# Multitasking in D

**Ali Çehreli**

May 13, 2015
ACCU, Silicon Valley

# Introduction

- The D programming language
- Features of D that help with multitasking
- Parallelism
- Concurrency by message passing
- Concurrency by data sharing
- Cooperative multitasking with fibers

# The D programming language

- Strongly statically typed with type and attribute inference
- Familiar to users of C, C++, Java, C#, Python, and more
- Imperative, object oriented, generic, generative, functional
- Contract programming, **`@safe`**, **`unittest`**, **`immutable`**, etc. help with program correctness
- Compilers: **`dmd`** (reference), **`ldc`** (LLVM backend), **`gdc`** (GCC backend)
- High level system language

```
int i = *ptr;
++ptr;
```

```
stdin
.byLine(KeepTerminator.yes)
.map!(a => a.dup)
.array
.sort()
.copy(stdout.lockingTextWriter());
```

# Some D users

- Facebook
- weka.io - Storage startup (stealth mode)
- Sociomantic - real-time bidding for eCommerce
- Remedy Games - game development
- Funkwerk IT Karlsfeld - passenger information systems
- EMSI - economic modeling
- RandomStorm - network security products and services
- 2night.it - italian website about nightlife
- SR Labs - Eye Tracking Systems Integrator
- Funatics Software GmbH - MMO Game Developer
- Infognition - mostly video processing technologies and products
- Social Magnetics - Distributed Storage, Processing & Messaging PaaS (stealth mode)
- Large Hedge Fund - receiving market data, electronic trading, processing large text files
- Adroll: big data marketing

# UFCS: Universal function call syntax

Enables member function call syntax for regular functions.

```
int dividedBy(int x, int y)
{
    return x / y;
}

    int a = 42;
    assert(dividedBy(a, 2) == 21);    // regular syntax
    assert(a.dividedBy(2) == 21);     // UFCS syntax
```

```
    stdin
    .byLine(KeepTerminator.yes)
    .map!(a => a.dup)
    .array
    .sort()
    .copy(stdout.lockingTextWriter());
```

The equivalent of the expression above:

```
copy(sort(array(map!(a => a.dup)(byLine(stdin,
                                        KeepTerminator.yes)))),
     lockingTextWriter(stdout));
```

# Templates

```
        stdin
        .byLine(KeepTerminator.yes)
        .map!(a => a.dup)    // ← explicit instantiation
        .array
        .sort()
        .copy(stdout.lockingTextWriter());
```

Template parameters are specified within parentheses:

```
struct Coordinate(T, size_t N)
{
    T[N] values;
}

alias Point3D = Coordinate!(int, 3);

auto mirror(P)(P p)
{
    p.values[] *= -1;
    return p;
}

void main()
{
    auto p = Point3D([ 1, 2, 3 ]);
    assert(mirror(p) == Point3D([ -1, -2, -3 ]));
}
```

# Lambdas and delegates

```
        stdin
        .byLine(KeepTerminator.yes)
        .map!(a => a.dup)
        .array
        .sort()
        .copy(stdout.lockingTextWriter());
```

*Another* way of defining the above lambda:

```
    auto func = (string a) {
                    return a.dup;
                };
```

# Ranges

D uses a range abstraction (not iterator) for element access.

The primitives **`front`**, **`empty`**, and **`popFront`** make an **`InputRange`**:

```
struct Fibonacci
{
    long front = 0;                 // ← usually a function
    long next = 1;

    static const empty = false;     // ← usually a function

    void popFront() {
        const nextNext = front + next;
        front = next;
        next = nextNext;
    }
}

static assert(isInputRange!Fibonacci);
```

# Range kinds

The following are the other ranges with the primitives that they require additionally:

```
InputRange
ForwardRange            .save
BidirectionalRange      .back, .popBack()
RandomAccessRange       [] (.length if finite)

OutputRange             .put
```

Dynamic arrays already satisfy **RandomAccessRange**, the most capable range kind:

```
int[] arr = [ 1, 2, 3 ];
static assert(isRandomAccessRange!(typeof(arr)));
```

# Range algorithms

Phobos (D standard library) has a large number of range algorithms:

```
    auto r = Fibonacci()
             .take(10)
             .map!(a => a * 10);

    writeln(r);    // ← 'r' is lazily consumed here
```

```
[0, 10, 10, 20, 30, 50, 80, 130, 210, 340]
```

