D Features in the D Standard Library

Ali Çehreli

DConf 2022 • August 1 • London
Introduction

• Main idea from Andrei Alexandrescu:

"take some piece of smart code ([...]) and dissect it to show how various features are put to work together to great effect"
Introduction

• Main idea from Andrei Alexandrescu:

"take some piece of smart code ( [... ] ) and dissect it to show how various features are put to work together to great effect"

• Some help from the Turkish D community
Introduction

• Main idea from Andrei Alexandrescu:

"take some piece of smart code ([…]) and dissect it to show how various features are put to work together to great effect"

• Some help from the Turkish D community

• There will be walls of code
Introduction

• Main idea from Andrei Alexandrescu:

  "take some piece of smart code ([...]) and dissect it to show how various features are put to work together to great effect"

• Some help from the Turkish D community
• There will be walls of code
• Will touch on iota, parallel, static if, std.concurrency.receive, and SumType
Introduction

• Main idea from Andrei Alexandrescu:

"take some piece of smart code ([...]) and dissect it to show how various features are put to work together to great effect"

• Some help from the Turkish D community
• There will be walls of code
• Will touch on iota, parallel, static if, std.concurrency.receive, and SumType
• Can be overwhelming:

```cpp
auto iota(B, E)(B begin, E end)
if (!isIntegral!(CommonType!(B, E)) &&
   !isFloatingPoint!(CommonType!(B, E)) &&
   !isPointer!(CommonType!(B, E)) &&
   is(typeof((ref B b) { ++b; })) &&
   (is(typeof(B.init < E.init)) || is(typeof(B.init == E.init)))) )
{
   // ...
}
```
Introduction

• Main idea from Andrei Alexandrescu:

"take some piece of smart code ([...]) and dissect it to show how various features are put to work together to great effect"

• Some help from the Turkish D community
• There will be walls of code
• Will touch on iota, parallel, static if, std.concurrency.receive, and SumType
• Can be overwhelming:

    ```cpp
    auto iota(B, E)(B begin, E end)
    if (!isIntegral!(CommonType!(B, E)) && 
        !isFloatingPoint!(CommonType!(B, E)) && 
        !isPointer!(CommonType!(B, E)) && 
        is(typeof((ref B b) { ++b; })) && 
        (is(typeof(B.init < E.init)) || is(typeof(B.init == E.init))) )
    {
        // ...
    }
    ```

• The numbers at the corners of the slides are number of steps, not number of slides.
D is excellent

With the killer feature of a collection of adjectives:

• Simpler
• Safer
• More correct
• Faster
• Time saving
• Sane
• Has a great community
• ...

D is excellent

With the killer feature of a collection of adjectives:

• Simpler
• Safer
• More correct
• Faster
• Time saving
• Sane
• Has a great community
• ...

Emergent properties:

• Pragmatic
• Refactorable (Moldable)
• Huge amount of unwritten code
• Fun
• ...

The standard library is Phobos

As of dmd 2.100.0, there are 54 std modules:

std.algorithm std.array std.ascii std.base64 std.bigint std.bitmanip std.checkedint std.compiler std.complex
std.concurrency std.container std.conv std.csv std.datetime std.demangle std.digest std.encoding std.exception
std.file std.format std.functional std getopt std.int128 std.json std.math std.mathspecial std.meta std.mmfile
std.numeric std.outbuffer std.parallelism std.path std.process std.random std.range std.regex std.signals
std.socket std.stdint std.stdio std.string std.summary std.system std.traits std.typecons std.typeutil std.uni
std.uri std.utf std.variant std.xml std.zip std.zlib

As well as some core and etc, and the object modules:

core.atomic core.attribute core.bitop core.builtins core.checkedint core.cpuid core.demangle core.exception
core.int128 core.lifetime core.math core.memory core.runtime core.simd core.thread core.time core.vararg
core.volatile

etc.c.zlib etc.c.curl etc.c.odbc.sql etc.c.odbc.sqltypes etc.c.odbc.sqlgeom etc.c.odbc.sqlquery etc.c.sqlite3
etc.linux.memoryerror

object
No special compiler keyword

The standard library is written in the D programming language.
A readable standard library

Accessible to all; e.g. on an Arch-based Linux distribution:

/usr/include/dlang/...
A readable standard library

Accessible to all; e.g. on an Arch-based Linux distribution:

/usr/include/dlang/...

