s = DtoS.transform(3.14159);
    auto m = new DtoS.Modificator(1.618); // DtoS.Modificator is a class
                                          // storing a double and a string.
}
```

You must keep in mind that instantiating a template means generating code. Using different arguments at different places in your code will instantiate *as many differently named scopes*. This is a major difference with *generics* in languages like Java or C#, where generic code is created only once and type erasure is used to link all this together. On the other hand, instantiating the same template, with the same arguments, will create only one piece of code.

```
module differentinstantiations;
import declaration;

alias StoD = Transformer!(string,double);
alias DtoS = Transformer!(double,string);
alias StoI = Transformer!(string,int);
// Now we can use three different functions and three different classes.
```

> **void.** Note that `void` is a valid D type and, as such, a possible template argument for a type parameter. Take care: many templates make no sense when `void` is used as a type. In the following sections and in the appendix, you'll see ways to restrict arguments to only certain types.

Template Building Blocks

Up to now, templates may not seem that interesting to you, even with a simple declaration and instantiation syntax. But wait! D introduced a few nifty tricks that both simplify and greatly expand template uses. This section will introduce you to your future best friends, the foundations on which your templates will be built.

The Eponymous Trick

If a template declares a symbol with the same name (greek: *epo-nymous*) as the enclosing template, that symbol is assumed to be referred to when the template is instantiated. This one is pretty good to clean your code:

```
module pair;

template pair(T)
{
    // template 'pair' declares only a 'pair' member
    T[] pair(T t) { return [t,t];}
}

auto array = pair!(int)(1);  // no need to do pair!(int).pair(1)
```

or:

```
module nameof1;

template nameOf(alias name)
```

```
{
    enum string nameOf = name.stringof;
}

struct Example { int i;}

void main()
{
    Example example;

    auto s1 = nameOf!(Example);
    auto s2 = nameOf!(example);

    assert(s1 == "Example");
    assert(s2 == "example");
}
```

There used to be a limitation in that the eponymous trick worked *only* if you defined one (and only one) symbol. Even if the other symbols were private, they would break the eponymous substitution. This was changed recently (Fall 2012) and now, we can do:

```
module record;

template Record(T, U, V)
{
    import std.typecons: Tuple, tuple;

    // The real work is done here
    // Use as many symbols as you need.
    alias Tuple!(T,U) Pair;
    alias Pair[V] AssocArray;
    alias AssocArray[] Record;
}
```

Note that in this case, the eponymous member is an alias, whereas it was a function or a manifest constant (an `enum`) in previous examples. As was already said, any member with the same name will do.

And then, to use `Record`:

```
module using_record;
import record;

Record!(int,string,double[]) recordslist;
/* ... */
```

Although, in this case, the eponymous member hides the other members: `Pair` and `AssocArray` cannot be accessed any more. That seems logical, since if you use the eponymous trick, it's to provide a simplified interface to your users.

Inner alias

A common use for templates is to do some type magic: deducing types, assembling them in new way, etc. Types are not first-class entities in D (there is no `type` type), but they can easily be manipulated as any other symbol, by aliasing them. So, when a template has to expose a type, it's done by aliasing it to a new name.

```
module allarrays;

template AllArraysOf(T)
{
    alias T    Element;
    alias T*   PointerTo;
    alias T[]  DynamicArray;
    alias T[1] StaticArray;
    alias T[T] AssociativeArray;
}
```

> **Exposing Template Parameters.** Though they are part of a template's name, its parameters are *not* directly accessible externally. Keep in mind that a template name is just a scope name. Once it's instantiated, all the `T`s and `U`s and such do not exist anymore. If you need them externally, expose them through a template member, as is done with `AllArraysOf.Element`. You will find other examples of this in section Struct Templates and section Class Templates.

`static if`

Syntax The `static if` construct[2] lets you decide between two code paths at compile time. It's not specific to templates (you can use it in other parts of your code), but it's incredibly useful to have your templates adapt themselves to the arguments. That way, using compile-time-calculated predicates based on the template arguments, you'll generate different code and customize the template to your need.

The syntax is:

```
static if (compileTimeExpression)
{
```

[2] It's *both* an statement and a declaration, so I'll call it a construct.

```
    /* Code created if compileTimeExpression is evaluated to true */
}
else /* optional */
{
    /* Code created if it's false */
}
```

Something really important here is a bit of compiler magic: once the code path is selected, the resulting code is instantiated in the template body, but without the curly braces. Otherwise that would create a local scope, hiding what's happening inside and would drastically limit the power of `static if`. So the curly braces are there only to group the statements together.

If there is only one statement, you can get rid of the braces entirely, something you'll see frequently in D code. For example, suppose you need a template that 'returns' `true` if the passed type is a dynamic array and `false` otherwise (this kind of predicate template is developed a bit more in section [predicates]).

```
module isdynamicarray;

template isDynamicArray(T)
{
    static if (is(T t == U[], U))
        enum isDynamicArray = true;
    else
        enum isDynamicArray = false;
}
```

As you can see, with no curly braces after `static if` and the eponymous trick (`isDynamicArray` is a symbol defined by the template and its type is automatically deduced by the compiler), results in a very clean syntax. The `is()` expression part is a way to get compile-time introspection which goes hand in hand with `static if`. There is a crash course on it at the end of this document (see [appendix-isExpression]).

Optional Code A common use of `static if` is to enable or disable code: a single `static if` without an `else` clause will generate code only when the condition is true. You can find many examples of this idiom in std.range where higher-level ranges (ranges wrapping other ranges) will activate some functionality if and only if the wrapped range can support it, like this:

```
/* We are inside a MyRange templated struct, wrapping a R. */

    R innerRange;
```

```
/* Code that exists in all instantiations of MyRange */
(...)

/* optional code */
static if (hasLength!R) // does innerRange have a .length() method?
    auto length()       // Then MyRange has one also.
    {
        return innerRange.length;
    }

static if (isInfinite!R)     // Is innerRange an infinite range?
    enum bool empty = false; // Then MyRange is also infinite.
// And so on...
```

Nested `static if`s `static if`'s can be nested: just put another `static if` after `else`. Here is a template selecting an alias:

```
module selector;
import std.traits: isIntegral, isFloatingPoint;

template selector(T, alias intFoo, alias floatFoo, alias defaultFoo)
{
    static if (isIntegral!T)
       alias intFoo selector;
    else static if (isFloatingPoint!T)
       alias floatFoo selector;
    else // default case
        alias defaultFoo selector;
}
```

If you need a sort of `static switch` construct, see section [examples-staticswitch].

Recursion with `static if`

Rank: Now, let's use `static if` for something a bit more complicated than just dispatching between code paths: recursion. What if you know you will receive n-dimensional arrays (simple arrays, arrays of arrays, arrays of arrays of arrays, ...), and want to use the fastest, super-optimized numerical function for the 1-dim array, another one for 2D arrays and yet another one for higher-level arrays? Abstracting this away, we need a template doing some introspection on types, that will return 0 for an element (anything that's not an array), 1 for a 1-dim array (`T[]`, for some `T`), 2 for a 2-dim array (`T[][]`), and so on.

Mathematicians call this the *rank* of an array, so we will use that. The definition is perfectly recursive:

```
1  module rank1;
2
3  template rank(T)
4  {
5      static if (is(T t == U[], U))        // is T an array of U, for some type U?
6          enum size_t rank = 1 + rank!(U); // then let's recurse down.
7      else
8          enum size_t rank = 0;            // Base case, ending the recursion.
9  }
```

Lines 5 and 6 are the most interesting: with some `is` magic, `U` has been deduced by the compiler and is accessible inside the `static if` branch. We use it to peel one level of `[]` off the type and recurse downward, using `U` as a new type for instantiating `rank`. Either `U` is itself an array (in which case the recursion will continue) or it will hit the base case and stop there. Since the template defines a member named like itself, the result is directly accessible: any instantiation of `rank` will be a value of type `size_t`.

Let's use it:

```
module using_rank1;
import rank1;

static assert(rank!(int)       == 0);
static assert(rank!(int[])     == 1);
static assert(rank!(int[][])   == 2);
static assert(rank!(int[][][]) == 3);

/* It will work for any type, obviously */
struct S {}

static assert(rank!(S)  == 0);
static assert(rank!(S[])== 1);
static assert(rank!(S*) == 0);
```

> **static assert.** Putting `static` before an `assert` forces the `assert` execution at compile-time. Using an `is` expression as the test clause gives assertion on types. One common use of `static assert` is to stop the compilation, for example if we ever get in a bad code path, by using `static assert(false, someString)` (or `static assert(0, someString)`). The string is then emitted as a compiler error message.

Rank for Ranges: D has an interesting sequence concept called a *range*. The Phobos standard library comes with predefined testing templates in std.range. Why not extend `rank` to have it deal with ranges and see if something is a range of ranges or more? A type can be tested to be a range with `isInputRange` and its element type is obtained by applying `ElementType` to the range type. Both templates are found in std.range. Also, since arrays are included in the range concept, we can entirely ditch the array part and use only ranges. Here is a slightly modified version of `rank`:

```
module rank2;
import std.range;

template rank(T)
{
    static if (isInputRange!T)                          // is T a range?
        enum size_t rank = 1 + rank!(ElementType!T); // if yes, recurse
    else
        enum size_t rank = 0;                           // base case, stop there
}

unittest
{
    auto c = cycle([[0,1],[2,3]]); // == [[0,1],[2,3],[0,1],[2,3],[0,1]...
    assert(rank!(typeof(c)) == 2); // range of ranges
}
```

Base Element Type: With `rank`, we now have a way to get the number of `[]`'s in an array type (`T[][][]`) or the level of nesting in a range of ranges. The complementary query would be to get the base element type, `T`, from any array of arrays ... of `T` or the equivalent for a range. Here it is:

```
module baseelementtype;
import std.range;
import rank2;

template BaseElementType(T)
{
    static if (rank!T == 0)       // not a range
        static assert(0, T.stringof ~ " is not a range.");
    else static if (rank!T == 1) // simple range
        alias ElementType!T                         BaseElementType;
    else                          // at least range of ranges
        alias BaseElementType!(ElementType!(T)) BaseElementType;
}
```

Line 8 is an example of `static assert` stopping compilation if we ever get into a bad code path. Line 12 is an example of a Matrioshka call: a template using another template's call as its parameter.

Generating Arrays: Now, what about becoming more generative by inverting the process? Given a type `T` and a rank `r` (a `size_t`), we want to obtain `T[][]...[]`, with `r` levels of `[]`'s. A rank of 0 means producing `T` as the result type.

```
1  module ndim;
2
3  template NDimArray(T, size_t r)
4  {
5      static if (r == 0)
6          alias T NDimArray;
7      else
8          alias NDimArray!(T, r-1)[] NDimArray;
9  }
```

Here, recursion is done on line 8: we instantiate `NDimArray!(T,r-1)`, which is a type, then create an array of them by putting `[]` at the end and expose it through an alias. This is also a nice example of using an integral value, `r`, as a template parameter.

```
module using_ndim;
import ndim;

alias NDimArray!(double, 8) Level8;
static assert(is(Level8 == double[][][][][][][][]));
static assert(is(NDimArray!(double, 0) == double));
```

Repeated composition: As a last example, we will use an alias template parameter in conjunction with some `static if` recursion to define a template that creates the 'exponentiation' of a function, aka its repeated composition. Here is what I mean by this:

```
module using_power;
import repeatedcomposition;

// standard function
string foo(string s) { return s ~ s;}

// function templates. You'll see them soon.
```

```
Arr[] makeArray(Arr)(Arr array) { return [array,array];}

void main()
{
   // power!(foo, n) is a function.
   assert(power!(foo, 0)("a") == "a");                // identity function
   assert(power!(foo, 1)("a") == foo("a"));           // "aa"
   assert(power!(foo, 2)("a") == foo(foo("a")));      // "aaaa"
   assert(power!(foo, 3)("a") == foo(foo(foo("a")))); // "aaaaaaaa"

   // It's even better with function templates:
   assert(power!(makeArray, 0)(1) == 1);
   assert(power!(makeArray, 1)(1) == [1,1]);
   assert(power!(makeArray, 2)(1) == [[1,1],[1,1]]);
   assert(power!(makeArray, 3)(1) == [[[1,1],[1,1]],[[1,1],[1,1]]]);
}
```

First, this is a template that 'returns' (becomes, rather) a function. It's easy to do with the eponymous trick: just define inside the template a function with the same name. Secondly, it's clearly recursive in its definition, with two base cases: if the exponent is zero, then we shall produce the identity function and if the exponent is one, we shall just return the input function itself. That being said, `power` writes itself:

```
1  module repeatedcomposition;
2
3  template power(alias fun, uint exponent)
4  {
5      static if (exponent == 0) // degenerate case -> id function
6          auto power(Args)(Args args) { return args; }
7      else static if (exponent == 1) // end-of-recursion case -> fun
8          alias fun power;
9      else
10         auto power(Args...)(Args args)
11         {
12             return .power!(fun, exponent-1)(fun(args));
13         }
14 }
```

.power If you are wondering what's with the `.power` syntax on line 12, it's because by defining an eponymous template, we hide the parent template's name. So inside `power(Args...)`, `power` refers to `power(Args...)` and not `power(alias fun, uint exponent)`. Here we want a new `power` to be generated so we call on the global `power` template with the 'global scope' operator (`.`).

In all three branches of `static if`, `power` exposes a `power` member, activating the eponymous template trick and allowing for an easy use by the client. Note that this template will work not only for unary (one argument) functions but also for n-args functions [3], for delegates and for structs or classes that define the `()`(ie, `opCall`) operator and for function templates... [4]

Now, are you beginning to see the power of templates?

> **Curried Templates?** No, I do not mean making them spicy, but separating the template's arguments, so as to call them in different places in your code. For `power`, that could mean doing `alias power!2 square;` somewhere and then using `square!fun1`, `square!fun2` at your leisure: the `exponent` parameter and the `fun` alias are separated. In fact, `power` is already partially curried: `fun` and `exponent` are separated from `Args`. For more on this, see section Templates In Templates. Given a template `temp`, writing a `curry` template that automatically generates the code for a curried version of `temp` is *also* possible, but outside the scope of this document.

Templates Specializations Up to now, when we write a `T` in a template parameter list, there is no constraint on the type that `T` can become during instantiation. Template specialization is a small 'subsyntax', restricting templates instantiations to a subset of all possible types and directing the compiler into instantiating a particular version of a template instead of another. If you've read [appendix-isexpression] on the `is()` expression, you already know how to write them. If you didn't, please do it now, as it's really the same syntax. These specializations are a direct inheritance from C++ templates, up to the way they are written and they existed in D from the very beginning, long before `static if` or templates constraints were added.

The specializations are added in the template parameter list, the `(T, U, V)` part of the template definition. `Type : OtherType` restricts `Type` to be implicitly convertible into `OtherType`.

```
module specialization1;

template ElementType(T : U[], U) // can only be instantiated with arrays
{
    alias U ElementType;
```

[3] Except for the degenerate, $n = 0$ case, since the identity function defined above accepts only one arg. A version with more than one argument is possible, but would need to return a tuple.

[4] I cheated a little bit there, because the resulting function accepts any number of arguments of any type, though the standard function parameters checks will stop anything untowards to happen. A cleaner (but longer, and for template functions, more complicated) implementation would propagate the initial function parameter typetuple.

```
}

template ElementType(T : U[n], U, size_t n) // only with static arrays
{
    alias U ElementType;
}

class Array { alias int ElementType;}

template ElementType(T : Array)
{
    alias Array.ElementType ElementType;
}
```

Now, the idea may seem strange to you: if you know you want to restrict `Type` to be `AnotherType`, why make it a template parameter? It's because of templates specializations' main use: you can write different implementations of a template (with the same name, obviously), and when asked to instantiate one of them, the compiler will automatically decide which one to use, taking the 'most adapted' to the provided arguments. 'Most adapted' obeys some complicated rules you can find on the D Programming Language website, but they act in a natural way most of the time. The neat thing is that you can define the most general template *and* some specialization. The specialized ones will be chosen when it's possible.

```
module specialization2;

template InnerType(T : U*, U) // Specialization for pointers
{
    alias U InnerType;
}

template InnerType(T : U[], U) // Specialization for dyn. arrays
{ /*...*/ }

template InnerType(T) // Standard, default case
{ /*...*/ }

void main()
{
    int* p;
    int i;
    alias InnerType!(typeof(p)) Pointer; // pointer spec. selected
    alias InnerType!(typeof(i)) Default; // standard template selected
}
```

This idiom is frequently used in C++, where there is no (built-in) `static if` construct or template constraints. Oldish D templates used it a lot, too, but since other ways have been around for some years, recent D code seems to be more constraint-oriented: have a look at heavily templated Phobos modules, for example std.algorithm or std.range.

> **Specializations or static if or Templates Constraints?** Yes indeed. Let's defer this discussion for when we have seen all three subsystems.

Default Values

Like functions parameters, templates parameters can have default values. The syntax is the same: `Param = defaultValue`. The default can be anything that makes sense with respect to the parameter kind: a type, a literal value, a symbol or another template parameter.

```
module def;

template Default(T = int, bool flag = false)
{
    static if (flag)
        alias T Default;
    else
        alias void Default;
}

alias Default!(double) D1;       // Instantiate Default!(double, false)
alias Default!(double, true) D2; // Instantiate Default!(double, true) (Doh!)
alias Default!() D3;             // Instantiate Default!(int, false)
```

In contrast to function parameters, thanks to Templates Specializations or IFTI, some template parameters can be automatically deduced by the compiler. So, default template parameters are not required to be the final parameters in the list:

```
module deduced;
import std.typecons: Tuple;

template Deduced(T : U[], V = T, U)
{
    alias Tuple!(T,U,V) Deduced;
}
```

```d
alias Deduced!(int[], double) D1; // U deduced to be int. Force V to be a double.
alias Deduced!(int[]) D2; // U deduced to be int. V is int, too.
```

> **Specialization and Default Value?** Yes you can. Put the specialization first, then the default value. Like this: `(T : U[] = int[], U)`. It's not commonly used, though.

As for functions, well-chosen defaults can greatly simplify standard calls. See for example std.algorithm.sort. It's parameterized on a predicate and a swapping strategy, but both are adapted to what most people need when sorting. That way, most client uses of the template will be short and clean, but customization to their own need is still possible.

> **TODO** Maybe something on template dummy parameters, like those used by std.traits.ReturnType. Things like `dummy == void`.

Function Templates

Syntax

If you come from languages with generics, maybe you thought D templates were all about parameterized classes and functions and didn't see any interest in the previous sections (acting on types?). Fear not, you can also do type-generic functions and such in D, with the added generative power of templates.

As we have seen in The Eponymous Trick, if you define a function inside a template and use the template's own name, you can call it easily:

```d
module function_declaration1;
import std.conv : to;

// declaration:
template myFunc(T, int n)
{
    auto myFunc(T t) { return to!int(t) * n;}
}

void main()
{
    // call:
    auto result = myFunc!(double,3)(3.1415);

    assert(result == to!int(3.1415)*3);
}
```

Well, the full story is even better. First, D has a simple way to declare a function template: just put a template parameter list before the argument list:

```
module function_declaration2;
import std.conv:to;

string concatenate(A,B)(A a, B b)
{
    return to!string(a) ~ to!string(b);
}

Arg select(string how = "max", Arg)(Arg arg0, Arg arg1)
{
    static if (how == "max")
        return (arg0 < arg1) ? arg1 : arg0;
    else static if (how == "min")
        return (arg0 < arg1) ? arg0 : arg1;
    else
        static assert(0,
        "select: string 'how' must be either \"max\" or \"min\".");
}
```

Nice and clean, uh? Notice how the return type can be templated too, using `Arg` as a return type in `select`.

auto return

Since you can select among code paths, the function return type can vary widely, depending on the template parameters you passed it. Use `auto` to simplify your code:

```
module morph;

// What morph will return will heavily depend on T and U
auto morph(alias f, T, U)(U arg)
{
    static if (is(T == class))
        return new T(f(arg));
    else static if (is(T == struct))
        return T(f(arg));
    else
        return; // void-returning function.
}
```

auto ref. A function template can have an `auto ref` return type. That means that for templates where the returned values are lvalues, the template will get the **`ref`**ed version. And the non-**`ref`** version if not. I should add some examples for that behaviour.

IFTI

Even better is Implicit Function Template Instantiation (IFTI), which means that the compiler will generally be able to automatically determine a template's parameters by studying the function arguments. If some template arguments are pure compile-time parameters, just provide them directly:

```
module ifti;
import function_declaration2;

struct Foo {}

void main()
{
    string res1 = concatenate(1, 3.14); // A is int and B is double
    string res2 = concatenate("abc", Foo()); // A is string, B is Foo

    auto res3 = select(3,4); // how is "max", Arg is int.
    auto res4 = select!"min"(3.1416, 2.718); // how is "min", Arg is double.
}
```

As you can see, this results in very simple calling code. So we can both declare function templates and call them with a very clean syntax. The same can be done with structs or classes and such, as you will see in the next sections. In fact, the syntax is so clean that, if you are like me, you may forget from time to time that you are *not* manipulating a function (or a struct, etc.): you are manipulating a template, a parameterized piece of code.

A Mantra. `XXX` templates are not `XXX`s, they are templates. With `XXX` being any of (function, struct, class, interface, union). Templates are parameterized scopes and scopes are not first-class in D: they have no type, they cannot be assigned to a variable, they cannot be returned from functions. That means, for example, that you *cannot* return function templates, you cannot inherit from class templates and so on. Of course, *instantiated* templates are perfect examples of functions, classes, and such. Those you can inherit, return...

We may encounter **The Mantra** again in this tutorial.

Example: Flattening Arrays and Ranges

Let's use what we have just seen in a concrete way. In D, you can manipulate 2D and 3D arrays, but sometimes need to process them linearly. As of this writing, neither std.algorithm nor std.range provide a `flatten` function. Beginning with simple arrays, here is what we want:

```
module using_flatten1;
import flatten1;

void main()
{
    assert( flatten([[0,1],[2,3],[4]]) == [0,1,2,3,4] );
    assert( flatten([[0,1]]) == [0,1] );
    assert( flatten([0,1]) == [0,1] );
    assert( flatten(0) == 0 );

    assert( flatten([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]])
            == [0,1,2,3,4,5,6,7,8] );
}
```

So, studying the examples, we want a simple array (rank == 1) or a non-array (rank == 0) to be unaffected by `flatten`: it just returns them. For arrays of rank 2 or higher, it collapses the elements down to a rank-1 array. It's classically recursive: we will apply `flatten` on all sub-arrays with std.algorithm.map and concatenate the elements with std.algorithm.reduce:

```
module flatten1;
import std.algorithm;
import rank2;

auto flatten(Arr)(Arr array)
{
    static if (rank!Arr <= 1)
        return array;
    else
    {
        auto children = map!(.flatten)(array);
        return reduce!"a~b"(children); // concatenate the children
    }
}
```

We make good use of D `auto` return parameter for functions there. In fact, a single call to `flatten` will create one instance per level, all with a different return type.

Note that `flatten` works perfectly on ranges too, but is not lazy: it eagerly concatenates all the elements down to the very last one in the innermost range. Ranges being lazy, a good `flatten` implementation for them should itself be a range that delivers the elements one by one, calculating the next one only when asked to (and thus, would work on infinite or very long ranges too, which the previous simple implementation cannot do). Implementing this means creating a struct template with a factory function. You will find this as an example here.

From our current `flatten`, it's an interesting exercise to add another parameter: the number of levels you want to flatten. Only the first three levels or last two innermost, for example. Just add an integral template parameter that gets incremented (or decremented) when you recurse and is another stopping case for the recursion. Positive levels could mean the outermost levels, while a negative argument would act on the innermost ones. A possible use would look like this:

```
flatten!1([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // == [[[0,1],[],[2],[3], [4,5],[],[6,7,8]]
flatten!2([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // == [0,1,2,3,4,5,6,7,8]
flatten!0([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // ==[[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]
flatten!(-1)([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // ==[[[0,1]], [[2]], [[3,4,5]], [[]], [[6,7,8]]]
```

Anonymous Function Templates

In D, you can define anonymous functions (delegates even, that is: closures):

```
module anonymous_function1;

auto adder(int a)
{
    return (int b) { return a+b;};
}

unittest
{
    auto add1 = adder(1); // add1 is an int delegate(int)
    assert(add1(2) == 3);
}
```

In the previous code, `adder` returns an anonymous delegate. Could `adder` be templated? Ha! Remember **The Mantra**: function templates are templates and cannot be returned. For this particular problem, there are two possible solutions. Either you do not need any new type and just use `T`:

```d
module anonymous_function2;

auto adder(T)(T a)
{
    return (T b) { return a+b;};
}

unittest
{
    auto add1f = adder(1.0); // add1f is an float delegate(float)
    assert(add1f(2.0) == 3.0);

    import std.bigint;

    // addBigOne accepts a BigInt and returns a BigInt
    auto addBigOne = adder(BigInt("1000000000000000"));
    assert(addBigOne(BigInt("1")) == BigInt("1000000000000001"));

    // But:
    // auto error = add1(3.14); // Error! Waiting for an int, getting a double.
}
```

In the previous example, the returned anonymous delegate is *not* templated. It just happens to use `T`, which is perfectly defined once instantiation is done. If you really need to return something that can be called with any type, use an inner struct (see the section on inner structs).

Now, it may come as a surprise to you that D *does* have anonymous function templates. The syntax is a purified version of anonymous functions:

```d
(a,b) { return a+b;}
```

Yes, the previous skeleton of a function is an anonymous template. But, remember The Mantra: you cannot return them. And due to (to my eyes) a bug in **alias** grammar, you cannot alias them to a symbol:

```d
alias (a){ return a;} Id; // Error
```

So what good are they? You can use them with template alias parameters, when these stand for functions and function templates:

```d
module calltwice;

template callTwice(alias fun)
{
```

```
    auto callTwice(T)(T t)
    {
        return fun(fun(t));
    }
}

unittest
{
    alias callTwice!( (a){ return a+1;}) addTwo;
    assert(addTwo(2) == 4);
}
```

Since they are delegates, they can capture local symbols:

```
module using_calltwice;
import calltwice;

unittest
{
    enum b = 3; // Manifest constant, initialized to 3
    alias callTwice!( (a){ return a+b;}) addTwoB;
    assert(addTwoB(2) == 2 + 3 + 3);
}
```

TODO. Some paragraphs on the new `(a,b) => a+b` syntax.

Closures are a Poor Man's Objects

D closures can wrap runtime environment and keep them near their hot little hearts. We can of course create them with a template. To obtain the equivalent of an object, let's create a function that returns a tuple (a std.typecons.Tuple with named arguments.

```
module makecounter;
import std.traits;
import std.typecons;

auto makeCounter(T)(T _counter = T.init) if (isNumeric!T)
{
    bool sense = true;
    auto changeSense = () { sense = !sense;};
    auto inc = (T increment)
               { _counter += (sense ? increment : -increment); };
    auto dec = (T decrement)
```

```
                { _counter += (sense ? -decrement : decrement); };
    auto counter = (){ return _counter;};

    return Tuple!( typeof(changeSense), "changeSense"
                 , typeof(inc), "inc"
                 , typeof(dec), "dec"
                 , typeof(counter), "counter")
                 (changeSense, inc, dec, counter);
}
```

The `if` part after the argument list is just sanity-check template constraint (they are described in the section on constraints). The returned `Tuple` is a bit heavy for my taste, but using named tuple fields gives us a nice object-like call syntax.[5] Otherwise, a simple `return tuple(changeSense, inc, dec, counter);` could have been used but then the inner closures would have to be accessed by their index and not a name.

Here is how to use it:

```
module using_makecounter;
import makecounter;

void main()
{
    auto c = makeCounter(0); // T is int.
    auto c2 = makeCounter!int; // The very same.

    c.inc(5);
    assert(c.counter() == 5);
    c.inc(10);
    assert(c.counter() == 15);
    c.changeSense(); // now each increment will in fact decrement
    c.inc(5);
    assert(c.counter() == 10);
}
```

Function Overloading

Unfinished OK, I need to write something on this.

Storage Classes

As seen in section Instantiating, storage classes get applied to types during instantiation. It also works for function templates arguments:

[5] See the example in section Named Field Tuples for a way to extend Phobos `tuple`.

```d
module storage;

void init(T)(ref T t)
{
    t = T.init;
}

unittest
{
    int i = 10;
    init(i);
    assert(i == 0);
}
```

Should the need arise, this means you can customize your storage classes according to template arguments. There is no built-in syntax for that, so you'll have to resort to our good friend `static if` and the eponymous trick:

```d
// Has anyone a better example?
module init;

template init(T)
{
    static if (is(T == immutable) || is(T == const))
        void init(T t) {} // do nothing
    else static if (is(T == class))
        void init(ref T t)
        {
            t = new T();
        }
    else
        void init(ref T t)
        {
            t = T.init;
        }
}
```

Properties are Automatically Deduced

In D, a function can have the following properties:

- A function can be tagged with the `pure` property, which means it does not have side-effects: the value you get back is the only thing that matters.

- They can also be tagged with `@safe`, `@trusted` and `@system`. `@safe` means a function cannot corrupt memory. A `@trusted` function can call `@safe` ones, but offers no other guaranty concerning memory. And a `@system` function may do whatever it wants.
- The last property is `nothrow` which means the function will not throw any exceptions.

As the compiler gets complete access to a function template code, it can analyze it and automatically deduce properties for you. This feature is still quite new as of this writing, but it seems to work. So, *all* of your function templates will get a smattering of properties when they are instantiated (these properties will of course vary with the template parameters).

in and out Clauses

The `in` and `out` clauses for a function are given full access to a template's parameters. As for other parameterized code, that means you can use `static if` to enable or disable code, depending on the template arguments.

```
module inoutclauses;
import std.complex, std.math, std.traits;

auto squareRoot(N)(N n) if (isNumeric!N || isComplex!N)
in
{
    // no need to do that for a complex.
    static if (isNumeric!N)
        assert(n > 0);
}
body
{
    return sqrt(n);
}
```

Modifying Functions

This section will show you how to use wrapper templates to add new functionalities to pre-defined functions. More powerful examples are shown in **??**, but they use templates we have not seen yet. Be sure to have a look at them when you can, though.

> **Unfinished!** I want to put some small function-wrapping templates:

- making a function accept tuples

- making a function have named parameters (sort of)
- making a function have default values for its args
- making a function accept more args than the original
- making a function accept arguments of a different type (that's useful when mapping on tuples, like in section [Tuples As Sequences])

Accepting a Tuple

```
module acceptingtuple;

template tuplify(alias fun)
{
    auto tuplify(T...)(Tuple!T tup)
    {
        return fun(tup.expand);
    }
}
```

Another interesting (and much more complicated) example is `juxtapose` (see section [juxtapose]).

Mapping n ranges in parallel

> **TODO** Some explanation. Showing the interest of mapping n ranges in parallel.