# static if

Enables code sections depending on compile time conditions:

```
struct Map(alias func, R)
    if (isCallable!func && isInputRange!R)
{
    R range;

    @property bool empty() { return range.empty; }
    @property auto front() { return func(range.front); }
    void popFront()        { range.popFront(); }

  static if (isRandomAccessRange!R) {
    auto opIndex(size_t i) { return func(range[i]); }
  }
}

// Convenience function template
auto map(alias func, R)(R range) {
    return Map!(func, R)(range);
}

void main() {
    auto arr = [ 0, 1, 2 ];
    auto r = arr.map!(a => a * 10);
    assert(r[2] == 20);
}
```

***static if*** *will not be added to C++. (See "“Static If” Considered", document N3613, at http://isocpp.org/files/papers/ by Bjarne Stroustrup and others.)*

# shared

Data is thread-local by default. **shared** makes it sharable:

```
    int a;              // thread-local
    shared(int) b;      // can be shared between threads

    shared int c;       // alternative syntax
```

**shared** is an overloadable attribute:

```
void foo(       int  a) { /* ... */ }
void foo(shared(int) a) { /* ... */ }

struct S
{
    int i;

    this(int i)         { /* ... */ } // constructing an S
    this(int i) shared { /* ... */ } // constructing a shared(S)
}
```

Disclaimer: **shared** spec is not finalized yet (e.g. are memory barriers inserted?).

# immutable

- **immutable** means "this variable never changes"

```
    immutable id = 42;
    string s = "hi";   // string is immutable(char)[]
```

  - No need to lock **immutable** data in concurrency.
  - Compiler has more opportunities to optimize.

Both **const** and **immutable** are transitive:

```
struct S { int id; }

const module_level_S = S(42);

int foo(ref const(S) s)
{
    s.id = 100;             // ← COMPILATION ERROR
    s = module_level_S;     // ← COMPILATION ERROR
}
```

(It is possible to rebind **const** class references with **Rebindable**.)

# Parallelism

To execute tasks in parallel mainly for performance reason.

```
import std.parallelism;

void foo(int a)
{
    writefln("Working on %s", a);
}

    auto arr = [ 1, 2, 3, 4 ];

    arr.parallel.each!foo;
```

```
Working on 1
Working on 4
Working on 2
Working on 3
```

Alternatively:

```
    foreach (i; arr.parallel) {
        writefln("Working on %s", i);
    }
```

# Some functions of `std.parallelism`

- **`parallel`**: Visit the elements of a range in parallel.
- **`task`**: Creates tasks that are executed in parallel.
- **`asyncBuf`**: Iterates the elements of an InputRange semi-eagerly in parallel.
- **`map`**: Calls functions with the elements of an InputRange semi-eagerly in parallel.
- **`amap`**: Calls functions with the elements of a RandomAccessRange fully-eagerly in parallel.
- **`reduce`**: Makes calculations over the elements of a RandomAccessRange in parallel.

# Message-passing concurrency

To execute multiple tasks concurrently (which usually interact).

```
import std.conv;
import std.concurrency;

struct Exit {}

void converter()    // ← (can receive parameters as well)
{
    bool done = false;

    while (!done) {
        receive(
            (string a) { ownerTid.send(a.to!int);    },
            (int a)    { ownerTid.send(a.to!string); },
            (Exit _)   { done = true;                }
        );
    }
}

void main()
{
    auto worker = spawn(&converter);

    worker.send("42");
    assert(receiveOnly!int() == 42);

    worker.send(100);
    assert(receiveOnly!string() == "100");

    worker.send(Exit());
}
```

# Data-sharing concurrency

Prone to race conditions.

```
import core.thread;
import core.atomic;
import std.concurrency;

void incrementor(shared(int) * i)
{
    atomicOp!"+="(*i, 1);    // ++(*i) is a compiler warning
}

void main()
{
    shared(int) i = 0;

    spawn(&incrementor, &i);

    thread_joinAll();

    assert(i == 1);
}
```

# synchronized

Critical section with implicit lock:

```
    synchronized {
        // ...
    }
```

Critical section with explicit lock(s):

```
    synchronized (a, b) {
        // ...
    }
```