An excerpt:

/usr/include/dlang/dmd/std/range/package.d

// ...
module std.range;
// ...

auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E))) || isPointer!(CommonType!(B, E)))
   && isIntegral!S)
{
   // ...
   void popFront()
   {
      assert(!empty);
      if (current == last) step = 0;
      else current += step;
   }
   // ...
}
Ranges

Phobos uses the range abstraction.
"Values from 0 to 10 (exclusive), increment by 2:"

\[
\text{iota}(0, 10, 2) \quad // \quad \text{Generates 0, 2, 4, 6, and 8}
\]
A range example

A limited \textit{iota} wannabe:

```c
struct MyNumbers {
    int begin;       // LIMITATION: Works only with int; better be templatized
    int end;
    int step;

    bool empty() {
        return begin >= end;
    }

    int front() {
        return begin;
    }

    void popFront() {
        begin += step;  // INACCURACY: For floating point types (assuming templatized)
    }
}
```
A range example

A limited iota wannabe:

```cpp
struct MyNumbers {
  int begin;     // LIMITATION: Works only with int; better be templatized
  int end;
  int step;

  bool empty() {
    return begin >= end;
  }

  int front() {
    return begin;
  }

  void popFront() {
    begin += step;    // INACCURACY: For floating point types (assuming templatized)
    begin += step;
  }
};
```

Its convenience function:

```cpp
MyNumbers myNumbers(int begin, int end, int step) {
  return MyNumbers(begin, end, step);
}
```
A range example

A limited iota wannabe:

```cpp
struct MyNumbers {
    int begin;  // LIMITATION: Works only with int; better be templatized
    int end;
    int step;

    bool empty() {
        return begin >= end;
    }

    int front() {
        return begin;
    }

    void popFront() {
        begin += step;  // INACCURACY: For floating point types (assuming templatized)
    }
};
```

Its convenience function:

```cpp
MyNumbers myNumbers(int begin, int end, int step) {
    return MyNumbers(begin, end, step);
}
```

A unit test:

```cpp
unittest {
    assert(myNumbers(0, 10, 2).equal([0, 2, 4, 6, 8]));
}
```
With a Voldemort type

Moving the `struct` into the convenience function:

```cpp
auto myNumbers(int begin, int end, int step) {
    struct MyNumbers {
        // This time, no members; uses the parameters.
        bool empty() {
            return begin >= end;
        }

        int front() {
            return begin;
        }

        void popFront() {
            begin += step;
        }
    }

    return MyNumbers();
}
```
With a Voldemort type

Moving the `struct` into the convenience function:

```cpp
auto myNumbers(int begin, int end, int step) {
    struct MyNumbers {
        // This time, no members; uses the parameters.
        bool empty() {
            return begin >= end;
        }
        int front() {
            return begin;
        }
        void popFront() {
            begin += step;
        }
    }
    return MyNumbers();
}
```

Disclaimer: Will be unnecessarily expensive because a dynamically allocated context will be kept alive for the `delegate`. You may want to use the following equivalent:

```cpp
auto myNumbers(int begin, int end, int step) {
    static struct MyNumbers {
        // ...
    }
    return MyNumbers(begin, end, step);
}
```
auto return type means "Deduce the return type automatically."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if (isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E))
    && isIntegral!S)
{ /* ... */ }
```
Explanations for iota

auto return type means "Deduce the return type automatically."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
    && isIntegral!S)
{ /* ... */ }
```

Template parameters mean "B, E, and S are some types."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
    && isIntegral!S)
{ /* ... */ }
```
Explanations for `iota`

`auto` return type means "Deduce the return type automatically."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
    && isIntegral!S)
{ /* ... */ }
```

