D Templates: Some Notes

Philippe Sigaud

Saturday 17th December, 2011

Contents

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 any D user should know about them. But, based on C++'s templates as they are, they 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 doing 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, nifty things like `static if`, `alias` or tuples cured me of that impression. I hope this document will help you also.

Part I 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 (4), struct (5) and class (6) templates. Throughout the text, examples will present applications of these concepts.

Part II is about more advanced topics a D template user will probably use, but not on a daily basis, like template constraints (8), mixin templates (12) or operator overloading (11).

Part III presents other metaprogramming tools: string mixins (16), compile-time function evaluation (17) and `__traits` (18). 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.

Part IV 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 (A) completes this document.

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

Part I
Basics

A template is a recipe, a blueprint that will generate code at your command and according to compile-time parameters you'll give. Templates are *parameterized code*. Each template definition is written once in a module and can then be instantiated many times with different parameters, possibly resulting in quite different code, depending on the arguments you used.

1 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: 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: These will capture 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. They will *not* accept built-in types as arguments, however. You declare them with `alias identifier`.

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

Template parameters tuples: Template parameters tuples will capture under one identifier an entire list of template parameters (types, names, literals, ...). Tuples will store any template argument you will throw at them. If no argument is passed, you will just get a 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 10. The syntax is `identifier...` (yes, three dots) and tuples must be the last parameter of a template.

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, described in section 17). 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 (`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

And then there is a catch: code inside a `template` declaration must only by syntactically correct D code (that is: code that looks like D code). The semantics is checked only during 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 (see section 5) and class (6) templates.

Here are some template declaration examples:

```
template ArrayOf(T) // T is a type
{
    alias T[] ArrayType;
    alias T ElementType;
    immutable T t; // storage class

}

template Transformer(From, To) // From and To are types, too
{
    To transform(From from) { /* some code that returns a To*/}

    class Modificator
    {
        From f;
```

```
        To t;
        this(From f) { ... }
    }
}

template NameOf(alias a)
{
    enum string name = a.stringof; // enum: manifest constant
                                   // determined at compile-time
}

template ComplicatedOne(T, string s, alias a, bool b, int i)
{ /* some code using T, s, a, b and i */ }

template Minimalist() {} // Zero-parameter template declaration.

template OneOrMore(FirstType, Rest...) // Rest is a tuple.
{ ... }

template ZeroOrMore(Types...) // Types is a tuple.
{ ... }

template Multiple(T)     { ... } // One arg version.
template Multiple(T,U)   { ... } // Two args,
template Multiple(T,U,V) { ... } // and three.
```

The real 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 3.4, default values in section 3.5, instantiation constraints in 8 and more on tuples in section 10).

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

enum

In the previous code, see line 22? 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.

2 Instantiating a Template

To instantiate a template, use the following syntax:

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

Note the '`!`' before the comma-separated argument list. If the argument list contains only one argument (one token), you can drop the parenthesis:

```
templateName!argument
```

Templates as templates 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)`.

The compiler will have a look at the declarations (if more than one template were 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 in section 3.4 and template constraints in section 8).

When you instantiate a template, the global effect is that a new named scope is created in the template declaration scope. The name of this new scope is the template name with its argument list: `templateName!(args)`. Inside the scope, the parameters are now 'replaced' with the corresponding arguments (storage classes get applied, variables are initialized, ...). Here's what possible instantiations of the templates declared in section 1 might look like:

```
ArrayOf!int

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

struct MyStruct { ... }
NameOf!(MyStruct) // "MyStruct" is a identifier, captured by alias

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

Minimalist!()
// or even:
Minimalist

OneOrMore!( int                    // FirstType is int.
          , double, string, "abc" // Rest is (double,string,"abc")
          )

ZeroOrMore!(int)               // Types is a 1-element tuple: (int)
ZeroOrMore!(int,double,string) // Types is (int,double,string)
ZeroOrMore!()                  // Types is the empty tuple: ()

Multiple!(int)               // Selects the one-arg version
Multiple!(int,double,string) // The three args version.
```

```
Multiple!()                    // Error! No 0-arg version
```

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

```
// 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.

// the transform function is accessible. Instantiated like this,
// it's a function from string to int.
auto i = Transformer!(string,double).transform("abc"); // i is an int
```

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

```
alias Transformer!(string,double) StoD;

auto i = StoD.transform("abc");
auto m = new StoD.Modificator("abc"); // StoD.Modificator is a class
                                      // storing a string and a double.
```

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, trying to instantiate many times the 'same' template (ie: with the same arguments) will only create one piece of code.

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

`void`

Note that `void` is a D type and, as such, a possible template argument for a type parameter. Just be careful because 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 some types.

3 Templates Building Blocks

Up to now, templates must seem not 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 templates use. This section will introduce you to your future best friends, the foundations on which your templates will be built.

3.1 The Eponymous Trick

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

```
template pair(T)
{
    T[] pair(T t) { return [t,t];} // declares only pair
}

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

template NameOf(alias name)
{
    enum string NameOf = name.stringof;
}

int foo(int i) { return i+1;}
auto s1 = NameOf!(foo); // s1 is "foo"
auto s2 = NameOf!(NameOf)
```

A limitation is that the eponymous trick works *only* if you define one (and only one) symbol. Even if the other symbols are private and such, they will break the eponymous substitution. This may change at a latter time, as the D developers have shown interest for changing this, but for the time being, you cannot do:

```
template ManyAlias(T, U)
    T[] firstArray;
    U[] secondArray;
    T[U] ManyAlias; // Halas, ET substitution doesn't work here
}

// Hoping for ManyAlias!(int,string).ManyAlias
ManyAlias!(int, string) = ["abc":0, "def":1];

ManyAlias!(int, string).firstArray = [0,1,2,3];
```