# core.atomic

- **atomicOp**: Executes binary operations atomically

```
    auto c = atomicOp!"+"(a, b);
```

- **cas**: Compare and swap; enables *lock-free data structures*
- *more...*

# core.sync

Classic concurrency primitives:

- **core.sync.barrier**
- **core.sync.condition**
- **core.sync.config**
- **core.sync.exception**
- **core.sync.mutex**
- **core.sync.rwmutex**
- **core.sync.semaphore**

# Preemptive multitasking

- Parallelism and concurrency are based on **`core.thread`**
- **`core.thread`** uses operating system threads
- OS threads are subject to preemptive multitasking

Disadvantages of preemtive multitasking:

- Relinquishing remainder of scheduling quantum due to lock convoy, I/O wait, etc.
- CPU data cache lost
- Memory management unit (MMU) state lost

Enter *cooperative multitasking* with fibers...

# Cooperative multitasking

- Same thread, same address space
- Tasks yield to each other explicitly
- Knows as: fiber, green thread, greenlet, goroutine, etc.

**Pros:**

- Thread safety is not an issue; locking or atomic operations are not necessary (however, watch out for a rogue yield!)
- Faster for various reasons: Cache coherency, no locking, fully utilized time slice

**Cons:**

- The parent and its fibers run on the same core, while other cores may be idle (consider different threading models like the *M:N threading model*)
- The parent and its other fibers are suspended until the running fiber yields

# Call stack

Local state of every function call is kept in a *stack frame*.

The stack frames of currently active function calls is called the *call stack*.

```
void main()
{
    int a;
    int b;

    int c = foo(a, b);
}

int foo(int x, int y)
{
    bar(x + y);
    return 42;
}

void bar(int param)
{
    string[] arr;
    // ...
}
```

```
The call stack grows upward
as function calls get deeper.
       ▲  ▲
       |  |
 +--------------+
 | int param    | ← bar's frame
 | string[] arr |
 +--------------+
 | int x        |
 | int y        | ← foo's frame
 | return value |
 +--------------+
 | int a        |
 | int b        | ← main's frame
 | int c        |
 +--------------+
```

***Disclaimer:*** *It actually grows downward on all common CPUs.*

# Call stack is especially useful in recursion

No matter how deep the layers of function calls are, the call stack takes care of it:

```
import std.array;

int sum(int[] arr,
        int currentSum = 0)
{
  if (arr.empty) {
      return currentSum;
  }

  return sum(arr[1..$],
             currentSum + arr.front);
}

void main()
{
  assert(sum([1, 2, 3]) == 6);
}
```

```
+--------------------------+
| arr        == []         | ← final call
| currentSum == 6          |
+--------------------------+
| arr        == [3]        | ← third call
| currentSum == 3          |
+--------------------------+
| arr        == [2, 3]     | ← second call
| currentSum == 1          |
+--------------------------+
| arr        == [1, 2, 3]  | ← first call
| currentSum == 0          |
+--------------------------+
|            ...           | ← main's frame
+--------------------------+
```

***Note:*** *You should use* ***std.algorithm.sum*** *instead.*

***Disclaimer:*** *"Tail-call optimization" can eliminate stack frames.*

# Some data structures and their algorithms are trivial and elegant with recursion

For example, the node structure of a binary tree:

```
struct Node
{
    int element;
    Node * left;
    Node * right;

    void print() const
    {
        if (left) {
            left.print();
            write(' ');
        }

        write(element);

        if (right) {
            write(' ');
            right.print();
        }
    }

    // ...
}
```

# Surprising complexity

Implementing a range (or iterator) type for a tree is very hard especially considering how trivial it is with recursion.

```
struct Tree
{
// ...

    struct InOrderRange
    {
        ... What should the implementation be? ...
    }

    InOrderRange opSlice() const
    {
        return InOrderRange(root);
    }
}
```

Some tree iterator implementations require an additional **`Node*`** to point at the parent node.

