Grigoriy Okopnik

# HardCode

- general-purpose programming language
- cloud service development and administration platform
- next gen IDE: IT content delivery platform

# Fatal run-time errors

## Most loved errors currently:

- Null pointer exception
- Index out of range
- Unhandled exception

# Fatal run-time errors

## Null pointer exception: example in Go

```
type abc struct {
    field int
}

func getAbc() *abc {
    return nil
}

func main() {
    v := getAbc()
    print(v.field)            // Crash
    if v != nil {
        print(v.field)        // No crash, wow...
    } else {
        print(v.field)        // Crash
    }
    print(v.field)            // Crash
    if v != nil || 3*3 == 9 { // Extra condition
        print(v.field)        // Crash
    }
    if v == nil {             // Typo
        print(v.field)        // Crash
    }
}
```

# Fatal run-time errors

## Null pointer exception: why?

Why is this “feature” “implemented”
in all existing languages??

(Except for Elm where things are even worse)

# Fatal run-time errors

## Null pointer exception: the way it should be

```
struct abc (
    int field;
)

optional struct abc getAbc () { // optional abc == variant<struct abc|null>
    return null
}

void main () {
    optional struct abc v = getAbc ()
    print (v.field)             // Compile-time error
    let (struct abc& v_ref = v) {
        print (v_ref.field)     // No crash
    } else {
        print (v_ref.field)     // Compile-time error
    }
    print (v.field)             // Compile-time error
    if (let (abc& v_ref = v) || 3*3 == 9) { // Warning: "v_ref" can't be exposed
        print (v_ref.field)     // Compile-time error
    }
    if (let (abc& v_ref = v) && 3*3 == 9) {
        print (v_ref.field)     // No crash
    } else {
        print (v_ref.field)     // Compile-time error
    }
}
```

# Fatal run-time errors

## Index out of range: example in Go

```
func getArray() []int {
    return []int{}
}

func main() {
    a := getArray()
    print(a[0])                 // Crash
    if len(a) >= 1 {
        print(a[0])             // No crash, wow...
    }
    print(a[0])                 // Crash
    if len(a) >= 1 || 3*3 == 9 { // Extra condition
        print(a[0])             // Crash
    }
    if len(a) >= 0 {            // Typo
        print(a[0])             // Crash
    }
}
```

# Fatal run-time errors

## Working with indices: the way it should be

```
int[] getArray () {
    return int[] ()
}

void main () {
    int[] a = getArray ()
    print (a[0])                    // Compile-time error (Can only be accessed by index object!)
    let (index (a) i = 0) {
        print (a[i])                // No crash
    }
    print (a[0])                    // Compile-time error
    if (let (index (a) i = 0) || 3*3 == 9) { // Warning: `i` is not exposed (should use `_`)
        print (a[i])                // Compile-time error
    }
    if (let (index (a) i = 0) && 3*3 == 9) {
        print (a[i])                // No crash
    } else {
        print (a[i])                // Compile-time error
    }
    mut int[] b = int[] (1, 2, 3)   // Everything is const by default, we need keyword "mut" here
    let (index (b) i = 2) {
        b[i] = 500                  // Can assign too with same syntax
        print (b[i])
    }
}
```

# Fatal run-time errors

## Working with indices: index validations are quiet rare

```
void sort2 (int& a, int& b)
{
    if (a > b)
        swap (a, b)
}

void sort (int[] elements) // No run-time index check
{
    for desc (word:ranged i: #elements - 1, 1) // If using “ranged” with no arguments
        for (word:ranged j: 0, i - 1)          // range is statically deduced
            sort2 (elements[j], elements[j + 1])
}

void sort (int[] elements) // Same thing but verbose for clarity
{
    for desc (word:ranged (1, #elements - 1) i: #elements - 1, 1)
        for (word:ranged (0, #elements - 2) j: 0, i - 1) {
            word:ranged (0, #elements - 2) index_a = j
            word:ranged (1, #elements - 1) index_b = j + 1
            sort2 (elements[index_a], elements[index_b])
        }
}
```

# Fatal run-time errors

## Exception vs ...

- Exceptions allow expressive separation of successful branch from erroneous
- No need to handle errors at each level
- No way to use returned value if an error has occurred
- Unhandled exception:
  easy to forget to handle exception at necessary level
- Unclear exit points at function: can’t see if some code throws an exception

# Fatal run-time errors

## … vs Error object

- No Unhandled exceptions
- Explicit exit points at function: return is explicit, no exceptions
- Error handling branches aren’t distinguished from successful branches
- Easy to forget validating Error object and start working with invalid result
- Errors must be handled at each level

# Fatal run-time errors

## What should we choose: exceptions or error objects?

These guys new the wisdom.

Sometimes you can’t afford choosing.
You have to work your own way.

# Fatal run-time errors

## Hybrid error handling
## Turning “value or error” pattern into part of the language

```
// Some syntactic sugar

variant<Type|error@> val = get_value()
must (Type exact_val = val) {                    let (Type exact_val = val) {
    ...                                  ::=         ...
}                                                } else return val.error


Type exact_val = must (val)              ::=    Type exact_val = val.value else return val.error


                    maybe TypeX          ::=    variant<TypeX|error@>


                  optional TypeX         ::=    variant<TypeX|null>
```

# Fatal run-time errors

## Hybrid error handling: verbosity of Go

```
func mix(a, b, alpha float64) (float64, error) {
    if alpha < 0.0 || alpha > 1.0 {
        return 0.0, errors.New("argument 'alpha' not in range[0.0..1.0]")
    }
    return (a*alpha + b*(1.0-alpha)), nil
}

type rgb struct {
    r, g, b float64
}

func mix_rgb(c1, c2 rgb, alpha float64) (rgb, error) {
    r, err := mix(c1.r, c2.r, alpha)
    if err != nil {
        return rgb{}, err
    }
    g, err := mix(c1.g, c2.g, alpha)
    if err != nil {
        return rgb{}, err
    }
    b, err := mix(c1.b, c2.b, alpha)
    if err != nil {
        return rgb{}, err
    }
    return rgb{
        r,
        g,
        b,
    }, nil
}

func main() {
    mixed, err := mix_rgb(rgb{0.1, 0.3, 0.25}, rgb{0.7, 0.2, 0.4}, 0.8)
    if err == nil {
        fmt.Printf("Color (%v, %v, %v)\n", mixed.r, mixed.g, mixed.b)
    } else {
        fmt.Printf("Failed to mix colors: %v\n", err.Error())
    }
}
```

# Fatal run-time errors

## Hybrid error handling: expressiveness of HardCode

```
maybe double mix (double& a, double& b, double& alpha) {
    if (alpha < 0.0 || alpha > 1.0) {
        return @@error ("argument 'alpha' not in range[0.0..1.0]")
    }
    return (a*alpha + b*(1.0 - alpha))
}

struct rgb (
    double r, g, b;
)

maybe struct rgb mix_rgb (struct rgb& c1, struct rgb& c2, double& alpha) {
    return (
        r = must mix (c1.r, c2.r, alpha), // Argument of "must" must be variant<typeof (lvalue)|error>
        g = must mix (c1.g, c2.g, alpha), // if argument of "must" contains error "return error" is called
        b = must mix (c1.b, c2.b, alpha), // else alternative value is returned
    )
}

void main () {
    match (mix_rgb ((r = 0.1, g = 0.3, b = 0.25), (r = 0.7, g = 0.2, b = 0.4), 0.8)) {
        case (struct rgb& mixed) {
            print ("Color (%v, %v, %v)\n", mixed.(r, g, b))
        }
        case (error@ err) {
            print ("Failed to mix colors: %v\n", err.message)
        }
    } // No "missing default case" here
}
```

# Fatal run-time errors

## Self-documented functions pattern
## The way Google does it

```
func some_function() (value *Type)
// Should we check the result for nil?

func some_other_function() (value *Type, err error)
// What cases may than function return: all 4 cases?
// Should I check value for nil if err == nil?
// Is value valid (partial result) if err != nil or it is an implementation error?
```

# Fatal run-time errors

## Self-documented functions pattern
## The proper way

```
// May be null:
optional Type* some_function()

// May not be null:
Type* some_function()


// Value or error:
maybe Type some_function()

// Value, error or null:
variant<Type|error@|null> Type some_function()

// Maybe an error + always a value:
optional error@ some_function(out Type value)

// Maybe an error + maybe a value:
optional error@ some_function(out optional Type value)
```

# Fatal run-time errors

## Hybrid error handling
## Error object

```
struct error (
    string domain; // no need for enum's
    string code;   // no need for enum's
    ubyte[] message;
    polymorphic userdata;
    optional error@ child;
)
- doesn't use enum's but stores error domain and code
- may store arbitrary user data

string type:
- specified strictly by literals
- internated (and therefore deduplicated)
- created at link time (not at run-time)
- managed by reference counting
- comparison operation is as fast as pointer comparison operation
- don't present bottleneck for multithreaded programming
```

# Fatal run-time errors

## Working with error stack

```
maybe struct rgb mix_rgb (struct rgb& c1, struct rgb& c2, double& alpha) {
    double r = @must ("Failed to interpolate red component") mix (c1.r, c2.r, alpha)
    double g = @must ("mix_rgb", "interpolation error", "Failed to interpolate green component") mix (c1.g, c2.g, alpha)
    match (mix (c1.b, c2.b, alpha)) {
        case (double& b) {
            return (r = r, g = g, b = b)
        }
        case (error@ err) {
            return @@error ("mix_rgb", "interpolation error", "Failed to interpolate blue component", err)
        }
    }
}
```

# Fatal run-time errors

## Hybrid error handling Summary

- Successful branch is separated from error handling branch syntactically
- Errors have to be passed (concisely) to upper level at each level:
  helps to trace error propagation paths
- Explicit exit points from a function: return and must are obvious, no exceptions
- No way to use (invalid) value in case of error
- No «Unhandled exceptions»
- Return values cases are explicitly defined
- Error stack and error domains allow for convenient detailed error handling

# Synchronization tools

## Everlasting synchronization issues

- Race condition:
  nobody knows in which order things will happen
- Deadlock:
  forever hung up program
- Livelock:
  something happens, but the program is still hung up
- Such issues are very hard to catch witch autotesting