```
module nmap;
import std.algorithm;
import std.typetuple: allSatisfy;
import acceptingtuple;

// Very easy to do, now:
auto nmap(alias fun, R...)(R ranges) if (allSatisfy!(isInputRange,R))
{
    return map!(tuplify!fun)(zip(ranges));
}
```

More complicated: std.algorithm.map accepts more than one function as template arguments. In that case, the functions are all mapped in parallel on the range, internally using std.functional.adjoin. Here we can extend `nmap` to accept n functions in parallel too. There is a first difficulty:

```
auto nmap(fun..., R...)(R ranges) if (allSatisfy!(isInputRange, R))
{ ... }   ^^^^^^^^^^^^ Uh?
```

See the problem? We need both a variable list of functions and a variable list of ranges. But template parameters tuples must be the last parameter of a template: there can be only one. Double-stage templates come to the rescue:

```
template nmap(fun...) if (fun.length >= 1)
{
    auto nmap(R...)(R ranges) if (allSatisfy!(isInputRange, R))
    {...}
}
```

Now the two parts are well separated. Which gives us the final code:

```
module nmap2;
import std.algorithm;
import std.functional : adjoin;
import std.range;
import std.typetuple : allSatisfy;
import acceptingtuple; // tuplify

template nmap(fun...) if (fun.length >= 1)
{
    auto nmap(R...)(R ranges) if (allSatisfy!(isInputRange, R))
    {
        alias adjoin!(staticMap!(tuplify, fun)) _fun;
        return map!(_fun)(zip(ranges));
    }
}
```

TODO. Give an example with `max`, it works!

And here is the n-ranges version of std.algorithm.filter:

```
module nfilter;
import std.algorithm;
import std.range;
import std.typetuple : allSatisfy;

import acceptingtuple; // tuplify

auto nfilter(alias fun, R...)(R ranges) if (allSatisfy!(isInputRange, R))
{
    return filter!(tuplify!fun)(zip(ranges));
}
```

TODO. Examples, lots of examples.

Struct Templates

Syntax

As you might have guessed, declaring a struct template is done just by putting a list of template parameters after the struct name, like this:

```
module tree1;
import std.array;

struct Tree(T)
{
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}
    /* More tree functions: adding children, removing some,... */
}
```

> **Tree[] or Tree!(T)[]?** Remember that inside a template declaration, the template's name refers to the current instantiation. So inside `Tree(T)`, the name `Tree` refers to a `Tree!T`.

This gives us a run-of-the-mill generic tree, which is created like any other template:

```
module using_tree;
import tree1;

void main()
{
    auto t0 = Tree!int(0);
    auto t1 = Tree!int(1, [t0,t0]);
    Tree!int[] children = t1.children;
}
```

As with all previous templates, you can parametrize your structs using much more than simple types:

```
module heap1;

bool lessThan(T)(T a, T b) { return a<b;}

struct Heap(Type, alias predicate = lessThan, float reshuffle = 0.5f)
```

```
{
    // predicate governs the internal comparison
    // reshuffle deals with internal re-organizing of the heap
    Type[] values;
/*...*/
}
```

Struct templates are heavily used in std.algorithm and std.range for lazy evaluation, have a look there.

Factory Functions

Now, there is one limitation: struct constructors do not do activate IFTI like template functions do. In the previous subsection to instantiate `Tree(T)`, I had to explicitly indicate `T`:

```
auto t0 = Tree!int(0); // Yes.

auto t1 = Tree(0); // Error, no automatic deduction that T is int.
```

This is because templated constructors are possible and may have template parameters differing from that of the global struct template. But honestly that's a pain, even more so for struct templates with many template arguments. There is a solution, of course: use a template function to create the correct struct and return it. Here is an example of such a factory function for `Tree`:

```
module tree2;
import tree1;

auto tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}

void main()
{
    auto t0 = tree(0); // Yes!
    auto t1 = tree(1, [t0,t0]); // Yes!

    static assert(is( typeof(t1) == Tree!int ));

    auto t2 = tree(t0); // Yes! typeof(t2) == Tree!(Tree!(int))
}
```

Once more, have a look at std.algorithm and std.range, they show numerous examples of this idiom.

Giving Access to Inner Parameters

As was said in section Inner Alias, template arguments are not accessible externally once the template is instantiated. For the `Tree` example, you might want to get an easy access to `T`. As for any other templates, you can expose the parameters by aliasing them. Let's complete our `Tree` definition:

```
module tree3;
import std.array;

struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}
}

void main()
{
    Tree!string t0 =Tree!string("abc");
    alias typeof(t0) T0;

    static assert(is( T0.Type == string ));
}
```

Templated Member Functions

A struct template is a template like any other: you can declare templates inside, even function templates. Which means you can have templated member functions.

> **Crippled D.** Okay, since I haven't introduce some powerful D features such as mixin templates and string mixins, I'll have to do some code duplication here. Sorry for that, but that's the condition for having all named code sample compile.

Mapping on a Tree Let us use a templated member function to give our `Tree` a mapping ability. For a range, you can use std.algorithm.map to apply a function in turn to each element, thus delivering a transformed range. The same process can be done for a tree, thereby keeping the overall *shape* but modifying the elements. We could easily make that a free function, but this *is* the member function section.

Let's think a little bit about it before coding. `map` should be a function template that accepts any function name as a template alias parameter (like std.algorithm.map). Let's call this alias `fun`. The `value` member should be transformed by `fun`, that's easy to do. We want to return a new `Tree`, which will have as its type parameter the result type of `fun`. If `fun` transforms `A`s into `B`s, then a `Tree!A` will be mapped to a `Tree!B`. However, since `fun` can be a function template, it may not have a pre-defined return type that could be obtained by std.traits.ReturnType. We will just apply it on a `T` value (obtained by `T.init` and take this type. So `B` will be `typeof(fun(T.init))`.

What about the children? We will map `fun` on them too and collect the result into a new children array. They will have the same type: `Tree!(B)`. If the mapped `Tree` is a leaf (ie: if it has no children), the process will stop.

Since this is a recursive template, we have to help the compiler a bit with the return type. Here we go:[6]

```
module tree4;
import std.array;

auto tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}

struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}

    Tree!(typeof(fun(T.init))) map(alias fun)()
    {
        alias typeof(fun(T.init)) MappedType;
        MappedType mappedValue = fun(value);
        Tree!(MappedType)[] mappedChildren;
        foreach(child; children) mappedChildren ~= child.map!(fun);
        return tree(mappedValue, mappedChildren);
    }
}
```

Let's use it:

[6] The difference with Phobos `map` is that our version isn't lazy.

```
module using_tree4;
import std.conv;
import tree4;

int addOne(int a) { return a+1;}

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
    auto t2 = tree(2, [t1, t0, tree(3)]);

/* t2 is       2
             / | \
            1  0  3
           / \
          0   0
*/

    // t2 is a Tree!(int)
    static assert(is( t2.Type == int ));

    // Adding one to all values

    auto t3 = t2.map!(addOne);

/* t3 is       3
             / | \
            2  1  4
           / \
          1   1
*/

    assert(t3.value == 3);

    // Converting all values to strings
    auto ts = t2.map!(to!string); // we convert every value into a string;

/* ts is      "2"
             / | \
          "1""0""3"
           / \
        "0"  "0"
*/

    assert(is( ts.Type == string ));
```

```
    assert(ts.value == "2");
}
```

Folding a Tree You may feel `map` is not really a member function: it does not take any argument. Let's make another transformation on `Tree`s: folding them, that is: collapsing all values into a new one. The range equivalent is std.algorithm.reduce which collapses an entire (linear) range into one value, be it a numerical value, another range or what have you

For a tree, folding can for example generate all values in pre-order or post-order, calculate the height of the tree, the number of leaves... As for ranges, folding is an extremely versatile function. It can in fact be used to convert a `Tree` into an array or another `Tree`. We will do just that.

Taking inspiration from `reduce`, we need an seed value and *two* folding functions. The first one, `ifLeaf`, will be called on childless nodes, for `fold` to return `ifLeaf(value, seed)`. The second one, `ifBranch`, will be called nodes with children. In this case, we first apply `fold` on all children and then return `ifBranch(value, foldedChildren)`. In some simple cases, we can use the same function, hence a default case for `ifBranch`. Here is the code:[7]

```
module tree5;
import std.array;

Tree!(T) tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}

struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}

    typeof(ifLeaf(T.init, S.init))
    fold(alias ifLeaf, alias ifBranch = ifLeaf, S)(S seed)
    {
        if (isLeaf)
        {
            return ifLeaf(value, seed);
        }
```

[7] Technically, std.algorithm.reduce is a *left fold*, while what is shown here is a *right fold*. The difference is not essential here.

```
        else
        {
            typeof(Tree.init.fold!(ifLeaf, ifBranch)(seed))[] foldedChildren;
            foreach(child; children)
                foldedChildren ~= child.fold!(ifLeaf, ifBranch)(seed);
            return ifBranch(value, foldedChildren);
        }
    }
}
```

Let's play a bit with it. First, we want to sum all values of a tree. For leaves, we just return the node's `value` plus `seed`. For branches, we are given `value` and an array containing the sums of values for all children. We need to sum the values of this array, add it to the node's `value` and return that. In that case, we do not about the seed value.

```
module summingtree;
import std.algorithm;
import tree5;

typeof(T.init + S.init)
sumLeaf(T, S)(T value, S seed)
{
    return value + seed;
}

T sumBranch(T)(T value, T[] summedChildren)
{
    return value + reduce!"a+b"(summedChildren);
}

import std.stdio;

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
    auto t2 = tree(2, [t1, t0, tree(3)]);

    int sum = t2.fold!(sumLeaf, sumBranch)(0);
    assert(sum == 2 + (1 + 0 + 0) + (0) + (3));
}
```

In the same family, but a bit more interesting is getting all values for in-order iteration: given a tree node, return an array containing the local value and then the values for all nodes, recursively.

```
module inordertree;
import std.algorithm;
import tree5;

T[] inOrderL(T, S)(T value, S seed)
{
    return [value] ~ seed;
}

T[] inOrderB(T)(T value, T[][] inOrderChildren)
{
    return [value] ~ reduce!"a~b"(inOrderChildren);
}

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
    auto t2 = tree(2, [t1, t0, tree(3)]);

    int[] seed; // empty array
    auto inOrder = t2.fold!(inOrderL, inOrderB)(seed);
    assert(inOrder == [2, 1, 0, 0, 0, 3]);
}
```

And as a last use, why not build a tree?

TODO. Write just that.

Templated Constructors

Struct constructors are member function, so they can be templated too. They need not have the same template parameters as the struct definition:

```
module templatedconstructors;

struct S(T)
{
    this(U)(U u) { /*...*/ }
}

void main()
{
    auto s = S!string(1); // T is string, U is int.
}
```

As you can see, IFTI works for constructors. `U` is automatically deduced, though you have to indicate `T` in this case. However, this example is drastically limited: you cannot have any value of type `U` in the struct, because `U` does not exist outside the constructor. A bit more useful would be to collect an alias (a function, for example) and use it to initialize the struct. If it's used only for initialization, it can be discarded afterwards. But then, IFTI is not activated by an alias...

The most interesting use I've seen is to make conversions during the struct's construction:

```
module holder;
import std.conv;

struct Holder(Type)
{
    Type value;

    this(AnotherType)(AnotherType _value)
    {
        value = to!Type(_value);
    }
}

void main()
{
    Holder!int h = Holder!int(3.14);
    assert(h.value == 3);
}
```

That way, a `Holder!int` can be constructed with any value, but if the conversion is possible, it will always hold an `int`.

Inner Structs

You can create and return inner structs and use the local template parameters in their definition. We could have a factory function for `Heap` like this:

```
module heap2;

auto heap(alias predicate, Type)(Type[] values)
{
    struct Heap
    {
        Type[] values;
        this(Type[] _values)
```

```
        {
            /* some code initializing values using predicate */
        }
        /* more heapy code */
    }

    return Heap(values); // alias predicate is implicit there
}
```

In that case, the `Heap` struct is encapsulated inside `heap` and uses the `predicate` alias inside its own engine, but it's not a templated struct itself. I did not use `Tree` as an example, because with recursive types it becomes tricky.

By the way, strangely enough, though you cannot declare 'pure' templates inside functions, you can declare struct templates. Remember the `adder` function in section Anonymous Functions? It didn't need to be templated with one type for each argument, as most of the time when you add numbers, they have more or less the same type. But what about a function that converts its arguments to `string`s before concatenating them?

```
module anonymous_error;

auto concatenate(A)(A a)
{
    /* !! Not legal D code !! */
    return (B)(B b) { return to!string(a) ~ to!string(b);};
}
```

The previous example is not legal D code (remember **The Mantra**). Of course, there is a solution: just return a struct with a templated member function, in that case the `opCall` operator:

```
module innerconcatenate;
import std.conv;

auto concatenate(A)(A a)
{
    struct Concatenator
    {
        A a;

        auto opCall(B)(B b) @trusted
        {
            return to!string(a) ~ to!string(b);
        }
```

```
    }

    Concatenator c;
    c.a = a; // So as not to activate opCall()

    return c;
}

void main()
{
    auto c = concatenate(3.14);
    auto cc = c("abc");
    assert(cc == "3.14abc");
}
```

See section operator overloading.

What about templated inner structs inside struct templates? It's perfectly legal:

```
module templatedinner;

struct Outer(O)
{
    O o;

    struct Inner(I)
    {
        O o;
        I i;
    }

    auto inner(I)(I i) { return Inner!(I)(o,i);}
}

auto outer(O)(O o) { return Outer!(O)(o);}

void main()
{
    auto o = outer(1); // o is an Outer!int;
    auto i = o.inner("abc"); // Outer.Outer!(int).Inner.Inner!(string)
}
```

Template This Parameters

Unfinished! Some kind of example would be great here.

Inside a struct or class template, there is another kind of template parameter: the template this parameter, declared with `this identifier`. `identifier` then gets the type of the `this` reference. It's useful mainly for two uses:

- mixin templates, where you do not know the enclosing type beforehand. Please see this section for some examples.
- to determine how the type of the `this` reference is qualified (`const`, `immutable`, `shared`, `inout` or unqualified).

Example: a Concat / Flatten Range

We will use what we've learned about struct templates to create a lazy range that flattens ranges of ranges into linear ranges. Remember the `flatten` function from section Flattening Arrays and Ranges? It worked quite well but was *eager*, not *lazy*: given an infinite range (a cycle, for example) it would choke on it. We will here make a lazy flattener.

If you look at the ranges defined in std.range, you will see that most (if not all) of them are structs. That's the basic way to get laziness in D: the struct holds the iteration state and exposes the basic range primitives. At the very least to be an *input range* —the simplest kind of range— a type must have the following members, be they properties, member functions or manifest constants:

front returns the range's first element.
popFront discards the first element and advances the range by one step.
empty returns `true` if the range has no more elements, `false` otherwise.

From this simple basis, powerful algorithms can be designed that act on ranges. D defines more refined range concepts by adding other constraints. A *forward range* adds the `save` member that's used to store a range internal state and allows an algorithm to start again from a saved position. A *bidirectional range* also has the `back` and `popBack` primitives for accessing the end of the range, and so on.

Here we will begin by creating a simple input range that takes a range of ranges and iterates on the inner elements. Let's begin with the very basics:

```
module flatten2;

import std.range;
import rank2;

struct Flatten(Range)
{
    Range range;
```

```
    /*...*/
}

auto flatten(Range)(Range range)
{
    static if (rank!Range == 0)
        static assert(0, "flatten needs a range.");
    else static if (rank!Range == 1)
        return range;
    else
        return Flatten!(Range)(range);
}
```

So we have a struct template and its associated factory function. It doesn't make sense to instantiate `Flatten` with any old type, so `Range` is checked to be a range, using the `rank` template we saw on Rank for Ranges. We haven't seen template constraints yet (they are described in section Constraints), but they would be a good fit here too.

A range of ranges can be represented like this:

```
[ subRange1[elem11, elem12,...]
, subRange2[elem21, elem22,...]
, ... ]
```

We want `Flatten` to return elements in this order: `elem11`, `elem12`, ... `elem21`, `elem22`, ... Note that for ranges of rank higher than 2, the `elemxy`s are themselves ranges. At any given time, `Flatten` is working on a sub-range, iterating on its elements and discarding it when it's empty. The iteration will stop when the last subrange has been consumed, that is when `range` itself is empty.

```
1  module flatten3;
2  import std.range;
3
4  import rank2;
5
6  struct Flatten(Range)
7  {
8      alias ElementType!Range    SubRange;
9      alias ElementType!SubRange Element;
10
11     Range range;
12     SubRange subRange;
13
```

```
14      this(Range _range) {
15          range = _range;
16          if (!range.empty) subRange = range.front;
17      }
18
19      Element front() { return subRange.front;}
20
21      bool empty() { return range.empty;}
22
23      void popFront() {
24          if (!subRange.empty) subRange.popFront;
25          while(subRange.empty && !range.empty) {
26              range.popFront;
27              if (!range.empty) subRange = range.front;
28          }
29      }
30  }
```

- I cheat a little bit with D standard bracing style, because it eats vertical space like there is no tomorrow.
- We begin on line 8 and 9 by defining some new types used by the methods. They are not strictly necessary but make the code easier to understand and expose these types to the outer world, if they are needed.
- A constructor is now necessary to initialize the struct.
- `front` returns a subrange element.

But then, this only works for ranges of ranges (of rank < 2). We want something that flattens ranges of any rank down to a linear range. This is easily done, we just add recursion in the factory function:

```
module flatten4;
import rank2;
public import flatten3;

auto flatten(Range)(Range range)
{
    static if      (rank!Range == 0)
        static assert(0, "flatten needs a range.");
    else static if (rank!Range == 1)
        return range;
    else static if (rank!Range == 2)
        return Flatten!(Range)(range);
    else          // rank 3 or higher
        return flatten(Flatten!(Range)(range));
}
```

And, testing:

```
module using_flatten4;
import std.algorithm;
import std.range;
import std.string;
import rank2;
import flatten4;

void main()
{
    auto rank3 = [[[0,1,2],[3,4,5],[6] ]
                 ,[[7],[],[8,9],[10,11]]
                 ,[[],[12]             ]
                 ,[[13]                ]];

    auto rank1 = flatten(rank3);
    assert(rank!(typeof(rank1)) == 1); // Yup, it's a linear range
    assert(equal( rank1, [0,1,2,3,4,5,6,7,8,9,10,11,12,13] ));

    auto stillRank1 = flatten(rank1);
    assert(equal( stillRank1, rank1 )); // No need to insist

    auto text =
   "Sing, O goddess, the anger of Achilles son of Peleus,
    that brought countless ills upon the Achaeans.
    Many a brave soul did it send hurrying down to Hades,
    and many a hero did it yield a prey to dogs and vultures,
    for so were the counsels of Jove fulfilled
    from the day on which the son of Atreus, king of men,
    and great Achilles, first fell out with one another.";

    auto lines = text.splitLines;   // array of strings
    string[][] words;
    foreach(line; lines) words ~= array(splitter(line, ' '));
    assert( rank!(typeof(words)) == 3); // range of range of strings
                                        // range of range of array of chars
    auto flat = flatten(words);

    assert(equal(take(flat, 50),
             "Sing,Ogoddess,theangerofAchillessonofPeleus,thatbr"));
}
```

Here it is. It works and we used a struct template, static if, inner member alias), factory functions and IFTI.

Class Templates

> **This Section Needs You!** I'm not an OOP programmer and am not used to create interesting hierarchies. If anyone reading this has an example of class templates that could be used throughout the section, I'm game.

Syntax

No surprise here, just put the template parameters list between `class` and the optional inheritance indication:

```
module classsyntax1;

class Base {}
interface Interface1 {}
interface Interface2 {}

class MyClass(Type, alias fun, bool b = false)
    : Base, Interface1, Interface2
{ /*...*/ }
```

What's more fun is that you can have parameterized inheritance: the various template parameters are defined before the base class list, you can use them here:

```
module classsyntax2;

class Base(T) {}
interface Interface1 {}
interface Interface2(alias fun, bool b) {}

class MyClass(Type, alias fun, bool b = false)
    : Base!(Type), Interface1, Interface2!(fun,b)
{ /*...*/ }
```

> **Interface Templates?** Yes you can. See section Other Templates

This opens interesting vistas, where what a class inherits is determined by its template arguments (since `Base` may be many different classes or even interfaces depending on `Type`). In fact, look at this:

```
module classsyntax3;
```

```d
enum WhatBase { Object, Interface, BaseClass }

template Base(WhatBase whatBase = WhatBase.Object)
{
    static if (is(T == WhatBase.Object))
        alias Object Base; // MyClass inherits directly from Object
    else static if(is(T == WhatBase.Interface))
        alias TheInterface Base;
    else
        alias TheBase Base;
}

class MyClass(Type) : Base!Type {}
```

With this, `MyClass` can inherit from `Object`, the root of D's class hierarchy, from an interface, or from another class. Obviously, the dispatching template could be much more refined. With a second template parameter, the base class could itself be parameterized, and so on.

What this syntax *cannot* do however is change the number of interfaces at compile-time.[8] It's complicated to say: 'with *this* argument, `MyClass` will inherit from `I`, `J` and `K` and with *that* argument, it will inherit only from `L`.' You'd need the previous interfaces to all participate in the action, to all be templates and such. If the needed interfaces are all pre-defined and not templated, you need wrapping templates. It's a pain. However, type tuples can be used to greatly simplify this.

Methods Templates

An object's methods are nothing more than delegates with a reference to the local `this` context. As seen for structs methods can be templates too.

> **Unfinished!** I need to write something on overriding methods with templates. Also, I need to find some interesting method example. Man, I do not do classes.

`invariant` clauses

In the same family than `in`/`out` clauses for functions (section [inandoutclauses]), a class template's `invariant` clause has access to the template parameter. You cannot make it disappear totally, but you can get it to be empty with a `static if` statement.

[8] Except, maybe, by having an interface template be empty for certain parameters, thus in effect disappearing from the list.

```
module classinvariant;

class MyClass(T, U, V)
{
    /*...*/

    invariant ()
    {
        static if (is(T == U))
        {
            /* invariant code */
        }
        else
        { } /* empty invariant */
    }
}
```

Inner Classes

It's the same principle as for structs. You can define inner classes using the template parameters. You can even give them method templates that use other template arguments. There is really nothing different from inner structs.

Anonymous Classes

In D, you can return anonymous classes directly from a function or a method. Can these be templated? Well, they cannot be class templates, that wouldn't make sense. But you can return anonymous classes with templated methods, if you really need to.

```
module anonymousclass;

// stores a function and a default return value.
auto acceptor(alias fun, D)(D defaultValue)
{
    return new class
    {
        this() {}

        auto opCall(T)(T t)
        {
            static if (__traits(compiles, fun(T.init)))
                return fun(t);
            else
```

```
                return defaultValue;
        }
    };
}

unittest
{
    int add1(int i) { return i+1;}
    auto accept = acceptor!(add1)(-1);

    auto test1 = accept(10);
    assert(test1 == 11);

    auto test2 = accept("abc");
    assert(test2 == -1); // default value
}
```

For `__traits(compiles, ...)`, see here online and section ___traits in this document.

Parametrized Base Class

You can use a template parameter directly as a base class:

```
module parametrizedbaseclass;

interface ISerializable
{
    size_t serialize() @property;
}

class Serializable(T) : T, ISerializable
{
    size_t serialize() @property
    {
        return this.tohash;
    }
}
```

In this example, a `Serializable!SomeClass` can act as a `SomeClass`. It's not different from what you would do with normal classes except the idiom is now abstracted on the base class: you write the template once, it can then be used on any class.

If you have different interfaces like this, you can nest these properties:

```
auto wrapped = new Serializable!(Iterable!(SomeClass))(/*...*/);
```

Of course, the base class and the interface may themselves be parameterized:

```
module serializetemplate;

enum SerializationPolicy { policy1, policy2 }

interface ISerializable
(SerializationPolicy policy = SerializationPolicy.policy1)
{
    static if (is(policy == SerializationPolicy.policy1))
        void serialize() { /*...*/ }
    else
        void serialize() { /*...*/ }
}

class Serializable(T, Policy) : T, ISerializable!Policy
{
    /*...*/
}
```

In D, you can also get this kind of effect with an 'alias X this;} declaration in your class or struct. You should also have a look at mixin templates and wrapper templates for other idioms built around the same need.

Adding Functionalities Through Inheritance

Here is an example taken from Timon Gehr. Given a base class `Cell` with a private field and get/set methods, it's possible to provide additional functionalities through inheritance. By defining templates for `Cell` subclasses, each with a new possibility (doubling the value, logging the value, etc.), it's possible to select those new actions by inheritance:

```
/**
 * Timon Gehr timon.gehr@gmx.ch via puremagic.com
 */
module inheritanceexample;

import std.stdio;

abstract class Cell(T)
{
    abstract void set(T value);
```

```
    abstract const(T) get();
private:
    T field;
}

class AddSetter(C: Cell!T,T): C
{
    override void set(T value){field = value;}
}

class AddGetter(C: Cell!T,T): C
{
    override const(T) get(){return field;}
}

class DoubleCell(C: Cell!T,T): C
{
    override void set(T value){super.set(2*value);}
}

class OneUpCell(C: Cell!T,T): C
{
    override void set(T value){super.set(value+1);}
}

class SetterLogger(C:Cell!T,T): C
{
    override void set(T value)
    {
        super.set(value);
        writeln("cell has been set to '",value,"'!");
    }
}

class GetterLogger(C:Cell!T,T): C
{
    override const(T) get()
    {
        auto value = super.get();
        writeln("'",value,"' has been retrieved!");
        return value;
    }
}

// ConcreteCell has a 'true' getter and a 'true' setter
class ConcreteCell(T): AddGetter!(AddSetter!(Cell!T)){}
```

```
// OneUpDoubleSetter has a setter that adds 1 to the stored value
// and then doubles it
class OneUpDoubleSetter(T): OneUpCell!(DoubleCell!(AddSetter!(Cell!T))){}

// doubleOneUpSetter has a setter that doubles the stored value and add 1
class DoubleOneUpSetter(T): DoubleCell!(OneUpCell!(AddSetter!(Cell!T))){}

void main()
{
    Cell!string x;
    x = new ConcreteCell!string;
    x.set("hello");
    writeln(x.get());

    Cell!int y;
    y = new SetterLogger!(ConcreteCell!int);
    y.set(123); // prints: "cell has been set to '123'!

    y = new GetterLogger!(DoubleCell!(ConcreteCell!int));
    y.set(1234);
    y.get(); // prints "'2468' has been retrieved!"

    y = new AddGetter!(OneUpDoubleSetter!int);
    y.set(100);
    writeln(y.get()); // prints "202"
    y = new AddGetter!(DoubleOneUpSetter!int);
    y.set(100);
    writeln(y.get()); // prints "201"
}
```

For those of you knowing the Scala programming language, this is a bit reminiscent of *traits*, which in Scala are sort-of stub classes that can be added to other classes to provide new functions. In D, apart from the above feature, you can also use mixin templates and string mixins.

The Curiously Recurring Template Pattern

```
module crtp;

class Base(Child) { /*...*/ }

class Derived : Base!Derived { /*...*/ }
```

Hold on, what does that mean? `Base` is easy to understand. But what about `Derived`? It says it inherits from another class that is templated on... `Derived` *itself*? But `Derived` is not defined at this stage! Or is it? Yes, that works. It's called CRTP, which stands for Curiously Recurring Template Pattern (see Wikipedia on this). But what could be the interest of such a trick?

As you can see in the Wikipedia document it's used either to obtain a sort of compile-time binding or to inject code in your derived class. For the latter, D offers mixin templates which you should have a look at. CRTP comes from C++ where you have multiple inheritance. In D, I fear it's not so interesting. Feel free to prove me wrong, I'll gladly change this section.

Example: Automatic Dynamic Dispatch

This example comes from Andrej Mitrovic. You can also find it on the D wiki: http://wiki.dlang.org/Dispatching_an_object_based_on_its_dynamic_type.

Sometimes you may need to call a function with an object where the function only accepts a certain derived object type. In such a case you need to ensure that the dynamic type of an object is of a specific type before attempting to cast it to that type and passing it over to a function. For example:

```
class A { }
class B : A { }
class C : A { }

void foo(B b) { }
void foo(C c) { }

void main()
{
    A b = new B;
    A c = new C;

    foo(b);  // not accepted, object must be casted to either B or C
    foo(c);  // ditto
}
```

Since the objects `b` and `c` of static type `A` could be of any dynamic type derived from class `A`, the user would have to try and cast the objects to know whether they can call `foo` with such objects. The code to do the dispatching mechanism by hand would typically look like:

```
import std.stdio;
```

```
class A { }
class B : A { }
class C : A { }

void foo(B b) { writeln("called foo(B b);"); }
void foo(C c) { writeln("called foo(C c);"); }

void main()
{
    A b = new B;
    A c = new C;

    if (auto obj = cast(C)b)
        foo(obj);
    else if (auto obj = cast(B)b)
        foo(obj);

    if (auto obj = cast(C)c)
        foo(obj);
    else if (auto obj = cast(B)b)
        foo(obj);
}
```

However this is both inefficient and hard to type, it introduces copy-pasted code, and is error-prone since adding a new leaf class means the user has to inspect and edit the dispatch code. An alternative method is to use the `classinfo` structure (retrieved via `typeid()`) associated with every instantiated object, and to compare this to all existing classinfo structs. Once we have a match we can safely do a static cast of an object and pass it to a function.

With the help of templates we can automate the entire process. The only information such a template needs is the list of leaf classes so it can construct a tree class to properly dispatch the object. A full implementation is provided here:

```
import std.stdio;
import std.typetuple;
import std.traits;
import std.string;

template ClassTreeImpl(Leaves...)
{
    static if (Leaves.length > 1)
    {
        alias TypeTuple!(Leaves[0], BaseClassesTuple!(Leaves[0]),
                         ClassTreeImpl!(Leaves[1..$])) ClassTreeImpl;
```

```
    }
    else
    static if (Leaves.length == 1)
    {
        alias TypeTuple!(Leaves[0], BaseClassesTuple!(Leaves[0])) ClassTreeImpl;
    }
    else
    {
        alias TypeTuple!() ClassTreeImpl;
    }
}

template ClassTree(Leaves...)
{
    alias DerivedToFront!(NoDuplicates!(ClassTreeImpl!(Leaves))) ClassTree;
}

template AutoDispatch(Leaves...)
{
    void AutoDispatch(alias func, Args...)(Args args)
        if (Args.length >= 1 && is(Args[0] == class))
    {
        auto objInfo = typeid(args[0]);
        foreach (Base; ClassTree!Leaves)
        {
            if (objInfo == Base.classinfo)
            {
                static if (__traits(compiles, { // avoid CT errors due to unrolled static fo
                    return func(cast(Base)(cast(void*)args[0]), args[1..$]); }() ))
                {
                    return func(cast(Base)(cast(void*)args[0]), args[1..$]);
                }
            }
        }

        string[] arguments;
        arguments ~= objInfo.toString();
        foreach (arg; args[1..$])
            arguments ~= typeof(arg).stringof;

        assert(0, format("function '%s' is not callable with types '(%s)'",
                         __traits(identifier, func), arguments.join(", ")));
    }
}

class A { }
```

```
class B : A { }
class C : B { }
class D : B { }

void foo(C c, int x) { writefln("foo(C) : received %s", x); }
void foo(D d, int x, int y) { writefln("foo(D) : received %s %s", x, y); }

void main()
{
    A c = new C;
    A d = new D;
    A a = new A;

    alias AutoDispatch!(C, D) callFunc;

    callFunc!foo(c, 1);    // ok
    callFunc!foo(d, 2, 3); // ok
    callFunc!foo(a, 3);    // will assert at runtime
}
```

`AutoDispatch` takes a list of leaf classes, and extracts the tree class by using the traits found in std.traits. The inner `AutoDispatch` function can then be used to dispatch an object and any additional arguments to a function. The implementation only works for single-object arguments, but a more general solution that dispatches multiple objects is possible to implement.

Other Templates?

Two other aggregate types in D can be templated using the same syntax: interfaces and unions.

Interface Templates

The syntax is exactly what you might imagine:

```
module interfacesyntax;

interface Interf(T)
{
    T foo(T);
    T[] bar(T,int);
}
```

Templated interfaces are sometimes useful but as they look very much like class templates, I won't describe them. As before, remember The Mantra: interface templates are *not* interfaces, they are a blueprint to produce interfaces at your leisure.

Union Templates

Here is the syntax, no suprise there:

```
module unionsyntax;

union Union(A,B,C) { A a; B b; C c;}
```

Union templates seem like a good idea, but honestly I've never seen one. Any reader of this document, please give me an example if you know one.

Enumeration Templates?

Strangely, enumerations do not have the previous simplified syntax. To declare a templated enumeration, use the eponymous template trick:

```
module enumsyntax;

template Enum(T)
{
    enum Enum : T { A, B, C}
}
```

Some More Advanced Considerations

In the previous part, we saw what everyone should know about D templates. But in fact, there is much more to them than that. What follows is not necessarily more complicated, but it's probably a little less commonly used. As this document matures, some subjects may flow from Basics into Advanced and the other way round.

Constraints

Template constraints are a way to block a template instantiation if some condition is not met. Any condition that can be determined at compile-time is authorized, which makes constraints a superset of template specializations. As such, their usage grew rapidly once they were introduced and, if Phobos is any indication, templates specializations are on the contrary becoming less common.

Syntax

To obtain a constraint, put an `if` clause just after the template parameter list, and before the enclosed scope:

```
template templateName(T,U,V) if (someCondition on T, U or V)
{
   ...
}
```

When the compiler tries to instantiate a template, if will first check the constraint. If it evaluates to `false`, the template declaration is not part of the considered set. That way, using constraints, you can keep or drop templates at your leasure. `is` expressions are your friend there, allowing you to get compile-time introspection on types. See the appendix on is expression for a crash course on it.

You may have many template declarations with the same name and differing constraints (in fact, that's the very use case for constraints). Depending on the activated constraints, some or all will be considered by the compiler.