But, will you ask, what if I want client code to be clean and readable by using the eponymous trick, when at the same time I need to internally create many symbols in my template? In that case, create a secondary template with as many symbols as you need, and expose its final result through a (for example) `result` symbol. The first template can instantiate the second one, and refer only to the `.result` name:

```
/* your code */

// primary template
template MyTemplate(T, U, V)
{
    // eponymous trick activated!
```

```
    enum MyTemplate = MyTemplateImpl!(T,U,V).result;
}

// secondary (hidden) template
template MyTemplateImpl(T, U, V)
{
    // The real work is done here
    // Use as many symbols as you need.
    alias symbol1 ...
    enum otherName = ...
    enum result = ...
}

/* client code */
MyTemplate!(int,string,double[]) someValue;
(...)
```

You can find an example of this two-templates idiom in Phobos, for example in `std.functional.unaryFun` or `std.functional.binaryFun`.

3.2 Inner `alias`

A common use for templates is to do some type magics: 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.

```
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 5 on structs templates and section 6 on class templates.

3.3 static if

3.3.1 Syntax

The static if construct[2] let you decide between two code paths at compile time. It's not specific to templates (you can use it in other part 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)
{
     /* 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 to affect the outside and would drastically limit the interest of static if. So the curly braces are only there 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 9).

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

As you can see, no curly braces after static if and we are also using the eponymous trick (isDynamicArray is the only symbol defined by the template and its type is automatically deduced by the compiler), resulting in a very clean syntax. The is() 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 A).

3.3.2 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

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

ranges (ranges wrapping other ranges) will activate some functionality only if the wrapped range can support it, like this:

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

    R innerRange;

/* some code that exist in all instantiations of MyRange */
(...)

/* optional code */
static if (hasLength!R) // does innerRange has 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...
```

3.3.3 Nested `static if`s

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

```
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 26.

3.3.4 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:

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

Line 4 is the most interesting: with some `is` magic, `U` has been determined 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.

```
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(0, someString)`. The string is then emitted as a compiler error message.

Rank for Ranges: D has an interesting sequence concept that's called *range* and 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 (the '`U`') 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`:

```
import std.range;

template rank(T)
{
    static if (isInputRange!T)                // is T a range?
        enum rank = 1 + rank!(ElementType!T); // if yes, recurse
```

```
    else                                        // base case, stop there
        enum rank = 0;
}

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:

```
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!Range                     BaseElementType;
    else                         // at least range of ranges
        alias BaseElementType!(ElementType!(Range)) BaseElementType;
}
```

Line 4 is an example of `static assert` stopping compilation if we ever get into a bad code path. Line 8 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 inversing the process? Given a type `T` and a rank `r` (an `int`), we want to obtain `T[][]...[]`, with `r` levels of `[]`'s. A rank of 0 means producing `T` as the result type.

```
template NDimArray(T, int r)
{
    static if (r < 0)
        static assert(0, "NDimArray error: the rank must be positive.");
    else static if (r == 0)
        alias T NDimArray;
    else // r > 0
        alias NDimArray!(T, r-1)[] NDimArray;
}
```

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 as a template parameter.

```
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, its repeated composition. Here is what I mean by this:

```
string foo(string s) { return s ~ s;}

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

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:

```
template power(alias fun, uint exponent)
{
    static if (exponent == 0) // degenerate case -> id function
        auto power(Args)(Args args) { return args; }
    else static if (exponent == 1) // end-of-recursion case -> fun
        alias fun power;
    else
        auto power(Args...)(Args args)
        {
            return .power!(fun, exponent-1)(fun(args));
        }
}
```

.power

If you are wondering what's with the `.power` syntax on line 10, it's just that inside a template, the template's own name refers to the local (current) instantiation. So inside `power(Args...)` it 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, for delegates and for structs or classes that define the `()`(ie, `opCall`) operator and for function templates...[3]

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

Curried Templates?

No, I do not mean making them spicy, but separating the templates 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 14.

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.

3.4 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 A 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` and `Type == OtherType` restricts `Type` to be exactly `OtherType`.

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

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

// Say Array is a class, which has an ElementType type alias defined.
```

[3] 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 : Array)
{
    alias Array.ElementType ElementType;
}
```

Now, the `Type == AnotherType` syntax may seem strange to you: if you know you want to restrict `Type` to be *exactly* `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 based the 'most adapted' to the provided arguments. 'Most adapted' obeys some complicated rules you can find on the D Programming Language web site, 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. For

```
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
{ ... }

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

This idiom is used frequently 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.

3.5 Default Values

Like functions paramters, 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.

```
template Default(T = int, bool flag = false)
{ ... }

Default!(double);       // Instantiate Default!(double, false)
Default!(double, true); // Instantiate Default!(double, true) (Doh!)
Default!();             // Instantiate Default!(int, false)
```

A difference with function parameters default values is that, due to specializations (3.4) or IFTI (4.3), some parameters can be automatically deduced by the compiler. So, default templates parameters are not forced to be the last parameters:

```
template Deduced(T : U[], V = U, U)
{ ... }

Deduced!(int[], double); // U deduced to be int. Force V to be a double.
Deduced!(int[]); // U deduced to be int. V is int, too.
Deduced!(); // Error, T is not some array.
```

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 the sort function template found in std.algorithm. 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`.