# Synchronization tools

## Deadlock in C++

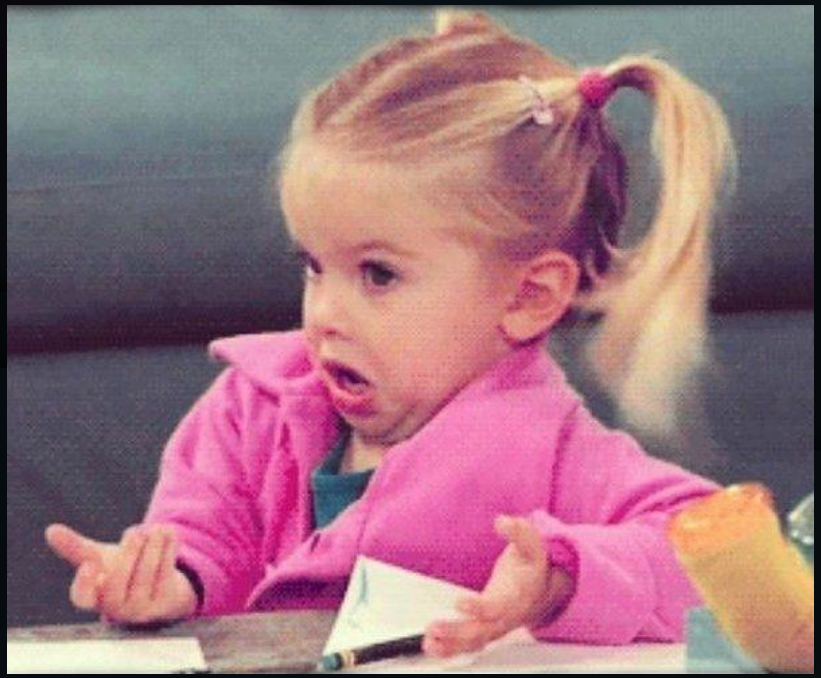
```
int main() {
    mutex m1, m2;
    thread t1([&m1, &m2] {
        cout << "Thread 1: 1. Acquiring m1." << endl;
        m1.lock();
        this_thread::sleep_for(chrono::milliseconds(10));
        cout << "Thread 1: 2. Acquiring m2." << endl;
        m2.lock();
        cout << "Thread 1: 3. Not reached" << endl;
    });
    thread t2([&m1, &m2] {
        cout << "Thread 2: 1. Acquiring m2." << endl;
        m2.lock();
        this_thread::sleep_for(chrono::milliseconds(10));
        cout << "Thread 2: 2. Acquiring m1." << endl;
        m1.lock();
        cout << "Thread 2: 3. Not reached" << endl;
    });

    t1.join();
    t2.join();

    return 0;
}
```

# Synchronization tools

But why do we need synchronization problems at all?



# Synchronization tools

## So how do we get rid of synchronization problems?

By following simple rules:

- No mutexes for application developer
- Instead of direct work with mutexes just mark
  object for shared access with keyword shared
  (non-shared object can only be accessed from thread where it was created)
- Shared objects can’t be nested
- One thread can access only one shared object at a time

# Synchronization tools

## Shared objects usage example

```
optional error func (context, // Control context
                     shared mut massive_container& big_object1,
                     shared mut massive_container& big_object2)
{
    big_object1.a.do_something () // Compilation error: shared object not locked
    must lock (big_object1, context) {
        big_object1.a.do_something ()
        big_object1.b.do_something2 ()
        lock (big_object2, context) { // Compilation error: nested lock
            big_object2.a.do_something ()
        }
    } /* else return error ("Cancelled by context") */
    let lock (big_object2, context) {
        big_object2.a.do_something ()
    } else {
        return error@ ("Object access timed out")
    }
    big_object1.a.do_something () // Compilation error: shared object not locked
    return null
}
```

# Synchronization tools

## What we get

- Shared object is accessed by one thread at a time (guaranteed by compiler)
- Synchronization errors are allowed by compiler
- Built-in analysis allows to guarantee function execution completion:
  no way to perform Lifelock
- Consequently no way to perform Deadlock
- RW lock variant of shared is available
- Logical hang up is possible because of network/IPC.
  But it is also analyzable.
  For that we need a network protocol with strict event model.
- Concurrent structured and safe nested locks aren’t part of MVP

# Memory model

## Old dull world

- Usually, either RAII or GC is present in a programming language
- When RAII is kinda combined with GC, RAII has to be forced (Python)
- GC eventually freezes
- No immutability guarantees
- Accidental object copying
- Zombie-objects
- Unnecessary object initialization and absence of object initialization

# Memory model

## Brave new world

- RAII even for objects managed by GC
- No need for zombie objects
- GC without freezes: there is a reference implementation of non-blocking multithreaded GC.
  The longest pause is 1 atomic operation per batch of GC events.
- Strict immutability semantics (fundamental for sandboxing)
- Initialization and mutation are separate operations
- Strict check form object initialization before usage.
  Initialization at any place exactly once.
  Including branch and initialization through reference passing.
- SIMD optimizations friendly language VM
- Non-existent accidental copy syntax
- No need for copy-constructors.
- All functions are stateless → module isolation, dynamic compilation, strict testability
- Analyzable stack depth

# Memory model

## Old dull world Garbage Collector

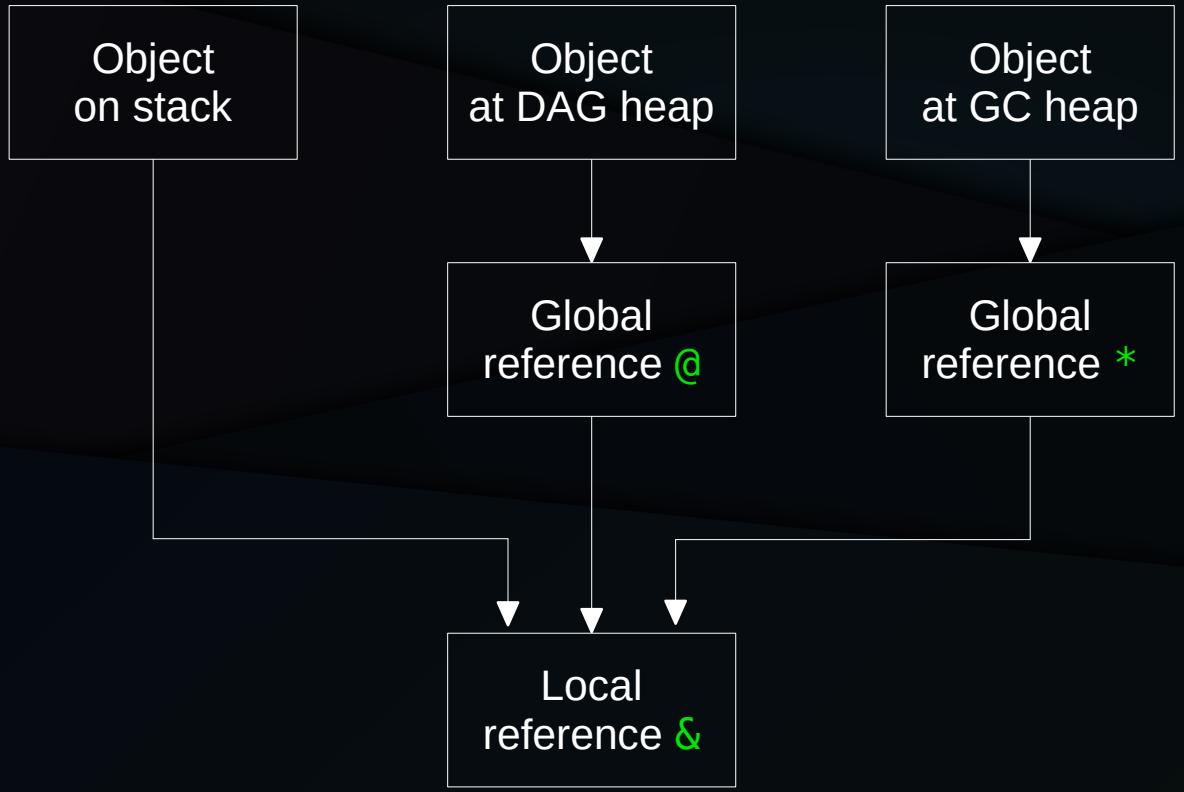
- Each memory management action locks the big mutex
- O(N) freezes are possible
  (despite the heuristic optimizations)

# Memory model

## Brave new world Garbage Collector

- The only point of synchronization is passing batch of events into graph handler:
  - after link deletion
  - after sending shared object (if handled by GC) to another thread
- The synchronization event duration is a duration of single atomic operation
- Graph management is handled by collecting events into queue
  instead of immediate application

# Memory model




```
Obj → &
  & → &
  @ → @
  * → *
  @ → &
  * → &
```

# Memory model

## Freezing GC, no RAII, Zombie-objects, ...

```
function undefined_file_close_time()
    local fh = assert(io.open("data1.txt", "w"))
    fh:write("hello\n")
    -- Lua GC: 'fh' is closed in undpredictable time
end

function zombie_object()
    local fh, err = io.open("data2.txt", "w")
    if fh then
        fh:write("hello\n")
        fh:close()
        -- 'fh' is zombie-object here
    else
        print(err)
    end
end
```

# Memory model

## Enough of this!

```
void basic_raii ()
{
    variant<mut io.file@|error@> maybe_fh = file_system.open ("data1.txt", io.READ_ONLY)
    match (maybe_fh) {
        case (mut io.file& fh) {
            let (ubyte[]& data = fh.read_all ()) {
                stdout.write (data)
                stdout.write ("\n")
            }
        }
        case (error@ err) {
            stderr.println (err.description ())
        }
    }
    // file is closed as in basic RAII
}
```

# Memory model

## Enough of this!

```
class gl_utils.mesh (
    ubyte[] url;
    ubyte[] local_path;
    .geometry geometry; /* .geometry <=> gl_utils.mesh.geometry */
    mut variant<mut io.file@|error@|null>* fh; // global reference: handled by GC
)

optional error@ load_mesh (mut struct mesh& mesh)
{
    variant<ubyte[]|error@> data // Not initialized here
    match (mesh.fh) { // Initializing 'data' in each branch
        case (mut io.file& fh) {
            data = fh.read_all ()
            // `mesh.fh := null` would be compile-time error
        }
        case (error@ err) {
            data = err
        }
        case (null) {
            data = error@ ("Resource already loaded")
        }
    }
    mesh.fh := null
    // No zombie object here
    return mesh.parse_mesh (must data)
}
```