Template parameters mean "B, E, and S are some types."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
    && isIntegral!S)
{ /* ... */ }
```

Function parameters mean "`iota` takes three parameters of such types."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
    && isIntegral!S)
{ /* ... */ }
```
Explanations for `iota`

**Auto return type means** "Deduce the return type automatically."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
   && isIntegral!S)
{ /* ... */ }
```

**Template parameters mean** "B, E, and S are some types."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
   && isIntegral!S)
{ /* ... */ }
```

**Function parameters mean** "iota takes three parameters of such types."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
   && isIntegral!S)
{ /* ... */ }
```

**Template constraint means** "Use when B and E are either integrals or pointers and S is integral."

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
   && isIntegral!S)
{ /* ... */ }
```
Multiple definitions of \texttt{iota}

1) Most parameterized:

```
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E))) || isPointer!(CommonType!(B, E)))  // (*)
    && isIntegral!S)
{ /* ... */ }
```
Multiple definitions of \texttt{iota}

1) Most parameterized:

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))  // (*)
     && isIntegral!S)
{ /* ... */ }
```

2) Without the \texttt{step} parameter:

```cpp
auto iota(B, E)(B begin, E end)
if (isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
{
    return iota(begin, end, CommonType!(B, E)(1));
}
```
Multiple definitions of \texttt{iota}

1) Most parameterized:

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if ((isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E))) // (*)
    && isIntegral!S)
{ /* ... */ }
```

2) Without the \texttt{step} parameter:

```cpp
auto iota(B, E)(B begin, E end)
if (isIntegral!(CommonType!(B, E)) || isPointer!(CommonType!(B, E)))
{
    \textbf{return} iota(begin, end, CommonType!(B, E)(1));
}
```

3) Without the \texttt{begin} parameter:

```cpp
auto iota(E)(E end)
if (is\textbf{typeof}(iota(E(0), end))) // (*)
{
    E begin = E(0);
    return iota(begin, end);
}
```
Multiple definitions of iota (continued)

4) Most parameterized for floating point types:

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if (isFloatingPoint!(CommonType!(B, E, S)))
{
    // ...
    Value front() const { assert(!empty); return start + step * index; }
    void popFront()
    {
        assert(!empty);
        ++index;  // [Ali]: start += step would not work for floating point types
    }
    // ...
}
```
Multiple definitions of iota (continued)

4) Most parameterized for floating point types:

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if (isFloatingPoint!(CommonType!(B, E, S)))
{
    // ...
    Value front() const { assert(!empty); return start + step * index; }
    void popFront()
    {
        assert(!empty);
        ++index; // [Ali]: start += step would not work for floating point types
    }
    // ...
}
```

5) Ditto without step:

```cpp
auto iota(B, E)(B begin, E end)
if (isFloatingPoint!(CommonType!(B, E)))
{ /* ... */ }
```
Multiple definitions of iota (continued)

4) Most parameterized for floating point types:

```cpp
auto iota(B, E, S)(B begin, E end, S step)
if (isFloatingPoint!(CommonType!(B, E, S)))
{
    // ...
    Value front() const { assert(!empty); return start + step * index; }
    void popFront()
    {
        assert(!empty);
        ++index; // [Ali]: start += step would not work for floating point types
    }
    // ...
}
```

5) Ditto without step:

```cpp
auto iota(B, E)(B begin, E end)
if (isFloatingPoint!(CommonType!(B, E)))
{ /* ... */ }
```

6) Catch-all specialization for user-defined types

```cpp
auto iota(B, E)(B begin, E end)
if (!isIntegral!(CommonType!(B, E)) &&
    !isFloatingPoint!(CommonType!(B, E)) &&
    !isPointer!(CommonType!(B, E)) &&
    is(typeof((ref B b) { ++b; })) &&
    (is(typeof(B.init < E.init)) || is(typeof(B.init == E.init))) )
{ /* ... */ }
```
CommonType

CommonType:

• "The type that all types can be implicitly converted to."
• "The type the ternary operator would choose."
**CommonType**

**CommonType:**
- "The type that all types can be implicitly converted to."
- "The type the ternary operator would choose."