4 Function Templates

4.1 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 section 3.1, if you define a function inside a template and use the template's own name, you can call it easily:

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

// call:
```

```
auto result = myFunc!(double,3)(3.1415);

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

Well, the true 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:

```
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`.

4.2 `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:[4]

```
auto morph(T, U, alias f)(U arg)
{
    static if (is(U == int) && is(T == class))
    {
        return new T(f(arg));
    }
    else static if (is(T == function))
    {} // void-returning function
    else static if (...)
    (...)
}
```

`auto ref`

A function template can have an `auto ref` return type. That means that for templates where the returned values are rvalues, the template will get the `ref`ed version. And the non-`ref` version if not.

[4] `auto` return functions used to have some limitations, like for example not appearing in docs. I remember having to code type-manipulating templates to get correct the return type.

4.3 IFTI

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

```
/* suppose the previous concatenate(A,B) function */
string res1 = concatenate(1, 3.14); // A is int and B is double

struct Foo {}
string res2 = concatenate("abc", Foo()); // A is string, B is Foo

/* suppose the previous select(string how = "max", Arg) function */
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. Them you can inherit, return...

We may encounter The Mantra again in this tutorial.

4.4 Example: Flattening Arrays and Ranges

Let's use what we have just seen in a concrete way. In D, you can manipulate 2D, 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:

```
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] );

assert( flatten([[[[[]]]]]) == [] ); // void[][][][][] -> void[]
```

So, studying the examples, we want a non-array or a simple array 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 concatenate the elements (with std.algorithm.reduce), thus losing one rank and call `flatten` again on the result;

```
import std.algorithm: reduce;

auto flatten(Arr)(Arr array)
{
    static if (rank!Arr <= 1)
        return array;
    else
    {
        auto r = reduce!"a~b"(array); // concatenates the elements
        return flatten(r);
    }
}
```

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 (5) with a factory function (5.2). You will find this as an example in section 5.8.

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]]]
```

4.5 Anonymous Function Templates

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

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

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 (page 19): 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`:

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

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 section 5.6).

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:

```
(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`, you cannot alias them to a symbol:

```
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:

```
template callTwice(alias fun)
{
    auto callTwice(T)(T t)
```

```
    {
        return fun(fun(t));
    }
}

alias callTwice!( (a){ return a+1;}) addTwo;

assert(addTwo(2) == 4);
```

Since they are delegates, they can capture local symbols, as long as these are defined at compile-time:

```
enum b = 3; // Manifest constant, initialized to 3

alias callTwice!( (a){ return a+b;}) addTwoB;

assert(addTwoB(2) == 2 + 3 + 3);
```

4.6 Functions Overloading

TODO: Write something on this.

4.7 Storage Classes

As seen in section 2, storage classes get applied to types during instantiation. It also works for function templates arguments:

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

int i = 10;
init(i);
assert(i == 0);
```

Should the need arises, that also means you can customize your storage classes according to templates 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:

```
// Has anyone a better example?
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;
        }
}
```

4.8 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 garanty concerning memory. And a `@system` function just laugh at you.
- The last property is `nothrow` which means you guarantee the function does not throw any exception.

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 does seem it works. 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).

4.9 `in` and `out` Clauses

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

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

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

4.10 Modifying Functions: Memoizing a Function

TODO: Should this section be moved in the struct templates section?

Let us use a template to wrap a function and provide some additional usefulness when calling the function. Memoizing is an interesting and useful example:

if the 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.

We have not seen struct templates yet (they are presented in section 5), but the following example should be easy to understand: the previous result 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.

```
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)()
{
    return Memoize!fun();
}
```

Usage is very simple:

```
int veryLongCalc(int i double d, string s) { ... }

auto vlcMemo = memoize!(veryLongCalc);

// calculate veryLongCalc(1, 3.1r4, "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.

```
enum MemoStoringPolicy {
    always,  // there is no tomorrow
    maximum  // sustainable growth
}

enum MemoDiscardingPolicy {
    oldest,   // only discard the oldest result
    fraction, // discard a fraction (0.5 == 50%)
    all       // burn, burn!
}

struct Memoize(alias fun,
               MemoStoringPolicy storing,
               MemoDiscardingPolicy discarding)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;

    static if (storing == MemoStoringPolicy.maximum)
    {
        Tuple!(PTT)[] argsQueue;
        size_t maxNumStored;
    }

    static if (discarding == MemoDiscardingPolicy.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,
            RT result = fun(args);    // compute the result and store it.
            memo[tuple(args)] == result;
            return result;
        }
    }
}
```

And a few factory function to help creating those `Memoize` structs:

```
// No runtime arg -> always store
Memoize!(fun, MemoStoringPolicy.always, MemoDiscardingPolicy.all)
memoize(alias fun)()
{
    return Memoize!(fun,
                    MemoStoringPolicy.always,
                    MemoDiscardingPolicy.all)();
}

// One runtime size_t arg -> maximum store / discarding all
Memoize!(fun, MemoStoringPolicy.maximum, MemoDiscardingPolicy.all)
memoize(alias fun)(size_t max)
{
    return Memoize!(fun,
                    MemoStoringPolicy.maximum,
                    MemoDiscardingPolicy.all)(max);
}

// Two runtime args (size_t, double) -> maximum store / discarding a fraction
Memoize!(fun, MemoStoringPolicy.maximum, MemoDiscardingPolicy.fraction)
memoize(alias fun)(size_t max, double fraction)
{
    return Memoize!(fun,
                    MemoStoringPolicy.maximum,
                    MemoDiscardingPolicy.fraction)(max, fraction);
}

// One compile-time argument (discarding oldest), one runtime argument (max)
Memoize!(fun, MemoStoringPolicy.maximum, discarding)
memoize(alias fun, MemoDiscardingPolicy discarding == MemoDiscardingPolicy.oldest)
(size_t max)
{
    return Memoize!(fun,
                    MemoStoringPolicy.maximum,
                    discarding)(max);
}
```

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:

```
int veryLongCalc(int i double d, string s) { ... }

// 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, MemoDiscardingPolicy.oldest)(20);
```

4.11 Modifying Functions: Currying a Function

```
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..$]).With!(U[1..$]);
    }
}


struct Curry(alias fun, int index = 0)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;
    PTT args;