# Memory model

## Function parameters

- 3 parameter types:
  - immutable
  - uninitialized
  - mutable
- All arguments are passed only by reference
- Local links don’t leave it’s scope
- Functions only have access to arguments and objects (if function is a method). No global variables, upvalues, etc.
- Nested initialization is optimal
- Return value access is optimal (no RVO ambiguity)

# Memory model

## Function parameters and return values

```
void check_eq (int& value, out bool& equal)
{
    equal = value == max
}

bool acc_max (int& next_value, out bool& equal, mut int& max)
{
    if (next_value > max) {
        max := next_value
        equal = false
        return true  // Sugar for { retval = true; return }
    } else {
        check_eq (next_value, equal)
        return false // Sugar for { retval = false; return }
    }
}
```

# Memory model

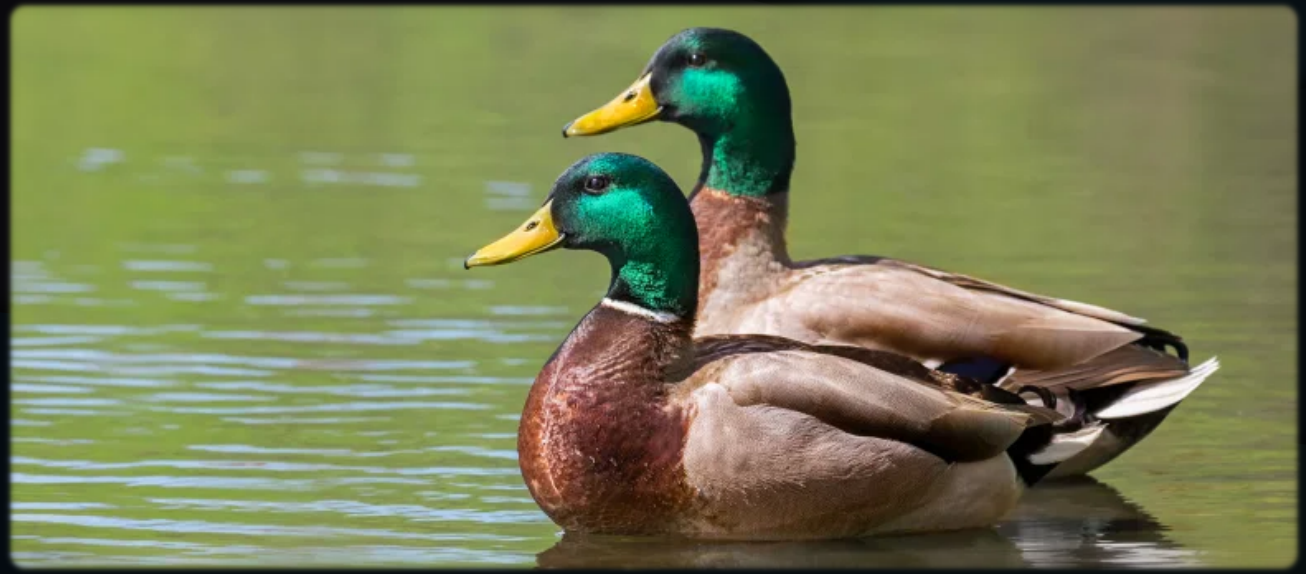
## Function parameters and return values

```
variant<double|double[2]|null|enum .> solve_square_equation (double& a, double& b, double& c)
{
    if (math.abs (a) < math.epsilon ()) {
        if (math.abs (b) < math.epsilon ()) {
            if (math.abs (b) < math.epsilon ()) {
                retval = enum .any
            } else {
                retval = null
            }
        } else {
            retval = c/b
        }
        return
    }

    double d = b*b - 4.0*a*c
    if (d >= 0.0) {
        double sqrt_d = math.sqrt (d)
        return double[2] ((b - sqrt_d)/(-2.0*a), (b + sqrt_d)/(-2.0*a)) // Sugar for { retval = VALUE; return }
    }
    retval = enum .complex
}
```

# Templates

Templates in C++, Java, C#



# Templates

## Templates in HardCode

- Template arguments are strictly typed
- Static polymorphism is a substituted implementation of dynamically polymorphic object.
- Or in simple words:
  - Polymorphic object is defined as such once
  - Then it can be used both dynamically and statically

## OOP in The World

“I made up the term object-oriented, and I can tell you I did not have C++ in mind.”

-- Alan Kay, OOPSLA '97

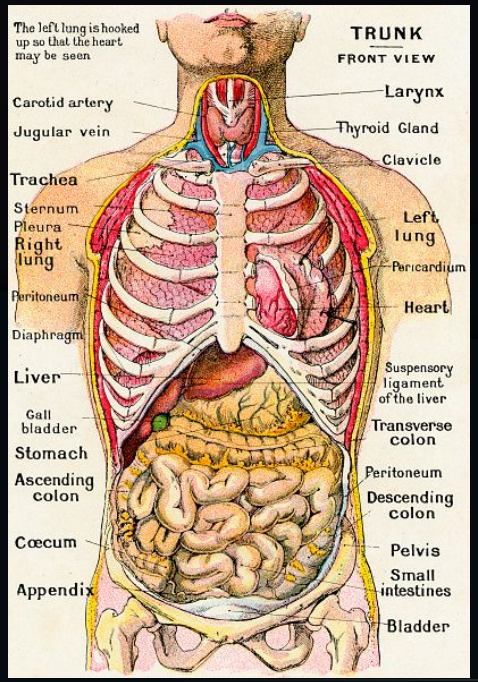
## OOP in HardCode

- Inspired by Game Development dynamic polymorphism: Entity model
- Composition over Inheritance
- Class always inherits an Interface, exactly one Interface
- Classes aren’t inherited

# OOP

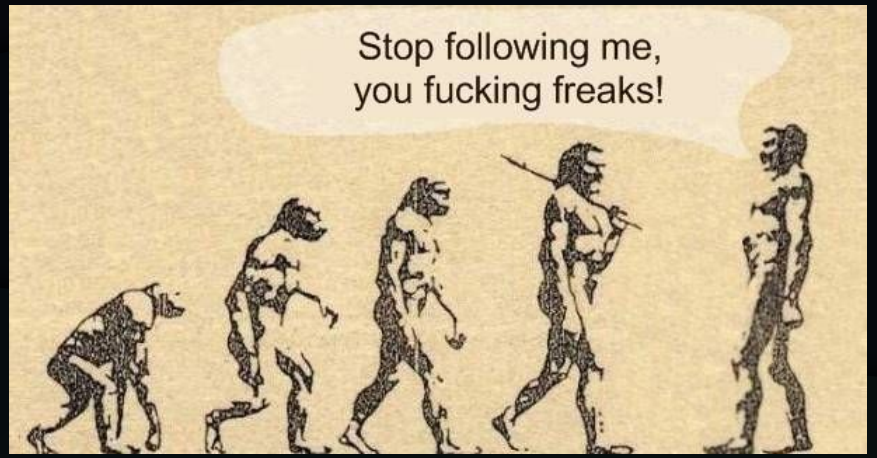
Composition over Inheritance

Composition is for Space



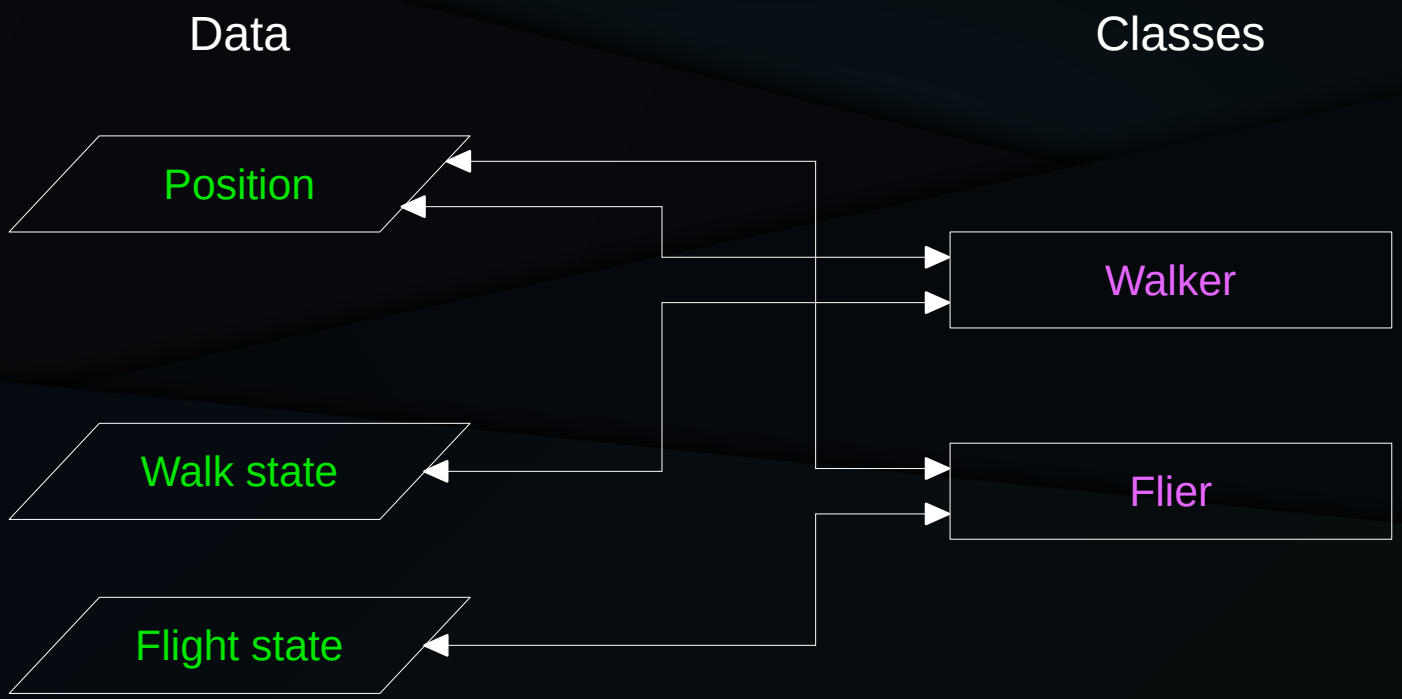

over

Inheritance is for Time



# Dynamic polymorphism

Data to functions relation: many to many



# Dynamic polymorphism

## Data to functions relation: many to many

```
// Basic data sections
class position (
    vec3 translation
    quat orientation
)
class walk_state (
    bool forward, back, left, right
    double phase
)
class flight_state (
    bool forward, back
    double flap_period
    double phase
)

// Proxy data sections
class walker: implements walker (
    mut class position* position
    mut class walk_state* walk_state
)
class flier: implements flyer (
    mut class position* position
    mut class flight_state* flight_state
)
```

```
// Interfaces
interface walker (
    void start_move (enum .direction& direction)
    void stop_move (enum .direction& direction)
    void update (double& delta_time)
)
interface flier (
    void start_move (enum .direction& direction)
    void stop_move (enum .direction& direction)
    void set_flap_period (double& flap_period)
    void update (double& delta_time)
)



{
    entity cute_thing = world.new_entity ()
    cute_thing.extend (walker ()) // +position, +walk_state
    cute_thing.extend (flier ())  // +flight_state
}
```