```
module constrained;

template Constrained(T)
    if (is(T : int)) { /*...*/ } // #1
template Constrained(T)
    if (is(T : string)) { /*...*/ } // #2
template Constrained(T,U)
    if (is(T : int) && !is(U : float)) { /*...*/ } // #3
template Constrained(T,U)
    if (is(T : int) && is(U : float)) { /*...*/ } // #4

alias Constrained!(int) C1;        // #1
// alias Constrained!(string) C2; // Error, no declaration fits (string)
alias Constrained!(int,string) C3;// #3 and #4 considered, but #4 is dropped.
                                  // So #3 it is.
```

This syntax is the same for the special-cases templates seen in sections Function Templates, Struct Templates, Class Templates and Other Templates. The only tricky part is for class templates, where you may wonder where to put the constraint: before or after the inheritance list? The answer is: before.

```
module constraintsyntax;

T theFunction(T)(T argument)
    if (is(T : int) || is(T : double))
```

```
{ return argument; }

struct TheStruct(T)
    if (is(T : int) || is(T : double))
{ /*...*/ }

class TheClass(T)
    if (is(T : int) || is(T : double))
    : BaseClass!T, Interface1
{ /*...*/ }
```

When you write constraints, just remember they are a compile-time construct. For `theFunction`, `argument` is not known at compile-time, only its type, `T`. Do not use `argument` in your constraint. If you need a value of type `T`, use `T.init`. For example:

```
module ctvalue;

auto callTwice(alias fun, T)(T arg)
    // Is it OK to do fun(fun(some T))?
    if (is(typeof({
                        fun(fun(T.init));
                    }())))
{
    return fun(fun(arg));
}
```

Constraints Usage

Constraints come from the same idea as C++0x `concept`, er..., concept, although simpler to define, understand and, as shown by D, implement. The idea is to define a set of conditions a type must respect to be a representative of a 'concept', and check for it before instantiating.

Have a look at constraints poster-child: *ranges*.[9] They were rapidly described in section flatten.

std.range defines a set of templates that check the different ranges concepts, called `isInputRange`, `isForwardRange`... I call these `bool`-becoming templates *predicate templates* and talk about them in this section. Usage is quite simple:

```
module rangewrapper;
import std.range;
```

[9] Ranges are overdue a tutorial.

```
struct RangeWrapper(Range)
    // Does Range comply with the input range 'concept'?
    if (isInputRange!Range)
{
   /* Here we know that Range has at least three member functions:
      .front(), .popFront() and .empty(). We can use them happily.*/
}

// In the factory function too.
auto rangeWrapper(Range)(Range range) if (isInputRange!Range)
{
    return RangeWrapper!(Range)(range);
}
```

In fact, it's a bit like a compile-time interface or compile-time duck-typing: we do *not* care about `Range`'s 'kind': it may by a `struct` or a `class` for all we know. What is important is that it respects the *input range* concept.

The good news is that the compiler will complain when it cannot instantiate a template due to constraints being not respected. It gives better error messages this way (although not as good as you might need).

Constraints Limits

The main problem is that, compared to templates specializations, you cannot do:

```
module constraints_error;

template Temp(T)
if (is(T:int))   // #1, specialized for ints
{ /*...*/ }

template Temp(T) // #2, generic case
{ /*...*/ }

alias Temp!int TI;// Error!
```

Why an error? Because the compiler finds that both the `int`-specialized and the generic version can be instantiated. It cannot decide which one you mean and, quite correctly, does nothing, humble code that it is. No problem, says you, we will just add a constraint to the generic version:

```
template Temp(T) if (is(T:int)) // #1
{ ... } // specialized for ints
```

```
template Temp(T) if (!is(T:int))// #2
{ ... } // generic case

Temp!int // Works!
```

Now, when you try to instantiate with an `int`, template #2 is not present (its constraint is false and was dropped from the considered template list) and we can have #1. Hurrah? Not quite. The #1-constraint wormed its way into the generic version, adding code where none was initially. Imagine you had not one, but three different specialized versions:

```
template Temp(T) if (is(T:int[])) // #1a
{ ... } // specialized for arrays of ints

template Temp(T) if (isRange!T) // #1b
{ ... } // specialized for ranges

template Temp(T) if (is(T:double[n], int n)) // #1c
{ ... } // specialized for static arrays of double

template Temp(T) // #2, generic
   if ( /* What constraint? */
{ ... }
```

OK, quick: what constraint for #2? The complement to *all* other constraints. See:

```
template Temp(T) // #2, generic
   if ( !(is(T:int[]))
     && !isRange!T
     && !is(T:double[n], int n))
{ ... }
```

That's becoming complicated to maintain. If someone else adds a fourth specialization, you need to add a fourth inverted version of its constraint. Too bad, you still compile and calls `Temp`: `Temp!(int[])`. And there: error! Why? Because constraints #1a and #1b are not mutually exclusive: an `int[]` is also a input range. Which means that for #1b, you need to add a clause excluding arrays of `int` and maybe modify constraint #2.

Ouch.

So, yes, constraints are wonderful, but they do have drawbacks. As a data point, this author uses them all the time, even though specializations are sometimes more user-friendly: most of what I want to impose and check on my types cannot be done by specializations.

Constraints, Specializations and `static if`

I mean, come on! Three different ways to decide if your template exists or not?

```
template Specialized(T : U[], U)
{ ... }

template Constrained(T) if (is(T : U[], U))
{ ... }

template StaticIfied(T)
{
    static if (is(T : U[], U))
    { ... }
    else // stop compilation
        static assert(0, "StaticIfied cannot be instantiated.");
}
```

What were the D designers thinking? Well, they got specializations from D's cousin, C++. The two other subsystems were added a few years later, as the power of D compile-time metaprogramming became apparent and more powerful tools were needed. So, the 'modern' subsystems are constraints and **`static if`** (Static If). Constraints are much more powerful than specializations, as anything you can test with specialization, you can test with an **`is`** expression in a constraint. And **`static if`** is wonderfully useful outside of template instantiation, so these two are well implanted in D and are there to stay.

What about specializations, now? First, they are quite nice to have when porting some C++ code. Second, they have a nice effect that constraints do *not* have: when more than one definition could be instantiated, priority is given to the more specialized. You saw the explanation in the previous subsection.

So in the end, the conclusion is a bit of *D Zen*: you are given tools, powerful tools. As these are powerful, they sometimes can do what other options in your toolbox can do also. D does not constrain (!) you, chose wisely.

Predicate Templates

When you find yourself typing again and again the same **`is`** expression or the same complicated constraint, it's time to abstract it into another template, a `bool`-becoming one. If you have a look at this section, you'll see a way to test if a particular piece of D code is OK (compilable) or not. Another way to obtain this is by using `__traits(compiles, some Code)`.

In Examples, the section Type Sorcery shows another example of a predicate template.

Testing for a member

For example, if you want to test if some type can be serialized, through a `size_t serialize()` member function:

```
module isserializable;

template isSerializable(Type)
{
    static if (__traits(compiles, {
                                    Type type;
                                    size_t num = type.serialize;
                                   }))
        enum bool isSerializable = true;
    else
        enum bool isSerializable = false;
}
```

Testing for Operations

As seen in previous sections (Struct Flatten, Constraints Usage), we are writing here a kind of compile-time interface. Any type can pass this test, as long as it has a `.serialize` member that returns a `size_t`. Of course, you're not limited to testing member functions. Here is a template that verify some type has arithmetic operations:

```
module hasarithmeticoperations;

template hasArithmeticOperations(Type)
{
    static if (__traits(compiles,
                        {
                            Type t, result;
                            result = t + t; // addition
                            result = t - t; // substraction
                            result = t * t; // multiplication
                            result = t / t; // division
                            result = +t;    // unary +
                            result = -t;    // unary -
                        }))
        enum bool hasArithmeticOperations = true;
    else
        enum bool hasArithmeticOperations = false;
}
```

```
static assert(hasArithmeticOperations!int);
static assert(hasArithmeticOperations!double);

struct S {}
static assert(!hasArithmeticOperations!S);
```

As you can see, you can test for any type of D code, which means it's *much* more powerful than templates specializations or the `is` expression.

You may also get a certain feel of a... *pattern* emerging from the previous two examples. All the scaffolding, the boilerplate, is the same. And we could easily template it on what operator to test, for example. It's possible to do that, but it means crafting code at compile-time. Wait until you see string mixins and CTFE in Around Templates.

Completing the Flatten range

Let's come back to `Flatten` from section Struct Flatten. Using concept-checking templates, we will verify the range-ness of the wrapper type and promote `Flatten` to forward range status if `Range` itself is a forward range:

```
import std.range;

struct Flatten(Range) if (isInputRange!Range)
{
    /* same code as before */

    static if (isForwardRange!Range)
        Flatten save() @property
        {
           return this;
        }
}
```

The struct is enriched in two ways: first, it cannot be instantiated on a non-range. That's good because with the code from section Struct Flatten, you could bypass the factory function and manually create a `Flatten!int`, which wouldn't do. Now, you cannot. Secondly, if the wrapped range is a forward range, then `Flatten!Range` is one also. That opens up whole new algorithms to `Flatten`, for just a quite-readable little piece of code.

You could extend the pattern in the same way by allowing `Flatten` to be a bidirectional range, but you would need to introduce a `backSubRange` member that keeps trace of the range's back state.

Template Tuple Parameters

Definition and Basic Properties

And now comes one of my favourite subjects: template tuple parameters. As seen in section Declarations these are declared by putting a `identifier...` at the last parameter of a template. The tuple will then absorb any type, alias or literal passed to it. For this very reason (that it can be a bunch of types interspersed with symbols), some people consider it a mongrel addition to D templates. That is true, but the ease of use and the flexibility it gives us is in my opinion well worth the cost of a little cleanliness. D template tuples have a `.length` member (defined at compile-time, obviously), their elements can be accessed using the standard indexing syntax and they can even be sliced (the `$` symbol is aliased to the tuple length):

```
module dropfront;

template DropFront(T...)
{
    static if ( T.length > 0     // .length. At least 1 element
             && is(T[0] == int)) // Indexing
        alias T[1..$] DropFront; // Slicing
    else
        alias void DropFront;
}

alias DropFront!(int, double, string) Shortened;
static assert(is( Shortened[0] == double));
static assert(is( Shortened[1] == string));
```

You can declare a value of type `tuple'. This value (called an expression tuple) also has a length, can be indexed and can be sliced. You can also pass it directly to a function if the types check with a function parameter list. If you throw it into an array, it will`melt' and initialize the array:

```
module tupledemonstration;

template TupleDemonstration(T...)
{
    alias T TupleDemonstration;
}

unittest
{
```

```
    TupleDemonstration!(string, int, double) t;

    assert(t.length == 3);
    t[0] = "abc";
    t[1] = 1;
    t[2] = 3.14;
    auto t2 = t[1..$];
    assert(t2[0] == 1);
    assert(t2[1] == 3.14);

    void foo(int i, double d) {}
    foo(t2); // OK.

    double[] array = [t2]; // see, between [ and ]
    assert(array == [1.0, 3.14]);
}
```

The simplest possible tuple is already defined in Phobos in std.typetuple.TypeTuple:

```
template TypeTuple(T...)
{
   alias T TypeTuple; // It just exposes the T's
}

alias TypeTuple!(int, string, double) ISD;
static assert(is( TypeTuple!(ISD[0], ISD[1], ISD[2]) == ISD ));
```

Pure template parameter tuples are auto-flattening: they do *not* nest:

```
module ttflatten;
import std.typetuple;

alias TypeTuple!(int, string, double) ISD;
alias TypeTuple!(ISD, ISD) ISDISD;
// ISDISD is *not* ((int, string, double), (int, string, double))
// It's (int, string, double, int, string, double)
static assert(is(ISDISD == TypeTuple!(int,string,double,int,string,double)));
```

This is both a bug and a feature. On the negative side, that condemns us to linear structures: no trees of type tuples. And since a branching structure can give rise to a linear, that would have been strictly more powerful. On the positive side, that allow us to concatenate tuples easily (and from that, to iterate easily). If you need recursive/branching structures, you can have them by using std.typecons.Tuple or really any kind of struct/class template: the types are not

flattened there. See for example section Polymorphic Tree for a fully polymorphic tree.

The last property tuples have is that they can be iterated over: use a **foreach** expression, like you would for an array. With **foreach**, you can iterate on both type tuples and expression tuples. The indexed version is also possible, but you cannot ask directly for a **ref** access to the values (but see the example below). This iteration is done at compile-time and is in fact one of the main ways to get looping at compile-time in D.

```
module ttiteration;

import std.typetuple;

unittest
{
    alias TypeTuple!(string, int, double) T;
    T t;
    t[0] = "abc";
    t[1] = 1;
    t[2] = 3.14;

    string[T.length] s;

    foreach(index, Type; T) // Iteration on types.
                            // Type is a different, er, type at each position
    {
        static if(is(Type == double))
            s[index] = Type.stringof;
    }
    assert(s ==  ["", "", "double"]);

    void bar(T)(ref T d) { T t; d = t;}

    foreach(index, value; t) // Iteration on values.
                             // value has a different type at each position!
    {
        bar(t[index]); // use t[iindex], not 'value' to get a ref access
    }

    assert(t[0] == "");
    assert(t[1] == 0);

    import std.math;
    assert(std.math.isnan(t[2]));
}
```

As values of this type can be created and named, they are *almost* first-class. They have two limitations, however:

- There is no built-in syntax for declaring a tuple. In the previous example, calling `T.stringof` returns the string `"(string, int,double)"`. But you cannot write `(string, int, double) myTuple;` directly. Paradoxically, if you have a `(string, int, double)` type tuple called `T`, you *can* do `T myTuple;`.
- These tuples cannot be returned from a function. You have to wrap them in a struct. That's what std.typecons.Tuple offers.

`Tuple`, `tuple`, `T...` and `.tupleof`

A common question from newcomers to D is the difference and definition between the different tuples found in the language and the standard library. I will try to explain:

Template tuple parameters are internal to templates. They are declared with `T...` at the last position in the parameter list. They group together a list of template parameters, be they types, values or alias. Two 'subtypes' are commonly used:

Type tuples are template tuple parameters that hold only types.

Expression tuples are tuples that hold only expressions. They are what you get when you declare a variable of type 'type tuple'.

Function parameter tuples You can get a function parameter type tuple from std.traits.ParameterTypeTuple. It's exactly a type tuple as seen before. A value of this type can be declared and can be passed to a function with the same parameters.

The .tupleof property is a property of aggregate types: classes and structs. It returns an expression tuple containing the members's values.

Member names tuple is a tuple of strings you get by using `__traits(members, SomeType)`. It contains all `SomeType` members names, as strings (including the methods, constructors, aliases and such).

std.typetuple.TypeTuple std.typetuple.TypeTuple is a pre-defined template in Phobos that's the simplest possible template holding a tuple. It's the common D way to deal with type tuples. The name is bit of a misnomer, because it's a standard template parameter tuple: it can hold types, but also values.

std.typecons.Tuple std.typecons.Tuple and std.typecons.tuple are pre-defined struct/function templates in Phobos that gives a simple syntax to manipulate tuples and return them from functions.

The Type of Tuples

You can get a tuple's type by using `typeof(tuple)`, like any other D type. There are two limit cases:

One-element tuples. There is a difference between a tuple of one element and a lone type. You cannot initialize a standard value with a 1-element tuple. You have to extract the first (and only) element before. In the same idea, the 1-element tuple has a length and can be sliced: actions that do not make sense for a standard type.

Zero-element tuples. It's possible to have an empty tuple, holding zero type, not to be confused with a uninitialized n-elements tuple or the tuple holding `void` as a type. In fact, the zero-element tuple can only have one value: its initialization value. For this reason, it's sometimes called the Unit type.[10]

void-containing tuples and empty tuples: A type tuple may hold the `void` type, like any other D type. It 'takes a slot' in the tuple and a tuple holding only a `void` is *not* the empty tuple.

```
module emptytuple;
import std.typetuple;

alias TypeTuple!(void) Void;
alias TypeTuple!() Empty;
static assert( !is(Void == Empty) );

static assert(!is( TypeTuple!(int, void, string) == TypeTuple!(int, string)));
```

Example: Variadic Functions Tuples are very useful to make function templates variadic (that is, accept a different number of parameters). Without restriction on the passe-in types, you will need most of the time another function template to process the arguments. A standard example for this is transforming all parameters into a `string`:

```
module variadic;

string toStrings(string sep = ", ", Args...)(Args args)
{
    import std.conv;
    string result;
    foreach(index, argument; args)
```

[10] `bool` is a type with *two* values (`true` and `false`). `()`, the empty tuple, is the type that has only *one* value. And `void` is the type that has *no* value.

```
    {
        result ~= to!string(argument);
        if (index != args.length - 1)
            result ~= sep; // not for the last one
    }
    return result;
}

unittest
{
    assert( toStrings(1, "abc", 3.14, 'a', [1,2,3])
                  == "1, abc, 3.14, a, [1, 2, 3]");
}
```

If you want to restrict the number of parameters or their types, use template constraints:

```
int howMany(Args...)(Args args)
    if (Args.length > 1 && Args.length < 10)
{
    return args.length; // == Args.length
}
```

Imagine you a have a bunch of ranges. Since they all have different types, you cannot put them in an array. And since most of them are structs, you cannot cast them to a base type, as you would for classes. So you hold them in a tuple. Then, you need to call the basic range methods on them: calling `popFront` on all of them, etc. Here is a possible way to do that:

```
module popallfronts;
import std.range, std.algorithm;
import areallranges;

void popAllFronts(Ranges...)(ref Ranges ranges)
    if(areAllRanges!Ranges)
{
    foreach(index, range; ranges)
        ranges[index].popFront; // to get a ref access
}

unittest
{
    auto arr1 = [0,1,2];
    auto arr2 = "Hello, World!";
    auto arr3 = map!"a*a"(arr1);
```

```
    popAllFronts(arr1, arr2, arr3);

    assert(arr1 == [1,2]);
    assert(arr2 == "ello, World!");
    assert(equal( arr3, [1,4]));
}
```

It works for any number of ranges, that's cool. And it's checked at compile-time, you cannot pass it an `int` discretly, hoping no one will see: it's the job of `areAllRanges` to verify that. Its code is a classical example of recursion on type tuples:

```
module areallranges;
import std.range;

template areAllRanges(Ranges...)
{
    static if (Ranges.length == 0) // Base case: stop.
        enum areAllRanges = true;
    else static if (!isInputRange!(Ranges[0])) // Found a not-range:stop.
        enum areAllRanges = false;
    else // Continue the recursion
        enum areAllRanges = areAllRanges!(Ranges[1..$]);
}
```

People used to languages like lisp/Scheme or Haskell will be right at home there. For the others, a little explanation might be in order:

- when you get a typetuple, either it's empty or it's not.
  - If it's empty, then all the elements it holds are ranges and we return `true`.
  - If it's not empty, it has at least one element, which can be accessed by indexing. Let's test it: either it's a range or it's not.
    - If it isn't a range, the iteration stops: not all elements are ranges, we return `false`.
    - If it's a range... we have not proved anything, and need to continue.

The recursion is interesting: by defining an `areAllRanges` manifest constant, we will activate the eponymous template trick, which gets initialized to the value obtained by calling the template on a shortened tuple. With slicing, we drop the first type (it was already tested) and continue on the next one. In the end, either we exhausted the tuple (the length == 0 case) or we find a non-range.

One-Element Tuples: Accepting Types and Alias Sometimes it makes sense for a template to accept either a type parameter or an alias. For example, a template that returns a string representing its argument. In that case, since type parameter do not accept symbols as arguments and the same way round for alias, you're doomed to repeat yourself:

```
module nameof2;

template nameOf(T)
{
    enum string nameOf = T.stringof;
}

template nameOf(alias a)
{
    enum string nameOf = a.stringof;
}

unittest
{
    assert(nameOf!(double[]) == "double[]");

    int i;
    assert(nameOf!(i) == "i");
}
```

Since tuples can accept both types and alias, you can use them to simplify your code a bit:

```
module nameof3;

template nameOf(T...) if (T.length == 1) // restricted to one argument
{
    enum string nameOf = T[0].stringof;
}
```

> **TODO** A better explanation is in order. I'm not that convinced myself.

Example: Inheritance Lists

> **Unfinished** All this section should be rewritten. The compiler is more accepting than I thought.

Using class templates, we might want to adjust the inheritance list at compile-time. Type tuples are a nice way to it: first define a template that alias itself to a type tuple, then have the class inherit from the template:

```
module interfacelist;
import std.typetuple;

interface I {}
interface J {}
interface K {}
interface L {}

class BaseA {}
class BaseB {}

template Inheritance(Base) if (is(Base == class))
{
    static if (is(Base : BaseA))
        alias TypeTuple!(Base, I, J, K) Inheritance;
    else static if (is(Base : BaseB))
        alias TypeTuple!(Base, L)       Inheritance;
    else
        alias Base                      Inheritance;
}

// Inherits from Base
class MyClass : Inheritance!BaseA { /*...*/ }
class MyOtherClass : Inheritance!BaseB { /*...*/ }
```

Here I templated `Inheritance` on the base class, but you could easily template it on a global `enum`, for example. In any case, the selection is abstracted away and the choice-making code is in one place, for you to change it easily.

> **TODO** Here I need to rewrite the junction between the two parts of this section.

Let's begin with something more simple: given a type and a type tuple, eliminate all occurrences of the type in the type tuple.

```
1 module eliminate;
2 import std.typetuple;
3
4 template Eliminate(Type, TargetTuple...)
5 {
```

```
6       static if (TargetTuple.length == 0) // Tuple exhausted,
7           alias TargetTuple
8                 Eliminate;    // job done.
9       else static if (is(TargetTuple[0] : Type))
10          alias Eliminate!(Type, TargetTuple[1..$])
11                Eliminate;
12      else
13          alias TypeTuple!(TargetTuple[0], Eliminate!(Type, TargetTuple[1..$]))
14                Eliminate;
15  }
16
17  unittest
18  {
19      alias TypeTuple!(int,double,int,string) Target;
20      alias Eliminate!(int, Target) NoInts;
21      static assert(is( NoInts == TypeTuple!(double, string) ));
22  }
```

The only difficulty is on line 13: if the first type is not a `Type`, we have to keep it and continue the recursion:

```
Eliminate!(Type,  Type0, Type1, Type2, ...)
->
Type0, Eliminate!(Type,  Type1, Type2, ...)
```

We cannot juxtapose types like I just did, we have to wrap them in a template. Phobos defines `TypeTuple` in std.typetuple for that use.

Now that we know how to get rid of all occurrences of a type in a type tuple, we have to write a template to eliminate all duplicates. The algorithm is simple: take the first type, eliminate all occurrences of this type in the remaining type tuple. Then call the duplicate elimination anew from the resulting type tuple, while at the same time collecting the first type.

```
module noduplicates;
import std.typetuple;
import eliminate;

template NoDuplicates(Types...)
{
    static if (Types.length == 0)
        alias Types NoDuplicates; // No type, nothing to do.
    else
        alias TypeTuple!(
                        Types[0]
```

```
                        , NoDuplicates!(Eliminate!(Types[0], Types[1..$]))
                        ) NoDuplicates;
}

static assert(is( NoDuplicates!(int,double,int,string,double)
                  == TypeTuple!(int,double,string)));
```

By the way, code to do that, also called `NoDuplicates`, is already in Phobos. It can be found in std.typetuple. I found coding it again a good exercise in type tuple manipulation. You can find a few examples of this kind of templates in Type Sorcery.

The last piece of the puzzle is to get a given class inheritance list. The `is` expression give us that, by way of type specializations:

```
module superlist;
import std.typetuple;

template SuperList(Class) if (is(Class == class))
{
    static if (is(Class Parent == super))
        alias TypeTuple!(Parent, SuperList!Parent) SuperList;
}

// for Object, is(Object Parent == super)) gives an empty typetuple
template SuperList()
{
    alias TypeTuple!() SuperList;
}

unittest
{
    class A {}
    class B : A {}
    class C : A {}
    class D : C {}

    static assert(is(SuperList!Object == TypeTuple!()));
    static assert(is(SuperList!A      == TypeTuple!(Object)));
    static assert(is(SuperList!B      == TypeTuple!(A, Object)));
    static assert(is(SuperList!D      == TypeTuple!(C, A, Object)));
}
```

TODO Rewrite the end.

Now we are good to go: given a base class, get its inheritance list with `SuperList`. Drop the base class to keep the interfaces. Stitch with the interfaces provided by `Inheritance` and call `NoDuplicates` on it. To make things clearer, I will define many aliases in the template. To keep the use of the eponymous trick, I will defer the aliasing in another template.

> **TODO** This is not needed anymore. As of DMD 2.060, the eponymous trick works even with internal members defined.

```
module checkedinheritance;
import superlist;
import interfacelist;

template CheckedInheritance(Base)
{
    alias CheckedImpl!(Base).Result CheckedInheritance;
}

template CheckedImpl(Base)
{
    /* Rewrite the
}

// It works!
class NoError : CheckedInheritance!(MyClass) { /*...*/ }
```

Operator Overloading

D allows users to redefine some operators to enhance readability in code. And guess what? Operator overloading is based on (member) templates. They are described here in the docs.

Syntax

The following sections give you the operators that you can overload and which template members you must define in your custom type. `u` is a value of type `U` for which you want to define operators. `v` is a value of type `V` with which `u` will interact. `V` can be the same as `U` or a different type altogether.

op is one of the 'concerned' operators introduced with each section.

Unary Operations Concerned operators: `+`, `-`, `++`, `--`, `*` and `~`.

Operation	Member template to define
*op*u	`opUnary(string s)() if (s == "`*op*`")`
*op*u[i0]	`opIndexUnary(string s)(size_t i0) if (s == "`*op*`")`
*op*u[i0, i1]	`opIndexUnary(string s)(size_t i0,size_t i1) if (s == "`*op*`")`
*op*u[i0, i1, in]	`opIndexUnary(string s)(size_t i0, ...) if (s == "`*op*`")`
*op*u[i..j]	`opSliceUnary(string s)(size_t i, size_t j) if (s == "`*op*`")`
*op*u[]	`opSliceUnary(string s)() if (s == "`*op*`")`

Binary Operations Concerned operators: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `in`, `&`, `|`, `^`, `<<`, `>>` and `>>>`.

Operation	Member template to define
u *op* v	`opBinary(string s, V)(V v) if (s == "`*op*`")`
v *op* u	`opBinaryRight(string s, V)(V v) if (s == "`*op*`")`

Assignement Concerned operators: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` and `>>>` (no `in`).

Operation	Member template to define
u *op* = v	`opOpAssign(string s, V)(V v) if (s == "`*op*`")`

Index Assignement Concerned operators: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` and `>>>` (no `in`).

Operation	Member template to define
u[i0, i1, in] *op* = v	`opIndexOpAssign(string s, V)(V v, size_t i0, ...)` `if (s == "`*op*`")`

Slice Assignement Concerned operators: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` and `>>>` (no `in`).

Operation	Member template to define

Operation	Member template to define
`u[i..j]` *op* `= v`	`opSliceOpAssign(string s, V)(V v, size_t i, size_t j)` `if (s == "`*op*`")`
`u[]` *op* `= v`	`opSliceOpAssign(string s, V)(V v)` `if (s == "`*op*`")`

Operation	Member template to define
`cast(T)u`	`T opCast(T)()`

Cast Operations Many other operators can be overloaded in D, but do not demand templates.

Example: Arithmetic Operators

TODO Tell somewhere that this is possible:

```
Foo opBinary(string op:"+")(...) { ... }
```

The idea behind this strange way to overload operators is to allow you to redefine many operators at once with only one method. For example, take this struct wrapping a number:

```
struct Number(T) if (isNumeric!T)
{
    T num;
}
```

To give it the four basic arithmetic operators with another `Number` and another `T`, you define `opBinary` for `+`, `-`, `*` and `/`. This will activate operations were `Number` is on the left. In case it's on the right, you have to define `opBinaryRight`. Since these overloadings tend to use string mixins, I'll use them even though they are introduced only on section String Mixins. The basic idea is: string mixins paste code (given as a compile-time string) where they are put.