    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;
}
```

5 Struct Templates

5.1 Syntax

As you might have guessed, declaring a struct template is done like this:

```
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:

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

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

```
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 iteration, have a look there.

5.2 Factory Functions

Now, there is one limitation: structs constructors do not do activate IFTI (4.3) 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 (see 5.5) 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`:

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

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.

5.3 Giving Access to Inner Parameters

As was said in section 3.2, 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:

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

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

auto t0 = tree("abc");
alias typeof(t0) T0;

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

5.4 Templated Member Functions

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

Mapping on a Tree Let us use that 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.[5]

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 for 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]

```
/* rest of Tree code */
    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);
    }
```

[5] We could easily make that a free function, but this *is* the member function section.
[6] The difference with Phobos `map` is that it's not lazy.

Let's use it:

```
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
int addOne(int a) { return a+1;}
auto t3 = t2.map!(add(1));

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

assert(t3.value = 3);

// Converting all values to strings
import std.conv:to;
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]

```
/* rest of Tree code */
auto fold(alias ifLeaf, alias ifBranch = ifLeaf, S)(S seed)
{
    if (isLeaf)
    {
        return ifLeaf(value, seed);
    }
    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 care for the seed.

```
auto sumLeaf(T, S)(T value, S seed)
{
    return value + seed;
}

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

auto 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.

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

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

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

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

And as a last use, why not build a tree?
TODO: Write just that.

5.5 Templated Constructors

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

```
struct S(T)
{
    this(U)(U u) { ... }
}

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

As you can see, IFTI (4.3) works for constructors. `U` is automatically deduced, though you have to indicate `T` in this case. The previous 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:

```
struct Holder(Type)
{
    Type value;

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

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`.

5.6 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:

```
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 4.5? 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?

```
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. Of course, there is a solution: just return a struct with a template member function (5.4), in that case the `opCall` operator:

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

        auto opCall(B)(B b)
```

```
        {
            return to!string(a) ~ to!string(b);
        }
    }

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

    return c;
}

auto c = concatenate(3.14);
auto cc = c("abc");
assert(cc == "3.14abc");
```

See section 11 for more on operators overloading.

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

```
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);}

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

5.7 Template This Parameters

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. If find this useful mainly for mixin templates (12), where you do not know the enclosing type beforehand. Please see this section for some examples.

TODO: Some kind of example?

5.8 Example: a Concat / Flatten Range

We will use what we've seen for a struct template a lazy range that flattens ranges of ranges into linear ranges. Remember the `flatten` function from section

4.4? 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: Which returns the range's first element.

popFront: Which discards the first element and advances the range by one step.

empty: Which returns `true` if the range has no more element, `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 iterate on the inner elements. Let's begin with the very basis:

```
import std.range;

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 page 12. We haven't seen template constraints yet (they are described in section 8), but they would be a good fit there 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.

```
struct Flatten(Range)
{
    alias ElementType!Range    SubRange;
    alias ElementType!SubRange Element;

    Range range;
    SubRange subRange;

    this(Range _range) {
        this.range = range;
        discardEmptySubRanges();
    },

    Element front() { return subRange.front;}

    bool empty() { return range.empty;}

    void popFront() {
        if (!subRange.empty) subRange.popFront;
        discardEmptySubRanges();
    }

    void discardEmptySubRanges() {
        while(subRange.empty && !range.empty) {
            range.popFront;
            if (!range.empty) subRange = range.front;
        }
    }
}
```

- I cheat a little bit with D standard bracing style, because it eats vertical space like there is no tomorrow.
- We begin on line 3 and 4 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 correctly initialize the struct.
- `front` returns a subrange element.
- The `discardEmptySubRanges` function does what it says on the can: we do not iterate on empty subranges.

Let see if this template works, by creating an infinite range and giving it to `flatten`:

```
import std.range;

auto cy = cycle(["Hello", "World"); // "Hello","World","Hello","World",...
auto flattened = flatten(cy);
assert(flattened.front == 'H');

auto takeTwelve = take(flattened, 12);
assert(array(takeTwelve) == "HelloWorldHe");
```

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:

```
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:

```
auto rank3 = [[[0,1,2],[3,4,5],[6]]
             ,[[7],[],[8,9],[10,11]]
             ,[[],[]]
             ,[[12]]  ];

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

auto reallyFlat = flatten(flat);
assert(equal( reallyFlat, flat )); // No need to insist

import std.string, std.algorithm;

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 (this section, 5), `static if` (3.3), inner member alias (3.2), factory functions (5.2) and IFTI (4.3).

6 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 interesting example of class templates that could be used throughout the section, I'm game.

6.1 Syntax

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

```
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:

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

Interface Templates?
Yes you can. See section 7

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:

```
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;
}
```

With this, `MyClass` can inherit either from `Object`, D root to the class hierarchy, or 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 (see section 10.5 for an example).

6.2 Methods Templates

An object's methods are nothing more than delegates with a reference to the local `this` context. As seen for structs (5.4), methods can be templates too.

TODO: Something on overriding methods with templates. TODO: Find some interesting method example. Man, I do not do classes.

6.3 `invariant` clauses

In the same family than `in`/`out` clauses for functions (section 4.9), 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.

```
class MyClass(T, U, V)
{
    (...)

    invariant
    {
        static if (/* some condition on T,U,V */ )
        {
            /* invariant code */
        }
        else
        { /* empty invariant */ }
    }
}
```

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

6.4 Inner Classes

It's the same principle than for structs (5.6). You can define inner classes using the template parameters. You can even give them methods templates that use other template arguments. There is really nothing different from inner structs.