# Source code storage

## Object model in C++

Directory (nestable)
⇩⇩⇩
File (not nestable)
⇩⇩⇩
Namespace (nestable and intersectable)
⇩⇩⇩
Class (nestable)
⇩⇩⇩
Method/Function (not nestable)

# Source code storage

## Demo

*In interactive presentation here would be demonstration*
*of previous implementation of following source code storage concept*

# Source code storage

## HardCode approach

- Source code is stored at dedicated versioned DB
- All entities (interfaces, classes, enumerations, functions, …) are stored as uniform tree
- Linkage information is generated automatically and stored explicitly
- Only valid source code is accepted for storage
- Source code access is performed through IDE
- Language is designed for dynamic linkage
- Module dependencies are encapsulated though deduplicated

# Source code storage

## HardCode approach advantages

- Hot development: no need to restart an application
- Uniform entity structure designed for convenient refactoring
- Strict source code testability for “pure” code (where no IO is involved)
- Mock-driven design out-of-the-box for IO testing
- Embedding (like in Lua, Python, JS, etc. but better)
- Sandboxing (like in Lua but better)
- Consistent server-side code generation (e. g. for Protobuf)
- Development-related procedures management:
  - Autotests execution
  - Pull request, rebase, virtual rebase, review
  - Server-side validation of production branches

# Source code storage

## Capabilities (features to allow/disallow per branch)

- Allow using GC
- Allow using DAG heap
- Allow functions with no guarantees of completion
- Allow dirty source code (with WARNINGs):
  good for development branches, bad for production
- Allow thread management
- Allow only clean history
  (like using `` `merge --no-ff` `` only when `` `merge --ff-only` `` is possible)

# Source code storage

## Refactoring example: original version

```
template <interface real $real>
math.mat4 math.mat4.inverse_affine ()
{
    mut $real v0 = m[2][0]*m[3][1] - m[2][1]*m[3][0]
    mut $real v1 = m[2][0]*m[3][2] - m[2][2]*m[3][0]
    mut $real v2 = m[2][0]*m[3][3] - m[2][3]*m[3][0]
    mut $real v3 = m[2][1]*m[3][2] - m[2][2]*m[3][1]
    mut $real v4 = m[2][1]*m[3][3] - m[2][3]*m[3][1]
    mut $real v5 = m[2][2]*m[3][3] - m[2][3]*m[3][2]

    $real t00 = + (v5*m[1][1] - v4*m[1][2] + v3*m[1][3])
    $real t10 = - (v5*m[1][0] - v2*m[1][2] + v1*m[1][3])
    $real t20 = + (v4*m[1][0] - v2*m[1][1] + v0*m[1][3])
    $real t30 = - (v3*m[1][0] - v1*m[1][1] + v0*m[1][2])

    $real inv_det = 1.0/(t00*m[0][0] + t10*m[0][1] + t20*m[0][2] + t30*m[0][3])

    retval[0][0] = t00*inv_det
    retval[1][0] = t10*inv_det
    retval[2][0] = t20*inv_det
    retval[3][0] = t30*inv_det

    retval[0][1] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][1] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][1] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][1] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det

    v0 := m[1][0]*m[3][1] - m[1][1]*m[3][0]
    v1 := m[1][0]*m[3][2] - m[1][2]*m[3][0]
    v2 := m[1][0]*m[3][3] - m[1][3]*m[3][0]
    v3 := m[1][1]*m[3][2] - m[1][2]*m[3][1]
    v4 := m[1][1]*m[3][3] - m[1][3]*m[3][1]
    v5 := m[1][2]*m[3][3] - m[1][3]*m[3][2]

    retval[0][2] = + (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][2] = - (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][2] = + (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][2] = - (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det

    v0 := m[2][1]*m[1][0] - m[2][0]*m[1][1]
    v1 := m[2][2]*m[1][0] - m[2][0]*m[1][2]
    v2 := m[2][3]*m[1][0] - m[2][0]*m[1][3]
    v3 := m[2][2]*m[1][1] - m[2][1]*m[1][2]
    v4 := m[2][3]*m[1][1] - m[2][1]*m[1][3]
    v5 := m[2][3]*m[1][2] - m[2][2]*m[1][3]

    retval[0][3] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][3] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][3] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][3] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
}
```

# Source code storage

## Refactoring example: step 1

```
math.mat4 math.mat4.inverse_affine ()
{
    $real inv_det

    {
        $real v0 = m[2][0]*m[3][1] - m[2][1]*m[3][0]
        $real v1 = m[2][0]*m[3][2] - m[2][2]*m[3][0]
        $real v2 = m[2][0]*m[3][3] - m[2][3]*m[3][0]
        $real v3 = m[2][1]*m[3][2] - m[2][2]*m[3][1]
        $real v4 = m[2][1]*m[3][3] - m[2][3]*m[3][1]
        $real v5 = m[2][2]*m[3][3] - m[2][3]*m[3][2]

        $real t00 = + (v5*m[1][1] - v4*m[1][2] + v3*m[1][3])
        $real t10 = - (v5*m[1][0] - v2*m[1][2] + v1*m[1][3])
        $real t20 = + (v4*m[1][0] - v2*m[1][1] + v0*m[1][3])
        $real t30 = - (v3*m[1][0] - v1*m[1][1] + v0*m[1][2])

        inv_det = 1.0/(t00*m[0][0] + t10*m[0][1] + t20*m[0][2] + t30*m[0][3])

        retval[0][0] = t00*inv_det
        retval[1][0] = t10*inv_det
        retval[2][0] = t20*inv_det
        retval[3][0] = t30*inv_det

        retval[0][1] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
        retval[1][1] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
        retval[2][1] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
        retval[3][1] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
    }

    {
        $real v0 = m[1][0]*m[3][1] - m[1][1]*m[3][0]
        $real v1 = m[1][0]*m[3][2] - m[1][2]*m[3][0]
        $real v2 = m[1][0]*m[3][3] - m[1][3]*m[3][0]
        $real v3 = m[1][1]*m[3][2] - m[1][2]*m[3][1]
        $real v4 = m[1][1]*m[3][3] - m[1][3]*m[3][1]
        $real v5 = m[1][2]*m[3][3] - m[1][3]*m[3][2]

        retval[0][2] = + (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
        retval[1][2] = - (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
        retval[2][2] = + (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
        retval[3][2] = - (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
    }

    {
        $real v0 = m[2][1]*m[1][0] - m[2][0]*m[1][1]
        $real v1 = m[2][2]*m[1][0] - m[2][0]*m[1][2]
        $real v2 = m[2][3]*m[1][0] - m[2][0]*m[1][3]
        $real v3 = m[2][2]*m[1][1] - m[2][1]*m[1][2]
        $real v4 = m[2][3]*m[1][1] - m[2][1]*m[1][3]
        $real v5 = m[2][3]*m[1][2] - m[2][2]*m[1][3]

        retval[0][3] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
        retval[1][3] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
        retval[2][3] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
        retval[3][3] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
    }
}
```

# Source code storage

## Refactoring example: step 2

```
void math.mat4.inverse_affine.`.fill_col01` (out math.mat4<$real> ret, out $real inv_det)
{
    $real v0 = m[2][0]*m[3][1] - m[2][1]*m[3][0]
    $real v1 = m[2][0]*m[3][2] - m[2][2]*m[3][0]
    $real v2 = m[2][0]*m[3][3] - m[2][3]*m[3][0]
    $real v3 = m[2][1]*m[3][2] - m[2][2]*m[3][1]
    $real v4 = m[2][1]*m[3][3] - m[2][3]*m[3][1]
    $real v5 = m[2][2]*m[3][3] - m[2][3]*m[3][2]

    $real t00 = + (v5*m[1][1] - v4*m[1][2] + v3*m[1][3])
    $real t10 = - (v5*m[1][0] - v2*m[1][2] + v1*m[1][3])
    $real t20 = + (v4*m[1][0] - v2*m[1][1] + v0*m[1][3])
    $real t30 = - (v3*m[1][0] - v1*m[1][1] + v0*m[1][2])

    inv_det = 1.0/(t00*m[0][0] + t10*m[0][1] + t20*m[0][2] + t30*m[0][3])

    retval[0][0] = t00*inv_det
    retval[1][0] = t10*inv_det
    retval[2][0] = t20*inv_det
    retval[3][0] = t30*inv_det

    retval[0][1] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][1] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][1] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][1] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
}
```

# Source code storage

## Refactoring example: step 2

```
void math.mat4.inverse_affine.`.fill_col2` (out math.mat4<$real> ret, $real& inv_det)
{
    $real v0 = m[1][0]*m[3][1] - m[1][1]*m[3][0]
    $real v1 = m[1][0]*m[3][2] - m[1][2]*m[3][0]
    $real v2 = m[1][0]*m[3][3] - m[1][3]*m[3][0]
    $real v3 = m[1][1]*m[3][2] - m[1][2]*m[3][1]
    $real v4 = m[1][1]*m[3][3] - m[1][3]*m[3][1]
    $real v5 = m[1][2]*m[3][3] - m[1][3]*m[3][2]

    retval[0][2] = + (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][2] = - (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][2] = + (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][2] = - (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
}
```