**Example:**

```cpp
static assert(is (CommonType!(double, int, short) == double));
```
is(typeof(expr))

Both `is` and `typeof` are evaluated at compile time.

* `typeof(expr)`: the type of the expression.
Both `is` and `typeof` are evaluated at compile time.

- `typeof(expr)`: the type of the expression.
- `is(Type)`: `true` if `Type` is semantically correct.
is(typeof(expr))

Both `is` and `typeof` are evaluated at compile time.

- `typeof(expr)`: the type of the expression.
- `is(Type)`: true if `Type` is semantically correct.

Example:

```javascript
// The following 'is' expression is 'false':
is(  // 3) false
  typeof(  // 2) The lambda does not have a type
    (string s) {    // 1) Illegal operation for string
      ++s;
    }
  )
)
Both `is` and `typeof` are evaluated at compile time.

- `typeof(expr)`: the type of the expression.
- `is(Type)`: true if `Type` is semantically correct.

Example:

```cpp
// The following 'is' expression is 'false':
is( // 3) false
typeof( // 2) The lambda does not have a type
    (string s) { // 1) Illegal operation for string
        ++s;
    }
)
```

An equivalent construct:

```cpp
// The following '__traits' expression is 'false':
__traits(compiles,
    (string s) { // 1) Illegal operation for string
        ++s;
    }
)
```
Template type deduction preserves qualifiers:

```c
void main() {
    const a = 42;
    foo(a);
}

void foo(A)(A a) {
    A result;    // A is deduced as const(int), and because of that:
    ++result;   // - Compilation ERROR
    }
```
Unqual

Template type deduction preserves qualifiers:

```c
void main() {
    const a = 42;
    foo(a);
}

void foo(A)(A a) {
    A result; // A is deduced as const(int), and because of that:
    ++result; // ← Compilation ERROR
}
```

Unqual saves the day:

```c
void foo(A)(A a) {
    Unqual!A result; // 'result' is 'int'
    ++result;       // Now compiles
}
```
The elements will be processed on all CPU cores in parallel (e.g. can be 4 times faster on a 4-core system):

```java
Student[] students;
// ...
foreach (s; students.parallel) {
    // ...
}
```
The elements will be processed on all CPU cores in parallel (e.g. can be 4 times faster on a 4-core system):

```java
Student[] students;
// ...
foreach (s; students.parallel) {
// ...
}
```

The equivalent without *universal function call syntax* (UFCS):

```java
foreach (s; parallel(students)) {
// ...
}
```
The elements will be processed on all CPU cores in parallel (e.g. can be 4 times faster on a 4-core system):

```cpp
Student[] students;
// ...
foreach (s; students.parallel) {
    // ...
}
```

The equivalent without *universal function call syntax* (UFCS):

```cpp
foreach (s; parallel(students)) {
    // ...
}
```

A function that dispatches to a member function of a global range object:

```cpp
ParallelForeach!R parallel(R)(R range)
{
    return taskPool.parallel(range);
}
```
The elements will be processed on all CPU cores in parallel (e.g. can be 4 times faster on a 4-core system):

```c++
Student[] students;
// ...
foreach (s; students.parallel) {
    // ...
}
```

The equivalent without *universal function call syntax (UFCS)*:

```c++
foreach (s; parallel(students)) {
    // ...
}
```

A function that dispatches to a member function of a global range object:

```c++
ParallelForeach!R parallel(R)(R range)
{    return taskPool.parallel(range);
}
```

The equivalent with optional parenthesis:

```c++
ParallelForeach!R parallel(R)(R range)
{    return taskPool().parallel(range);
}
```
A lazily-initialized global object:

```c
@property TaskPool taskPool() @trusted
{
    import std.concurrency : initOnce;
    gshared TaskPool pool;
    return initOnce!pool(
        auto p = new TaskPool(defaultPoolThreads);
        p.isDaemon = true;
        return p;
    )();
}
```

)initOnce uses initOnceLock, which uses a mutex.)
A lazily-initialized global object:

```typescript
@property TaskPool taskPool() @trusted
{
    import std.concurrency : initOnce;
    __gshared TaskPool pool;
    return initOnce!pool(
        auto p = new TaskPool(defaultPoolThreads);
        p.isDaemon = true;
        return p;
    )();
}
```

(initOnce uses initOnceLock, which uses a mutex.)

TaskPool.parallel returns a ParallelForeach object:

```typescript
final class TaskPool
{
    // ...
    ParallelForeach!R parallel(R)(R range)
    {
        // ...
    }
}
ParallelForeach supports foreach iteration through a pair of `opApply` functions:

```csharp
private struct ParallelForeach(R) {
    // ...
    int opApply(scope NoIndexDg dg) {
        static if (randLen!=R) {
            mixin(parallelApplyMixinRandomAccess);
        } else {
            mixin(parallelApplyMixinInputRange);
        }
    }
    int opApply(scope IndexDg dg) { /* ... */ }
}
```
ParallelForeach supports foreach iteration through a pair of opApply functions:

```csharp
private struct ParallelForeach(R)
{
    // ...
    int opApply(scope NoIndexDg dg)
    {
        static if (randLen!R) {
            mixin(parallelApplyMixinRandomAccess);
        } else {
            mixin(parallelApplyMixinInputRange);
        }
    
    int opApply(scope IndexDg dg) { /* ... */ }
}
```

The implementation comes from string mixins:

```csharp
private enum string parallelApplyMixinRandomAccess = q{
    // ...
    // Whether iteration is with or without an index variable.
    enum withIndex = Parameters!(typeof(dg)).length == 2;
    // ...

    void doIt()
    { // ...
    }

    submitAndExecute(pool, &doIt);
    return 0;
};
```
parallel (summary)

The magic:

```cpp
Student[] students;
// ...
foreach (s; students.parallel) {
    // ...
}
```
parallel (summary)

The magic:

```cpp
Student[] students;
// ...
foreach (s; students.parallel) {
    // ...
}
```

Some of the D features:

- UFCS
- Optional function call parenthesis
- Mutex-protected lazy initialization
- `foreach` support by `opApply`
- Design-by-introspection (DbI)
- String mixins
The power of design-by-introspection (DbI)

```plaintext
auto r = iota(100)
    .map!(n => n * n)
    .stride(2)   // (I know; 'iota' could be utilized for the same.)
    .take(5);
write!(r);
```

[0, 4, 16, 36, 64]
The power of design-by-introspection (DbI)

```cpp
auto r = iota(100)
  .map!(n => n * n)
  .stride(2)          // (I know; 'iota' could be utilized for the same.)
  .take(5);
writeln(r);
```

```
[0, 4, 16, 36, 64]
```

How about the following?

```cpp
writeln(r[2]);    // Really?
```
The power of design-by-introspection (DbI)

```javascript
auto r = iota(100)
    .map!(n => n * n)
    .stride(2) // (I know; 'iota' could be utilized for the same.)
    .take(5);

writeln(r);
```

```
[0, 4, 16, 36, 64]
```

How about the following?

```javascript
writeln(r[2]); // Really?
```

```
16
```

- It works because
- `take` supports it because
- `stride` supports it because
- `map` supports it because
- `iota` supports it.
The power of design-by-introspection (DbI) (continued)

D is one programming language with static if.
The power of design-by-introspection (DbI) (continued)

D is one programming language with static if.