```
module number;
import std.traits;

struct Number(T) if (isNumeric!T)
{
    T num;

    auto opBinary(string op, U)(U u)
         if ((op == "+" || op == "-" || op == "*" || op == "/")
         && ((isNumeric!U) || is(U u == Number!V, V)))
    {
 mixin("alias typeof(a" ~ op ~ "b) Result;
        static if (isNumeric!U)
            return Number!Result(a"~op~"b);
        else
            return Number!Result(a"~op~"b.num);");
    }
}
```

`op` being a template parameter, it's usable to do compile-time constant folding: in this case the concatenation of strings to generate D code. The way the code is written, `Number`s respect the global D promotion rules. A `Number!int` plus a `Number!double` returns a `Number!double`.

Special Case: in

Unfinished

Special Case: cast

Unfinished

Mixin Templates

Up to now, *all* the templates we have seen are instantiated in the same scope than their declaration. Mixin templates have a different behaviour: the code they hold is placed upon instantiation *right at the call site*. They are thus used in a completely different way than other templates.

Syntax

To distinguish standard templates from mixin templates, the latter have slightly different syntax. Here is how they are declared and called:

```d
/* Declaration */
mixin template NewFunctionality(T,U) { ... }


/* Instantiation */
class MyClass(T,U,V)
{
    mixin NewFunctionality!(U,V);

    ...
}
```

As you can see, you put `mixin` before the declaration and `mixin` before the instantiation call. All other templates niceties (constraints, default values, ...) are still there for your perusal. Symbols lookup is done in the local scope and the resulting code is included where the call was made, therefore injecting new functionality.

As far as I know, there is no special syntax for function, class and struct templates to be mixin templates. You will have to wrap them in a standard `template` declaration. In the same idea, there is no notion of eponymous trick with mixin templates: there is no question of how to give access to the template's content, since the template is cracked open for you and its very content put in your code.

TODO Test for mixin T foo(T)(T t) { return t;}

By the way, you *cannot* mix a standard template in. It used to be the case, but it's not possible anymore. Now mixin templates and non-mixin ones are strictly separated cousins.

Mixing Code In

What good are these cousins of the templates we've seen so far? They give you a nice way to place parameterized implementation inside a class or a struct. Once more, templates are a way to reduce boilerplate code. If some piece of code appears in different places in your code (for example, in structs, where there is no inheritance to avoid code duplication), you should look for a way to put it in a mixin template.

Also, you can put small functionalities in mixin templates, giving client code access to them to chose how they want to build their types.

Note that the code you place inside a mixin template doesn't have to make sense by itself (it can refer to `this` or any not-yet-defined symbols). It just has to be syntactically correct D code.

For example, remember the operator overloading code we saw in section Operator Overloading? Here is a mixin containing concatenating functionality:

```
module mixinconcatenate;
import std.typecons;

mixin template Concatenate()
{
    Tuple!(typeof(this), U) opBinary(string op, U)(U u)
    if (op == "~")
    {
        return tuple(this, u);
    }
}
```

As you can see, it uses `this`, even though there is no struct or class in sight. It's used like this, to give concatenation (as tuples) ability to a struct:

```
module usingmixinconcatenate;
import std.typecons;
import mixinconcatenate;

struct S
{
    mixin Concatenate;
}

unittest
{
    S s,t;
    auto result = s ~ t;
    assert(result == tuple(s,t));
}
```

The idea to take back home is: the concatenation code is written once. It is then an offered functionality for any client scope (type) that want it. It could easily have been arithmetic operations, `cast` operations or new methods like `log`, `register`, new members or whatever else. Build you own set of mixins and use them freely. And remember they are not limited to classes and structs: you can also use them in functions, module scopes, other templates...

> **Limitations.** Mixin templates inject code at the local scope. They cannot add an `invariant` clause in a class, or `in`/`out` clauses in a function. They can be injected into an `invariant`/`in`/`out` clause.

TODO Some explanation on `mixin templateName name;`, to allow mixing of more than one template. See http://dlang.org/template-mixin.html#MixinIdentifier.

Mixin Example: Subscriber and Stack

This example comes from Bjorn Lietz-Spendig, who was kind enough to allow me to use it there. Thanks Bjorn!

We will define two mixin templates `PublisherMixin` and `StackMixin`, the first one implementing a subscription/unsubscription engine, the second providing the standard stack operations (`push`, `pop`,...). I'd like you to notice two important things here:

- D allowing local imports, `PublisherMixin` imports the machinery necessary for it to function: std.functional.toDelegate and std.stdio. Local imports really allow D template mixins to provide a nicely wrapped functionality as a coherent whole.
- `alias typeof(this)` (here called `Me`) is also a nice feature to remember: the mixin will 'look around', get the local `this` type and can then provide generic ready-to-be-used code.

```
module publisher;

public mixin template PublisherMixin()
{
    import std.functional : toDelegate;
    import std.stdio;

    alias void delegate(Object sender, string event) CallBack;
    alias void function(Object sender, string event) CallBackFun;


    bool[CallBack] callBacks;  // int[0][CallBack] may be even smaller

    //Register subscriber
    void register(CallBack callBack)
    {
    // Ensure subscriber is not yet registered.
        if (callBack in callBacks)
            writeln("Subscriber is already registered.");
        else
            callBacks[callBack] = true; //from;
    }
```

```
// Register Subscriber via function ptr.
    void register(CallBackFun callBackFun)
    {
        register( toDelegate(callBackFun) );
    }

    // Remove Subscriber
    void unRegister(CallBack callBack)
    {
        if (callBack in callBacks)
            callBacks.remove(callBack);
        else
            writeln("Trying to remove an unknown callback.");
    }

    // Remove Subscriber via function ptr.
    void unRegister(CallBackFun callBackFun)
    {
        unRegister(toDelegate(callBackFun));
    }

    // Notify ALL Subscribers
    void notify(Object from, string evt)
    {
        foreach ( CallBack cb, bool origin ; callBacks )
        {
            cb( from, evt );
        }
    }
}

mixin template StackMixin()
{
    // get parent type
    alias typeof(this) Me;
    static Me[] stack;

protected:

    @property bool empty()  { return stack.length == 0;  }
    @property size_t count()   { return stack.length; }

    void push(Me element)
    {
        stack ~= element;
    }
```

```
    Me pop()
    {
        Me el = peek();
        stack.length -= 1;
        return el;
    }

    Me peek()
    {
        if ( stack.length == 0 )
            throw new Exception("peek on an empty statck.");

        Me el = stack[stack.length-1];
        return el;
    }
}
```

Now that the mixins are defined, we can inject them into any structure with a defined `this`. Here we will use classes to show that mixins are inherited. Let's get some `Friends`:

```
module friends;
import std.stdio;
import publisher;

class FriendStack
{
    private string name;
    private int age;

    // Our Mixins
    mixin StackMixin;
    mixin PublisherMixin;

    void pushFriend(string name, int age)
    {
        // Create new instance for the Stack.
        auto person = new FriendStack();
        person.name = name;
        person.age = age;

        push(person);

        // Notify all subscribers
```

```
        notify(person, "Push");
    }

    //Pop
    void popFriend()
    {
        auto person = pop();
        notify( person, "Pop");
    }

    // Subscriber method
    void inHouseInfo(Object sender, string msg)
    {
        auto p = cast(FriendStack)sender;
        writefln("Subscriber: in House , Name: %s, Age: %s, Message: %s \n",
                 p.name, p.age, msg);
    }
}

class ExtFriendStack : FriendStack // our VIPs
{
    private bool isDeveloper;

    // Push
    void pushFriend(string name, int age, bool D)
    {
        // Create new instance for the Stack.
        auto xperson = new ExtFriendStack();
        xperson.name = name;
        xperson.age = age;
        xperson.isDeveloper = D;

        push(xperson);

        // Notify all subscribers
        notify(xperson, "Push");
    }
}

/**
 *
 *  FRIENDSTACK Subscriber functions - Used by FriendStack AND ExtFriendStack.
 */
void twitter(Object sender, string msg)
{
    auto p = cast(FriendStack)sender;
```

```
    writefln("Subscriber: Twitter, Name: %s, Age: %s, Message: %s \n",
             p.name, p.age, msg);
}

void reddit(Object sender, string msg)
{
    auto p = cast(FriendStack)sender;
    writefln("Subscriber: Reddit, Name: %s, Age: %s, Message: %s \n",
             p.name, p.age, msg);
}

void mail(Object sender, string msg)
{
    auto p = cast(FriendStack)sender;
    writefln("Subscriber: eMail, Name: %s, Age: %s, Message: %s \n",
             p.name, p.age, msg);
}

/**
 *
 *  EXTENDED FRIENDSTACK Subscriber functions
 *  contains additional --isDeveloper-- information.
 */
void blog(Object sender, string msg)
{
    if ( cast(ExtFriendStack)sender )
    {
        auto p = cast(ExtFriendStack)sender;
        writefln("Subscriber: Blog, Name: %s, Age: %s, Is Developer ?: %s, Message: %s \n",
                 p.name, p.age, p.isDeveloper, msg);
    }
    else
    {
        writeln("----- Blog is for VIPs only ------ Message rooted to email");
        mail(sender, msg);

        // Downcast is also possible
        //auto p = cast(FriendStack)sender;
        //writefln("Subscriber: Blog, Name: %s, Age:  %s, Message: %s \n",
        //         p.name, p.age, msg);
    }
}
```

And now we can use the stacks to our heart's content and get some friends:

```
module usingfriends;
```

```
import friends;

void main()
{
    auto p = new FriendStack();
    auto p2 = new ExtFriendStack();

    // Register  some subscribers.
    p.register(  &twitter );
    p2.register( &twitter );
    p2.register( &blog );
    p2.register( &blog );

    // Push and Pop
    p.pushFriend( "Alex", 19 );
    p.pushFriend( "Tommy", 55);
    p2.pushFriend( "Hans", 42, false);
    p2.pushFriend( "Walt", 101, true);

    p.popFriend();
    p2.popFriend();
    p2.popFriend();

    p.unRegister( &twitter );
    p2.unRegister( &twitter );
    p.register( &blog );
    p.pushFriend( "Alexej", 33 );
}
```

If you run the previous code, you'll see that a `DeveloperStack` inherits the mixins from `PersonStack`: it's both a stack and a publisher. Thanks to `alias typeof(this)` defined in `StackMixin`, `ExtFriendStack` can contain additional information.

opDispatch

Syntax

`opDispatch` is a sort of operator overloading (it's in the same place in the online documentation) that deals with members calls (methods or value members). Its definition is the same as an operator:

```
... opDispatch(string name)()
... opDispatch(string name, Arg)(Arg arg)
... opDispatch(string name, Args...)(Args args)
```

The usual template constraints can be used: constraints on `name`, constraints on the arguments.

When a type has an `opDispatch` method and a member call is done without finding a defined member, the call is dispatched to `opDispatch` with the invoked name as a string.

```
module dispatcher;
import std.typetuple;

struct Dispatcher
{
    int foo(int i) { return i*i;}
    string opDispatch(string name, T...)(T t)
    {
        return "Dispatch activated: " ~ name ~ ":" ~ TypeTuple!(T).stringof;
    }
}

void main()
{
    Dispatcher d;

    auto i = d.foo(1); // compiler finds foo, calls foo.
    auto s1 = d.register("abc"); // no register member -> opDispatch activated;
    assert(s1 == "Dispatch activated: register:(string)");

    auto s2 = d.empty; // no empty member, no argument.
    assert(s2 == "Dispatch activated: empty:()");
}
```

Once `opDispatch` has the name called and the arguments, it's up to you to decide what to do: calling free functions, calling other methods or using the compile-time string to generate new code (see section String Mixins).

Since string mixins really go hand in hand with `opDispatch`, I'll use them even though I haven't introduced them right now. The executive summary is: they paste D code (given as a compile-time string) where they are called. There.

Getters and Setters

For example, suppose you have a bunch of members, all private and want client code to access them through good ol' `setXXX/getXXX` methods. Only, you do not want to write all these methods by yourself. You lucky you, `opDispatch` can help you.

```d
module getset;

class GetSet
{
    private int i;
    private int j;
    private double d;
    private string theString;

    auto opDispatch(string name)() // no arg version -> getter
    if (name.length > 3 && name[0..3] == "get")
    {
        enum string member = name[3..$]; // "getXXX" -> "XXX"
        // We test if "XXX" exists here: ie if is(typeof(this.XXX)) is true
        static if (__traits(compiles,
                            mixin("is(typeof(this." ~ member ~ "))")))
        mixin("return " ~ member ~ ";");
        else
            static assert(0, "GetSet Error: no member called " ~ member);
    }

    auto opDispatch(string name, Arg)(Arg arg) // setter
        if (name.length > 3 && name[0..3] == "set")
    {
        enum string member = name[3..$]; // "setXXX" -> "XXX"
        // We test if "member" can be assigned to. this.member = Arg.init
          static if (__traits(compiles, mixin("{" ~ member ~ " = Arg.init;}")))
          {
              mixin(member ~ " = arg;");
              mixin("return " ~ member ~ ";");
          }
          else
              static assert(0, "GetSet Error: no member called" ~ member);
    }
}

unittest
{
    auto gs = new GetSet();
    gs.seti(3);
    auto i = gs.geti;
    assert(i == 3);

    gs.settheString("abc");
    assert(gs.gettheString == "abc"); // "abc"
}
```

Nifty, eh? This could be a bit better by dealing with the capitalization of the first letter: `getTheString`, but this is good enough for now. Even better, you could put this code in a mixin template to give this get/set capacity to any struct or class (see section Mixin Templates).

Wrapping and Subtyping: Expanding Types

Wrapper Templates

> **Unfinished** The idea is to put some functionality inside before (and possibly, after) the dispatching code.

We've seen how to inject code with mixin templates) or use template class inheritance to modify you classes' code (Adding Functionalities Through Inheritance). We've also seen how you can define a wrapper struct around a range to expose a new iteration scheme for its element (Struct Flatten). All these idioms are way to modify pre-existing code.

But what you want to put a logging functionality around a predefined struct, so that any method call is logged? For class, you can inherit from the class and defined a subclass with new, modified, methods. But you have to do that 'by hand', so to speak. And for a struct, you're out of luck.

But, templates can come to the rescue, with a bit of `opDispatch` magic.

> **TODO** Finish this.

- put Type wrapped into a Logger struct.
- get Type.tupleof
- call typeof() on this.
- opDispatch Test if wrapped.foo() is legal.

alias this: Transparent Types

We will play a little bit with `alias this`. Let us define a small wrapper type:

```
module transp;

struct Transparent(T)
{
    T value;
    alias value this;
}
```

```
// factory function
Transparent!T transparent(T)(T t)
{
    return Transparent!T(t);
}
```

This defines `Transparent`, a small struct holding a lone value of type `T`. If a `Transparent!T` is asked to do something for which it has no method nor the correct type (that is to say, almost every action you can imagine!), `alias this` activates and uses `value` in lieu of `Transparent!T`. This gives us a very transparent wrapper:

```
module usingtransparent;
import transp;

void main()
{
    auto i = transparent(10);
    assert(i == 10); // equality testing

    i = 1; // Assignment from an int

    ++i; // ++ acts on i.value
    assert(i == 2);

    // int-accepting function
    int foo(int ii) { return ii+1;}

    assert(foo(i) == 3); // function calls

    int trueInt = i; // transmits its value

    assert(trueInt == 2);
    assert(i == 2);
    assert(i == trueInt);

    // Successive calls all collapse into one Transparent.
    i = transparent(transparent(i));
}
```

As you can see, `Transparent` behaves very gracefully: in almost all circumstances where an `int` could fit, `Tranparent!int` will act as a real `int`. Of course, its type is still OK:

```
assert(is(typeof(i) == Transparent!int));
```

Another wrapper can be added:

```
module transp2;
import transp;

struct Thin(T)
{
    T value;
    alias value this;
}

Thin!T thin(T)(T t)
{
    return Thin!T(t);
}

unittest
{
    auto i = transparent(10);
    auto t = thin(i); // Works.
    assert(t == 10); // Yep, value was transmitted correctly.

    assert(is(typeof(t) == Thin!(Transparent!int)));
}
```

What's the use of such an invisible wrapper? First, it's a perfect subtype, which is the very goal of `alias X this`. You can add functionalities to your type: new methods, etc. Looks a bit like class inheritance, isn't it? Except that `Transparent` can wrap around (ie: become the subtype of) any type it's aimed at. If you are interested in multiple ways to subtype a type, see section Library Typedef, showing Trass3r's library typedef.

Note that this is subtly different from the idea shown in section Wrapper Templates. In section Wrapper Templates, use of `opDispatch` lets the wrapper intercept calls *before* dispatching them. That way, you can add logging to a type, for instance. This is something you cannot do with `alias this`. On the other hand, `alias this` dispatching is automatically done by the compiler, no need to write intercepting code.

For example, here is another wrapper that gives the description of the held type:

```
module descriptor;
import std.stdio;

struct Descriptor(T)
{
```

```
    T value;
    alias value this;

    string description() @property
    {
        import std.conv;

        return "Descriptor holding a " ~ T.stringof
             ~ " of value " ~ to!string(value) ~ ".\n"
             ~ "(sizeof: " ~ to!string(T.sizeof) ~ " bytes)";
    }
}

void main()
{
    auto d = Descriptor!double(1.0);
    writeln(d.description); // "Descriptor holding a double of value 1
                            //  (sizeof: 8 bytes)"
}
```

One day, D will get multiple **alias this**, since it's described in TDPL (TDPL). This will open the possibility of a type being subtype to many other types at the same time, literally acting as a chameleon depending on circumstances. We will be able to group disparate types and functionalities in one place. I wait for it with increasing interest.[11]

However, as far as I'm concerned, the main interest of this subtyping is not so much the possibility to add 'parallel' functionality as encoding information in the type itself. See the next section:

Library Typedef

As we've seen in the previous section, **alias this** gives the perfect subtype, able to mimic its parent type in all circumstances. But what if another derived types is needed? D used to have a **typedef** keyword to defines a new type (as opposed to **alias** redefinition which just define a new symbol/name for a type), but it was dropped from the language in 2011. However, the need to define new types in relation with pre-existing types is still there. Templates can help in defining a library-level solution, which is described here.

Mainly, when a new type is defined in relation with another, already existing, target type, there can be four different relationships shown in the following figure:

[11] This was written in January 2012, let see when we get multiple alias this.

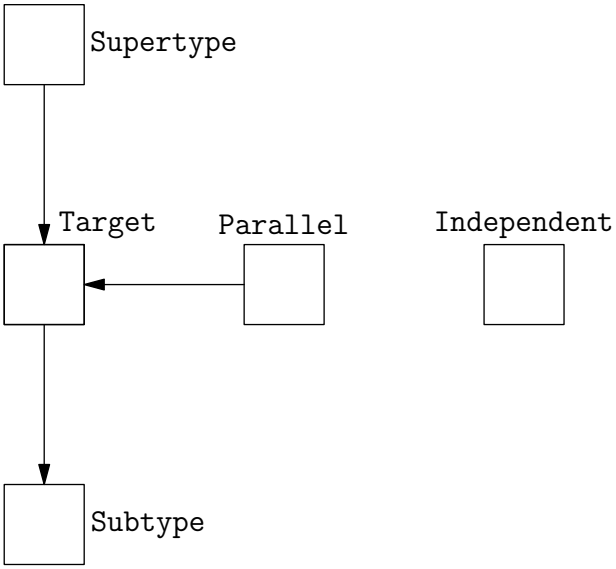

Figure 1: Typedef relationships

subtype That's the standard subclass or `alias this` relationship: a subtype can act *in lieu of* its parent type in all circumstances. Whatever its parent can do, the subtype does also. Moreover, the subtype can be forcibly cast as its parent type (think classes).

supertype The converse: a supertype can be assigned a value of type 'target', its subtype (since a subtype can act as its supertype), and that's about it. Once the supertype and functions acting on it are created, the original target type can be used there also.

parallel A parallel type is neither a subtype not a supertype of the original target (it cannot mimic the target), but it can be created with target values. It's more or less the behaviour of the now-deceased `typedef` keyword.

independent An independent type is just there and bears no relationship whatsoever with the original, target type. I cite it there but the interest of defining such a type while still having a 'target' type in mind is limited...

Here is clean little piece of code from Trass3r that groups all these notions in one template:

```
module librarytypedef;

enum Relationship
{
```

```
    Independent,
    Super,
    Sub,
    Parallel,
}

struct Typedef( Target,
                Relationship relation = Relationship.Sub,
                Target init = Target.init,
                string _f = __FILE__,
                int _l = __LINE__ )
{
    Target payload = init;

    static if ( relation != Relationship.Independent )
        this( Target value )
        {
            payload = value;
        }

    static if ( relation == Relationship.Sub)
        // typedef int foo; foo f;
        // f.opCast!(t)() == cast(t) f
        Target opCast(Target)()
        {
            return payload;
        }

    static if ( relation == Relationship.Sub
             || relation == Relationship.Parallel )
        alias payload this;

    static if ( relation == Relationship.Super )
        typeof( this ) opAssign( Target value )
        {
            payload = value;
            return this;
        }
    else static if ( relation == Relationship.Sub )
        @disable void opAssign( Target value );
}
```

TODO Give some examples.

Types as Information

User-Defined Literals

In http://drdobbs.com/blogs/tools/229401068, Walter Bright makes a convincing case for using templates as user-defined literals. His example is now the std.conv.octal wrapper that allowed D to (see if it's possible to) ditch octal literals as a compiler construct and push them into library-space. That way, you can use:

```
auto o1 = octal!"755";
// or even:
auto o2 = octal!755;
```

as a stand-in replacement for the `0755` octal literal. It's no more complicated to type (a few chars to add at most), it's clearer for a reader of your code (who may miss the beginning `0`) that he's dealing with an octal there. Best of all, the implementation is there as library code: easy to reach, easy to debug, less intimidating than diving into compiler code, and easy to duplicate for other encodings. That way, this could be done:

```
auto h1 = hexa!"deadbeef";
auto b1 = binary!1011011;
// Or even:
auto number = base!(36, "4d6r7th2h7y");
```

Behind the scene, `octal` reads its `string` or `int` argument and converts it into an `int` or a `long` depending on the argument size. It's a nice piece of work that could pave the way to similar well-integrated language extensions.

> **Unfinished** Also, DSL in strings in D, see Statically-Checked Writeln, Encoding Informations with Types and Annotating Types).

Encoding Information With Types

This is an extension of the previous section's idea, that can be found for example in std.range.assumeSorted and std.range.SortedRange. These Phobos constructs encode some information in a type (in this case, the fact than the range is sorted, with an associated predicate). That way, subsequent operations acting on a range can use better algorithm if they know it's sorted.

This kind of encoding can be used for many different schemes:

- For a matrix library, you could have a, for example, `assumeHermitian` or `assumeTriangular` that just wraps a pre-existing matrix.

- For a XML/HTML library, you could imagine a `Validated` struct that's used to indicate, er, validated content. Any external input will have to pass trough a `validate` function that delivers the `Validated` object. Subsequent operations will work only on `Validated` data, not raw data.
- A units library (as in, $kg.m/s^2$, not unit-testing) is mainly a bunch of `double` (or complex) values wrapped in a multi-checked type that allows only some operations, depending on types. `meter(5) + kilo(gram(10))` is a no-no, but `meter(5)*kilo(gram(10))` is OK.

Moreover, it's easy to provide additional information:

```
struct MinMax(T)
{
    T min;
    T max;

    T value;

    alias value this;
}
```

In this case, `MinMax` means the wrapped value is between `min` and `max` (the enforcing code would be in the factory function, for example).

Things get interesting when using multiple wrappers inside one another. Imagine we have three wrappers, for numerical ranges with an ordering operation:

- `minmaxed`, which ascertains (well, at least transmits the message) that the wrapped value is between to extrema.
- `periodic`, which encodes the idea that a range has a period, accessible through the `.period` member.
- `derivative`, which says successive elements' difference is no more than a `.slope` number (in absolute).

```
import std.range: cycle;

auto range = cycle([0,1,2,3,4,3,2,1]);
auto withMetadata = minmaxed(
                            periodic(
                                     derivative( range
                                               , 1)
                                     , 8
                            , 0, 10);

assert(withMetadata.period == 8); // propagated all the way up.
```

So, in the previous sample, `withMetadata`'s elements are at the same time limited in variation, limited in value and periodic. The trouble is when we want to shift metadata around, comparing types with metadata ordered in a different way: obviously for us humans, a range with suprema and periodic is *also* periodic and with suprema. But as far as D is concerned a `minmaxed(periodic())` and a `periodic(minmaxed())` do not have the same type. For more on this notion, see section annotatingtypes on annotating types.

TODO A unit example in the Examples part? It may be a bit big...

Templates in Templates

Sometimes, you know users of your code will send to your template a first list (of indefinite length) of parameters, followed by a second list. Seeing that, you may want to write code like the following:

```
template MyTemp(A..., B...)
{ ... }
```

Alas, two template parameters tuples do not make sense (see Template Declarations. But it's not a dead-end. First, you could try to write:

```
template(AB...)
{ ... }
```

And filter the `AB`s to find the ones you need. It can be done by `staticFilter`, presented in Static Filter (and its associated function on a function arguments would be `tupleFilter` shown in Filtering Tuples). In this case, the two series of arguments can be completely mixed (and not separated in two lists), which is strictly more powerful. On the other hand, it means you must be able to separate them by their types alone, which may not be possible.

Templates All the Way Down

Happily, the initial problem can easily be solved this way:

```
template MyTemp(A...)
{
    template MyTemp(B...)
    {
    (...)
    }
}
```

Remember, a template can have template members. So you can nest templates within templates, each with its own constraints and intermediate code. If you use the same name through and through the eponymous trick will be activated, but sometimes using a different name at each stage can make your code more readable.

For example, what if you want to compare two template tuples to see if they contain the same arguments?

```
module compare;

template Compare(First...)
{
    template With(Second...)
    {
        static if (First.length != Second.length)
            enum With = false;
        else static if (First.length == 0) // End of comparison
            enum With = true;
        else static if (!is(First[0] == Second[0]))
            enum With = false;
        else
            enum With = Compare!(First[1..$]).With!(Second[1..$]);
    }
}

//Usage:
unittest
{
    alias Compare!(int, double, string).With!(int, double, char) C;
    static assert(C == false);
}
```

In that case, using `With` inside `Compare` let the code be quite easy to use. Notice that the eponymous trick is done only on `With`, because it's this inner template that we want the result of.

Going back to `MyTemp`, using it is slightly more complicated:

```
1 // Yes:
2 alias MyTemp!(int, double, string) MyTemp1;
3 MyTemp1!(char, void);
4
5 // No:
6 MyTemp!(int, double, string)!(char, void);
7 // No :
8 MyTemp!(int, double, string).!(char, void);
```

The D grammar does not authorize multiple template calls like the ones on lines 6 and 8. You must use an alias for the intermediate stage. It's not that drastic a limitation, because if you created the template as a multi-step construction, it's most probably because you wanted to do a multi-step invocation...

Double-Stage Function Templates

For function templates, this can give very powerful things. You can have a place in your code where compile-time parameters are given, which delivers another, crafted-just-for-your-needs, template which you can instantiate later on. See for example String Interpolation and the `interpolate` function-in-template.

Policies are particularly good with this idiom. Section Memoizing a Function presents a template that transforms a standard D function into a memoized one. Here is what could be done if it was a two-steps template:

```
/*
 * memoizer will store the first million tuple of args
 * and discard half of them when the maximum is reached,
 * to free some memory. At this stage, the function it
 * will memoize is not known. The user decided this
 * particular instantiation was the best memoizing
 * strategy in her code. memoizer could (and will!) be
 * applied on many different functions.
 */
alias memoize!( Storing.maximum
              , 1_000_000
              , Discarding.fraction
              , 0.5f
              ) memoizer;

auto longCalculation1(int i, double d) { ... }
auto longCalculation2(double d, string s) { ... }

/*
 * longCalculation1 and longCalculation2 will both profit
 * from the memoization even though they have different
 * signatures, different arguments and return a different type.
 */
alias memoizer!longCalculation1 mlc1;
alias memoizer!longCalculation2 mlc2;
```

> **TODO** OK, now maybe I should provide an updated version of `memoize` in Memoizing a Function.

Named-Fields Tuples

Let's have another example, to use IFTI. In Phobos, the function std.typecons.tuple lets you create a tuple on the fly. It's a very nice example of IFTI in action:

```
module usingtuple;
import std.typecons;

void main()
{
    // tuple1 is a Tuple!(int,double,string)
    auto tuple1 = tuple(1, 3.14159, "abc");
    // tuple2 is a Tuple!(char,char,char)
    auto tuple2 = tuple('a','b','c');
}
```

But Phobos' `Tuple` is more powerful than that. It can have named parameters:

```
Tuple!(int, "counter", string, "name") myCounter;
myCounter.counter = -1;
myCounter.name = "The One and Only Counter Around There";
```

As of this writing, Phobos doesn't provide a `tuple` factory function allowing named arguments in an nice, automated manner. Here is what I'd like to have:

```
module usingnamedtuple;
import namedtuple;

void main()
{
    auto myCounter = tuple!("counter", "name")
                           (-1, "Who's his Daddy's counter?");

    myCounter.counter = 1;

    // Or even:
    alias tuple!("counter", "name") makeCounter;

    auto c1 = makeCounter(0, "I count ints!");
    auto c2 = makeCounter("Hello", "I'm a strange one, using strings");

    c2.counter ~= " World!";
}
```

In the previous example, `c1` is a `Tuple!(int, string)`, whereas `c2` is a `Tuple!(string, string)`. That means `makeCounter` is a factory function for tuples with two fields, named `counter` and `name`, which see their types determined later on. I want this, so let's code it.

First, it's obvious we need a two-stage template:

```
import std.typetuple: allSatisfy;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args)
    {
    (...)
    }
}
```

The constraint is here to check that the user gives at least one name and also that all passed `names` are indeed string literals template parameters. I use std.typetuple.allSatisfy to verify the condition on all of them. I cannot use directly std.traits.isSomeString because this template acts on types, whereas I need something checking a string literal.