# Source code storage

## Refactoring example: step 2

```
void math.mat4.inverse_affine.`.fill_col3` (
    out math.mat4<$real> ret, $real& inv_det)
{
    $real v0 = m[2][1]*m[1][0] - m[2][0]*m[1][1]
    $real v1 = m[2][2]*m[1][0] - m[2][0]*m[1][2]
    $real v2 = m[2][3]*m[1][0] - m[2][0]*m[1][3]
    $real v3 = m[2][2]*m[1][1] - m[2][1]*m[1][2]
    $real v4 = m[2][3]*m[1][1] - m[2][1]*m[1][3]
    $real v5 = m[2][3]*m[1][2] - m[2][2]*m[1][3]

    retval[0][3] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][3] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][3] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][3] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
}
```

# Source code storage

## Refactoring example: step 2

```
- math
  - mat4
    - inverse_affine
      - .fill_col01
      - .fill_col2
      - .fill_col3

math.mat4 math.mat4.inverse_affine ()
{
    $real inv_det
    .fill_col01 (retval=, inv_det=)
    .fill_col2 (retval=, inv_det)
    .fill_col3 (retval=, inv_det)
}
```

# Source code storage

## Refactoring example: step 3

```
void math.mat4.inverse_affine.`.fill_col01`.`.calc_det` ($real& t00, $real& t10, $real& t20, $real& t30)
{
    return 1.0/(t00*m[0][0] + t10*m[0][1] + t20*m[0][2] + t30*m[0][3])
}
```

# Source code storage

## Refactoring example: step 3

```
void math.mat4.inverse_affine.`.fill_col01` (out math.mat4<$real> ret, out $real inv_det)
{
    $real v0 = m[2][0]*m[3][1] - m[2][1]*m[3][0]
    $real v1 = m[2][0]*m[3][2] - m[2][2]*m[3][0]
    $real v2 = m[2][0]*m[3][3] - m[2][3]*m[3][0]
    $real v3 = m[2][1]*m[3][2] - m[2][2]*m[3][1]
    $real v4 = m[2][1]*m[3][3] - m[2][3]*m[3][1]
    $real v5 = m[2][2]*m[3][3] - m[2][3]*m[3][2]

    $real t00 = + (v5*m[1][1] - v4*m[1][2] + v3*m[1][3])
    $real t10 = - (v5*m[1][0] - v2*m[1][2] + v1*m[1][3])
    $real t20 = + (v4*m[1][0] - v2*m[1][1] + v0*m[1][3])
    $real t30 = - (v3*m[1][0] - v1*m[1][1] + v0*m[1][2])

    inv_det = .calc_det (t00, t10, t20, t30)

    retval[0][0] = t00*inv_det
    retval[1][0] = t10*inv_det
    retval[2][0] = t20*inv_det
    retval[3][0] = t30*inv_det

    retval[0][1] = - (v5*m[0][1] - v4*m[0][2] + v3*m[0][3])*inv_det
    retval[1][1] = + (v5*m[0][0] - v2*m[0][2] + v1*m[0][3])*inv_det
    retval[2][1] = - (v4*m[0][0] - v2*m[0][1] + v0*m[0][3])*inv_det
    retval[3][1] = + (v3*m[0][0] - v1*m[0][1] + v0*m[0][2])*inv_det
}
```

# Source code storage

## Refactoring example: step 3

```
- math
  - mat4
    - inverse_affine
      - .fill_col01
        - .calc_det
      - .fill_col2
      - .fill_col3

math.mat4 math.mat4.inverse_affine ()
{
    $real inv_det
    .fill_col01 (retval=, inv_det=)
    .fill_col2 (retval=, inv_det)
    .fill_col3 (retval=, inv_det)
}
```

# Dynamic linkage

## Why would one want dynamic typing?

Only to simplify dynamic linkage (JS example):

```
function (a, b) {
    console.log(a, b)
}

function (...args) {
    var a = args[0]
    var b = args[1]
}
```

– all functions in JS are identical to linker

# Dynamic linkage

## Dynamic linkage, static typing

- Is convenient due to linkage information generation and storage at VCS
- Executes much faster than code with dynamic typing
- Eliminates (not by itself)
  fatal run-time errors
- Allows to automatically recompile only dependent functions without application restart
- Embedding
- Sandboxing
- Actual Embedding and Sandboxing in HardCode

# Complexity analysis

## Misconception from school

Thesis:
«One can’t simply prove
Turing-complete program terminability»

Misconception:
«We care about Turing-complete programs»

# Complexity analysis

## One more misconception from school

«We need recursions»

# Complexity analysis

## Some other conception

- We need recursions very much!
  To write unanalyzable programs.
- We need hanging up programs!
  So our code would hang up from time to time
  and we got payed for codebase support forever.
  Job security per se.

# Complexity analysis

## But really

- Some handy things for development:
  - recursions
  - infinite loops
  - unused variables and other dirty code –
    all the stuff triggering WARNINGs

  Recursions and infinite loops are very handy
  for implementing some complex recursive algorithm.
  Allowing for dirty code is handy for refactoring and debug
  (try refactoring Golang code – unused variable errors will get you suicidal).
  But all the things above are absolutely intolerable for production code!!

- Recursions by themselves aren’t friendly with analysis.
  On the hand, recursion patterns probably are.

# Complexity analysis

## Noble traits of decent programs

- A program guarantee interruptability for any possible input data.
  This must be a guarantee by a **programming language**.

- No unanalyzable behavior is allowed in the language:
  - unsafe memory operations
  - unconditional jump operator `goto`
  - loops without strict exit condition (*allowed in development branch*)
  - undefined behavior due to synchronization errors
  - memory management bottlenecks
  - recursions (*allowed in development branch*)

# Complexity analysis

## Analysis example: Heapsort

```
void heapify (mut int[]&:check (#this > 0) elements)
void extract_top (mut int[]&:check (#this > 0) elements)
void drown_top (mut int[]&:check (#this > 0) elements)

void sort (mut int[]& elements)
{
    mut int[]&:check (#this > 0) elements_ne = must elements
    heapify (elements_ne)
    index (elements_ne) head = 0
    for desc (index (elements_ne) c: #elements_ne - 1, 0) {
        extract_top (elements_ne[head:c + 1])
        drown_top (elements_ne[head:c])
    }
}
```

# Complexity analysis

## Heapsort implementation

```
void heapify (mut int[]&:check (#this > 0) elements)
{
    word:check (#this > 0) element_count = #elements
    word height = log2floor (element_count)
    for (word h: 0, height) {
        word b = (1 << height) - 2
        word e = (1 << h) >> 1
        for desc (word i: b, e - 1) {
            index (elements) idx1 = must i
            index (elements) idx2 = must (i*2 + 1)
            if (elements[idx1] < elements[idx2])
                swap (elements[idx1], elements[idx2])
            index (elements) idx3 = must (idx2 + 1)
            if (elements[idx1] < elements[idx3])
                swap (elements[idx1], elements[idx3])
        }
    }
}
```

```
void heapify (int* elements, ssize_t count)
{
    ssize_t height = log2floor (count); // Undefined behavior for count <= 0
    for (ssize_t h = 0; h < height; ++h) {
        ssize_t b = (1 << height) - 2;
        ssize_t e = (1 << h) >> 1;
        for (ssize_t i = b; i >= e; --i) {
            ssize_t idx1 = i;
            if (idx1 >= count)
                return;
            ssize_t idx2 = i*2 + 1;
            if (idx2 >= count)
                return;
            if (elements[idx1] < elements[idx2])
                swap (&elements[idx1], &elements[idx2]);
            ssize_t idx3 = idx2 + 1;
            if (idx3 >= count)
                return;
            if (elements[idx1] < elements[idx3])
                swap (&elements[idx1], &elements[idx3]);
        }
    }
}
```

# Complexity analysis

## Heapsort implementation

```
void extract_top (mut int[]&:check (#this > 0) elements)
{
    index (elements) head = 0
    index (elements) tail = #elements - 1
    swap (elements[head], elements[tail])
}
```

```
void extract_top (int* elements, ssize_t count)
{
    swap (&elements[0], &elements[count - 1]);
}
```

# Complexity analysis

## Heapsort implementation

```
void drown_top (mut int[]&:check (#this > 0) elements)
{
    word:check (this > 0) element_count = #elements
    word height = log2floor (element_count)
    index (elements) idx = 0
    for (word _: 0, height) {
        index (elements) idx1 = idx
        index (elements) idx2 = must (idx*2 + 1)
        let (index (elements) idx3 = idx2 + 1) {
            if (elements[idx1] < elements[idx2] &&
                elements[idx2] > elements[idx3]) {
                swap (elements[idx1], elements[idx2])
                idx = idx2
            } else if (elements[idx1] < elements[idx3]) {
                swap (elements[idx1], elements[idx3])
                idx = idx3
            } else {
                return
            }
        } else {
            if (elements[idx1] < elements[idx2])
                swap (elements[idx1], elements[idx2])
            return
        }
    }
}
```

```
void drown_top (int* elements, ssize_t count)
{
    ssize_t idx = 0;
    ssize_t height = log2floor (count); // Undefined behavior for count <= 0
    for (ssize_t i = 0; i < height; ++i) {
        ssize_t idx1 = idx;
        ssize_t idx2 = idx*2 + 1;
        ssize_t idx3 = idx2 + 1;
        if (idx2 >= count)
            return;
        if (idx3 >= count) {
            if (elements[idx1] < elements[idx2])
                swap (&elements[idx1], &elements[idx2]);
            return;
        }
        if (elements[idx1] < elements[idx2] &&
            elements[idx2] > elements[idx3]) {
            swap (&elements[idx1], &elements[idx2]);
            idx = idx2;
        } else if (elements[idx1] < elements[idx3]) {
            swap (&elements[idx1], &elements[idx3]);
            idx = idx3;
        } else {
            return;
        }
    }
}
```