6.5 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.

```
// stores a function and a default return value.
auto acceptor(alias fun, D)(D default)
{
    return new class
    {
        auto opCall(T)(T t)
        {
            static if (__traits(compiles, fun(T.init)))
                return fun(t);
            else
                return default;
        }
    };
}

int add1(int i) { return i+1;}
auto accept = acceptor!(add1)(0);
auto test1 = accept(1);
assert(test1 == 2);

auto test2 = accept("abc");
assert(test2 == 0); // default value
```

TODO: Test this!
For `__traits(compiles, ...)`, see here.

6.6 Parameterized Base Class

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

```
interface ISerializable
{
    size_t serialize() @property;
}

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

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:

```
enum SerializationPolicy { policy1, policy2 }

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

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 (12) and wrapper templates (13.3) for other idioms built around the same need.

6.7 Another Example

```
/**
Timon Gehr timon.gehr@gmx.ch via puremagic.com
This is an useful pattern. I don't have a very useful example at hand, but this one should
*/
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;
	}
}

class ConcreteCell(T): AddGetter!(AddSetter!(Cell!T)){}
class OneUpDoubleSetter(T): OneUpCell!(DoubleCell!(AddSetter!(Cell!T))){}
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"

	// ...
}
```

6.8 The Curiously Recurring Template Pattern

```
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 (12) 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.

6.9 Example

TODO: An example with duplicator, a template that takes a class and creates another class which is its clone: same base, same interfaces, etc.

TODO: What about an expression template?

7 Other Templates?

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

7.1 Interface Templates

The syntax is exactly what you might imagine:

```
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 (see page 19): interface templates are *not* interfaces.

7.2 Union Templates

Here is the syntax, no suprise there:

```
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.

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

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

Part II
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 Part I into Part II and the other way round.

8 Constraints

Templates 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 templates specializations (see 3.4). 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.

8.1 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 (A) 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.

```
template Constrained(T)
    if (is(T : int)) { ... } // #1
template Constrained(T)
    if (is(T : float)) { ... } // #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

Constrained!(byte) // #1
```

```
Constrained!(string) // Error, no declaration fits (string)
Constrained!(int,string) // #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 4, 5, 6 and 7. 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.

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

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`. So you should not use `argument` in your constraint. If you need a value of type `T`, use `T.init`. For

```
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));
}
```

8.2 Constraints Usage

Constraints come from the same idea than 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 4.4.

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 section 9. Usage is quite simple:

```
import std.range;

struct RangeWrapper(Range)
    // Does Range comply with the input range 'concept'?
    if (isInputRange!Range)
{
```

[9] Ranges are overdue a tutorial.

```
   /* 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) if (isInputRange!Range)
{
    return RangeWrapper!(Range)( ... );
}
```

In fact, it's a bit like a sort of 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[10] 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).

8.3 Constraints Limits

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

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

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

Temp!int // 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:

[10] I think *compiler* and *complain* must have the same root.

```
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 (3.4) are sometimes more user-friendly: most of what I want to impose and check on my types cannot be done by specializations.

8.4 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 (3.4) 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` (3.3). 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 effet 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.

9 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 a section A.2, 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)`.

9.1 Testing for a member

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

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

9.2 Testing for operations

As seen in previous sections (5.8, 8.2), 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:

```
template hasArithmeticOperations(Type)
{
    static if (__traits(compiles,
```

```
                        {
                            Type t;
                            t + t; // addition
                            t / t; // substraction
                            t * t; // multiplication
                            t / t; // division
                            +t;    // unary +
                            -t;    // unary -
                        }))
        enum bool hasArithmeticOperations = true;
    else
        enum bool hasArithmeticOperations = false;
}

static assert(hasArithmeticOperation!int);
static assert(hasArithmeticOperation!double);

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

As you can see, you can test for any type of D code, which means it's *much* more powerful than templates specializations (3.4) or the `is` expression (Appendix A).

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 (16) and CTFE (17) in Part III.

9.3 Completing the `Flatten` range:

Let's come back to `Flatten` from section 5.8. 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 than 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 5.8, 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.

10 Template Tuple Parameters

10.1 Definition and Basic Properties

And now comes one of my favourite subjects: template tuple parameters. As seen in section 1 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 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):

```
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:

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

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` as `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:

```
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), as you'll see in sections 10.3 and 10.5. 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 28 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.

```
// keeping the same t and T than the previous examples.
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);
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.traits.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` and `tuple` are pre-defined struct/function templates in Phobos that gives a simple syntax to manipulate tuples and return them from functions.

10.2 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.[11]

`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.

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

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

10.3 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`:

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

[11] Look: `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;
}

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:

```
import std.range, std.algorithm;

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

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:

```
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`.[12]
- 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 (3.1), 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.

10.4 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:

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

template nameOf(alias a)
{
    Enum string nameOf = to!string(a);
}

static assert(nameOf!(double[]) == "double[]");
static assert(nameOf!(nameOf) == "nameOf");
```

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

[12] You might not like it, but it's cleaner mathematically this way.

```
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.

10.5 Example: Inheritance Lists

TODO: All this section should be rewritten. The compiler is more accepting than I thought.

Using class templates (6.1), 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:

```
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!MyOtherClass { ... }
```

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.

There is a catch if you use it again on a derived class:

```
// Error! I,J and K are already listed through MyClass
Class ErrorClass : Inheritance!(MyClass) { ... }
```

The interfaces are already listed in `MyClass` while `Inheritance` injects them again. That gets us a compilation error, because in D you cannot put an interface twice in an inheritance list. We have to do something a little more complicated: given an inheritance type list, we must eliminate all double interfaces.

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

```
template Eliminate(Type, TargetTuple...)
{
    static if (TargetTuple.length == 0) // Tuple exhausted,
        alias TargetTuple Eliminate;    // job done.
    else static if (is(TargetTuple[0] : Type))
        alias Eliminate!(Type, TargetTuple[1..$]) Eliminate;
    else
        alias
TypeTuple!(TargetTuple[0], Eliminate!(Type, TargetTuple[1..$])) Eliminate;
}

alias TypeTuple!(int,double,int,string) Target;
alias Eliminate!(int, Target) NoInts;
static assert(is( NoInts == TypeTuple!(double, string) ));
```

The only difficulty is on line 9: 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.