For example, the Take struct that is returned by the take function:

```d
struct Take(Range)
// ...
{
    // ...
    static if (isRandomAccessRange!R)
    {
        auto ref opIndex(size_t index)
        {
            assert(index < length,
                    "Attempting to index out of the bounds of a "
                    ~ Take.stringof);
            return source[index];
        }
    } // ...
}
```
Pattern matching

D does not provide *pattern matching* as a language feature. But we can provide some form of it.
Pattern matching

D does not provide pattern matching as a language feature. But we can provide some form of it.

For example, `std.concurrent.receive` can dispatch to different delegates by message type(s):

```cpp
receive{
    (LinkTerminated msg) {
        // The worker terminated
        // ...
    },

    (Result result) {
        // The worker sent a result
        // ...
    },

    (Foo foo, Bar bar) {
        // The worker sent both a Foo and a Bar
        // ...
    },
};
```
D does not provide *pattern matching* as a language feature. But we can provide some form of it.

For example, `std.concurrency.receive` can dispatch to different delegates by message type(s):

```cpp
receive(
    (LinkTerminated msg) {
      // The worker terminated
      // ...
    },
    (Result result) {
      // The worker sent a result
      // ...
    },
    (Foo foo, Bar bar) {
      // The worker sent both a Foo and a Bar
      // ...
    });
```

`std.concurrency` uses `Variant` to send various types of messages:

```cpp
struct Message {
    MsgType type;
    Variant data;
    // ...
}
```
Pattern matching by linear searching at run time

In the following excerpt

• `ops` is the array of operations (e.g. delegates) provided to receive

• `Ops` is a tuple of their types

```plaintext
foreach (i, t; Ops)
{
    alias Args = Parameters!(t);
    auto op = ops[i];

    // ...

    if (msg.convertsTo!(Args)) // ← Boils down to Variant.convertsTo
    {
        // Found the matching operation.
        // ... calls 'op' and returns ...
    }
}
```
Aside: Useful error messages

assert and static assert can provide useful error messages.
Aside: Useful error messages

assert and static assert can provide useful error messages.

For example, inside std.concurrency.MessageBox.get:

```cpp
class MessageBox {
    // ...

    bool get(T...)(scope T vals) {
        // ...
        static assert(T.length, "T must not be empty");
        // ...
    }

    // ...
}
```
assert and static assert can provide useful error messages.

For example, inside `std.concurrency.MessageBox.get`:

```cpp
class MessageBox
{
    // ...

    bool get(T...)(scope T vals)
    {
        // ...
        static assert(T.length, "T must not be empty");
        // ...
    }
    // ...
}
```

More:

```cpp
static assert(a1.length != 1 || !is(a1[0] == Variant),
              "function with arguments " + a1.stringof + 
              " occludes successive function");
```
Discriminated union implementations in Phobos

Multiple:

- **Variant**: Can hold a value of any type
- **Algebraic**: Can hold a value of a set of types known at compile-time (Not recommended; use SumType instead)
- **SumType**: Better Algebraic written by Paul Backus
SumType

Copying from its documentation:

• Pattern matching
• Support for self-referential types
• Full attribute correctness (pure, @safe, @nogc, and nothrow are inferred whenever possible)
• A type-safe and memory-safe API compatible with DIP 1000 (scope)
• No dependency on runtime type information (TypeInfo)
• Compatibility with BetterC
SumType example

Definition:

```plaintext
struct Fahrenheit { double degrees; }
struct Celsius { double degrees; }
struct Kelvin { double degrees; }

alias Temperature = SumType!(Fahrenheit, Celsius, Kelvin);
```
SumType example

Definition:

```cpp
struct Fahrenheit { double degrees; }
struct Celsius { double degrees; }
struct Kelvin { double degrees; }

alias Temperature = SumType!(Fahrenheit, Celsius, Kelvin);
```

Construction:

```cpp
Temperature t1 = Fahrenheit(98.6);
Temperature t2 = Celsius(100);
Temperature t3 = Kelvin(273);
```
SumType example

Definition:

```cpp
struct Fahrenheit { double degrees; }
struct Celsius { double degrees; }
struct Kelvin { double degrees; }

alias Temperature = SumType!(Fahrenheit, Celsius, Kelvin);
```