# Complexity analysis

## Heapsort implementation

```
void sort (mut int[]& elements)
{
    mut int[]&:check (#this > 0) elements_ne = must elements
    heapify (elements_ne)
    index (elements_ne) head = 0
    for desc (index (elements_ne) c: #elements_ne - 1, 0) {
        extract_top (elements_ne[head:c + 1])
        drown_top (elements_ne[head:c])
    }
}
```

```
void sort (int* elements, ssize_t count)
{
    if (count <= 0)
        return;
    heapify (elements, count);
    for (ssize_t c = count; c > 1; --c) {
        extract_top (elements, c);
        drown_top (elements, c - 1);
    }
}
```

# Complexity analysis

## Heapsort: classification

```
void heapify (mut int[]&:check (#this > 0) elements)
{
    word:check (this > 0) element_count = #elements   // load (ram, 64)
    word height = log2floor (element_count)           // bitwise (64)
    for (word h: 0, height) {                         // copy (reg, 64)
                                                      // log2floor (#elements) x iarith (64)
                                                      // log2floor (#elements) x icmp (64)
                                                      // log2floor (#elements) x {
        word b = (1 << height) - 2                    //     bitwise (64)
                                                      //     iarith (64)
        word e = (1 << h) >> 1                        //     bitwise (64)
                                                      //     bitwise (64)
        for desc (word i: b, e - 1) {                 //     copy (reg, 64)
                                                      //     iarith (64)
                                                      //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x iarith (64)
                                                      //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x icmp (64)
                                                      //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x {
            index (elements) idx1 = must i            //         copy (reg, 64)
                                                      //         icmp (reg, 64)
            index (elements) idx2 = must (i*2 + 1)    //         iarith (64)
                                                      //         iarith (64)
                                                      //         icmp (reg, 64)
            if (elements[idx1] < elements[idx2])      //         load (ram, 32)
                                                      //         load (ram, 32)
                                                      //         icmp (32)
                swap (elements[idx1], elements[idx2]) //         load (ram, 32)
                                                      //         load (ram, 32)
                                                      //         copy (reg, 32)
                                                      //         store (ram, 32)
                                                      //         store (ram, 32)
            index (elements) idx3 = must (idx2 + 1)   //         iarith (64)
                                                      //         icmp (reg, 64)
            if (elements[idx1] < elements[idx3])      //         load (ram, 32)
                                                      //         load (ram, 32)
                                                      //         icmp (32)
                swap (elements[idx1], elements[idx3]) //         load (ram, 32)
                                                      //         load (ram, 32)
                                                      //         copy (reg, 32)
                                                      //         store (ram, 32)
                                                      //         store (ram, 32)
        }                                             //     } // (b - e)
    }                                                 // } // log2floor(#elements)
}
```

# Complexity analysis

## Heapsort: classification

```
void extract_top (mut int[]&:check (#this > 0) elements)
{
    index (elements) head = 0                          // copy (reg, 64)
    index (elements) tail = #elements - 1              // iarith (64)
    swap (elements[head], elements[tail])              // load (ram, 32)
                                                       // load (ram, 32)
                                                       // copy (reg, 32)
                                                       // store (ram, 32)
                                                       // store (ram, 32)
}
```

# Complexity analysis

## Heapsort: classification

```
void drown_top (mut int[]&:check (#this > 0) elements)
{
    word:check (this > 0) element_count = #elements
    word height = log2floor (element_count)
    index (elements) idx = 0
    for (word _: 0, height) {



        index (elements) idx1 = idx
        index (elements) idx2 = must (idx*2 + 1)


        let (index (elements) idx3 = idx2 + 1) {

            if (elements[idx1] < elements[idx2] &&


                elements[idx2] > elements[idx3]) {




                swap (elements[idx1], elements[idx2])




                idx = idx2

            } else if (elements[idx1] < elements[idx3]) {


                swap (elements[idx1], elements[idx3])




                idx = idx3
            } else {
                return
            }

        } else {
            if (elements[idx1] < elements[idx2])


                swap (elements[idx1], elements[idx2])



            return
        }
    }
}
```

```
// load (ram, 64)
// bitwise (64)
// copy (reg, 64)
// copy (reg, 64)
// log2floor (#elements) x iarith (64)
// log2floor (#elements) x icmp (64)
// log2floor (#elements) x {
//     copy (reg, 64)
//     iarith (64)
//     iarith (64)
//     icmp (32)
//     max {
//         iarith (64)
//         load (ram, 32)
//         load (ram, 32)
//         icmp (32)
//         load (ram, 32)
//         load (ram, 32)
//         icmp (32)
//         bool (64)
//         max {
//             load (ram, 32)
//             load (ram, 32)
//             copy (reg, 32)
//             store (ram, 32)
//             store (ram, 32)
//             copy (reg, 64)
//         } or {
//             load (ram, 32)
//             load (ram, 32)
//             icmp (32)
//             max {
//                 load (ram, 32)
//                 load (ram, 32)
//                 copy (reg, 32)
//                 store (ram, 32)
//                 store (ram, 32)
//                 copy (reg, 64)
//             } or {

//             }
//         }
//     } or {
//         load (ram, 32)
//         load (ram, 32)
//         icmp (32)
//         load (ram, 32)
//         load (ram, 32)
//         copy (reg, 32)
//         store (ram, 32)
//         store (ram, 32)

//     }
// }
```

# Complexity analysis

## Heapsort: classification

```
void sort (mut int[]& elements)
{
    mut int[]&:check (#this > 0) elements_ne = must elements
    heapify (elements_ne)
    index (elements_ne) head = 0
    for desc (index (elements_ne) c: #elements_ne - 1, 0) {



        extract_top (elements_ne[head:c + 1])


        drown_top (elements_ne[head:c])


    }
}
```

```
// icmp (64)
// call "heapify (elements_ne)"
// set (reg, 64)
// iarith (64)
// (#elements - 1) x iarith (64)
// (#elements - 1) x icmp (64)
// (#elements - 1) x {
//     iarith (64)
//     slice
//     call "extract_top (elements_ne[head:c + 1])"
//     copy (64)
//     slice
//     call "drown_top (elements_ne[head:c])"
// }
```

# Complexity analysis

## Rules for score assignment

```
 load (ram) = 1 + 1 mt (ram)
store (ram) = 1 + 1 mt (ram)
 copy (reg) = 1
  set (reg) = 1
    bitwise = 2
       bool = 2
     iarith = 5
       icmp = 2
      slice = 2 + 2 mt (ram)
```

# Complexity analysis

## Score evaluation

```
void heapify (mut int[]&:check (#this > 0) elements)
{
    word:check (#this > 0) element_count = #elements          // 1 + 1 mt (ram)
    word height = log2floor (element_count)                   // 2
    for (word h: 0, height) {                                 // 1
                                                              // log2floor (#elements) x 5
                                                              // log2floor (#elements) x 2
                                                              // log2floor (#elements) x {
        word b = (1 << height) - 2                            //     2
                                                              //     5
        word e = (1 << h) >> 1                                //     2
                                                              //     2
        for desc (word i: b, e - 1) {                         //     1
                                                              //     5
                                                              //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x 5
                                                              //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x 2
                                                              //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x {
            index (elements) idx1 = must i                    //         1
                                                              //         2
            index (elements) idx2 = must (i*2 + 1)            //         5
                                                              //         5
                                                              //         2
            if (elements[idx1] < elements[idx2])              //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
                                                              //         2
                swap (elements[idx1], elements[idx2])         //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
                                                              //         1
                                                              //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
            index (elements) idx3 = must (idx2 + 1)           //         5
                                                              //         2
            if (elements[idx1] < elements[idx3])              //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
                                                              //         2
                swap (elements[idx1], elements[idx3])         //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
                                                              //         1
                                                              //         1 + 1 mt (ram)
                                                              //         1 + 1 mt (ram)
        }                                                     //     } // (b - e)
    }                                                         // } // log2floor(#elements)
}
```

# Complexity analysis

## Score evaluation

```
void heapify (mut int[]&:check (#this > 0) elements)
{
    // 4 + 1 mt (ram)
    // log2floor (#elements) x 24
    // log2floor (#elements) x { | h: [0, log2floor (#elements)); asc
    //     (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2) x 47 + 12 mt (ram)
    // }
}
void extract_top (mut int[]&:check (#this > 0) elements)
{
    // 13 + 4 mt (ram)
}
void drown_top (mut int[]&:check (#this > 0) elements)
{
    // 5 + 1 mt (ram)
    // log2floor (#elements) x 45 + 10 mt (ram)
}
void sort (mut int[]& elements)
{
    // 8
    // call "heapify (elements_ne)"
    // (#elements - 1) x 17 + 4 mt (ram)
    // (#elements - 1) x { | c: [#elements_ne - 1, 0); desc
    //     17 + 4 mt (ram)
    //     call "extract_top (elements_ne[0:c + 1])"
    //     call "drown_top (elements_ne[0:c])"
    // }
}
```

# Complexity analysis

## Score evaluation

```
void sort (mut int[]& elements)
{
    // 12 + 1 mt (ram)
    // log2floor (#elements) x 24
    // log2floor (#elements) x { | h: [0, log2floor (#elements)); asc
    //     max (((1 << h) >> 1) - (1 << log2floor (#elements)) + 2, 0) x 47 + 12 mt (ram)
    // }
    // max (0, #elements - 1) x 53 + 13 mt (ram)
    // max (0, #elements - 1) x { | c: [#elements - 1, 0); desc
    //     log2floor (c + 1) x 45 + 10 mt (ram)
    // }
}
```

# Complexity analysis

## Score evaluation (Bubble sort)

```
void bubble_sort (mut int[]& elements)
{
    // 1
    // max (0, #elements - 1) x 13
    // max (0, #elements - 1) x {
    //     j x { | j: [#elements - 2, 0); desc
    //         16 + 6 mt (ram)
    //     }
    // }
}
```