```
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;
}

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.

The last piece of the puzzle is to get a given class inheritance list. The `is` expression give us that by way of types specializations (A.4):

```
template SuperList(Class) if (is(Class = class))
```

```
{
    static if (is(Class list == super))
        alias TypeTuple!(list) SuperList;
    else // Object
        alias TypeTuple!() SuperList;
}
```

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, as seen in section 3.1.

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

template CheckedImpl(Base)
{
    // Get the inheritance list, getting rid of the base class
    static if (SuperList!(Base).length > 0)
        alias SuperList!(Base)[1..$] InList;
    else // Object is the only class with a zero-length inh. list
        alias SuperList!(Base)       InList;

    alias TypeTuple!( InList
                    , Inheritance!(Base)[1..$]) AllInterfaces;

    alias TypeTuple!( Inheritance!(Base)[0] // base class
                    , NoDuplicates!(AllInterfaces)) Result;
}

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

11 Operator Overloading

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

11.1 Syntax

Table 1 gives you the operator that you can overload and which function template you must define:

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

Category	Operators	Template to define

Table 1: Operator Overloading

11.2 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 this these overloading tend to use string mixins, I'll use them even though they are introduced only on section 16. The basic idea is: string mixins paste code (given as a compile-time string) where they are put.

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

    auto opBinary(string op, 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`.

11.3 Special Case: `in`

11.4 Special Case: `cast`

12 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.

12.1 Syntax

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

```
/* 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.

12.2 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.

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 11? Here is a mixin containing concatenating functionality:

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

    Tuple!(U, This) opBinaryRight(string op, this This)(This u)
    if (op == "~")
    {
        return tuple(u, this);
    }
}
```

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:

```
struct S
{
    /* some code */

    mixin Concatenate;
}

S s,t,u;

auto result = s ~ t ~ u;
assert(result == tuple(s, tuple(t,u)));
```

In this particular case, we should test for tuples already containing the current type and flatten them, so as get `tuple(s,t,u)`.

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.

13 opDispatch

13.1 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 than an operator:

```
... opDispatch(string name, Args)(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.

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

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 16 on 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.

13.2 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

```
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 (mixin("is(typeof(this." ~ name ~ "))"))
            mixin("return " ~ name ~ ";");
        else
            static assert(0, "GetSet Error: no member called " ~ name);
    }

    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 "name" can be assigned to. this.name = Arg.init
        static if (__traits(compiles,
                            mixin("this." ~ name ~ " = Arg.init;")))
            mixin("return " ~ name ~ ";");
        else
            static assert(0, "GetSet Error: no member called " ~ name);
    }
}

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

gs.settheString("abc");
writeln(gs.gettheString); // "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 12).

13.3 Wrapper Templates

We've seen how to inject code with mixin templates (12) or use template class inheritance to modify you classes' code (10.5). We've also seen how you can define a wrapper struct around a range to expose a new iteration scheme for its element (5.8). 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. If yes, call

X X X X X X X X

14 Templates All the Way Up

TODO: Write this section.

Double-decker templates.

Curried templates.

To separate two type tuples.

15 `__FILE__` and `__LINE__`

In section 3.5, 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:

```
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 (2).

```
// file thefile.d
module thefile;

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:

```
template Debug(alias toTest, file == __FILE__, 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);
}

/* Usage */
Debug!(templateToBeTested).With!(Arguments);
```

That way, no need to modify your beautiful templates.

TODO: Test that.

Part III
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 (16), compile-time function evaluation (17) and `__traits` (18), 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!

16 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.

16.1 Syntax

The syntax is slightly different from mixin templates (12):

```
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.

16.2 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 alreday saw that in section 11 on operator overloading and section 13 on `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 mine;
import named;

mixin(Named!"First");  // creates struct First { ... }
mixin(Named!"Second"); // and struct Second { ... }

First f1, f2;
Second s1;

assert(is( typeof(s1) == mine.First));
```

Here comes the generating code:

```
module named;

template Named(string name)
{
    enum string Named = "struct " ~ name ~ " { "
                        ~ "/+ some code +/"
                        ~ " }";
}

/* For example, name == "First" ->
   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 module containing `Named`, 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 in different modules, the code must be organized a bit differently: the structs must be generated in the template module (for example):

```
module named;

template Named(string name)
{
    alias NamedImpl!(name).result Named;
}

template NamedImpl(string name)
{
    enum string Named = "struct " ~ name ~ " { "
                        ~ "/+ some code +/"
                        ~ " }";
    mixin(Named);
    mixin("alias " ~ name ~ " result;");
}
```

```
module mine;
import named;

named!"First" f1, f2;
named!"Second" s1;
```

Usage is a different, as you can see. In this case, `First` is generated inside `NamedImpl` 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 named;

template Named(string name)
{
    alias NamedImpl!(name).result Named;
}

template NamedImpl(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` (3.3):

```
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);
}

S!(GetSet.yes, int) gs;

/* Generates:

struct S!(GetSet.yes, int)
{
    private int value;
    int get() @property { return value;}
    void set(int _value) { value = _value;}
}
```

```
*/

gs.set(1);
assert( gs.get == 1);
```

16.3 Limitations

Code crafting is still a bit awkward, because I haven't introduced CTFE yet (see 17). 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 4.1 to generate the error message for `select`. Or you can put strings between `q{` and `}`.

17 Compile-Time Function Evaluation

17.1 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. The limits to what can be done with CTE functions are pushed farther away at each new release. At the time of this writing, the limitations are mostly: no classes and no exceptions (and so, no `enforce`).[13] All the `foreach`, `while`, `if`/`then`/`else` statements, arrays manipulation, struct manipulation, function manipulation... are there. You can even do pointer arithmetics!

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:

[13] No doubt this sentence will become obsolete rapidly.

17.2 __ctfe

17.3 Templates and CTFE

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:

17.4 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.

Concretly, here is the getting-setting code from section 16.2, reloaded:

```
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));
}

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 in section 16.2: 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`.