```
module isastringliteral;
import std.traits: isSomeString;

template isAStringLiteral(alias name)
{
    enum isAStringLiteral = isSomeString!(typeof(name));
}
```

That being in place, we need to create the correct `Tuple`. Arguments names are provided right after each type (so as to allow for a mix between named and anonymous fields:

```
/*
 * The first two fields are named, and the fourth.
 * The third is anonymous.
 */
alias Tuple!( int,    "counter"
            , string, "name"
            , double  /* Anonymous */
            , double, "total"
            ) MyTuple;
```

For our function, we consider that, given n names, the first n arguments will be named and the remaining (if any) will be anonymous. That give us another constraint: if the user provide less than n arguments, we refuse the input and stop the compilation right there.

```
import std.typetuple: allSatisfy;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args) if (T.length >= names.length)
    {
    (...)
    }
}
```

Now, we just need the alternate the names and the argument types. Happily, this document describes a `Interleave` template in Interleaving Types, that does just that:

```
module namedtuple;
import std.typecons;
import std.typetuple;
import interleave;
import isastringliteral;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args) if (T.length >= names.length)
    {
        return Tuple!(Interleave!(T).With!(names))(args);
    }
}
```

And, presto, here we have our named-tuple factory function. Isn't that nice? The closure example in Closures Are A Poor Man's Objects could use it to simplify its returned value.

> **TODO** And a curried template.

`__FILE__` and `__LINE__`

> **Unfinished** This section needs some heavy testing. My D config was broken when I wrote this part. Take everything in there with a *big* grain of salt. It's on my todo list to test and rewrite everything.

In section Default Values, we've seen that template parameters can have default values. There are also two special, reserved, symbols that are defined in D: `__FILE__` and `__LINE__`. They are used in standard (non-`mixin`) templates, but their behaviour will remind you of mixins: when instantiated, they get replaced by strings containing the file name and the line in the file of the *instantiation call site*. Yes, it's a sort of two-way dialogue: module `a.d` defines template `T`. Module `b.d` asks for a `T` instantiation. This instantiation is done in module `a.d`, but will line and filename taken from `b.d`!

They are mostly declared like this:

```
module filelinetagging;

struct Unique(T, string file, size_t line)
{
    enum size_t l = line;
    enum string f = file;
    T t;
}

auto unique(T, string file = __FILE__, size_t line = __LINE__)(T t)
{
    return Unique!(T, file, line)(t);
}
```

As `Unique`'s name suggests, this is a way to obtain unique instantiations. Except if you call the very same template twice in the same line of your file, this pretty much guarantee your instantiation will be the only one. Remember that template arguments become part of the template scope name when instantiation is done (Instantiating a Template).

```
module usingunique;
import filelinetagging;

void main()
{
    auto u = unique(1); // Unique!(int, "thefile.d", 4)

    auto v = unique(1); // Unique!(int, "thefile.d", 6)

    static assert(!is( typeof(v) == typeof(u) ));
}
```

Even though `u` and `v` are declared the same way, they have different types.

Apart from *one-of-a-kind* types, this is also useful for debugging: you can use the strings in error messages:

```
auto flatten(Range, file == __FILE__, line == __LINE__)(Range range)
{
    static if (rank!Range == 0)
        static assert(0, "File: " ~ file ~ " at line: " ~ line
                       ~ ", flatten called with a rank-0 type: "
                       ~ Range.stringof);
    else static if (rank!Range == 1)
        return range;
    else
        return Flatten!(Range)(range);
}
```

And here is a little gift:

```
module debugger;

/** Usage:
 * Debug!(templateToBeTested).With!(Argument0, Argument1, Argument2);
 */
template Debug(alias toTest, string file = __FILE__, size_t line = __LINE__)
{
    template With(Args...)
    {
        static if (is( toTest!Args ))
            alias toTest!Args With;
        else
            static assert(0, "Error: " ~ to!string(toTest)
                           ~ " called withs arguments: "
                           ~ Args.stringof);
    }
}
```

That way, no need to modify your beautiful templates.

TODO Test that.

Around Templates: Other Compile-Time Tools

There is more to compile-time metaprogramming in D than *just* templates. This part will describe the most common tools: string mixins, compile-time function evaluation (CTFE) and `__traits`, as seen in relation with templates. For the good news is: they are all interoperable. String mixins are wonderful to inject code in your templates, compile-time-evaluable functions can act as template

parameters and can be templated. And, best of best, templated compile-time functions can return strings which can in turn be mixed-in in your templates. Come and see, it's fun!

String Mixins

String mixins put D code were they are called, just before compilation. Once injected, the code is *bona fide* D code, like any other. Code is manipulated as strings, hence the name.

Syntax

The syntax is slightly different from mixin templates:

```
mixin("some code as a string");
```

You must take care not to forget the parenthesis. String mixins are a purely compile-time tool, so the string must also be determined at compile-time.

Mixing Code In, With Templates

Of course, just injecting predefined code is a bit boring:

```
mixin("int i = 3;"); // Do not forget the two semicolons
                     // one for the mixed-in code,
                     // one for the mixin() call.
i++;
assert(i == 4);
```

There is no interest in that compared to directly writing standard D code. The fun begins with D powerful constant folding ability: in D, strings can be concatenated at compile-time. That's where string mixins meet templates: templates can produce strings at compile-time and can get strings as parameters. You already saw that in section Operator Overloading and section `opDispatch`, since I couldn't help doing a bit of foreshadowing.

Now, imagine for example wanting a template that generates structs for you. You want to be able to name the structs as you wish. Say we would like the usage to look like that:

```
module usingmystruct1;
import mystruct1;
```

```
mixin(MyStruct!"First");  // creates a new type called First (a struct)
mixin(MyStruct!"Second"); // and another one called Second

void main()
{
    // "First" code was injected right there, in module 'mine'.
    First f1, f2;
    Second s1;

    assert(is( typeof(f1) == First));
}
```

Here comes the generating code:

```
module mystruct1;

template MyStruct(string name)
{
    enum string MyStruct = "struct " ~ name
                         ~ " { "
                         ~ "/+ some code +/"
                         ~ " }";
}

// For example, with name == "First", it will return
//   "struct First { /+ some code +/ }"
//
```

In this case, the string is assembled inside the template during instantiation, exposed through the eponymous trick and then mixed in where you want it. Note that the string is generated in the `utils` module containing `MyStruct`, but that `First` and `Second` are defined exactly where the `mixin()` call is. If you use the mixin in different modules, this will define as many different structs, all named the same way. This might be exactly what you want, or not.

To get the same struct type in different modules, the code must be organized a bit differently: the structs must be generated in the template module.

```
module mystruct2;

template MyStruct(string name)
{
    alias MyStructImpl!(name).result MyStruct;
}
```

```
template MyStructImpl(string name)
{
    enum string code = "struct " ~ name
                     ~ " { "
                     ~ "/+ some code +/"
                     ~ " }";
    mixin(code);
    mixin("alias " ~ name ~ " result;");
}
```

```
module usingmystruct2;
import mystruct2;

MyStruct!"First" f1, f2;
MyStruct!"Second" s1;
```

Usage is a different, as you can see. In this case, `First` is generated inside `MyStructImpl` and exposed through an alias (this particular alias statement is itself generated by a string mixin). In fact, the entire code could be put in the mixin:

```
module mystruct3;

template MyStruct(string name)
{
    alias MyStructImpl!(name).result MyStruct;
}

template MyStructImpl(string name)
{
mixin("struct " ~ name
    ~ " {"
    ~ "/* some code */"
    ~ " }\n"
    ~ "alias " ~ name ~ " result;");
}
```

Here is an example using the ternary `?:` operator to do some compile-time selection of code, similar to what can be done with **`static if`** (Static If):

```
module getset2;

enum GetSet { no, yes}
```

```
struct S(GetSet getset = GetSet.no, T)
{
    enum priv = "private T value;\n"
              ~ "T get() @property { return value;}\n"
              ~ "void set(T _value) { value = _value;}";

    enum pub = "T value;";

    mixin( (getset == GetSet.yes) ? priv : pub);
}
```

The code:

```
import getset2;

void main()
{
    S!(GetSet.yes, int) gs;
    gs.set(1);
    assert( gs.get == 1);
}
```

generates:

```
struct S!(GetSet.yes, int)
{
    private int value;
    int get() @property { return value;}
    void set(int _value) { value = _value;}
}
```

Limitations

Code crafting is still a bit awkward, because I haven't introduced CTFE yet (see CTFE). So we are limited to simple concatenation for now: looping for example is possible with templates, but far easier with CTFE. Even then, it's already wonderfully powerful: you can craft D code with some 'holes' (types, names, whatever) that will be completed by a template instantiation and then mixed in elsewhere. You can create other any kind of D code with that.

You can put `mixin()` expressions almost were you want to, but...

TODO Test the limits:inside static if expressions, for example

Escaping strings. One usual problem with manipulating D code as string is how to deal with strings in code? You must escape them. Either use `\"` to create string quotes, a bit like was done in section Function Template Syntax to generate the error message for `select`. Or you can put strings between `q{` and `}` or between backquotes.

Compile-Time Function Evaluation

Evaluation at Compile-Time

Compile-Time Function Evaluation (from now on, CTFE) is an extension of the constant-folding that's done during compilation in D code: if you can calculate `1 + 2 + 3*4` at compile-time, why not extend it to whole functions evaluation? I'll call evaluable at compile-time functions CTE functions from now on.

It's a very hot topic in D right now and the reference compiler has advanced by leaps and bounds in 2011 and 2012. The limits to what can be done with CTE functions are pushed farther away at each new release. All the `foreach`, `while`, `if`/`else` statements, arrays manipulation, struct manipulation, function manipulation are there. You can even do pointer arithmetics! When I began this document (DMD 2.055), the limitations were mostly: no classes and no exceptions (and so, no `enforce`). This was changed with DMD 2.057, allowing the manipulation of classes at compile-time. DMD 2.058-2.061 then killed a huge amount of CTFE-related bugs.

In fact danger lies the other way round: it's easy to forget that CTE functions must also be standard, runtime, functions. Remember that some actions only make sense at compile-time or with compile-time initialized constants: indexing on tuples for example:

`__ctfe`

Unfinished Write something on this new functionality, which enables testing inside a function whether we are at compile-time or runtime.

Templates and CTFE

Unfinished Some juicy examples should be added.

That means: you can feed compile-time constants to your classical D function and its code will be evaluated at compile-time. As far as templates are concerned, this means that function return values can be used as template parameters and as `enum` initializers:

```
/* */
```

Template functions can very well give rise to functions evaluated at compile-time:

Templates and CTFE and String Mixins, oh my!

And the fireworks is when you mix(!) that with string mixins: code can be generated by functions, giving access to almost the entire D language to craft it. This code can be mixed in templates to produce what you want. And, to close the loop: the function returning the code-as-string can itself be a template, using another template parameters as its own parameters.

Concretely, here is the getting-setting code from section Getters and Setters, reloaded:

```
module getset3;
import std.conv;

enum GetSet { no, yes}

string priv(string type, string index)
{
    return
    "private "~type~" value"~index~";\n"
  ~ type~" get"~index~"() @property { return value"~index~";}\n"
  ~ "void set"~index~"("~type~" _value) { value"~index~" = _value;}";
}

string pub(string type, string index)
{
    return type ~ "value" ~ index ~ ";";
}

string GenerateS(GetSet getset = GetSet.no, T...)()
{
    string result;
    foreach(index, Type; T)
        static if (getset == GetSet.yes)
             result ~= priv(Type.stringof, to!string(index));
        else
            result ~= pub(Type.stringof, to!string(index));
    return result;
}
```

```
struct S(GetSet getset = GetSet.no, T...)
{
    mixin(GenerateS!(getset,T));
}

void main()
{
    S!(GetSet.yes, int, string, int) gs;

/* Generates:
struct S!(GetSet.yes, int, string, int)
{
    private int value0;
    int get0() @property { return value0;}
    void set0(int _value) { value0 = _value;}

    private string value1;
    string get1() @property { return value1;}
    void set1(string _value) { value1 = _value;}

    private int value2;
    int get2() @property { return value2;}
    void set2(int _value) { value2 = _value;}
}
*/

    gs.set1("abc");
    assert(gs.get1 == "abc");
}
```

This code is much more powerful than the one we saw previously: the number of types is flexible, and an entire set of getters/setters is generated when asked to. All this is done by simply plugging `string`-returning functions together, and a bit of looping by way of a compile-time `foreach`.

Simple String Interpolation

All this play with the concatenating operator (~) is becoming a drag. We should write a string interpolation function, evaluable at compile-time of course, to help us in our task. Here is how I want to use it:

```
import stringinterpolation;

alias interpolate!"struct #0 { #1 value; #0[#2] children;}" makeTree;
```

```
enum string intTree = makeTree("IntTree", "int", 2);
enum string doubleTree = makeTree("DoubleTree", "double", "");

static assert(intTree
          == "struct IntTree { int value; IntTree[2] children;}");
static assert(doubleTree
          == "struct DoubleTree { double value; DoubleTree[] children;}");
```

As you can see, the string to be interpolated is passed as a template parameter. Placeholders use a character normally not found in D code: `#`. The n^{th} parameter is `#n`, starting from 0. As a concession to practicality, a lone `#` is considered equivalent to `#0`. Args to be put into the string are passed as standard (non-template) parameters and can be of any type.

```
module stringinterpolation;
import std.conv;

template interpolate(string code)
{
    string interpolate(Args...)(Args args) {
        string[] stringified;
        foreach(index, arg; args) stringified ~= to!string(arg);

        string result;
        int i;
        int zero = to!int('0');

        while (i < code.length) {
            if (code[i] == '#') {
                int j = 1;
                int index;
                while (i+j < code.length
                    && to!int(code[i+j])-zero >= 0
                    && to!int(code[i+j])-zero <= 9)
                {
                    index = index*10 + to!int(code[i+j])-zero;
                    ++j;
                }

                result ~= stringified[index];
                i += j;
            }
            else {
                result ~= code[i];
```

```
                ++i;
            }
        }

        return result;
    }
}
```

TODO The syntax could be extended somewhat: inserting multiple strings, inserting a range of strings, all arguments to the end.

Example: extending std.functional.binaryFun

Unfinished This one is dear to my heart. Mapping n ranges in parallel is one of the first things that I wanted to do with ranges, for examples to create ranges of structs with constructor taking more than one parameter.

Phobos has two really interesting templates: std.functional.unaryFun and std.functional.binaryFun.

TODO Explain that this aims to extend that to n-args functions.

TODO Augh, the introduction, as of DMD 2.058 of a nice closure syntax more or less alleviate the need for such a construction.

```
bool isaz(char c) {
    return c >= 'a' && c <= 'z';
}

bool isAZ(char c) {
    return c >= 'A' && c <= 'Z';
}

bool isNotLetter(char c) {
    return !isaz(c) && !isAZ(c);
}

int letternum(char c) {
    return to!int(c) - to!int('a') + 1;
}

int arity(string s) {
    if (s.length == 0) return 0;
```

```
    int arity;
    string padded = " " ~ s ~ " ";
    foreach(i, c; padded[0..$-2])
        if (isaz(padded[i+1])
         && isNotLetter(padded[i])
         && isNotLetter(padded[i+2]))
            arity = letternum(padded[i+1]) > arity ?
                    letternum(padded[i+1])
                  : arity;
    return arity;
}

string templateTypes(int arit) {
    if (arit == 0) return "";
    if (arit == 1) return "A";

    string result;
    foreach(i; 0..arit)
        result ~= "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i] ~ ", ";

    return result[0..$-2];
}

string params(int arit) {
    if (arit == 0) return "";
    if (arit == 1) return "A a";

    string result;
    foreach(i; 0..arit)
        result ~= "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i]
               ~ " " ~ "abcdefghijklmnopqrstuvwxyz"[i]
               ~ ", ";

    return result[0..$-2];
}

string naryFunBody(string code, int arit) {
    return interpolate!"auto ref naryFun(#0)(#1) { return #2;}"
                       (templateTypes(arit), params(arit), code);
}

template naryFun(string code, int arit = arity(code))
{
    mixin(naryFunBody(code, arit));
}
```

Sorting Networks

Sorting networks are a nice example of what compile-time code generation can buy you. Sorting is a very vast subject and obtaining near-optimal sort in many circumstances is quite difficult. But in some cases, if you know your input length, you can build a pre-defined list of comparisons between elements and swap them if they are not in the correct order. Here, I will represent sorting networks as pairs of indices in the input range: given indices i and j, compare `input[i]` and `input[j]`, and so on.

For example, given an array or a random-access range of length n, the following table presents possible (optimal, in this case) sorting networks.

n	Sorting network
2	[[0, 1]]
3	[[0, 2], [0, 1], [1, 2]]
4	[[0, 2], [1, 3], [0, 1], [2, 3], [1, 2]]
5	[[0, 4], [0, 2], [1, 3], [2, 4], [0, 1], [2, 3], [1, 4], [1, 2], [3, 4]]

Table 7: the first sorting networks

The code to generate the list of indices can be found in Knuth's *The Art of Computer Programming* or on the net. The code given below I translated from Common Lisp (!) to D, and is taken from Doug Hoyte's Let Over Lambda, a very opinionated book about Common Lisp's macros.

```
module sortingnetwork;

int ceilLog2(int n)
{
    int i;
    if ((n & (n-1)) == 0) i = -1;
    while (n > 0) { ++i; n/= 2;}
    return i;
}

/**
 * Given a length n, returns an array of indices pairs
 * corresponding to a sorting network for length n.
 * Looks a bit like C code, isn't it?
 */
int[2][] sortingNetwork(int n)
```

```
{
    int[2][] network;
    auto t = ceilLog2(n);
    auto p = 1 << (t-1);
    while (p > 0)
    {
        auto q = 1 << (t-1);
        auto r = 0;
        auto d = p;
        while (d > 0)
        {
            for(int i = 0; i<=(n-d-1); ++i)
            {
                if (r == (i & p)) network ~= [i, i+d];
            }
            d = q-p;
            q /= 2;
            r = p;
        }
        p /= 2;
    }
    return network;
}
```

`sortingNetwork` returns an array of indices pairs. From this, it's easy to generate code. The basic building block done by `interpolate` (see Simple String Interpolation):

```
module buildsortingcode;
import stringinterpolation;
import sortingnetwork;

string buildSortingCode(size_t l)()
{
    enum network = sortingNetwork!(l);
    string result;
    foreach(elem; network) result ~=
        interpolate!(
        "t1 = input[#0];
         t2 = input[#1];
         if (!binaryFun!pred(t1, t2))
         {
             auto temp = t2;
             input[#1] = t1;
             input[#0] = temp;
```

```
        }\n")(elem[0], elem[1]);
    return result;
}
```

And from there, I want a template that pre-generates a templated sorting network function. As for std.algorithm.sort, a predicate `pred` determines the way individual range elements are compared.

```
module networksort;
import std.range;
import std.functional;
import std.exception;
import buildsortingcode;
import stringinterpolation;

template networkSort(size_t l)
{
    mixin(
interpolate!(
    "void networkSort(alias pred = \"a < b\", R)(ref R input)
     if (isRandomAccessRange!R)
     {
         enforce(input.length >= #,
                 \"Calling networkSort!# with a range of less than # elements\");
          ElementType!R t1, t2;")(l)
      ~ buildSortingCode!(l)
 ~ "}");
}
```

The strange mixin-in-a-template construction is there to cut the template in two, to allow user code like this:

```
// somewhere in your code
alias networkSort!32 sort32;

// somewhere else
sort32(myArr);
sort32!"a > b"(myArr);
```

`sort32` is designed to work on 32-elements-long random-access ranges, without knowing in advance the nature of the range nor the predicate used to sort. If you call it on ranges with a length greater than 32, it will sort only the first 32 elements. For less than 32 elements, the enforcement will fail.

So, what does that little sorting routine buy us? It appears to be quite efficient for small-size arrays, compared to std.algorithm.sort, but sees its performance degrade after a point. The following table compares one million sorting of n-elements randomly-shuffled arrays done with `networkSort` and `std.algorithm.sort` and gives the ratio of speed-up brought by pre-computing the sorting code.

n	Sorting network (ms)	Standard sort (ms)	ratio
5	324	532	1.642
10	555	1096	1.975
10	803	1679	2.091
20	1154	2314	2.005
25	1538	3244	2.109
30	2173	3508	1.614
35	4075	4120	1.011
40	5918	5269	0.890
45	7479	5959	0.797
50	9179	6435	0.701

Table 8: Comparing `networkSort` and `std.algorithm.sort`

So, at least on this benchmark, `networkSort` outperforms Phobos with a speed-up of up to 100% for ranges between 5 and 40 elements.[12] Of course, Phobos' `sort` is much more versatile, since it works on ranges with a length known only at runtime, but if you know your input's length in advance `networkSort` can be nice to use.

In any cases, the idea to take home away is that, if you've a good idea of what your runtime data will look like, you probably can pre-generate optimal code for it at compile time and then do some triaging at runtime.

`__traits`

The general `__traits` syntax can be found online here. Traits are basically another compile-time introspection tool, complementary to the `is` expression (see the Appendix). Most of time, `__traits` will return `true` or `false` for simple type-introspection questions (is this type or symbol an abstract class, or a final function?). As D is wont to do, these questions are sometimes ones you could

[12] FWIW, on my computer, the cut-off seems to be for about 38-39 elements.

ask using `is` or template constraints, but sometimes not. What's interesting is that you can do some introspection on types, but also on symbols or expressions.

Yes/No Questions with `__traits`

Seeing how this is a document on templates and that we have already seen many introspection tools, here is a quick list of what yes/no questions you can ask which can or *cannot* be tested with `is`:[13]

Question	Doable with other tools?
isArithmetic	Yes
isAssociativeArray	Yes
isFloating	Yes
isIntegral	Yes
isScalar	Yes
isStaticArray	Yes
isUnsigned	Yes
isAbstractClass	No
isFinalClass	No
isVirtualFunction	No
isAbstractFunction	No
isFinalFunction	No
isStaticFunction	No
isRef	No
isOut	No
isLazy	No
hasMember	No (Yes?)
isSame	No
compiles	Yes (in a way)

Table 9: Comparison between `__traits` and other introspection tools

[13] As with any other affirmation in this document, readers should feel free to prove me wrong. That shouldn't be too difficult.

These can all be quite useful in your code, but I'll shy away from them since they are not heavily template-related. More interesting in my opinion is using `__traits` to get new information about a type. These are really different from other introspection tools and I will deal with them in more detail right now.

`identifier`

`identifier` gives you a symbol's name as a string. This is quite interesting, since some symbols are what I'd call *active*: for example, if `foo` is a function, `foo.stringof` will try to first call `foo` and the `.stringof` will fail. Also, `.stringof`, though eminently useful, sometimes returns strangely formatted strings. `identifier` is much more well-behaved.

Let's get back to one of the very first templates in this doc, `nameOf` here. Initially, it was coded like this:

```
template nameOf(alias a)
{
    enum string name = a.stringof; // enum: manifest constant
                                   // determined at compile-time
}
```

But, this fails for functions:

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

auto name = nameOf!foo; // Error, 'foo' must be called with 2 arguments
```

It's much better to use `__traits` (also, the eponymous trick):

```
module nameof;

template nameOf(alias a)
{
    enum string nameOf = __traits(identifier, a);
}

unittest
{
    int foo(int i, int j) { return i+j;}
    enum name = nameOf!foo; // name is available at compile-time

    assert(name == "foo");
}
```

Note that this works for many (all?) kinds of symbols: template names, class names, even modules:

```
import std.typecons;
import nameof;

enum name2 = nameOf!(nameOf); // "nameOf(alias a)"
enum name3 = nameOf!(std.typecons); //"typecons"
```

getMember

In a nutshell, `__traits(getMember, name, "member")` will give you direct access to `name.member`. This is the real member: you can get its value (if any), set it anew, etc. Any D construct with members is OK as a `name`. If you wonder why the aggregate is called directly by its own name whereas the member needs a string, it's because the aggregate is a valid symbol (it exists by itself), when the member's name has no existence outside the aggregate (or even worse, may refer to another, unrelated, construct).

> **Aggregates.** I'll use *aggregate* as a catch-all term for any D construct that has members. Structs and classes are obvious aggregates, as are interfaces, but it's interesting to keep in mind that templates too have members (remember section Instantiating a Template? A template is a named, parameterized scope). So all `__traits` calls shown in this section can be used on templates. That's interesting to keep in mind. Even more interesting, to my eyes, is that *modules* are also aggregates even though they are not first-class citizens in D-land. You'll see examples of this in the following sections.

allMembers

This one is cool. Given an aggregate name, `__traits(allMembers, aggregate)` will return a tuple of string literals, each of which is the name of a member. For a class, the parent classes' members are also included. Built-in properties (like `.sizeof`) are not included in that list. Note that I did say 'tuple': it's a bit more powerful than an array, because it can be iterated over at compile-time.

The names are not repeated for overloads, but see the next section for a way to get the overloads.

```
module usingallmembers1;

class MyClass
{
```

```
    int i;

    this() { i = 0;}
    this(int j) { i = j;}
    ~this() { }

    void foo() { ++i;}
    int foo(int j) { return i+j;}
}

void main()
{
    // Put in an array for a more human-readable printing
    enum myMembers = [__traits(allMembers, MyClass)];

    // See "i" and "foo" in the middle of standard class members
    // "foo" appears only once, despite it being overloaded.
    assert(myMembers == ["i", "__ctor", "__dtor", "foo", "toString",
                         "toHash", "opCmp", "opEquals", "Monitor", "factory"]);
}
```

So the above code is a nice way to get members, both fields (like `i`) and methods (like `foo`). In case you wonder what `"__ctor"` and `"__dtor"` are, it's the internal D name for constructors and destructors. But it's perfectly usable in your code! For structs, the list far less cluttered since they only get the constructor and destructor's names and `opAssign`, the assignment operator (`=`).

Since this trait returns strings, it can be plugged directly into `getMember`. See a bit farther down a way to get a nice list of all members and their types.

Now, let's ramp things up a bit: what about class and struct templates? Let's try this out:

```
module allmemberstemplate;

class MyClass(T)
{
    T field;
}

static assert([__traits(allMembers, MyClass)] == ["MyClass"]);
```

Oh, what happened? Remember from Basics that struct, classes and functions templates are just syntactic sugar for the 'real' syntax:

```
template MyClass(T)
{
```

```
    class MyClass
    {
        T field;
    }
}
```

So, `MyClass` is just the name of the external template, whose one and only member is the `MyClass` class. So all is well. If you instantiate the template, it functions as you might expect:

```
module allmemberstemplate2;
import allmemberstemplate;

static assert([__traits(allMembers, MyClass!int)]
           == ["field", "toString", "toHash", "opCmp",
               "opEquals", "Monitor", "factory"]);
```

If you remember one of the very first uses we saw for templates in Instantiating a Template, that is as a named, parameterized scope, this can give rise to interesting introspection:

```
module templateintrospection;

template Temp(A, B)
{
    A a;
    B foo(A a, B b) { return b;}
    int i;
    alias A    AType;
    alias A[B] AAType;
}

static assert([__traits(allMembers, Temp)]
           == ["a", "foo", "i", "AType", "AAType"]);
static assert([__traits(allMembers, Temp!(double,string))]
           == ["a","foo", "i", "AType", "AAType"]);
```

As you can see, you also get aliases' names. By the way, this is true for structs and templates also.

Another fun fact is that D modules are amenable to `__traits`'s calls, though that's true only for packaged modules (that is, `import pack.mod;` imports the `pack` and `mod` symbols, but `import mod;` imports nothing).

```d
module allmembersmodule;
import std.algorithm;
import std.compiler;

// Huuuge list of names
enum algos = [__traits(allMembers, std.algorithm)];
// A bit shorter
enum compiler = [__traits(allMembers, std.compiler)];

void main()
{
    assert(compiler == ["object", "name",
                        "Vendor", "vendor",
                        "version_major", "version_minor",
                        "D_major", "D_minor"]);
}
```

In the previous code, you see that among `a` members, there is `"object"` (the implicit `object.d` module module} imported by the runtime), and `"std"`, the global package that shows here whenever you import a `std.*` module. It would be easy to imagine a template that recursively explores the members, finds the modules and tries to recurse into them to get a complete import-tree with a template. Alas, `"std"` blocks that, since the package itself does not have a member.[14]

> **introspection.** I'm pretty sure auto-introspection (a module calling `allMembers` on its own name) used to work in Fall 2011. Now it's 2012 and this doesn't work anymore. Hmmm.

Like for the other aggregates, you get the aliased names also, and the unit tests defined in the module.[15] [16]

> **What's the point of inspecting a module?** Well, first that was just for fun and to see if I could duplicate a module or create a struct with an equivalent members list (all forwarding to the module's own members). But the real deal for me was when using string mixins to generate some type. If the user uses the mixin in its own module, it could create conflicts with already-existing names. So I searched for a way for a mixin template to inspect the module it's currently

[14] If someone finds a way to determine with a template that `b` imports `std.algorithm`, I'd be delighted to see how it's done!

[15] For those of you curious about it, they are named `__unittestXXX` where `XXX` is a number. Their type is more or less `void delegate()`.