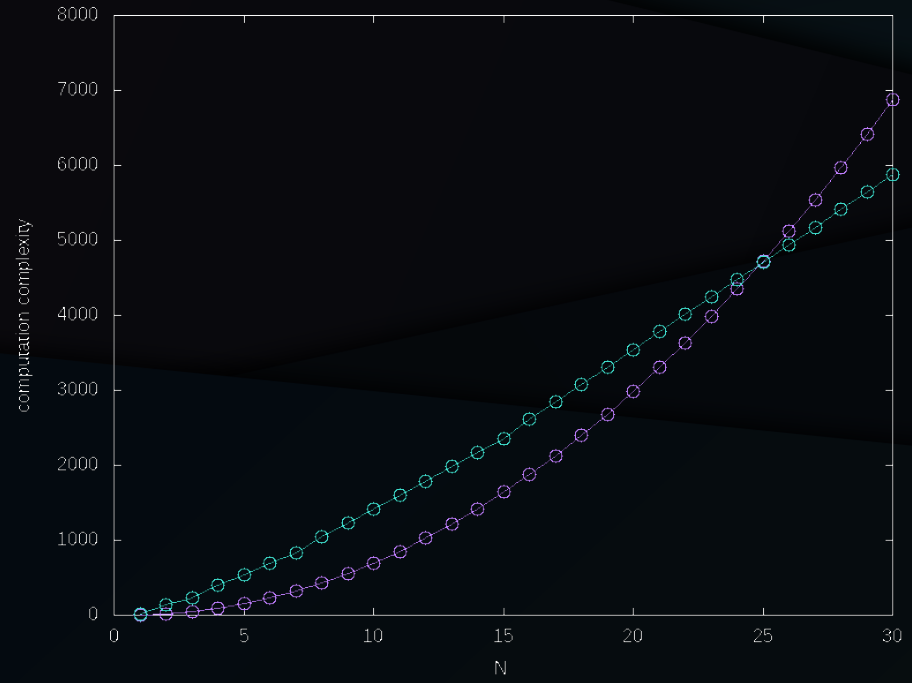
# Complexity analysis

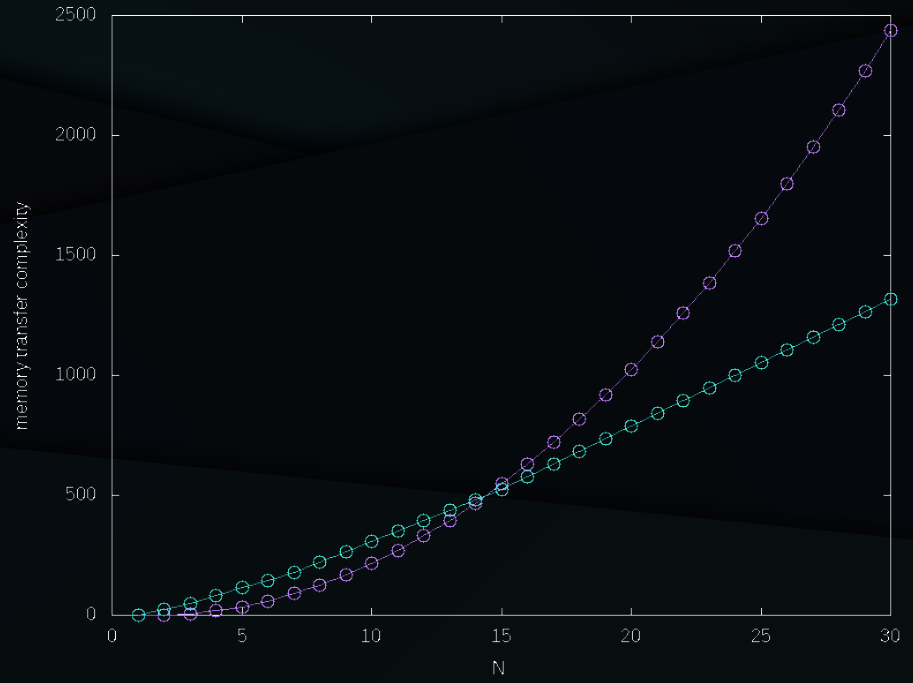
## Dimensions substitution

```
void test_heap_sort ()
{
    {
        mut int[]& elements = make_random (8)
        heap_sort (elements) // 1040 + 222 mt
    }
    {
        mut int[]& elements = make_random (27)
        heap_sort (elements) // 5176 + 1159 mt
    }
}
void test_bubble_sort ()
{
    {
        mut int[]& elements = make_random (8)
        bubble_sort (elements) // 428 + 126 mt
    }
    {
        mut int[]& elements = make_random (27)
        bubble_sort (elements) // 5539 + 1950 mt
    }
}
```

# Complexity analysis

## Merging algorithms



# Complexity analysis

## Particular case analysis

- Tracking conditional jumps `if`, `let`, `must`, `match`
- Tracking `return` and `break`
- Getting score for actually (not probably) executed instructions

# Complexity analysis

## Distribution analysis

- Generating random data
- Or take actual requests from production
- Or sometimes even generating full coverage automatically
- Getting score distribution for actually (not probably) executed instructions

# Complexity analysis

## Stochastic analysis

- E. g., hash arrays have worst case of O(N) for single element access
- For such cases worst case analysis is worthless
- We’ll have to relax our analysis a little bit:
  a kind of leniency for functions heavily reliant on data distribution
- Objects marked as stochastic for analysis
  must be under control of a compiler (as some kind of IO)
- And so user functions using stochastic IO have
  to be considering average case of data distribution

# Complexity analysis

## Complexity before of uninterruptible section analysis

- Functions working with system resources (such as network or file system), are mostly in state of interruptable waiting
- For such functions we care about complexity of the longest uninterruptible section rather than whole function execution complexity
- Interruption may be caused by system resource state change including timeout
- Interruption itself is guaranteed by compiler

# Complexity analysis

## Complexity before of uninterruptible section analysis

```
variant<message|error|null> read_and_parse (context, mut io.socket& socket, word& size_limit)
{
    mut request_parser parser = ()
    loop (!parser.finished ()) {
        select (context, socket) {              // Interruption point
            case (error@ err) {
                return err
            }
            case (enum io.EOF) {
                break
            }
            case (enum context.interrupt) {
                break
            }
        }
        loop {
            ubyte[] buffer = socket.read (4096) // Non-blocking read
            if (#buffer == 0)
                break
            parser.consume (buffer)
            select (context, socket) {          // Interruption point
                case (enum context.interrupt) {
                    break
                }
            }
            if (parser.finished ())
                break
        }
    }
    return parser.message ()
}
```

# Complexity analysis

## HardCode coroutines

- Coroutines are implemented as generators – special reentrant functions
- Allows using arbitrary schedulers:
  - prioritized for fastest request completion
  - prioritized for parallelism
  - custom – bound by developers imagination
- Complexity before interruption analysis allows finding perfect balance between responsiveness and performance
- yield call is possible only at reentrant function body

# Complexity analysis

## CPU instruction analysis

- Allows to score complexity more precisely
- Don’t require physical access to specific CPU model
- Outside MVP

# Complexity analysis

## Analysis with consideration for CPU cache

- Quiet mystical
- Working cache model is quiet possible (in theory)
- Reliability of cache modeling requires heavy research
- Outside MVP

# Complexity analysis

## Application for service development

- Scoring worst case scenario of full execution
- Scoring worst case scenario of longest uninterruptible section
- If we like the score
  then consider the component development complete
- Else if we like longest uninterruptible section score
  then consider the component development complete
- Else fix our program and repeat analysis

# Complexity analysis

## Application using IDE

- Comparing algorithm complexity
- Searching for bottlenecks
- Regression analysis
- Analysis based on data dumped from production service

# Sandboxing

## What Sandboxing is

```
#!/usr/bin/lua

require "canvas"

local user_func_source = get_untrusted_function()    -- Getting some function from untrusted source
--[[
draw_pixel (10, 20, 0xff0000)
draw_pixel (30, 40, 0x00ff00)
draw_pixel (200, 500, 0x0000ff)
]]

local env_proxy = {
   draw_pixel = canvas.draw_pixel,                   -- The only function available for user environment
}

local env = setmetatable ({}, {__index = env_proxy}) -- User environment is read-only from inside

local user_func = assert(load (user_func_source, "user func", "t", env))

user_func ()                                         -- Feeling safe calling untrusted function
```

# Sandboxing

## What Sandboxing currently is

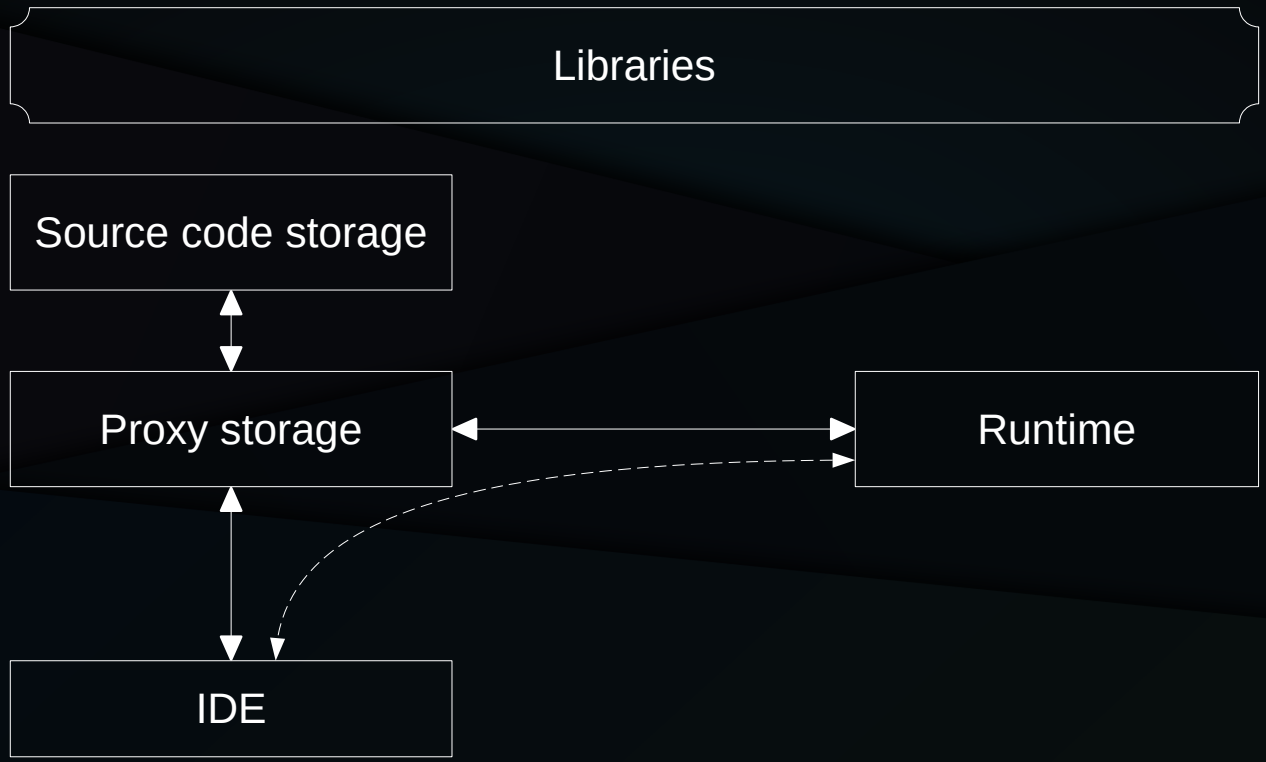
```
function small_little_fast_user_callback()
    while true do
       -- No way to interrupt this without killing an application!!
    end
end
```