17.5 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:

```
alias interpolate!"struct #0 { #1 value; #0[#2] children;}" makeTree;

enum string intTree = makeTree("IntTree", "int", 2);
enum string doubleTree = makeTree("DoubleTree", "double", "");

assert(intTree
       == "struct IntTree { int value; IntTree[2] children;}");
assert(doubleTree
       == "struct DoubleTree { double value; IntTree[] 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.

```
template interpolate(string code)
{
    string interpolate(Args...)(Args args) {
        string[] stringified;
        foreach(index, arg; args) stringified ~= to!string(arg);

        string result;
        int i;

        while (i < code.length) {
            if (code[i] == '#') {
                int j = 1;
                int index;
                auto zero = to!int('0');
                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.

17.6 Example: extending `std.functional.binaryFun`

std.functional has two really interesting templates: `unaryFun` and `binaryFun`.

```
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));
}
```

18 __traits

The general __traits syntax can be found online here. Traits are basically another compile-time introspection tool, complementary to the is expression (see appendix A). 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 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.

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:[14]

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 2: Comparison between __traits and other introspection tools

More interesting in my opinion is using __traits to get new information

[14] As with any other affirmation in this document, readers should feel free to prove me wrong. That shouldn't be too difficult.

from the type. These are really different from other introspection tools and I will deal with them in more detail right now.

- identifier (example with foo)
- getMember (returns an expression)
- getOverloads (returns an array of overloads)
- getVirtualFunctions (returns an arrays of the virtual overloads)
- parent (get the parent directly). Example: recurse upwards.
- allMembers (tuple of string literals, names of members of a type. If it's a class, it will include inherited members. No repetition -> getOverloads). 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("a",5))
- derivedMembers (tuple of string literals, names of members of type. If it's a class it will *not* include inherited members. No repetition -> getOverloads)

Example: iterating on members, extracting types from an aggregate.

19 Wrapping it all Together

Using traits to crack a type open.
templated CT functions to generate strings to be mixed-in.
Example: cloning a type Example: naryFun?

Part IV
Examples

20 Type Sorcery

20.1 Mapping, Filtering and Folding Types

TODO: Say something about mapping with a typetuple-returning template

```
template Qual(T)
{
    alias TypeTuple!(T,T[],const(T)[],immutable(T)[], shared(T)[],
const T, immutable T, shared T, const T[], immutable T[], shared T[]) Qual;
}

alias TypeTuple!(
      bool,ubyte,byte,ushort,short,uint,int,ulong,long,
      float,double,real,char,wchar,dchar) ValueTypes;

alias staticMap!(Qual,ValueTypes) QualifiedTypes;
```

20.2 Interspersing Types, Crossing Types

```
/**
Helper template. Given T0, T1, T2, ..., Tn, Tn+1, ... T2n, will returns
the interleaving of the first part with the second part:
T0, Tn+1, T1, Tn+2, ... Tn, T2n
It's fragile: no test, etc.
A better way to do this would be as a two-steps template: Interleave!(T...).With!(U...)
*/
template Interleave(T...)
{
    static if (T.length > 1)
        alias TypeTuple!(T[0], T[$/2], Interleave!(T[1..$/2], T[$/2+1 .. $])) Interleave;
    else
        alias T Interleave;
}
```

21 Relational Algebra

Inspiration for this example comes from This blog article.

TODO: 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..

22 Cloning, sort of

TODO: (Elsewhere) creating a class from a struct?

23 Recording Successive States

From Andrej Mitrovic.

```
import std.stdio;
import std.traits;

struct Shape
{
    int x, y;
    void foo(int val) { x += val; }
    int bar(int val) { y += val; return y; }
}

struct RecorderImpl(T)
{
    T t;
    T[] t_states;

    this(T t)
    {
        this.t = t;
        t_states ~= t;
    }

    auto opDispatch(string method, Args...)(Args args)
    {
        static if (mixin("is(ReturnType!(t." ~ method ~ ") == void)"))
        {
            mixin("t." ~ method ~ "(args);");
            t_states ~= t;
        }
        else
        {
            mixin("auto result = t." ~ method ~ "(args);");
            t_states ~= t;
            return result;
        }
    }

    auto opIndex(size_t index)
    {
        assert(index < t_states.length);
        return t_states[index];
    }

    auto opSlice(size_t lowerBound, size_t upperBound)
    {
        return t_states[lowerBound..upperBound];
    }
```

```
    auto opSlice()
    {
        return t_states[];
    }
}

auto Recorder(T)(T t)
{
    return RecorderImpl!T(t);
}

void main()
{
    auto shape = Recorder(Shape(0, 0));

    shape.foo(5);
    shape.bar(5);

    writeln(shape[0]);
    writeln(shape[]);
}
```

24 Fields

From Jacob's Carlborg Orange.