[16] I didn't try static constructors in modules. Don't hesitate to play with them and tell me what happens.

being instantiated in. Then, I wanted to write a template that, given a class name, would give me the entire hierarchy it's in (as the local module scope would see it, that was enough for me). This `Hierarchy` template should be shown in this document. Then, while testing std.traits.ParameterTypeTuple, I saw that it gives the parameter typetuple of *one* function, even when it's overloaded. So inspecting a module is also a way to get the full list of functions with a particular name and getting the parameter typetuple for each of them.

TODO Inject Hierarchy here. **TODO** Code a more powerful version of ParameterTypeTuple.

`derivedMembers`

Really it's the same as above, except that you do not get a class' parents members, only the class' own members.

`getOverloads`

Given:

- an aggregate name or an instance of that agregrate
- a member name as a string

Then, `__traits(getOverloads, name, "member")` will give you a tuple of all local overloads of `name.member`. By 'local', I mean that for classes, you do not get the parents classes overloads. There is a difference between using `getOverloads` on a type and on an instance: in the former case, you get a tuple of all the overloads. What's cool is that, by using `"__ctor"`, you also get a direct access to a type's constructor overloads. That can be quite handy in some cases.

Getting All Members, Even Overloaded Ones

Now, if you're like me, the urge to mix `allMembers` (which returns the members' names without overloads) and `getOverloads` (which returns the overload of *one* member) is quite great. So let's do that.

First, a bit of machinery: I want the members to be described by a name and a type. Let's create a struct holder, templated of course:

TODO Getting qualified names would be better.

```
module member;

struct Member(string n, T)
{
    enum name = n; // for external access
    alias T Type;
}
```

Given a member name, we'll need a template that delivers the associated `Member`. I also want a way to impose the name, this will come handy later on:

```
module makemember;
public import member;
import nameof;

template MakeMember(alias member)
{
    mixin( "alias Member!(\""
         ~ nameOf!member
         ~ "\", typeof(member)) MakeMember;");
}

template MakeNamedMember(string name)
{
    template MakeNamedMember(alias member)
    {
        mixin( "alias Member!(\""
             ~ name
             ~ "\", typeof(member)) MakeNamedMember;");
    }
}
```

Now, given an aggregate name and a member name (as a string, since these do not exist by themselves), we want a list of `Member` structs holding all the information:

```
module overloads1;
import std.typetuple;
import makemember;

template Overloads(alias a, string member)
{
    alias staticMap!(MakeMember, __traits(getOverloads,a, member))
          Overloads;
}
```

staticMap is explained in section Mapping on Type Tuples.

Now, that already works:

```
module myclass;

class MyClass
{
    int i; // field
    alias i j; // symbol alias

    alias int Int; // type alias

    struct Inner {} // Inner type

    template Temp(T) { alias T Temp;} // template

    this() { i = 0;} // constructor #1
    this(int j) { i = j;} // constructor #2
    ~this() { }

    void foo(int j) { ++i;} // foo overload #1
    int foo(int j, int k = 0) { return i+j;} // foo overload #2

    alias foo bar; // symbol alias

    unittest
    {
        int i;
    }
}

module usingoverloads1;
import std.stdio;
import overloads1;
import myclass;

void main()
{
    alias Overloads!(MyClass, "foo") o;

    /*
    prints:
    foo, of type: void(int j)
    foo, of type: int(int j, int k = 0)
    */
```

```
    foreach(elem; o)
        writeln(elem.name, ", of type: ", elem.Type.stringof);
}
```

We indeed got two `Member` instances, one for each overload. Each `Member` holds the name `"foo"` and the overload type. Except, there is a catch: for a field, there are no overloads. Aliases are also problematic and not correctly dealt with. We need to pay attention to that in `Overloads`:

```
module overloads2;
import std.typetuple;
import makemember;

/**
 * Gets the overloads of a given member, as a Member type tuple.
 */
template Overloads(alias a, string member)
{
    // a.member is a method
    static if (__traits(compiles, __traits(getOverloads, a, member))
        && __traits(getOverloads, a, member).length > 0)
        alias staticMap!(MakeNamedMember!(member), __traits(getOverloads, a, member))
              Overloads;
    else // field or alias
    // a.member is a field, or a symbol alias
        static if (is(typeof(__traits(getMember, a, member))))
            mixin( "alias Member!(\""
                 ~ member
                 ~ "\", typeof(__traits(getMember, a, member))) Overloads;");
    // a.member is a type alias
    else static if (mixin( "is(Member!(\""
                         ~ member
                         ~ "\", __traits(getMember, a, member)))"))
        mixin( "alias Member!(\""
             ~ member
             ~ "\", __traits(getMember, a, member)) Overloads;");
    // a.member is template
    else
        mixin( "alias Member!(\""
             ~ member
             ~ "\", void) Overloads;");
}
```

The entire template may be a bit daunting, but it's to give it a (mostly) correct way to deal with the many kinds of member an aggregate may have: fields,

methods, type aliases, symbols aliases, template names. As of this writing, it does not deal correctly with inner types: I think it should give them a `Outer.Inner` type, whereas here it produces only `Inner`.[17] Also, unittest blocks appear, but they are given a `void` type, the default case in this template. I think they should have a `void()` type.

The last step is to get this for all members of a given aggregate. It's quite easy:

```
1  module allmembers;
2  import overloads2;
3  import std.typetuple;
4
5  template GetOverloads(alias a)
6  {
7      template GetOverloads(string member)
8      {
9          alias Overloads!(a, member) GetOverloads;
10     }
11 }
12
13 template AllMembers(alias a)
14 {
15     alias staticMap!(GetOverloads!(a), __traits(allMembers, a)) AllMembers;
16 }
```

The strange `GetOverloads` two-stage construction is just a way to map it more easily on line 15. So, this was quite long to explain, but it works nicely:

```
module usingallmembers2;
import std.typetuple;
import std.stdio;
import myclass;
import allmembers;

void main()
{
    alias AllMembers!(MyClass) O;
    writeln(O.stringof);

    foreach(o; O)
        writeln(o.name, ", of type: " ~ o.Type.stringof);

    /*
    prints:
```

[17] Maybe by using the `qualifiedName` template shown in section parent.

```
    i, of type: int
    j, of type: int
    Int, of type: int
    (...)
    __ctor, of type: MyClass()
    __ctor, of type: MyClass(int j)
    (...)
    foo, of type: void(int j)
    foo, of type: int(int j, int k = 0)
    (...)
    */
}
```

That's cool, every member and overload is accounted for: the two contructors are there, the destructor also, and of course `i`, of type `int`.

> **TODO** Example: store all members in a hashtable or a polymorphic association list. As a mixin, to be put inside types to enable runtime reflection? `(a.send("someMethod", args), a.setInstanceVariable("i",5))`

Testing for Interface Implementation

The previous section gave us a way to get an aggregate member list. From there, it's an easy step to get an interface's member list and to see if a given symbol's members contains the entire interface member list:

```
module implements;
import std.typetuple;
import allmembers;

/**
 * Statically check if symbol 'a' implements interface I
 * (that is, if all members of I are found in members of a.
 */
template implements(alias a, I) if (is(I == interface))
{
    alias implementsImpl!(a, AllMembers!I) implements;
}

template implementsImpl(alias a, Items...)
{
    static if (Items.length == 0)
        enum implementsImpl = true;
    else static if (staticIndexOf!(Items[0], AllMembers!a) == -1)
```

```
        enum implementsImpl = false;
    else
        enum implementsImpl = implementsImpl!(a, Items[1..$]);
}

interface I
{
    int foo(int i);
    void foo();

    string toString();
}

class Bad
{
    void foo(int i) {}
}

struct Good
{
    int field;

    int foo(int i) { return i;}
    void foo() { field = 1;}

    string toString() { return "I'm a good struct!";}
}

unittest
{
    assert( implements!(Good, I));
    assert(!implements!(Bad, I));
}
```

`getVirtualFunctions`

It's in the same family than `getOverloads` and such. It'll give you the list of virtual overloads for a class method. Given a class name, finding all overloads of all fields, even overriden ones, is let as a exercise to the reader.

parent

`__traits(parent, symbol)` will return the symbol that's the parent of it. It's *not* the parent in a 'class hierarchy' sense,[18] it deals with qualified names and strip one level. Once you reach the toplevel scope, it returns the module name (this can be dangerous, because modules themselves do not have parents). See:

```
module cclass;
import nameof;

class C
{
    int i;
    int foo(int j)
    {
        int k; // k is "cclass.C.foo.k"
        assert(nameOf!(__traits(parent, k)) == "foo");
        return i+k;
    }
}

module parent;
import nameof;
import cclass;

// C is more precisely a cclass.C)
static assert(nameOf!(__traits(parent, C)) == "cclass");

void main()
{
    auto c = new C(); // c is "parent.main.c"
    assert(nameOf!(__traits(parent, c)) == "main");
    assert(nameOf!(__traits(parent, c.i)) == "C");
    c.foo(1);
}
```

Even if there is no `qualifiedIdentifier` in `__traits`, we can construct a template to get it:

```
module qualifiedname;
import nameof;

template qualifiedName(alias a)
```

[18] If you came here to see a way to get all parents of a class, see section Class Hierarchy.

```
{
    // does it have a parent?
    static if (__traits(compiles, __traits(parent, a)))
    // If yes, get the name and recurse
        enum qualifiedName = qualifiedName!(__traits(parent, a))
                           ~  "." ~ nameOf!(a);
    // if not, it's a module name. Stop there.
    else
        enum qualifiedName = nameOf!a;
}

module usingqualifiedname;
import qualifiedname;
import cclass;

void main()
{
    auto c = new C();
    assert(qualifiedName!c == "usingqualifiedname.main.c");

    assert(qualifiedName!(c.foo) == "cclass.C.foo"); // same in both cases
    assert(qualifiedName!(C.foo) == "cclass.C.foo");
}
```

Note that `c.foo` is *not* qualified as `usingqualifiedname.main.c.foo`.

Local Scope Name

Sometimes, when dealing with mixin templates or string mixins, you'll inject code in an unknown scope. To get your way round, it can be useful to get the local scope's name. Intuitively, the previous example could help with that: just create a local variable, get the qualified name of its parent to determine in which scope the mixin is. Then, expose the scope name. Let's call this one `scopeName` and the associated inquisitive template `getScopeName`.

```
module scopename;
public import qualifiedname;

mixin template getScopeName()
{
    enum scopeName = qualifiedName!(__traits(parent, scopeName));
}
```

The idea is to declare local enum called `scopeName` and take the qualified name of its own parent in the same expression (yes, that works!).

To use `getScopeName`, just mix it in where you need a local scope name:

BUG: Something changed since I last tested this code (2012). As of now (DMD 2.066, August 2014), this does not work anymore. I'll add it to my TODO list.

```
module usingscopename;
import std.stdio;
import scopename;

class C
{
    mixin getScopeName; // 1

    int i;
    int foo(int j)
    {
        int k;
        mixin getScopeName; // 2
        writeln(scopeName);
        return i+k;
    }
}

void main()
{
    auto c = new C();
    writeln(c.scopeName); // "test.C" (1)
    c.foo(1);             // "test.C.foo" (2)

    mixin getScopeName; // 3
    writeln(scopeName); // "test.main" (3)
}
```

Wrapping it all Together

> **Unfinished** I have to dig some cute examples in my code base. Using string mixins and CTFE and templates all together.

Examples

This part will present various examples showing what can be done with D templates, be it type manipulation, code generation or language extension Most examples are code snippets that were useful to me at one time or another. I hope they will be useful to you too.

Contributors welcome! Even more than for other parts, I welcome any short and synthetic example of what can be done with templates. Do not hesitate to chime in and push some code in this doc!

Type Sorcery

One of the most fundamental use of templates is type sorcery: type creation, type manipulation, etc. D being a statically type language, all of your creations will have a defined type. Sometimes, these can be cumbersome to write or manipulate. Templates can help you in this.

Mapping, Filtering and Folding Types

As we saw in section Template Tuple Parameters, template tuple parameters can hold type tuples (that's even their original role). Since these can be iterated, indexed or sliced, they are ideal candidates to some standard iteration algorithms. As for ranges, you can map another template on type tuples, filter the types you want to extract or fold (reduce) them into another type.

And non-types? What about acting on expression tuples? You can do that too. Even though this section is called *Type* Sorcery, all templates here can work on expression tuples too. Do not feel limited.

Mapping on Type Tuples Mapping a type tuple is simply applying another (unary) template on all types of a type tuple. Phobos already defines `staticMap` in std.typetuple, but it's a nice exercise to code it again. We want a template that takes another template name (as an `alias` parameter), say `Wrapper`, and a typetuple (`T0, T1, T2, ..., Tn`) and returns `Wrapper!T0, Wrapper!T1, Wrapper!T2, ..., Wrapper!Tn`.

```
module staticmap;
import std.typetuple;

template staticMap(alias M, T...)
{
    static if (T.length == 0) // End of sequence
        alias TypeTuple!() staticMap; // stop there
    else
        alias TypeTuple!(M!(T[0]), staticMap!(M, T[1..$])) staticMap;
}
```

We use the auto-flattening of type tuples to aggregate the results into a unique tuple. Notice how indexing and slicing make for a not-so-complicated piece of code. As it's already defined in std.typetuple, I'll use Phobos' template from now on.

Even a simple template such as this one can have great uses:

- Getting rid of all qualifiers in a type list, by mapping std.traits.Unqual on the types.
- Generating a huge quantity of types by using a typetuple-returning mapper (see below).
- Given a bunch of function types, getting their return types or parameter typetuples.

Example: Testing a Function The seconde use in the previous section's list can be useful to unit-test a part of your code. Suppose you have a templated function that is supposed to work on any built-in type. You do not need to generate all possible combinations of types. Just use `staticMap` to generate them for you:

```
module qualified;
import std.typetuple;

/**
* Given a type T, generates all qualified versions of T
* that you find interesting (eleven versions all in all).
*/
template Qualified(T)
{
    alias TypeTuple!(
                     T, const(T), immutable(T), shared(T),
                     T[],const(T)[],immutable(T)[], shared(T)[],
                     const(T[]), immutable(T[]), shared(T[])
                    ) Qualified;
}

// All 16 built-in types you're interested in.
alias TypeTuple!(
                 bool,
                 ubyte,byte,
                 ushort,short,
                 uint,int,
                 ulong,long,
                 float,double,real,
                 char,wchar,dchar
```

```
                ) ValueTypes;

// Bang, 11*16 types generated.
alias staticMap!(Qualified,ValueTypes) QualifiedTypes;

// If you're really a pervert (note that there will be duplicates)
alias staticMap!(Qualified, QualifiedTypes) DoublyQualifiedTypes;
```

Now, if your function is supposed to work on all the generated qualified types, just test it:

```
module tester;
import qualified;

void myFun(T)(T t) {}

template test(alias fun)
{
    void on(T...)()
    {
        foreach(Type; T)
            static if (!__traits(compiles, fun(Type.init)))
                pragma(msg, "Bad testing combination: "
                          ~ fun.stringof ~ " and " ~ Type.stringof);
    }
}

unittest
{
    test!(myFun).on!(QualifiedTypes);
}
```

Filtering Type Tuples You can search for and extract some types from a tuple, using a predicate to chose which type (or more generally, which tuple element) you want to keep. A predicate in this particular case means 'a template that, when given a type as argument, will return either `true` or `false`'. The test done on the tuple element can be as complicated as you want, particularly using `is()` expressions (see here).

That gives us the following code:

```
module staticfilter;
import std.typetuple;

template staticFilter(alias Pred, T...)
```

```
{
    static if (T.length == 0) // End of sequence
        alias TypeTuple!() staticFilter;
    else static if (Pred!(T[0]))
        alias TypeTuple!(T[0], staticFilter!(Pred, T[1..$])) staticFilter;
    else
        alias TypeTuple!(      staticFilter!(Pred, T[1..$])) staticFilter;
}
```

Using `staticFilter` is quite simple. Let's get integral types from a tuple, by way of std.traits.isIntegral:

```
module usingstaticfilter1;
import std.typetuple;
import std.traits;
import staticfilter;

alias TypeTuple!(int, double, float, string, byte, bool, float, void) Types;

alias staticFilter!(isIntegral, Types) OnlyIntegrals;

static assert(is(OnlyIntegrals == TypeTuple!(int, byte)));
```

What about separating types from non-types? First, let's create a template that is `true` for pure types and `false` on non-types:

```
module istype;

template isType(T)
{
    enum isType = true;
}

template isType(alias a)
{
    static if (is(a))
        enum isType = true;
    else
        enum isType = false;
}
```

Hey, wait! OK with having two specialised templates, one on template type parameters and another on aliases. But why the `static if`? It's because user-defined types (`MyClass` and such) are *both* a type and a symbol (we saw that

in section Template Declarations). For this particular use, I want them to be considered as types, contrary to other symbols (function names, module names, variables, ...). Hence the `is()` test. We can change things a bit to get a built-in types detector, which may also be interesting. Built-in types *cannot* be used as alias parameters, so let's test for that:

```
module isbuiltintype;

template isSymbol(alias a)
{
    enum isSymbol = true;
}

template isBuiltinType(T)
{
    static if (__traits(compiles, isSymbol!(T)))
        enum isBuiltinType = false;
    else
        enum isBuiltinType = true;
}

template isBuiltinType(alias a)
{
    enum isBuiltinType = false;
}
```

And, here we go:

```
module usingstaticfilter2;
import std.typetuple;
import staticfilter;
import istype;
import isbuiltintype;

class MyClass {}
int foo(int i) { return i;}

alias staticFilter!(isType, 1, int, 3.14, "abc", foo, MyClass) Types;
alias staticFilter!(isBuiltinType, 1, int, 3.14, "abc", foo, MyClass) Builtins;

static assert(is(Types == TypeTuple!(int, MyClass)));
static assert(is(Builtins == TypeTuple!(int)));
```

But that is, admittedly, pretty run-of-the-mill stuff. Though useful from time to time, it's quite rare for someone to be given a pure type tuple like this. A much more common use for the likes of `staticFilter` is when creating complex types.

Example: building a `Graph`

As a first example, imagine you have a `Graph(Node, Edge)` struct, templated on the nodes (vertice) and edges types (themselves templated). When you create a `Graph` with a factory function, it would be nice to be able to mix nodes and edges in a natural way. That is, given `graph`, `node` and `edge` functions that do the obvious thing, you want to autorize calls like:

```
/**
* Automatically creates a graph
*   - with four nodes labelled "A", "B", "C", "D", holding a double,
*   - with nodes linking "A" to "B", "D" to "A" and "D" to "C".
*/
auto g = graph(node("A", 3.14159), node("B", 1.0),
               edge("A","B"),
               node("C", 2.71828), node("D", 0.0),
               edge("D","A"), edge("D", "C"));
```

This allows the user building her `Graph` to create nodes and edges between these nodes in a natural way (as opposed to, say, batch-building all nodes and then adding edges between them). But, as a library writer, that means your `graph` factory function has the following signature:

```
auto graph(NodesOrEdges...)(NodesOrEdges args)
if (/* sanity check test on NodesOrEdges */)
```

Both the sanity check performed by the template constraint (Constraints) and the building code inside `graph` can be quite complicated. `staticFilter` helps by separating the arguments between nodes and edges. Without extending this example too much, say we have at our disposal the following predicate templates:

```
template isNode(N) {/* true iff N is a Node!(LabelType, ValueType)*/}
template isEdge(E) {/* true iff E is an Edge!(LabelType)*/}

template isNodeOrEdge(T)
{
    static if (isNode!T || isEdge!T)
        enum isNodeOrEdge = true;
    else
        enum isNodeOrEdge = false;
}
```

And let's suppose also all *bona fide* `Node`s and `Edge`s have `.LabelType` and `.ValueType` members exposing their inner types (as shown in Inner Alias).

Then, getting all nodes and edges is easy:

```
alias staticFilter!(isNode, NodesOrEdges) Nodes;
alias staticFilter!(isEdge, NodesOrEdges) Edges;
```

This is where things get interesting: obtaining the edges and nodes types is just a first building block. Now `graph` must check at a minimum the following elements:

- All arguments must be nodes or edges.
- Is there at least *one* node in the list?
- If yes, do all nodes have the same `LabelType` and the same `ValueType`, or, at least, is there a common type between all the labels' types and another one for the values stored in the nodes?
- Do edges `LabelTypes` have a common type? (Note that there can be zero edges).
- Do the edges labels have the correct type to refer to the provided nodes?

Note that *all* these checks are done only on types and thus can be done at compile-time, thereby ensuring a pretty solid static check on the graph built. What cannot be done in this way is verifying during compilation that edges do indeed refer to existing nodes.

Let's use what we have seen until now to create the `GraphCheck` template, before seeing another `staticFilter` example:

```
import std.traits: CommonType;

template GraphCheck(NodesOrEdges...)
{
    enum GraphCheck = GraphCheckImpl!(NodesOrEdges).result;
}

template GraphCheckImpl(NodesOrEdges...)
{
    alias staticFilter!(isNode, NodesOrEdges) Nodes;
    alias staticFilter!(isEdge, NodesOrEdges) Edges;

    // 1. All arguments must be nodes or edges
    static assert (Nodes.length + Edges.length != NodesOrEdges.length,
                   "Some args are not nodes or edges.");

    // 2. There must be at least one node
    static assert (Nodes.length == 0,
                   "You must provide at least one node.");

    // 3. Is there a common type for the nodes' labels and values?
```

```
    // First step: extracting labels and values
    template GetLabel(T) if (isNode!T || isEdge!T)
    {
        alias T.LabelType GetLabel;
    }

    template GetValue(T) if (isNode!T)
    {
        alias T.ValueType GetValue;
    }

    alias staticMap!(GetLabel, Nodes) NodesLabels;
    alias staticMap!(GetValue, Nodes) NodesValues;

    static assert (is(CommonType!(NodesLabels) == void), // no common type
                   "The nodes do not have all the same label type.");

    static assert (is(CommonType!(NodesValues) == void),
                   "The nodes do not have all the same value type.");

    // 4. Same for edges
    alias staticMap!(GetLabel, Edges) EdgesLabels;

    static assert (is(CommonType!(EdgesLabels) == void),
                   "The edges do not have all the same label type.");

    // 5. Edges - Node compatibility
    static assert(!is(CommonType!NodesLabels == CommonType!EdgesLabels),
                  "Nodes and edges do not have the same label type.");

    enum result = true;
}
```

This is one huge template, but `staticFilter` sits square in the middle and greatly simplifies the code. Note the use of `static assert`, slightly different from Now, `graph` signature is simply:

```
auto graph(NodesOrEdges...)(NodesOrEdges args) if (GraphCheck!NodesOrEdges)
{ ... }
```

> **Unfinished** The second example will be one code generating a struct, with string literal and type parameters.

Folding Type Tuples With mapping and filtering, folding (aka, reducing) is the third standard operation on sequences.[19] The idea is the same as std.algorithm.reduce: given a seed `S` and a binary function `bin`, calculate `bin(bin(bin(S, T[0]),T[1],T[2],...))`: apply `bin` on the seed and the first type of the tuple, then take the resulting type as a new seed and re-apply `bin` to this and the second type of the tuple, and so on, until the entire tuple is used.

So, what's the use of such a function? It's used to *collapse* a type tuple into one type. This one type can a simple type (for example, the 'biggest' type in the tuple, for some definition of big) or a complex structure built iteratively step by step along the tuple: a binary tree holding all the types, for example, or the reverse of the tuple, or even all the types but sorted according to a predicate.

Here, we will see two examples: getting the maximum type and sorting a type tuple.

But first, here is `staticReduce`:

```
module staticreduce;

template staticReduce(alias bin, Types...) // Types[0] is the seed
{
    static if (Types.length < 2)
        static assert(0, "staticReduce: tuple "
               ~ Types.stringof ~ " has not enough elements (min: 2 elements)");
    else static if (Types.length == 2) // end of recursion
        alias bin!(Types[0], Types[1]) staticReduce;
    else // recurse
        alias staticReduce!(bin, bin!(Types[0], Types[1])
                                , Types[2..$])
              staticReduce;
}
```

From here, how do we get the biggest type? Simple, just apply a `Max` binary template to your type list:

```
module maxtemplate;

template Max(T1, T2)
{
    static if (T1.sizeof >= T2.sizeof)
        alias T1 Max;
    else
        alias T2 Max;
}
```

[19] In fact, it's the mother of all operations on sequences, since map and filter can be defined using reduce.

```
module usingmax;
import std.typetuple;
import maxtemplate;
import staticreduce;

alias TypeTuple!(int, bool, double, float delegate(float), string[]) Types;

alias staticReduce!(Max, Types) MaxType;
static assert(is(MaxType == float delegate(float)));
```

You can vary your definition of `Max` according to taste. Here I used the built-in `.sizeof` property to compare two unknown types. To compare on the names, I'd have used `.stringof` and so on.

Sorting Types

> **Ivory Tower Wankery!** I mean, when would sorting types be useful? It can be useful, read on

When can sorting a tuple be useful? Mainly for the same reason you'd want to sort any sequence:

- easily find a type (in $\log(n)$),
- easily get afterwards the first p types or last p types,
- compare two type tuples for equivalence,
- easily getting rid of duplicates (transforming a tuple into a set of types).

I'll focus on the third use, comparing two type tuples. See for example the std.variant.Algebraic template struct. `Algebraic!(Type0, Type1, ... TypeN)` can hold a value of one of `Type0`, `TypeN`. But of course, in a certain sense, the previous type is also the same as `Algebraic!(Type1, TypeN, ... Type0)` or any other reordering of the types. But that's not the case currently:

```
module variant_error;
import std.variant;

void main()
{
    Algebraic!(int, double, string) algOne;
    algOne = 1;
    Algebraic!(double, int, string) algTwo = algOne; // fail!
}
```

Using sorted types internally (or even using a factory function to create algebraic values, that always returned sorted `Algebraic`) would allow for a seamless use of algebraic values.

Here is one way to sort types using `staticReduce`. The standard situation is the following: you have a list of already sorted types (the initial tuple's first n types) which is your current state, the value being constructed. `staticReduce` takes this list and puts the first remaining unsorted type in it and then reiterates on the next unsorted type. So the basic step is to add a new type to a sorted list.

A small problem arises: the state is just one type, but it has to store all the sorted types. It cannot be a naked type tuple, since these auto-flatten (see Tuple Properties). We will use std.typecons.Tuple to wrap it. Inner types of a `Tuple` are accessible through the `.Types` alias. I'm afraid that will uglify the code a bit.

Finally, what predicate to use? `.sizeof` is not adequate: many different types have the same size and that would have a bad consequence, as the initial order would influence the sorted order. I'll just use the stringified version of the types, obtained by the built-in `.stringof` property:

```
module sorttypes;
import std.typecons;
import staticreduce;

template Max(T1, T2)
{
    static if (T1.stringof >= T2.stringof)
        alias T1 Max;
    else
        alias T2 Max;
}

template AddToSorted(Sorted, Type)
{
// Length 0: already sorted
    static if (Sorted.Types.length == 0)
        alias Tuple!(Type) AddToSorted;
// Smaller than the first one: put Type in first place
    else static if (is(Max!(Sorted.Types[0], Type) == Sorted.Types[0]))
        alias Tuple!(Type, Sorted.Types) AddToSorted;
// Bigger than the last one: put Type at the end
    else static if (is(Max!(Sorted.Types[$-1], Type) == Type))
        alias Tuple!(Sorted.Types, Type) AddToSorted;
// Else, compare to the middle type and recurse left or right of it
    else static if (is(Max!(Sorted.Types[$/2], Type) == Type))
        alias Tuple!(Sorted.Types[0..$/2],
```

```
                    AddToSorted!(Tuple!(Sorted.Types[$/2..$]),Type).Types)
                AddToSorted;
    else
        alias Tuple!(AddToSorted!(Tuple!(Sorted.Types[0..$/2]),Type).Types,
                    Sorted.Types[$/2..$])
                AddToSorted;
}

template Sort(Types...)
{
    alias staticReduce!(AddToSorted, Tuple!(), Types) Sort;
}
```

As I said, in the end `Sort` is just applying `AddToSorted` to the target tuple. The initial (seed) state for `staticReduce` is an empty typetuple, `Tuple!()`.

Now, does that work? You bet:

```
module sortingtypes;
import std.typetuple;
import sorttypes;

alias TypeTuple!(int, bool, string function(int), float[]) Types1;
alias TypeTuple!(int, float[], string function(int), bool) Types2;

static assert(is(Sort!Types1 == Sort!Types2));
```

If you do not like sorting types alphabetically by name, you can resort to other definitions of `Max` or, even better, make a version of `Sort` that accepts another, binary, template as an argument and uses this template as way to determine ordering.

> **What about non-types?** As was said at the very beginning of the section, the templates presented here work most of the time for other template parameters: pure numbers, strings, aliases, Here you'd have to change `AddToSorted` second parameter to accept non-types. Or, another way to do it would be to first map a stringifier on the tuple's elements and *then* sort the resulting strings.

Scanning Types, Interleaving Types, Crossing Types

> **TODO** Determine if this subsection is really useful. Is there any short and telling example?