Construction:

```cpp
Temperature t1 = Fahrenheit(98.6);
Temperature t2 = Celsius(100);
Temperature t3 = Kelvin(273);
```

Pattern matching:

```cpp
Fahrenheit toFahrenheit(Temperature t) {
    return Fahrenheit(
        t.match!(
            (Fahrenheit f) => f.degrees,
            (Celsius c) => c.degrees * 9.0/5 + 32,
            (Kelvin k) => k.degrees * 9.0/5 - 459.4
        )
    );
}
```
SumType example

Definition:

```plaintext
struct Fahrenheit { double degrees; }
struct Celsius { double degrees; }
struct Kelvin { double degrees; }

alias Temperature = SumType!(Fahrenheit, Celsius, Kelvin);
```

Construction:

```plaintext
Temperature t1 = Fahrenheit(98.6);
Temperature t2 = Celsius(100);
Temperature t3 = Kelvin(273);
```

Pattern matching:

```plaintext
Fahrenheit toFahrenheit(Temperature t) {
    return Fahrenheit(
        t.match!(
            (Fahrenheit f) => f.degrees,
            (Celsius c) => c.degrees * 9.0/5 + 32,
            (Kelvin k) => k.degrees * 9.0/5 - 459.4
        )
    );
}
```

In fact, multiple dispatch:

```plaintext
match!(
    (Fahrenheit f1, Fahrenheit f2) => writeln("Both F"),
    (Celsius c1, Celsius c2) => writeln("Both C"),
    (Kelvin k1, Kelvin k2) => writeln("Both K"),
    (1, 2) => writeln("Different"),
    (t1, t2);
```
SumType at compile time (1/3)

Builds a handler lookup table:

```c++
private template matchImpl(Flag!"exhaustive" exhaustive, handlers...)
// ...
enum matches = ()
{
    size_t[numCases] matches;
    // ...
    static foreach (caseId; 0 .. numCases)
    {
        static foreach (hid, handler; handlers)
        {
            static if (canMatch!(handler, valueTypes!caseId))
            {
                // ...
                matches[caseId] = hid;
                // ...
            }
        }
    }
    return matches;
}();
```
Builds handler names:

```cpp
enum handlerName(size_t hid) = "handler" ~ toCtString!hid;
static foreach (size_t hid, handler; handlers)
{
    mixin("alias ", handlerName!hid, " = handler;" );
}
```
Builds a switch statement at compile time:

```java
immutable argsId = TagTuple(args).toCaseId;

final switch (argsId)
{
    static foreach (caseId; 0 .. numCases)
    {
        case caseId:
            static if (matches[caseId] != noMatch)
            {
                return mixin(handlerName!(matches[caseId]), "(" handlerArgs!caseId, ")");
            }
            else
            {
                static if (exhaustive)
                {
                    static assert(false,
                        "No matching handler for types `" ~ valueType!caseId.stringof ~ "`");
                }
                else
                {
                    throw new MatchException(
                        "No matching handler for types `" ~ valueType!caseId.stringof ~ "`");
                }
            }
    }

    assert(false, "unreachable");
}
```
SumType supports recursive data types

Again, from the documentation:

```csharp
// An expression is either
// - a number,
// - a variable, or
// - a binary operation combining two sub-expressions.
alias Expr = SumType!(
    double,
    string,
    Tuple!(Op, "op", This*, "lhs", This*, "rhs")
);

// ...
struct This {}
```
SumType supports recursive data types

Again, from the documentation:

```cpp
// An expression is either
// - a number,
// - a variable, or
// - a binary operation combining two sub-expressions.
alias Expr = SumType!(
    double,
    string,
    Tuple!(Op, "op", This*, "lhs", This*, "rhs")
);

// ...
struct This {}
```

Aside: Parts of Phobos documentation come from actual unittest blocks.
Conclusion

• D is very powerful
• Phobos is written in readable D
• Phobos takes advantage of D effectively