# Fiber operations

- A fiber (and its call stack) starts with a callable entity taking no parameter, returning nothing:

```
void fiberFunction() { /* ... */ }
```

- Created as an object of the **`core.thread.Fiber`** class hierarchy:

```
    auto fiber = new Fiber(&fiberFunction);
```

- Started and resumed by its **`call()`** member function:

```
    fiber.call();
```

- Pauses itself by **`Fiber.yield()`**:

```
void fiberFunction() { /* ... */ Fiber.yield(); /* ... */ }
```

- The execution state of a fiber is determined by its **`.state`** property:

```
    if (fiber.state == Fiber.State.TERM) { /* ... */ }
```

# Trivial example: the Fibonacci series

```
import core.thread;

void fibonacciSeries(ref int current) {
    current = 0;    // Note: 'current' is the parameter
    int next = 1;

    while (true) {
        Fiber.yield();

        /* Next call() will continue from this point */

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main() {
    int current;
    Fiber fiber = new Fiber(() => fibonacciSeries(current));

    foreach ( ; 0 .. 10) {
        fiber.call();

        import std.stdio;
        writef("%s ", current);
    }
}
```

Unfortunately, does not provide a range interface, uses a **`ref`** variable, and too low level.

# `Generator` to present a fiber as an `InputRange`

```
import std.stdio;
import std.range;
import std.concurrency;

/* Resolve the name conflict with std.range.Generator. */
alias FiberRange = std.concurrency.Generator;

void fibonacciSeries()
{
    int current = 0;    // <-- Not a parameter anymore
    int next = 1;

    while (true) {
        yield(current);

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main()
{
    auto series = new FiberRange!int(&fibonacciSeries);
    writefln("%(%s %)", series.take(10));
}
```

# Tree iterator as a fiber

```
struct Node
{
// ...
    auto opSlice() const
    {
        return byNode(&this);
    }
}

auto byNode(const(Node) * node)
{
    return new FiberRange!(const(Node)*)(
        () => nextNode(node));
}

void nextNode(const(Node) * node)
{
    if (node.left) {
        nextNode(node.left);
    }

    yield(node);    // <-- Elegant once again! ;)

    if (node.right) {
        nextNode(node.right);
    }
}
// ...
    writefln("%(%s %)", tree[]);
```

# Asynchronous input/output example

A sign-on flow of a service with the following protocol:

```
> hi                          ← Alice connects
Flow 0 started.
> 0 Alice
> 0 alice@example.com
> 0 20
Flow 0 has completed.         ← Alice finishes
Added user 'Alice'.
> hi                          ← Bob connects
Flow 1 started.
> 1 Bob
> 1 bob@example.com
> 1 30
Flow 1 has completed.         ← Bob finishes
Added user 'Bob'.
> bye
Goodbye.
Users:
  User("Alice", "alice@example.com", 20)
  User("Bob", "bob@example.com", 30)
```

```
    if (input == "hi") {
        signOnNewUser();    // ← WARNING: Blocking design
    }
```

# Fibers help with asynchronous input/output

The flow functionality can be implemented in linear fashion:

```
class SignOnFlow : Fiber
{
// ...
    void run()
    {
        name = inputData_;
        Fiber.yield();

        email = inputData_;
        Fiber.yield();

        age = inputData_.to!uint;
    }
}

// The caller:
void handleFlowData(/* ... */) {
// ...
    flow.inputData = input.data;
    flow.call();

    if (flow.state == Fiber.State.TERM) {
        // ... This flow has completed ...
    }
}
```

# The users are not blocked anymore

Alice's interaction is highlighted:

```
> hi                        ← Alice connects
Flow 0 started.
> 0 Alice
> hi                        ← Bob connects
Flow 1 started.
> hi                        ← Cindy connects
Flow 2 started.
> 0 alice@example.com
> 1 Bob
> 2 Cindy
> 2 cindy@example.com
> 2 40
Flow 2 has completed.       ← Cindy finishes
Added user 'Cindy'.
> 1 bob@example.com
> 1 30
Flow 1 has completed.       ← Bob finishes
Added user 'Bob'.
> 0 20
Flow 0 has completed.       ← Alice finishes
Added user 'Alice'.
> bye
Goodbye.
Users:
  User("Cindy", "cindy@example.com", 40)
  User("Bob", "bob@example.com", 30)
  User("Alice", "alice@example.com", 20)
```

# References

- Main site:

http://dlang.org

- Wiki:

http://wiki.dlang.org

- An asynchronous input/output framework based on fibers:

http://vibed.org

- Ali's book:

http://ddili.org/ders/d.en