Static Scan A *scan* is a sort of `reduce`/`fold` operation (see Folding Type Tuples), but giving the intermediary results. It's used by the `juxtapose` template in Juxtaposing Functions.

```
module staticscan;
import std.typetuple;

/**
Gives the TypeTuple resulting from the successive applications of F to reduce
the T list.
*/
template StaticScan(alias F, T...)
{
    static if (T.length == 0)
        alias TypeTuple!() StaticScan; // This case should never happen with normal use
    static if (T.length == 1)
        alias TypeTuple!(T[0]) StaticScan;
    else
        alias TypeTuple!(T[0], StaticScan!(F, F!(T[0], T[1]), T[2..$])) StaticScan;
}
```

Interleaving Types

```
module interleave;
import std.typetuple;

/**
 * Given (T0, T1, T2, ..., Tn) and (U0, U1, ..., Um) will returns
 * the interleaving of the first part with the second part:
 *
 * (T0, U0, T1, U1, ...
 *
 * If one of the inputs is shorter than the other,
 * the longer part is put at the end of the interleaving.
 */
template Interleave(First...)
{
    template With(Second...)
    {
        static if (First.length == 0)
            alias Second With;
        else static if (Second.length == 0)
            alias First With;
        else
            alias TypeTuple!( First[0], Second[0]
```

```
                                , Interleave!(First[1..$]).With!(Second[1..$]))
                    With;
    }
}
```

Annotating Types

> **Unfinished** The idea is to wrap values in a template adding some meta data. Ideally, I'd like to get things like the following code to work:

```
auto arr = [0,1,2,3,4]; // Array of ints
auto arr2 = sorted(arr); // Now, we know it's sorted
auto arr3 = positive(arr2); // Sorted *and* all elements are positive
```

Or, more generally:

```
auto arr = [0,1,2,3,4]; // Array of ints
auto arr2 = annotate!("sorted", (a,b) => a<b)(arr);
auto arr3 = annotate!("positive")(arr2);

assert("positive" in arr3.properties);
assert(arr3.Properties == TypeTuple!( Property!("sorted", (a,b) => a < b)
                                    , Property!("positive")));

// the wrapped value is still there:
auto arr4 = array(filter!((a) => a%2==0))(arr3);
// getting rid of some properties
auto arr5 = arr3.discardProperty!"positive";
assert(arr5.Properties == TypeTuple!(Property!("sorted", (a,b) => a < b)));

auto arr6 = annotate!("negative")([-4, -3, -2, -1]);
auto arr7 = annotate!("sorted", (a,b) => a<b)(arr6);

assert(arr3.property!"sorted" == arr7.property!"sorted"); // same predicate
```

Here is a first *very rough and unfinished* draft:

```
module annotation;
import std.typetuple;

struct Meta(string Name, alias Data)
{
    enum name = Name;
```

```
    alias Data data;
}

template isMeta(T)
{
    static if (__traits(hasMember, T, "name")
            && __traits(hasMember, T, "data"))
        enum isMeta = true;
    else
        enum isMeta = false;
}

template GetName(alias a)
{
    enum string GetName = a.name;
}

template isAnnotated(T)
{
    static if (__traits(compiles, T.Annotations))
        enum bool isAnnotated = true;
    else
        enum bool isAnnotated = false;
}

string getNames(Metadata...)() @property
{
    alias staticMap!(GetName, Metadata) names;
    string result;
    foreach(name; names)
        result ~= "\""~name~"\",";
    if (names.length) result = result[0..$-1];
    return "alias TypeTuple!(" ~ result ~ ") names;";
}

struct Annotated(T, Metadata...)
if (allSatisfy!(isMeta, Metadata))
{
    T value;
    alias value this;
    mixin(getNames!(Metadata));
    Metadata metadata;

    auto property(string s)() @property
    {
        static if (staticIndexOf!(s, names) != -1)
```

```
            return metadata[staticIndexOf!(s, names)];
        else
            static assert(false, "Unknown property: " ~ s);
    }

    bool hasAnnotation(string name) @property
    {
        foreach(n; names)
            if (name == n) return true;
        return false;
    }
}

// auto annotated(T)(T value)
// {
//     return Annotated!(T)(value);
// }

template annotated(Metadata...) if (Metadata.length)
{
    auto annotated(T)(T value)
    {
    //     alias TypeTuple!(Metadata) MetaTypes;
        static if (isAnnotated!(T))
            return Annotated!(T.AnnotatedType, T.Annotations, Metadata)(value.value);
        else
        {
            Annotated!(T, Metadata) a;
            a.value = value;
            return a;
        }
    }
}
```

Tuples as Sequences

Unfinished The idea here is to treat tuples of values (is in `tuple(1, "abc", 3.14, 3, 0)` as sequences: map a (polymorphic, aka templated) function on them, filter them, etc. Why do that: imagine for example having a bunch of ranges, all of a different type. If you want to group them in a range and process them, you can't because a range must be homogeneous in type. The functions presented there lift that restriction.

Mapping on Tuples

```
module maptuple;
import std.typecons;
import std.typetuple;
import std.functional;

/**
 * Helper template to get a template function return type.
 */
template RT(alias fun)
{
    template RT(T)
    {
        alias typeof(fun(T.init)) RT;
    }
}

/// Maps on a tuple, using a polymorphic function. Produces another tuple.
Tuple!(staticMap!(RT!fun, T)) mapTuple(alias fun, T...)(Tuple!T tup)
{
    StaticMap!(RT!fun, T) res;
    foreach(i, Type; T) res[i] = unaryFun!fun(tup.field[i]);
    return tuple(res);
}
```

Filtering Tuples

> **Unfinished** The idea here is to filter a tuple with a predicate acting on types. It's quite useful when you can get a bunch of parameters in a function and want only some of them. See Filtering Type Tuples to see an example with the `graph` function.

```
module filtertuple;
import std.typecons;
import std.typetuple;

template FilterTupleTypes(alias pred, alias tup)
{
    static if (tup.field.length)
    {
        static if (pred(tup.field[0]))
            alias TypeTuple!(tup.Types[0], FilterTupleTypes!(pred, tuple(tup.expand[1..$])))
                  FilterTupleTypes;
        else
```

```
            alias FilterTupleTypes!(pred, tuple(tup.expand[1..$]))
                  FilterTupleTypes;
    }
    else
    {
        alias TypeTuple!() FilterTupleTypes;
    }

}

template FilterTupleIndices(alias pred, alias tup, size_t ind)
{
    static if (tup.field.length)
    {
        static if (pred(tup.field[0]))
            alias TypeTuple!( ind
                            , FilterTupleIndices!( pred
                                                 , tuple(tup.expand[1..$])
                                                 , ind+1
                                                 )
                            ) FilterTupleIndices;
        else
            alias FilterTupleIndices!( pred
                                     , tuple(tup.expand[1..$])
                                     , ind+1
                                     ) FilterTupleIndices;
    }
    else
    {
        alias TypeTuple!() FilterTupleIndices;
    }

}

/// Filter a tuple on its values.
Tuple!(FilterTupleTypes!(pred, tup)) filterTuple(alias pred, alias tup)()
{
    FilterTupleTypes!(pred, tup) result;
    alias FilterTupleIndices!(pred, tup, 0) indices;
    foreach(i, ind; indices)
    {
        result[i] = tup.field[ind];
    }
    return tuple(result);
}
```

```
(1, "abc", 2, "def", 3.14)
->
((1,2),("abc","def"),(3,14))
```

Fun With Functions

This section will present some templates acting on functions or templated wrappers around functions, to expand them. It's not part of Function-Templates because it uses struct templates and it's not part of Struct Templates because the wrapper struct is not the main focus of attention.

Determining a Function's Number of Arguments

> **Unfinished** This is old code, see if anything changed about that in the past two years.

```
module functionarity;
import std.traits;

template arity(alias fun)
if (isFunction!fun)
{
    enum size_t arity = ParameterTypeTuple!(fun).length;
}
```

Memoizing a Function

When a function does long calculations, it might be efficient to store the computed results in an external structure and to query this structure for the result instead of calling the function again. This is called *memoizing* (not *memorizing*) and this section will show how to use a template to have some memoizing fun.

The previously-seen results are stored in an associative array, indexed on tuples of arguments. To get a function return type or parameter type tuple, just use Phobos' std.traits.ReturnType and std.traits.ParameterTypeTuple, which are templates that accept function *names* or types.

```
module memoize1;
import std.traits;
import std.typecons;

struct Memoize(alias fun)
{
```

```
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;
    RT[Tuple!(PTT)] memo; // stores the result, indexed by arguments.

    RT opCall(PTT args)
    {
        if (tuple(args) in memo)      // Have we already seen these args?
        {
            return memo[tuple(args)]; // if yes, use the stored result
        }
        else // if not, compute the result and store it.
        {
            RT result = fun(args);
            memo[tuple(args)] = result;
            return result;
        }
    }
}

Memoize!fun memoize(alias fun)()
{
    Memoize!fun memo;
    return memo;
}
```

Usage is very simple:

```
module usingmemoize1;
import memoize1;

int veryLongCalc(int i, double d, string s)
{
    /* ... cheating here ... */
    return i;
}

void main()
{
    auto vlcMemo = memoize!(veryLongCalc);

   // calculate veryLongCalc(1, 3.14, "abc")
   // takes minutes!
   int res1 = vlcMemo(1, 3.14, "abc");
   int res2 = vlcMemo(2, 2.718, "def");// minutes again!
   int res3 = vlcMemo(1, 3.14, "abc"); // a few ms to get res3
}
```

The above code is trivial and could be optimized in many ways. Mostly, a real memoizing template should also modify its behavior with storing policies. For example:

- No-limit or limited size store?
- In case of limited-size store: how to define the limit and what should be the eviction policy?
- First-in/First-out memo?
- Least recenly used memo?
- Least used?
- Time-to-live?
- Discard all and flush the store?
- Discard only a fraction?
- Stop memoizing?

The last X results could be stored in a queue: each time a result is pushed into the associative array, push the arguments tuples in the queue. Once you reach the maximum store limit, discard the oldest one or (for example) half the stored values.

Here is a possible small implementation. It makes for a nice example of enabling/disabling code with `static if` and `enum`-based policies. Note that I use D dynamic arrays as a primitive queue. A real queue could probably be more efficient, but there isn't one in the standard library as of this writing.

```
module memoize2;
import std.traits;
import std.typecons;

enum Storing {
    always,  // there is no tomorrow
    maximum  // sustainable growth
}

enum Discarding {
    oldest,   // only discard the oldest result
    fraction, // discard a fraction (0.5 == 50%)
    all       // burn, burn!
}
```

```
struct Memoize(alias fun,
               Storing storing,
               Discarding discarding)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;

    static if (storing == Storing.maximum)
    {
        Tuple!(PTT)[] argsQueue;
        size_t maxNumStored;
    }

    static if (discarding == Discarding.fraction)
        float fraction;

    RT[Tuple!(PTT)] memo; // stores the result, indexed by arguments.

    RT opCall(PTT args)
    {
        if (tuple(args) in memo)      // Have we already seen these args?
        {
            return memo[tuple(args)]; // if yes, use the stored result
        }
        else                          // if not,
        {
            static if (storing == Storing.always)
            {
                RT result = fun(args);// compute the result and store it.
                memo[tuple(args)] = result;
                return result;
            }
            else // Storing.maximum
            {
                if (argsQueue.length >= maxNumStored)
                {
                    static if (discarding == Discarding.oldest)
                    {
                        memo.remove(argsQueue[0]);
                        argsQueue = argsQueue[1..$];
                    }
                    else static if (discarding == Discarding.fraction)
                    {
                        auto num = to!size_t(argsQueue.length * fraction);
                        foreach(elem; argsQueue[0..num])
                            memo.remove(elem);
```

```
                    argsQueue = argsQueue[num..$];
                }
                else static if (discarding == Discarding.all)
                {
                    memo = null;
                    argsQueue.length = 0;
                }
            }

            RT result = fun(args);// compute the result and store it.
            memo[tuple(args)] = result;
            argsQueue ~= tuple(args);
            return result;
        }
    }
}
```

And a few factory function to help creating those `Memoize` structs:

```
module memoize3;
import memoize2;

// No runtime arg -> always store
Memoize!(fun, Storing.always, Discarding.all)
memoize(alias fun)()
{
    Memoize!(fun,
             Storing.always,
             Discarding.all) result;
    return result;
}

// One runtime size_t arg -> maximum store / discarding all
Memoize!(fun, Storing.maximum, Discarding.all)
memoize(alias fun)(size_t max)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.all) result;
    result.maxNumStored = max;
    return result;
}

// Two runtime args (size_t, double) -> maximum store / discarding a fraction
```

```
Memoize!(fun, Storing.maximum, Discarding.fraction)
memoize(alias fun)(size_t max, double fraction)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.fraction) result;
    result.maxNumStored = max;
    result.fraction = fraction;
    return result;
}

// One compile-time argument (discarding oldest), one runtime argument (max)
Memoize!(fun, Storing.maximum, discarding)
memoize(alias fun, Discarding discarding = Discarding.oldest)
(size_t max)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.oldest) result;
    result.maxNumStored = max;
    return result;
}
```

Note that, due to the introduction of an `opCall` operator, it's not possible to use a struct literal. We have to first create the struct, then initialize its fields.

Most of the time, the type of runtime arguments is enough to determine what you want as a memoizing/storing behavior. Only for the (rarer?) policy of discarding only the oldest stored result does the user need to indicate it with a template argument:

```
moduleusingmemoize3;
import memoize3;

int veryLongCalc(int i, double d, string s)
{
   /* ... cheating here ... */
   return i,;
}

void main()
{
    // Store the first million results, flush the memo on max
    auto vlcMemo1 = memoize!(veryLongCalc)(1_000_000);

    // Store the first million results, flush half the memo on max
```

```
    auto vlcMemo2 = memoize!(veryLongCalc)(1_000_000, 0.5f);

    // Store first twenty results, discard only the oldest
    auto vlcMemo3 = memoize!(veryLongCalc, Discarding.oldest)(20);
}
```

Currying a Function

Unfinished Some explanations would greatly help there.

Another useful transform on functions is to *curry* them:[20] to transform a n-args function into n one-parameter functions inside another.

TODO Show some example: mapping a range for example.

```
module checkcompatibility;

template CheckCompatibility(T...)
{
    template With(U...)
    {
        static if (U.length != T.length)
            enum With = false;
        else static if  (T.length == 0) // U.length == 0 also
            enum With = true;
        else static if (!is(U[0] : T[0]))
            enum With = false;
        else
            enum With = CheckCompatibility!(T[1..$]).With!(U[1..$]);
    }
}

module curry;
import std.traits;
import checkcompatibility;

struct Curry(alias fun, int index = 0)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;
    PTT args;
```

[20] from Haskell Curry, who formalized the idea.

```
    auto opCall(V...)(V values)
        if (V.length > 0
         && V.length + index <= PTT.length)
    {
        // Is fun directly callable with the provided arguments?
        static if (__traits(compiles, fun(args[0..index], values)))
            return fun(args[0..index], values);
        // If not, the new args will be stored. We check their types.
        else static if (!CheckCompatibility!(PTT[index..index + V.length]).With!(V))
            static assert(0, "curry: bad arguments. Waited for "
                           ~ PTT[index..index + V.length].stringof
                           ~ " but got " ~ V.stringof);
        // not enough args yet. We store them.
        else
        {
            Curry!(fun, index+V.length) c;
            foreach(i,a; args[0..index]) c.args[i] = a;
            foreach(i,v; values) c.args[index+i] = v;
            return c;
        }
    }
}

auto curry(alias fun)()
{
    Curry!(fun,0) c;
    return c;
}
```

Juxtaposing Functions

> **Unfinished** The idea is to glue functions together, to map on many ranges in parallel and create another multiple-value range afterwards.

The functions are juxtaposed, that is, given:

```
int foo(int i, int j) { return i+j;}
string bar() { return "Hello, World";}
double baz(double d) { return d*d;}
```

Then `juxtapose!(foo,bar,baz)` is a function accepting two `int`s and a `double` as arguments and returning a tuple holding an `int`, a `string` and a `double`.

```
module juxtapose;
import juxtaposehelper;
```

```
template juxtapose(Funs...)
{
    Tuple!(staticFilter!(isNotVoid, ReturnTypes!Funs))
    juxtapose(ParameterTypeTuples!Funs params)
    {
        typeof(return) result;
        alias SumOfArities!Funs arities;
        alias SumOfReturns!Funs returns;
        foreach(i, Fun; Funs)
        {
            enum firstParam = arities[i];
            enum lastParam = firstParam + arity!(Fun);
            static if (returns[i] != returns[i+1])
                result.field[returns[i]] = Fun(params[firstParam..lastParam]);
        }
        return result;
    }
}
```

Necessary scaffolding:

```
module juxtaposehelper;
import std.traits;
import functionarity;
import staticscan;
import maponalias;

template isNotVoid(T)
{
    enum bool isNotVoid = !is(T == void);
}

/**
 * Given a bunch of functions names, gives the typetuple of their return types.
 * Used by juxtapose.
 */
template ReturnTypes(Funs...)
{
    alias MapOnAlias!(ReturnType, Funs) ReturnTypes;
}

/**
 * Given a bunch of functions names, gives the (flattened) typetuple
 * of their return values. Used by juxtapose.
```

```
 */
template ParameterTypeTuples(alias fun, Rest...)
{
    alias MapOnAlias!(ParameterTypeTuple, fun, Rest) ParameterTypeTuples;
}

template SumOfArity(size_t zero, alias fun)
{
    enum size_t SumOfArity = zero + arity!fun;
}

template SumOfArities(F...)
{
    alias StaticScan!(SumOfArity, 0, F) SumOfArities;
}

template SumOfReturn(size_t zero, alias fun)
{
    static if (is(ReturnType!fun == void))
        enum size_t SumOfReturn = zero;
    else
        enum size_t SumOfReturn = zero + 1;
}

template SumOfReturns(Funs...)
{
    alias StaticScan!(SumOfReturn, 0, Funs) SumOfReturns;
}
```

Should I also add these?

```
module maponalias;
import std.typetuple;

/**
 * Maps the Mapper template on the alias list.
 */
template MapOnAlias(alias Mapper, alias current, Rest...)
{
    static if (Rest.length)
        alias TypeTuple!(Mapper!current, MapOnAlias!(Mapper, Rest)) MapOnAlias;
    else
        alias Mapper!current MapOnAlias;
}
```

```
template MapOnAlias(alias Mapper)
{
    alias TypeTuple!() MapOnAlias;
}
```

Relational Algebra

Inspiration for this example comes from This blog article.

> **TODO** I have the code somewhere. What it should do: extracting from a tuple: project, select. Also, natural/inner/outer join, cartesian product. And intersection/union/difference. `rename!("oldField", "newField")`. Databases are just dynamic arrays of tuples.

```
module relational;
import std.traits;
import std.typetuple;
import std.typecons;
import std.range;
import std.conv;
import isastringliteral;
import istype;
import half;
import interleave;

struct Entry(Data...)
if ( (Data.length % 2 == 0)
  && (allSatisfy!(isAStringLiteral, Half!Data))
  && (allSatisfy!(isType, Half!(Data[1..$], void))))
{
    alias Half!Data Headers;
    alias Half!(Data[1..$], void) Values;
    alias data this;

    Tuple!(Interleave!(Values).With!(Headers)) data;

    this(Values values)
    {
        foreach(i, Unused; Values) data[i] = values[i];
    }

    string toString() @property
    {
        string s = "[";
```

```
        foreach(i, Unused; Values)
            s ~= Headers[i] ~ ":" ~ to!string(data[i]) ~ ", ";
        return s[0..$-2] ~ "]";
    }
}

template entry(Headers...)
if (allSatisfy!(isAStringLiteral, Headers))
{
    Entry!(Interleave!(Headers).With!(Values)) entry(Values...)(Values values)
    if (Values.length == Headers.length)
    {
        return typeof(return)(values);
    }
}

template isEntry(E)
{
    enum isEntry = __traits(compiles,
                     {
                          void fun(T...)(Entry!T e) {}
                          fun(E.init);
                     });
}

auto rename(string from, string to, E)(E e)
if (isEntry!E)
{
    enum index = staticIndexOf!(from, E.Headers);
    static if (index == -1)
        static assert(false, "Bad index in rename: no header called "
                             ~ from ~ " in " ~ E.Headers.stringof);
    else
        return entry!(E.Headers[0..index],
                      to,
                      E.Headers[index+1..$])(e.data.expand);
}

auto rename(string from, string to, D)(D db)
if (isDatabase!D)
{
    ReturnType!(rename!(from, to, ElementType!D))[] result;
    foreach(i, elem; db)
        result ~= rename!(from, to)(elem);
    return result;
}
```

```
template isDatabase(D)
{
    static if (isDynamicArray!D && isEntry!(ElementType!D))
        enum bool isDatabase = true;
    else
        enum bool isDatabase = false;
}

template isSomeHeader(E)
if (isEntry!E)
{
    template isSomeHeader(string s)
    {
        static if (staticIndexOf!(s, E.Headers) != -1)
            enum bool isSomeHeader = true;
        else
            enum bool isSomeHeader = false;
    }
}

template HeaderValue(E)
if (isEntry!E)
{
    template HeaderValue(string header)
    if (staticIndexOf!(header, E.Headers) != -1)
    {
        alias TypeTuple!(header, E.Values[staticIndexOf!(header, E.Headers)])
              HeaderValue;
    }
}

template project(Headers...)
if (Headers.length > 0)
{
    auto project(E)(E e)
    {
        static if (isEntry!E && allSatisfy!(isSomeHeader!E, Headers))
        {
            alias staticMap!(HeaderValue!E, Headers)
                  HeadersAndValues;
            Entry!(HeadersAndValues) result;
            foreach(i, Unused; Headers)
                mixin("result." ~ Headers[i] ~ " = e." ~ Headers[i] ~ ";");
            return result;
```

```
        }
        else static if (isDatabase!E
                    && allSatisfy!(isSomeHeader!(ElementType!E), Headers))
        {
            alias staticMap!(HeaderValue!(ElementType!E), Headers)
                  HeadersAndValues;

            Entry!(HeadersAndValues)[] result;
            Entry!(HeadersAndValues)   elem;

            foreach(i,Unused; e)
            {
                foreach(j, Unused2; Headers)
                    mixin("elem." ~ Headers[j] ~ " = e[i]." ~ Headers[j] ~ ";");
                result ~= elem;
            }
            return result;
        }
        else
            static assert(0, "Cannot project on " ~ Headers.stringof[5..$]
                           ~ " for type " ~ E.stringof);
    }
}
```

`Half` is more or less the converse of `Interleave` (see section Interleaving Types): given a typetuple, it takes every other types:[21]

```
module half;
import std.typetuple;

template Half(T...)
{
    static if (T.length == 0)
        alias TypeTuple!() Half;
    else static if (T.length == 1)
        alias TypeTuple!(T[0]) Half;
    else
        alias TypeTuple!(T[0], Half!(T[2..$])) Half;
}

unittest
{
    alias TypeTuple!() Test0;
```

[21] A standard generalization would be a template taking on type in n: `Stride!(3, T0, T1, ..., Tn)`. I let that as an exercise to the reader.

```
    alias TypeTuple!(int) Test1;
    alias TypeTuple!(int, float) Test2;
    alias TypeTuple!(int, float, string) Test3;
    alias TypeTuple!(int, float, string, double) Test4;

    static assert(is(Half!Test0 == TypeTuple!()));
    static assert(is(Half!Test1 == TypeTuple!(int)));
    static assert(is(Half!Test2 == TypeTuple!(int)));
    static assert(is(Half!Test3 == TypeTuple!(int, string)));
    static assert(is(Half!Test4 == TypeTuple!(int, string)));
}
```

Which gives us:

```
module usingrelational;
import std.stdio;
import relational;

alias Entry!("Name",     string
            ,"EmployId", int
            ,"Dept", string) Employee;

alias Entry!("DeptName", string
            ,"Manager",  string) Dept;

void main()
{
    auto e = entry!("Name", "EmployId", "DeptName")("John", 1, "Tech");
    auto e2 = Employee("Susan", 2, "Financial");
    auto e3 = Employee("Bob", 3, "Financial");
    auto e4 = Employee("Sri", 4, "Tech");

    auto d1 = Dept("Tech", "Sri");
    auto d2 = Dept("Financial", "Bob");

    auto employees = [e2,e3,e4];
    auto depts = [d1, d2];

    writeln(employees);

    writeln(rename!("Dept", "DN")(employees));
}
```

Fun With Classes and Structs

Class Hierarchy

> **Unfinished** Two things I'll show here: how to get a class parents an how to determine an entire hierarchy of classes in a local scope.

Generic Maker Function

Like this:

```
class C
{
    int i, j;

    this(int _i) { i = _i; j = _i;}
    this(int _i, int _j) { i = _i; j =_j;}
}

alias make!C makeC;

auto theCs = map!makeC([0,1,2,3,4]);
auto theCs2 = map!makeC(zip([0,1,2,3,4],
                            [4,3,2,1,0]));
```

Emitting Events

This example template, comprising `Fields` and `Notify`, comes from Andrej Mitrovic. He has be kind enough to allow me to put it there and to give me some explanations for the logic behind this template. He's using it in his experimental GUI library to signal an event.

```
module fields;
import std.typetuple;
import isastringliteral;

mixin template Fields(T, fields...)
    if (allSatisfy!(isAStringLiteral, fields))
{
    alias typeof(this) This;

    static string __makeFields(T, fields...)()
    {
        string res;
```

```
        foreach(field; fields) res ~= T.stringof~ " " ~ field ~ ";\n";
        return res;
    }

    static string __makeOpBinaryFields(string op, fields...)()
    {
        string res;
        foreach(field; fields)
            res ~= "res." ~ field
                 ~ " = this." ~ field ~ op ~ " rhs." ~ field ~ ";\n";
        return res;
    }

    mixin(__makeFields!(T, fields)());

    This opBinary(string op)(This rhs)
    {
        This res;
        mixin(__makeOpBinaryFields!(op, fields)());
        return res;
    }

    void opOpAssign(string op)(This rhs)
    {
        mixin("this = this " ~ op ~ " rhs;");
    }
}
```

The user mix it in its own types:

```
module usingfields;
import fields;

struct Point {
    mixin Fields!(int, "x", "y", "z", "w");
}

struct Size {
    mixin Fields!(int, "width", "height");
}
```

This goes hand in hand with the `Notify` struct template:

```
module notify;
import std.conv;
```

```d
struct Notify(T)
{
    alias void delegate(ref T) OnEventFunc;
    OnEventFunc onEvent;

    void init(OnEventFunc func) {
        onEvent = func;
    }

    string toString()
    {
        return to!string(raw);
    }

    auto opEquals(T)(T rhs)
    {
        return rhs == rhs;
    }

    void opAssign(T)(T rhs)
    {
        if (rhs == raw)
            return;  // avoid infinite loops

        // temp used for ref
        auto temp = rhs;
        onEvent(temp);
    }

    auto opBinary(string op, T)(T rhs)
    {
        mixin("return raw " ~ op ~ " rhs;");
    }

    void opOpAssign(string op, T)(T rhs)
    {
        // temp used for ref
        mixin("auto temp = raw " ~ op ~ " rhs;");

        if (temp == raw)
            return;  // avoid infinite loops

        onEvent(temp);
    }
```

```
    public T raw;  // raw access when we don't want to invoke event()
    alias raw this;
}
```

Which you mix in classes you want to notify any changes to their internal state:

```
module usingnotify;
import std.stdio;
import usingfields;
import notify;

class Widget
{
    this()
    {
        point.init((ref Point pt) { writefln("changed point to %s", pt); });
        size.init((ref Size sz)   { writefln("changed size to %s", sz); });
    }

    Notify!Point point;
    Notify!Size size;
}
```

For example, a user might change the point (position) field of a widget via:

```
Point moveBy = Point(10, 0); widget.point += moveBy;}.
```

This doesn't modify the field yet, but only triggers `onEvent(moveBy)`, which in turn emits a signal containing the `Widget` reference and the requested position `doMove.emit(this, moveBy)`. This gets processed by a chain of an arbitrary number of listeners. These listeners take `moveBy` by reference and this comes in handy when e.g. a `Widget` is part of a `Layout`. The `Layout` simply adds itself to the chain of listeners and edits the ref argument if it wants to, or even returns `false` to completely deny any changes to a field.

This allows for some great flexibility. For example, say a user subclasses from `Widget` (let's call it `Child`), and overrides `onMove()` of the widget to limit it to the position of:

```
min: Point(10, 10) (top-left)
max: Point(100, 100) (bottom-right)
```

This `Widget` has a parent `Widget` (call it `Parent`), which has a layout set. The layout might allow any child `Widgets` of the `Parent` to only be set between the following positions:

```
min: Point(20, 20)
max: Point(80, 80)
```

Additionally the layout might take a child's size into account so the `Widget` never overflows the `min` and `max` points of the layout. If this `Widget` was at `Point(20, 20)` and had `Size(50, 50)` it means the layout will limit the `Wid\-get`'s minimum and maximum points to:

```
min: Point(20, 20)
max: Point(30, 30)
```

When the `Widget` is at `Point(30, 30)` its bottom-right point will be `Point(80, 80)`, which is the maximum bottom-right point the layout has set. The layout won't allow the `Widget` to be at position `Point(10, 10)` either, even though `Widget`'s `onMove` method allows it.

So if the user tries to call:

```
child.point = Point(120, 120);
```

First, `Child`'s `onMove` is the first listener of some `doMove` signal. It will modify the ref argument and set it to `Point(100, 100)` as the user coded it that way. Then, the layout will modify it to `Point(30, 30)` since it takes `Widget`'s size into account so it doesn't overflow.

There could be any number of listeners, and you could add a listener to any position in the chain of events (maybe you want to intercept and modify the argument before it reaches `Parent`'s layout if you want to set temporary limits to a `Widget`'s position).

Finally, there is always one last listener. This one is an internal function that actually modifies the `.raw` field, and calls into internal painting and blitting functions to make the `Widget` appear into its new position. Also, any listener can return `false` to break further processing of events and deny modifying the field.

Notice that assignments never happen, external code has to use the `.raw` field to actually modify the internal payload, which avoids calling `onEvent()`.

Fields

From Jacob Carlborg's Orange serialization library:

```
module fieldsof;
```

```
/**
 * Evaluates to an array of strings containing
 * the names of the fields in the given type
 */
template fieldsOf (T)
{
    const fieldsOf = fieldsOfImpl!(T, 0);
}

/**
 * Implementation for fieldsOf
 *
 * Returns: an array of strings containing the names of the fields in the given type
 */
template fieldsOfImpl (T, size_t i)
{
    static if (T.tupleof.length == 0)
        enum fieldsOfImpl = [""];

    else static if (T.tupleof.length - 1 == i)
        enum fieldsOfImpl = [T.tupleof[i].stringof[1 + T.stringof.length + 2 .. $]];

    else
        enum fieldsOfImpl = T.tupleof[i].stringof[1 + T.stringof.length + 2 .. $] ~ fieldsOf
}
```

Extending an enum

This code comes from Simen Kjaeras. It generates an enum definition as a string, by taking all the members of the old enum, and adding those passed in string parameters, and mixing it in.

```
module enumdefasstring;

string EnumDefAsString(T)()
if (is(T == enum))
{
   string result = "";
   foreach (e; __traits(allMembers, T))
       result ~= e ~ " = T." ~ e ~ ",";
   return result;
}
```

This piece of code iterates over all members of the passed `enum`T, generating a string containing all members and their values. For this enum:

```
enum bar
{
    a, b, c
}
```

The generated string looks like this (if you want to check this, feel free to call `EnumDefAsString` at run-time and print its result):

```
"a = bar.a,b = bar.b,c = bar.c"
```

As we can see, this is a valid body for an enum. That means we can use `mixin()` to generate this exact same enum. But wait - there's more:

```
module extendenum;
import enumdefasstring;

template ExtendEnum(T, string s)
   if (is(T == enum) &&
   is(typeof({mixin("enum a{"~s~"}");})))
{
   mixin(
   "enum ExtendEnum {"
   ~ EnumDefAsString!T()
   ~ s
   ~ "}");
}
```

This code concatenates the string generated from the previous function with the one passed to the function as parameter `s`. So with `bar` previously defined, and this instantiation:

```
ExtendEnum!(bar, "d=25")
```

The body of the function will look like this (after string expansion):

```
mixin(
   "enum ExtendEnum {"
   ~ "a = bar.a,b = bar.b,c = bar.c"
   ~ "d=25"
   ~ "}");
```

Concatenating those strings, we see that we have a valid enum definition:

```
enum ExtendEnum {a = bar.a,b = bar.b,c = bar.c,d=25}
```

The mixin then pastes it in, and it is compiled as regular D code.