TODO: Test typeof(s.tupleof)

```
/**
 * 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)
const fieldsOfImpl = [""];

else static if (T.tupleof.length - 1 == i)
const fieldsOfImpl = [T.tupleof[i].stringof[1 + T.stringof.length + 2 .. $]];
```

```
else
const fieldsOfImpl = T.tupleof[i].stringof[1 + T.stringof.length + 2 .. $] ~ fieldsOfImpl!
}
```

25 Extending an enum

```
string EnumDefAsString(T)() if (is(T == enum)) {
  string result = "";
  foreach (e; __traits(allMembers, T)) {
      result ~= e ~ " = T." ~ e ~ ",";
  }
  return result;
}

template ExtendEnum(T, string s) if (is(T == enum) &&
is(typeof({mixin("enum a{"~s~"}");}))) {
  mixin("enum ExtendEnum {" ~
      EnumDefAsString!T() ~ s ~
  "}");
}

unittest {
  enum bar {
      a = 1,
      b = 7,
      c = 19
  }

  import std.typetuple;

  alias ExtendEnum!(bar, q{ // Usage example here.
      d = 25
  }) bar2;

  foreach (i, e; __traits(allMembers, bar2)) {
      static assert( e == TypeTuple!("a", "b", "c", "d")[i] );
  }

  assert( bar2.a == bar.a );
  assert( bar2.b == bar.b );
  assert( bar2.c == bar.c );
  assert( bar2.d == 25 );

  static assert(!is(typeof( ExtendEnum!(int, "a"))));
  static assert(!is(typeof( ExtendEnum!(bar, "25"))));
}

//(Simen Kjaeraas)
```

26 Static Switching

TODO: What, no compile-time switch? Let's create one. Example of: tuples, type filtering (in constraints), recursion, etc.

```
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..$]);
}
```

27 Gobble

```
struct Gobbler(T...)
{
    T store;
    Gobbler!(T, string,U) opBinary(string op, U)(U u) if (op == "~")
    {
        return Gobbler!(T,string, U)(store, op, u);
    }
}

Gobbler!() gobble() { return Gobbler!()();}
```

28 A Polymorphic Tree

29 Polymorphic Association Lists

Usage: a bit like Lua tables: structs, classes (you can put anonymous functions in them?), namespaces. Also, maybe to add metadata to a type?

```
template Half(T...)
{
    static if (T.length <= 1)
        alias TypeTuple!() Half;
    else
        alias TypeTuple!(T[0], Half!(T[2..$])) 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)
    {
        static if ((staticIndexOf!(a, Keys) == -1) && (T.length % 2 == 1)) // key not foun
            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)];
    }
}

alias AList!( 1,     "abc"
            , 2,     'd'
            , 3,     "def"
            , "foo", 3.14
            ,        "Default") al;

writeln("Keys: ", al.Keys);
writeln("Values: ", al.Values);
writeln("at!1: ", al.at!(1));
writeln("at!2: ", al.at!(2));
writeln("at!\"foo\": ", al.at!("foo"));
writeln("Default: ", al.at!4);
```

30 Expression Templates

31 Wrapping a Function

Making it accept a tuple, for example.

```
template tuplify(alias fun)
{
    auto tuplify(T...)(Tuple!T tup)
```

```
    {
        return fun(tup.expand);
    }
}
```

Another interesting (and much more complicated) example is `juxtapose`.

32 Mapping n ranges in parallel

```
// 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 to take more than one function as template argument. 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))
{ ...
```

See the problem? 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))
    {...}
}
```

Final code:

```
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));
    }
}
```

Give an example with `max`, it works!
And here is the n-ranges version of `std.algorithm.filter`:

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

33 Statically-Checked Writeln

34 Extending a Class

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)`.

```
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.

35 Pattern Matching With Functions

36 Generating a Switch for Tuples

Case 0:, etc.

37 Tuples as Sequences

37.1 Mapping on Tuples

37.2 Filtering Tuples

```
(1, "abc", 2, "def", 3.14)
->
((1,2),("abc","def"),(3,14))
```

Appendices

A A Crash Course on the `is(...)` Expression

A.1 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 very useful in conjunction with `static if` (see section 3.3 and template constraints (see section 8). 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`.

A.2 `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

```
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:

```
bool validCall(alias fun, Args...)(Args args)
{
    static if (is( typeof({ /* code to test */
                            fun(args);
                            /* end of code to test */
                          }())))
        return true;
    else
        return false;
}

// Usage:
T add(T)(T a, T b) { return a+b;}

assert( validCall!add(1, 2.3));   // generates add!(double)
assert(!validCall!add(1, "abc")); // no template instantiation possible

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

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:

```
__traits(compiles, { /* some code */ })
```

`__traits` is another of D numerous Swiss Army knife constructs. You can find the `compiles` documentation here. Section 18 is dedicated to it.

A.3 `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)
```

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

```
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 3.2 and 5.3), 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.Tuples 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...)
(...)                                          ^^^^^^^^^^^^^
```

But sometimes D channels its inner perl and, lo! There is more than one way to do it! You can use IFTI (see 4.3) 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`:

```
template isTuple(T)
{
    static if (is(typeof({
              void tupleTester(InnerTypes...)(Tuple!(InnerTypes) tup) {}
              T.init possibleTuple;
              tupleTester(possibleTuple);
              }())))
        enum bool isTuple = true;
    else
        enum bool isTuple = false;
}
```

Line 4 defines the function template `tupleTester`, that only accepts `Tuple`s as arguments (even though it does nothing with them). We create something of type `T` on line 5, 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.

A.4 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 table 3.

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 or 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
`immutable`	`Type` is immutable	The type
`shared`	`Type` is shared	The type

Table 3: 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:

```
import std.algorithm;

template make(alias aggregate)
    if (is(typeof(aggregate) == class )
     || is(typeof(aggregate) == struct))
{
    auto make(Args...)(Args args)
    {
        alias typeof(aggregate) A;
```

```
        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;}}

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(equal(structRange, [new C(0), new C(1), new C(2), new C(3)]));
```

You can find another example of this kind of `is` in section 6, with the `duplicator` template.

Index