Static Switching

TODO What, no compile-time switch? Let's create one. Example of: tuples, type filtering (in constraints), recursion, etc.

```
module staticswitch;

template staticSwitch(List...) // List[0] is the value commanding the switching
                               // It can be a type or a symbol.
{
    static if (List.length == 1) // No slot left: error
        static assert(0, "StaticSwitch: no match for " ~ List[0].stringof);
    else static if (List.length == 2) // One slot left: default case
        enum staticSwitch = List[1];
    else static if (is(List[0] == List[1]) // Comparison on types
                || (  !is(List[0])         // Comparison on values
                   && !is(List[1])
                   && is(typeof(List[0] == List[1]))
                   && (List[0] == List[1])))
        enum staticSwitch = List[2];
    else
        enum staticSwitch = staticSwitch!(List[0], List[3..$]);
}
```

Generic Structures

Unfinished This section will present some generic structures of growing complexity.

Gobble

Let's begin with `Gobbler`, a small exercise in tuple manipulation and operator overloading. `Gobbler` is a struct wrapping a tuple and defining only one operator: the right-concatenation operator (`\~`).

```
module gobbler;

struct Gobbler(T...)
{
    alias T Types;
    T store;
    Gobbler!(T, U) opBinary(string op, U)(U u) if (op == "~")
    {
        return Gobbler!(T, U)(store, u);
```

```
    }
}

Gobbler!() gobble() { return Gobbler!()();}
```

`gobble` creates an empty gobbler and is there to activate the aspiration:

```
module usinggobbler;
import std.typetuple;
import gobbler;

void main()
{
    auto list = gobble ~ 1 ~ "abc" ~ 3.14 ~ "another string!";
    assert(is(list.Types == TypeTuple!(int, string, double, string)));
    assert(list.store[2] == 3.14);
}
```

TODO Indexing the `Gobbler`.

Polymorphic Association Lists

An association list as a sort of 'flat' associative array: it holds key-value pairs in a linear list. A polymorphic (aka, templated) one is a tuple holding a bunch of key-value keys, but with more flexibility on the types for keys and values. Different trade-off can be done here between the runtime or compile-time nature of keys and values.

Unfinished This section will present one solution.

Usage: a bit like Lua tables: structs, classes (you can put anonymous functions in them?), namespaces. Also, maybe to add metadata to a type?

```
module alist;
import std.typetuple;
import std.stdio;
import half;

struct AList(T...)
{
    static if (T.length >= 2 && T.length % 2 == 0)
        alias Half!T Keys;
    else static if (T.length >= 2 && T.length % 2 == 1)
```

```
        alias Half!(T[0..$-1]) Keys;
    else
        alias TypeTuple!() Keys;

    static if (T.length >= 2)
        alias Half!(T[1..$]) Values;
    else
        alias TypeTuple!() Values;

    template at(alias a)
    {
        // key not found, but default value present
        static if ((staticIndexOf!(a, Keys) == -1) && (T.length % 2 == 1))
            enum at = T[$-1]; // default value
        else static if ((staticIndexOf!(a, Keys) == -1) && (T.length % 2 == 0))
            static assert(0, "AList: no key equal to " ~ a.stringof);
        else //static if (Keys[staticIndexOf!(a, Keys)] == a)
            enum at = Values[staticIndexOf!(a, Keys)];
    }
}

void main()
{
    alias AList!( 1,     "abc"
                , 2,     'd'
                , 3,     "def"
                , "foo", 3.14
                ,        "Default") al;

    writeln("Keys: ", al.Keys.stringof);
    writeln("Values: ", al.Values.stringof);
    writeln("at!1: ", al.at!(1));
    writeln("at!2: ", al.at!(2));
    writeln("at!\"foo\": ", al.at!("foo"));
    writeln("Default: ", al.at!4);
}
```

A Polymorphic Tree

So, what's a polymorphic tree? It's just a tuple hoding other tuples as elements, as a standard tree container, only all values can be of a different type. Obviously this means that trees holding values of different types will also be of a different type, since the entire content's type is part of the tree signature. It can be a bit baffling to see one, but with a few helper functions to transform a tree or extract some values, it can be quite interesting to use.

Just to get a little feel for them and to use a less-used example for trees, imagine wanting to manipulate mark-up text in D. You could create your document as a D structure and then invoke some functions to transform it into DDoc text, or a LaTeX document, a Markdown one or even HTML:

```
auto doc =
document(
    title("Ranges: A Tutorial"),
    author("John Doe"),
    tableOfContents,

    /* level-one section */
    section!1(
        title("Some Range Definitions"),
        "Ranges are a nice D abstraction...",
        definition(
            "Input Range",
            "The most basic kind of range, it must define the following methods:",
            list(definition("front", "..."),
                 definition("popFront", "..."),
                 definition("empty", "..."))
        )
    )
    section!1(
        title("Some ranges examples"),
        "...",
        code(
            "auto m = map!((a) => a*a)([0,1,2,3]);
             assert(m.length == 4);"
        ),
        link("http://dlang.org/", "Website")
    )
    section!1(
        title("Beyond ranges"),
        "..."
    )
);

auto latex = doc.to!"LaTeX";
auto html = doc.to!"HTML";
auto ddoc = doc.to!"Ddoc";
auto simple = doc.to!"text";
```

In the previous (imaginary, but tempting for me) code, `doc` is a tuple made by the `document` factory function and holding small specifically marked pieces of text:

`title`, `section` or `link`. Each is a factory function producing a user-defined struct following a few simple conventions. If all types have a `to!"HTML"` member that transforms their content into HTML code, then the entire document can be dumped as a HTML file. The different types need not be classes inheriting from a common base and that must be shoe-horned into a defined hierarchy: template constraints do the verification for you. Think ranges.

This is an example of a polymorphic tree.

Expression Templates

Expression templates are a kind of polymorphic tree, but restricted to some known operations (most of the times unary/binary/ternary operators) and their operands. It allows one to store for example an arithmetic operation like this:

```
// From "x + 1 * y":
Binary!("+",
             Variable!"x",
             Binary!("*",
                        Constant(1),
                        Variable!"y"))
```

The advantages are that you can then manipulate the resulting tree to simplify the expression or avoid temporary evaluations. The previous expression could be simplified to hold the equivalent of `x + y` (getting rid of the multiplication by one.

More generally, you can encode a programming language expression in such a tree:

```
AST!"
if (x == 0)
then
{
    writeln(x);
}
else
{
    ++x;
    foo(x,y);
}"
=>
If(Comparison!("==", Symbol!"x", value(0)), // Condition
// then branch
   Block!( FunctionCall!("writeln", Symbol!"x") ),
```

```
// (optional) else branch
   Block!( Unary!("++", Symbol!"x"),
           FunctionCall!("foo", Symbol!"x", Symbol!"y")))
```

This way lies madness and the power of macros, because you can then manipulate the resulting Abstract Syntax Tree in any way you wish, rewrite the code it represents, convert it back into a string and write it into the file that will be given to the compiler.

So,

- Define a compile-time parser,
- feed it (a reasonable part of) the D grammar,
- define some new authorized constructs and the associated AST and the way this new constructs can be behind-the-scene assembled from existing part of the D language (aka, macros),
- write code in your new D extension, your *precious*,
- feed it with the macros to a program that will create the resulting AST, modifying it like you wish and reassemble it into *bona fide* D code.
- and will then feed it to the standard D compiler.

And *voila*, your own toy D extension. Or, you know, you could just bug Walter till he adds the syntax you want into the language.

Statically-Checked Writeln

This is an example of verifying a Domain-Specific Language (DSL) at compile-time in D. The goal is to take a format string for std.stdio.writef or std.stdio.writefln and to check the arguments' types before passing them to `writef`(`ln`).

For example, when writing:

```
writefln("For sample #%d, the results are (%s, %f)", num, name, f);
```

We know that `%d` means the argument must be an integral type, `%f` asks for a floating-point type and so on. That means in the previous sample, we know that:

- There must exactly 3 args, no more, no less.
- The first one must have an integral type.
- The second one can be of any type (more exactly, of any type that can be converted into a `string`).
- The third one must be of some floating point type.

These four conditions can be checked at compile-time, that's what we will do here. I won't code the entire POSIX specification for `printf`, the following table shows what will be checked.

Formatter	Asks For	Equivalent Constraint
%d, %i	an integral type	isIntegral
%u, %x, %X, %o	an unsigned integral type	isUnsigned
%f, %F, %e, %E, %g, %G	a floating point type	isFloatingPoint
%c	a char type	isSomeChar
%s	any type	isAnyType
%%	not a formatter	no check

Table 10: Standard formatters recognized by cwritef

For those interested in the details, this Wikipedia article makes for a nice reference. Note that not all formatters are implemented in std.stdio, for example the %p formatter for void* pointers seems not to work.

Most of the previous typechecking templates are in std.traits: isIntegral, isFloatingPoint, isUnsigned and isSomeChar are already implented in Phobos. The only one left is isAnyType, a quite complacent template:

```
module isanytype;

template isAnyType(T)
{
    enum isAnyType = true;
}
```

The only way for it to fail would be to give it a non-type.

Continuing with the previous chapters' example-driven development, here is what I want to obtain (cwritefln stands for checked-writefln):

```
module usingcheckedwrite;
import checkedwrite;

void main()
{
    cwritefln!"For sample #%d, the results are (%s, %f)"( 0, "foo", 3.14); // OK

 // NOK: bad number or args: waited for 3 args, got 2.
 // cwritefln!"For sample #%d, the results are (%s, %f)"( 0, "foo");
```

```
 // NOK: arg #3 of type double does not verify check isFloatingPoint
 // cwritefln!"For sample #%d, the results are (%s, %f)"( 0, 3.14, "foo");
}
```

Now, given a formatting string, the first thing is to extract the formatters and construct the constraints list. Here I'll use a string mixin and just need to build a string representing the desired final code:

```
module getformatters;
import std.conv;
import std.traits;

string getFormatters(S)(S s) if (isSomeString!S)
{
    dstring ds = to!dstring(s);
    bool afterPercent = false;
    bool error;
    string result = "alias TypeTuple!(";
    foreach(elem; ds)
    {
        if (error) break;
        if (afterPercent)
        {
            switch (elem)
            {
                case '%':
                    afterPercent = false;
                    break;
                case 'd':
                case 'i':
                    result ~= "isIntegral,"; // integers
                    afterPercent = false;
                    break;
                case 'u':
                case 'x':
                case 'X':
                case 'o':
                    result ~= "isUnsigned,"; // unsigned integral
                    afterPercent = false;
                    break;
                case 'f':
                case 'F':
                case 'e':
                case 'E':
                case 'g':
```

```
                case 'G':
                    result ~= "isFloatingPoint,"; // floating point
                    afterPercent = false;
                    break;
                case 'c':
                    result ~= "isSomeChar,"; // char
                    afterPercent = false;
                    break;
                case 's':
                    result ~= "isAnyType,"; // any string-convertible type
                    afterPercent = false;
                    break;
                /* flags, width, */
                case '+':
                case '-':
                case '#':
                case '.':
                case ' ':
                case '0':
                    ..
                case '9':
                    break;
                default:
                    error = true; // Error!
                    break;
            }
        }
        else
        {
            if (elem == '%') afterPercent = true;
        }
    }

    // Get rid of the last comma:
    if (result.length > 17) result = result[0..$-1];
    // finishing the alias code
    result ~= ") ArgsChecks;";

    if (afterPercent // finished the string but still in "afterPercent" mode
     || error)
        result = "static assert(0, \"Bad format string: \" ~ a);";

    return result;
}
```

It's quite a long sample, but the logic behind it is straightforward: it iterates

on all characters and looks for %x patterns. I included here a basic treatment for flags and such, but as I said earlier, this example does not deal with the entire POSIX specification: the goal is *not* to validate the formatting string, but to extract the formatters. When it determines the string is malformed, the generated code will be a `static assert`.

So, at the end, we get ready-to-be-mixed-in strings, like these:

```
// no formatter
"alias TypeTuple!() ArgsChecks;"

// the previous example
"alias TypeTuple!(isIntegral,isAnyType,isFloatingType) ArgsChecks;"

// Bad string
"static assert(0, \"Bad format string: %s and %z\");"
```

Once the tuple of checks is done, we need a template that verifies each argument in turn with the corresponding template. To get a better error message, I use an `int` template parameter, to count the number of args checked.

```
1  module verifychecks;
2  import std.conv;
3  import std.traits;
4  import std.typetuple;
5  import isanytype;
6  import getformatters;
7
8  template ArgsChecks(alias a) if (isSomeString!(typeof(a)))
9  {
10     mixin(getFormatters(a));
11 }
12
13 template VerifyChecks(int which, Checks...)
14 {
15     template on(Args...)
16     {
17         static if (Checks.length != Args.length)
18             static assert( 0
19                         , "ctwrite bad number of args: waited for "
20                         ~ to!string(Checks.length)
21                         ~ " args, got "
22                         ~ to!string(Args.length)
23                         ~ ".");
24         else static if (Checks.length == 0) // end of process
25             enum on = true;
```

```
26          else static if ({ alias Checks[0] C; return C!(Args[0]);}()) // recurse
27              enum on = VerifyChecks!(which+1, Checks[1..$]).on!(Args[1..$]);
28          else
29              static assert( 0
30                          , "cwrite bad arg: arg #"
31                          ~ to!string(which)
32                          ~ " of type "
33                          ~ Args[0].stringof
34                          ~ " does not verify check "
35                          ~ __traits(identifier, Checks[0]));
36      }
37  }
```

The `Verify` template is another example of a double-decker template, as seen in section Templates In Templates, to get a nice calling syntax:

```
Verify!(0, isIntegral, isFloatingPoint).on!(int, double)
```

The most subtle part is on line 26:

```
else static if ({ alias Checks[0] C; return C!(Args[0]);}()) // recurse
```

We want to apply the first checking constraint, `Check[0]`, on the first argument type, `Args[0]`. Alas, the D grammar does not allow the following construct:

```
else static if (Checks[0]!(Args[0])) // recurse
```

The standard way to do this would be:

```
alias Checks[0] Check;
// and then
else static if (Check!(Args[0]))
```

But that would put a `Check` symbol in the template local scope (at one time this would have broken the eponymous trick). It'd be possible to define a `VerifyImpl` template (see section impltrick), but using a local delegate works as well:

```
{ alias Checks[0] C; return C!(Args[0]);}()
```

The `{...}` part defines the delegate and `()` calls it, returning either `true` or `false`. This is then inserted inside the `static if`.

Anyway, once the checking code is done, the rest is easy:

```
module checkedwrite;
import std.stdio;
import std.traits;
import verifychecks;

void cwritef(alias a, Args...)(Args args)
if (isSomeString!(typeof(a))
 && VerifyChecks!(1, ArgsChecks!(a)).on!(Args))
{
    writef(a, args);
}

void cwritefln(alias a, Args...)(Args args)
if (isSomeString!(typeof(a))
 && VerifyChecks!(1, ArgsChecks!(a)).on!(Args))
{
    writefln(a, args);
}
```

Usage is then as shown here.

Extending a Class

TODO UFCS is now implemented. Things happened in 2012!

There is regularly a wish in the D community for something called Universal Function Call Syntax (UFCS): the automatic transformation of `a.foo(b)` into `foo(a,b)` when `a` has no member called `foo` and there *is* a free function called `foo` in the local scope. This already works for arrays (hence, for strings) but not for other types.

There is no way to get that in D for built-in types except by hacking the compiler, but for user-defined types, you can call templates to the rescue.

`opDispatch` can be used to forward to an external free function. A call `this.method(a,b)` becomes `method(this,a,b)`.

```
module forwarder;

mixin template Forwarder()
{
    auto opDispatch(string name, Args...)(Args args)
    {
        mixin("return " ~ name ~ "(args);");
    }
}
```

In D, a void `return` clause is legal:

```
return;
// or return void;
```

So if `name(this,a,b)` is a `void`-returning function, all is OK.

The main limitation of this trick is that it doesn't work across modules boundaries. Too bad.

Pattern Matching With Functions

> **Unfinished** The idea is to group a bunch of templates together and use their pattern matching ability. Maybe to be put in Function Templates?

Generating a Switch for Tuples

Case 0:, etc.

Or more generally, the idea to craft specific runtime code given compile-time information. See also Sorting Networks.

Appendices

The `is` expression

General Syntax

The `is(...)` expression gives you some compile-time introspection on types (and, as a side-effect, on D expressions). It's described here in the D website. This expression has a quirky syntax, but the basic use is very simple and it's quite useful in conjunction with `static if` (see section Static If) and template constraints (see section Constraints). The common syntaxes are:

```
is( Type (optional identifier) )
is( Type (optional identifier) :  OtherType,
         (optional template parameters list) )
is( Type (optional identifier) == OtherType,
         (optional template parameters list) )
```

If what's inside the parenthesis is valid (see below), `is()` returns `true` at compile-time, else it returns `false`.

`is(Type)`

Let's begin with the very first syntax: if `Type` is a valid D type in the scope of the `is` expression, `is()` returns `true`. As a bonus, inside a `static if`, the optional `identifier` becomes an alias for the type. For example:

```
module canbeinstantiated;

template CanBeInstantiatedWith(alias templateName, Types...)
{
    // is templateName!(Types) a valid type?
    static if (is( templateName!(Types) ResultType ))
    // here you can use ResultType (== templateName!(Types))
        alias ResultType CanBeInstantiatedWith;
    else
        alias void       CanBeInstantiatedWith;
}
```

Note that the previous code was done with templates in mind, but it is quite robust: if you pass as `templateName` something that's not a template name (a function name, for example), the `is` will see `templateName!(Types)` has no valid type and will return `false`. `CanBeInstantiatedWith` will correctly be set to `void` and your program does not crash.

> **Testing for an alias** Sometimes you do not know if the template argument you received is a type or an alias (for example, when dealing with tuples elements). In that case, you can use `!is(symbol)` as a test. If it really is an alias and not a type, this will return `true`.}

An interesting use for this form of `is`, is testing whether or not some D code is valid. Consider: D blocks are seen as delegates by the compiler (Their type is `void delegate()`). Using this in conjunction with `typeof` let you test the validity of a block statement: if `some code` is valid, `typeof({ some code }())` (note the `()` at the end of the delegate to 'activate' it) has a real D type and `is` will return true.

Let's put this to some use. Imagine you have a function template `fun` and some arguments, but you do not know if `fun` can be called with this particular bunch of arguments. If it's a common case in your code, you should abstract it as a template. Let's call it `validCall` and make it a function template also, to easily use it with the arguments:

```
module validcall;
import std.conv;
```

```d
bool validCall(alias fun, Args...)(Args args)
{
    return is( typeof({ /* code to test */
                        fun(args);
                        /* end of code to test */
                      }()));
}

// Usage:
T add(T)(T a, T b) { return a+b;}
string conc(A,B)(A a, B b) { return to!string(a) ~ to!string(b);}

void main()
{
    assert( validCall!add(1, 2));   // generates add!(int)
    assert(!validCall!add(1, "abc")); // no template instantiation possible

    assert( validCall!conc(1, "abc")); // conc!(int, string) is OK.
    assert(!validCall!conc(1)       ); // no 1-argument version for conc

    struct S {}

    assert(!validCall!S(1, 2.3)); // S is not callable
}
```

Note that the tested code is simply '`fun(args);`'. That is, there is no condition on `fun`'s type: it could be a function, a delegate or even a struct or class with `opCall` defined. There are basically two ways `fun(args);` can be invalid code: either `fun` is not callable as a function, or it is callable, but `args` are not valid arguments.

By the way, fun as it may be to use this trick, D provides you with a cleaner way to test for valid compilation:

```d
__traits(compiles, { /* some code */ })
```

`__traits` is another of D numerous Swiss Army knife constructs. You can find the `compiles` documentation here. Section `__traits` is dedicated to it.

`is(Type : AnotherType)` and `is(Type == AnotherType)`

The two other basic forms of `is` return true if `Type` can be implicitly converted to (is derived from) `AnotherType` and if `Type` is exactly `AnotherType`, respectively. I find them most interesting in their more complex form, with a list of template parameters afterwards. In this case, the template parameters act a bit as type variables in an equation. Let me explain:

```
is(Type identifier == SomeComplexTypeDependingOnUAndV, U, V)
```

The previous code really means: 'excuse me, Mr. D Compiler, but is `Type` perchance some complex type depending on `U` and `V` for some `U` and `V`? If yes, please give me those.' For example:

```
template ArrayElement(T)
{
    // is T an array of U, for some U?
    static if (is(T t : U[], U))
        alias U ArrayElement; // U can be used, let's expose it
    else
        alias void ArrayElement;
}

template isAssociativeArray(AA)
{
    static if (is( AA aa == Value[Key], Value, Key))
        /* code here can use Value and Key,
           they have been deduced by the compiler. */
    else
        /* AA is not an associative array
          Value and Key are not defined. */
}
```

Strangely, you can only use it with the `is(Type identifier, ...)` syntax: you *must* have `identifier`. The good new is, the complex types being inspected can be templated types and the parameter list can be any template parameter: not only types, but integral values,... For example, suppose you do what everybody does when encountering D templates: you create a templated n-dimensional vector type.

```
struct Vector(Type, int dim) { ... }
```

If you did not expose `Type` and `dim` (as aliases for example, as seen in sections Inner Alias and Giving Access to Inner Parameters), you can use `is` to extract them for you:

```
Vector!( ?, ?) myVec;
// is myVec a vector of ints, of any dimension?
static if (is(typeof(myVec) mv == Vector!(int, dim), dim))

// is it a 1-dimensional vector?
static if (is(typeof(myVec) mv == Vector!(T, 1), T))
```

is(A != B)? No, sorry, this doesn't exist. Use `is(A == B)`. But beware this will also fire if `A` or `B` are not legal types (which makes sense: if `A` is not defined, then by definition it cannot be equal to `B`). If necessary, you can use `is(A) && is(B) && !is(A == B)`.

is A a supertype to B? Hey, `is(MyType : SuperType)` is good to know if `MyType` is a subtype to `SuperType`. How do I ask if `MyType` is a supertype to `SubType`? Easy, just use `is(SubType : MyType)`.

For me, the main limitation is that template tuple parameters are not accepted. Too bad. See, imagine you use std.typecons.Tuple a lot. At one point, you need a template to test if something is a `Tuple!(T...)` for some `T` which can be 0 or more types. Though luck, `is` is a bit of a letdown there, as you cannot do:

```
template isTuple(T)
{
    static if (is(T tup == Tuple!(InnerTypes), InnerTypes...)
(...)                                          ^^^^^^^^^^^^^
```

TODO Yes you can! This was changed in DMD 2.060. This section needs updating!

But sometimes D channels its inner perl and, lo! There is more than one way to do it! You can use IFTI and our good friend the `is(typeof({...}()))` expression there. You can also use `__traits`, depending on you mood, but since this appendix is specifically on `is`:

```
1  module istuple;
2  import std.typecons;
3
4  template isTuple(T)
5  {
6      enum bool isTuple =
7          is(typeof({
8              void tupleTester(InnerTypes...)(Tuple!(InnerTypes) tup) {}
9              T.init possibleTuple;
10             tupleTester(possibleTuple);
11         }()));
12 }
```

Line 8 defines the function template `tupleTester`, that only accepts `Tuples` as arguments (even though it does nothing with them). We create something of type `T` on line 9, using the `.init` property inherent in all D types, and try to

call `tupleTester` with it. If `T` is indeed a `Tuple` this entire block statement is valid, the resulting delegate call indeed has a type and `is` returns `true`.

There are two things to note here: first, `isTuple` works for any templated type called `Tuple`, not only std.typecons.Tuple. If you want to restrict it, change `tupleTester` definition. Secondly, we do not get access to the inner types this way. For std.typecons.Tuple it's not really a problem, as they can be accessed with the `someTuple.Types` alias, but still...

By the way, the template parameter list elements can themselves use the `A : B` or `A == B` syntax:

```
static if (is( T t == A!(U,V), U : SomeClass!W, V == int[n], W, int n))
```

This will be OK if `T` is indeed an `A` instantiated with an `U` and a `V`, themselves verifying that this `U` is derived from `SomeClass!W` for some `W` type and that `V` is a static array of `int`s of length `n` to be determined (and possibly used afterwards). In the if branch of the `static if` `U`, `V`, `W` and `n` are all defined.

Type Specializations

There is a last thing to know about `is`: with the `is(Type (identifier) == Something)` version, `Something` can also be a type specialization, one of the following D keywords: `function`, `delegate`, `return`, `struct`, `enum`, `union`, `class`, `interface`, `super`, `const`, `immutable` or `shared`. The condition is satisfied if `Type` is one of those (except for `super` and `return`, see below). `identifier` then becomes an alias for some property of `Type`, as described in the following table.

Specialization	Satisfied if	`identifier` becomes
`function`	`Type` is a function	The function parameters type tuple
`delegate`	`Type` is a delegate	The delegate function type
`return`	`Type` is a function	The return type
`return`	`Type` is a delegate	The return type
`struct`	`Type` is a struct	The struct type
`enum`	`Type` is an enum	The enum base type
`union`	`Type` is an union	The union type
`class`	`Type` is a class	The class type
`interface`	`Type` is an interface	The interface type
`super`	`Type` is a class	The type tuple (Base Class, Interfaces)
`const`	`Type` is const	The type

Specialization	Satisfied if	`identifier` becomes
`immutable`	`Type` is immutable	The type
`shared`	`Type` is shared	The type

Table 11: Effect of type specializations in `is`

Let's put that to some use: we want a factory template that will create a new struct or a new class, given its name as a template parameter:

```
module maker;
import std.algorithm;
import std.conv;

template make(A)
    if (is(A a == class )
     || is(A a == struct))
{
    auto make(Args...)(Args args)
    {
        static if (is(A a == class))
            return new A(args);
        else
            return A(args);
    }
}

struct S {int i;}
class C
{
    int i;
    this(int ii) { i = ii;}
    override string toString() @property
    {
        return "C("~to!string(i)~")";
    }
}

void main()
{
    auto array = [0,1,2,3];

    auto structRange = map!( make!S )(array);
```

```
    auto classRange  = map!( make!C )(array);

     assert(equal(structRange, [S(0), S(1), S(2), S(3)]));
    assert(classRange.front.toString == "C(0)");
}
```

You can find another example of this kind of `is` in section Class Templates, with the `duplicator` template.

Resources and Further Reading

Unfinished Any new resource welcome!

Being a relatively young language, D doesn't have dozens of books or online sites dedicated to its learning. Still, there are a few resources on the Internet that pertains to D templates and associated subjects.

D Templates

Here are some resources on D templates.

D Reference Your first stop should be the dlang.org pages on templates. They are quite easy to read, don't hesitate to peruse them and bother the authors if anything is unclear or blatantly false. The `dlang.org` website is also a github project, here. The following table lists the pages dealing with templates.

Subject	URL
The general page on templates	template.html
D templates vs C++ ones	templates-revisited.html
An article on tuples	tuple.html
Template tuple parameters	variadic-function-templates.html
Template constraints	concepts.html
Mixin Templates	template-mixin.html
A small article on string mixins	mixin.html
`static if`	version.html
`is`	expression.html
Type Aliasing	declaration.html
Operator Overloading	operatoroverloading.html

Subject	URL

Table 12: pages dealing with templates

The D Programming Language The main D book as of this writing is of course `The D Programming Language` (also known as `TDPL`), by Andrei Alexandrescu. It's a very good reference book, and fun to read too. TDPL has a very interesting approach to templates: Andrei first introduces them as a natural extension of functions, classes, and structs syntax, to get parametrized code. It's only afterwards that templates-as-scopes are introduced. The following table lists the chapters that deals with templates.

Subject	Chapter
Functions Templates	5.3, 5.4, 5.10, and 5.12
Classes Templates	6.14
Struct Templates	7.1.10
`alias`	7.4
General Templates	7.5
Mixin Templates	7.6
Operator Overlaoding	12

Table 13: TDPL Chapters Dealing With Templates

Programming in D

Another quite interesting book is the translation from Turkish of Ali Cehreli's book `Programming in D`, which you can find for free here: http://ddili.org/ders/d.en/index.html. More precisely, as far as this document is concerned, the chapter on templates is already translated and can be found at http://ddili.org/ders/d.en/templates.html. Ali's book aims to be a gentle introduction to D, so the rhythm is less gung ho than TDPL. If you do not want to read the present document, the previously linked chapter is your best bet as a template tutorial.

The D Wiki

Apart from books, the D wiki has a tutorial on templates that can be found at http://wiki.dlang.org/Introduction_to_D_templates, that explains

std.functional.unaryFun very thoroughly, dealing in passing with various subjects such as templates, string mixins, and instantiation scope. I found it a very pleasant read and a nice explanation of a difficult part of Phobos.

D Metaprogramming

I found Nick Sabalausky's article at http://www.semitwist.com/articles/EfficientAndFlexible/SinglePage/ to be a very good read, and full of nifty ideas. In particular, Nick shows how to use compile-time information to craft code such that runtime arguments can then be used to "influence" compile-time values. Yes you read that right, and my description does not do it justice: read on, and see for yourselves. I definitely *must* put a section somewhere in this doc on this.

The code shown in the article can be found in a github project, at https://github.com/Abscissa/efficientAndFlexible

Templates in Other Languages

> **Unfinished** C++ templates, obviously. Articles, online books, and such. A small text on Boost. Also, maybe Java and C# generics.

Genericity in Other Languages

> **Unfinished** Haskell and ML concerning types. In general, any type trick that Haskellers live by: encoding information in types, using types to forbid unwanted states and so on.

Metaprogramming in Other Languages

macros

> **Unfinished** Maybe Lisp / Scheme / Clojure macros. Nemerle macros too.