# Sandboxing

## Sandboxing in HardCode

- Allows denying functions:
  - without completeness guarantees
  - exceeding computation/memory access complexity quote
  - exceeding longest uninterruptible section complexity quote
- Guaranteed specified responsiveness
- Sandboxed code has same performance as code in main environment
- All the above works with embedding

# Language architecture



# Language architecture

## Libraries

- Syntactic analyzer
- Semantic analyzer
- Linker
- Backends:
  - in-memory instructions
  - C code generation
  - VM instructions (for debug)
- Operation analyzer
- Runtime
  - Core
  - Garbage Collector
  - Module loader for bindings
- Serializer/parser (for source code storage)

# Comparing to other languages

## Lower than others, Higher than others

- High-level semantics
  - No unhandled runtime errors
  - No synchronization errors
  - Convenient memory model: RAII and GC together
  - Dynamic polymorphism “out-of-the-box”
  - Dynamic linkage, static typing
  - Actual Embedding, actual Sandboxing
  - Strict analysability
  - Consistent source code storage
- Low-level optimizations
  - Well optimized memory model
  - Arbitrary function inlining across whole application
  - Predictable stack usage
  - Adaptive function calling convention
  - Strict semantic guarantees for semantic tree and lower level optimizations
  - Strict optimizations on top of strict aliasing rules

# Language application

- Replacing every general-purpose language
- DSL (Domain Specific Language) construction
- General purpose browsing platform
- Service as a Service platform
- Embedding in Game Development

# Service as a Service platform

## Service development

- Today development looks like docker-compose:
  configuring whole service graph may take few weeks
- Service as a Service platforms allows to boot up entire service graph in 2 clicks
- The difference between monolith and microservices is opaque for a developer
- All the source code and resources are stored at the server-side,
  the only client-side application needed for development is IDE

- Effective development, flexible optimization
- Fast onboarding, flexibility and mobility for developer

# Service as a Service platform

## What a name stands for

- Service as a Service – next stage of Function as a Service paradigm
- Function as a Service – cloud service for executing uploaded functions
- Service as a Service – cloud service for service construction and serving

# Service as a Service platform

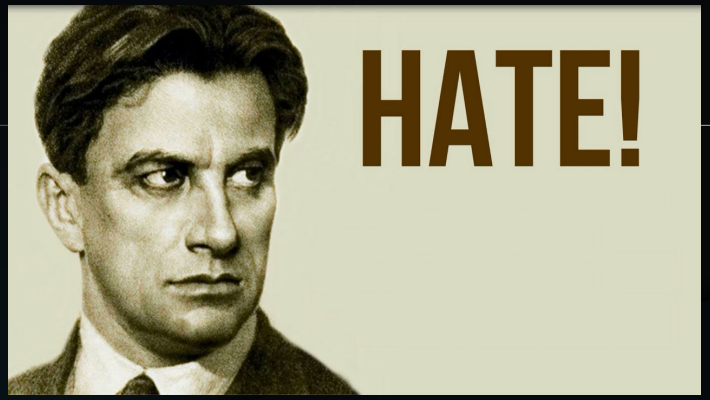
## Function as a Service today

- We may call some uploaded functions remotely
- Separate process for every function call
- Function termination is guaranteed by timeout (then serving process is killed)
- There are stateful functions
- The whole concept is big multiplier of overheads

## Service as a Service with HardCode

- Using smart API not limited to function calls
- API: fusion of evolved gRPC and SCTP on top of UDP
- Service API is stored in versioned DB, properly maintaining consistent API extendability
- Data serialization format: fusion of evolved Protocol buffers + Flat buffers
- Extended network protocol
- Cluster-oriented design out-of-the-box
- Library vs Microservice choice at service configuration time rather than at development time
- Event model is described at service schema: service request completion is analyzable
- Service client application completion is analyzable

# Service as a Service platform




## I hate TCP!!

- Warming up TCP connection pool
- Extra round-trips for connection establishment
- No packet duplication control
- Packet send repetition is configurable at kernel-level
- TCP connection timeouts are configurable at kernel level
- TCP connection timeout configuration isn't sent to remote peer
- TCP connections can't survive proxy restart
- TCP connections can't be transferred across network routes
- Message order is supported where it's not needed (increases latency dramatically)
- Implementations on top of TCP use to link session to TCP connection (like in gRPC)
- No idempotent request support out-of-the-box

# World of inappropriate design: TCP, SCTP and IPSec

- TCP is helpful for Hello World, outside Hello World TCP is a burden
- TCP, SCTP and IPSec are implemented at OSI level,
  where no regular update is possible
- IPv6 and SCTP experience showed inability of fast regular firmware update
  on both house routers and backbone network equipment
- Datagram and stream transmission control scheme
  require tight integration encryption messaging scheme.
  E. g., TLS and QUIC uses stapling while it isn’t introduced in dTLS.
- Datagram and stream transmission control and encryption implementation
  requires frequent update
- Conclusive proper design: datagram and stream transmission control and encryption
  must be implemented on top of UDP

# Proper network protocol design

- The whole user-level protocol is implemented on top of UDP
- The whole interaction is made though single port and with built-in keepalive feature (lesson learnt from RTP issues)
- Session management is part of protocol:
  - session may survive network failures on any level as well as server reboots (if required)
  - cloud input balancers are used for routing by session ID rather than proxying
  - service API schema (like in gRPC) allows for session-bound and session-unbound interactions
- Service API schema allows for streams and single datagrams attached to a session and streams not attached to a session
- The protocol implements unordered messaging on top of multiple statefully encrypted streams (managed as circular buffer) thus combining UDP latency with TLS security (unlike dTLS)
- Message duplication and sending repetition control is also part of the protocol

# Service as a Service platform

## Platform features

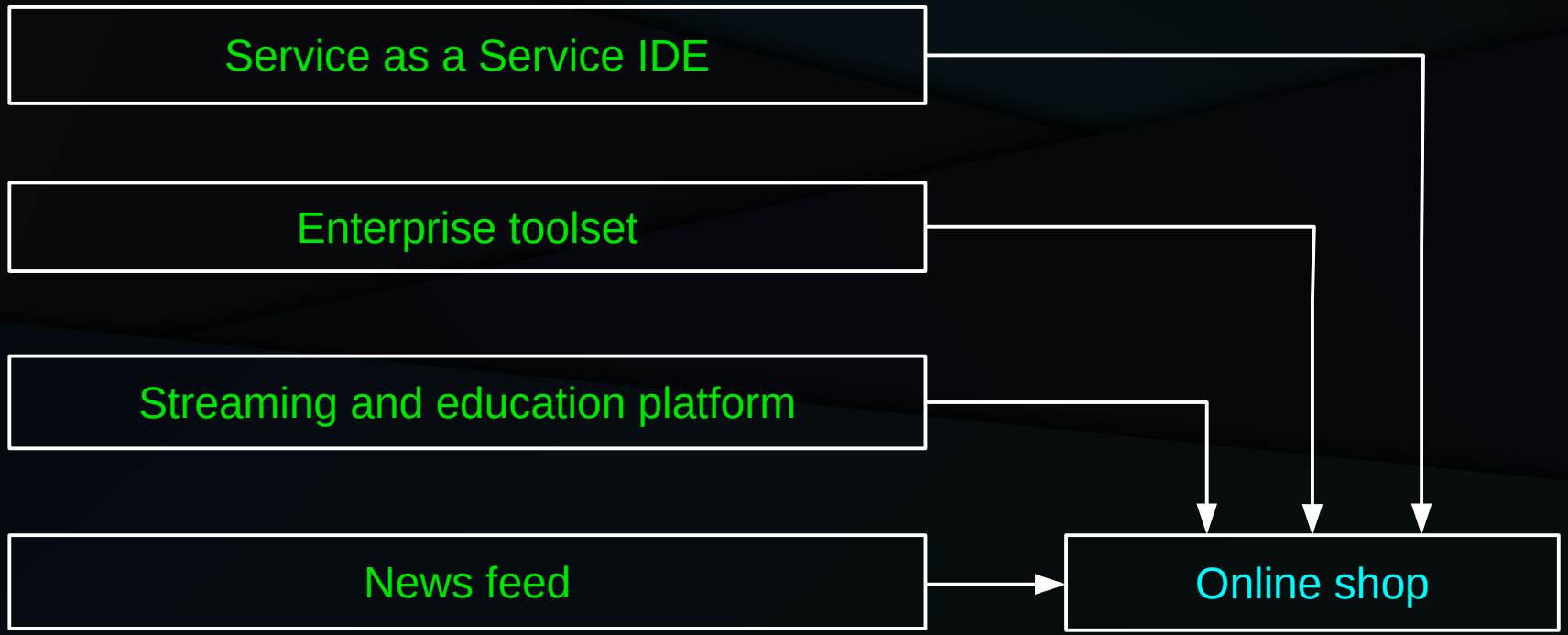
- The service is simply a tool for network schema design and callback implementation (similar to FaaS + Swagger, but with no band-aid design)
- Most code wrapping and service maintenance features are already implemented, are configurable and are parts of the the platform
  *E. g. metrics and logs are collected «out-of-the-box» without losses*
- Deployment and request event history is collected and maintained by the platform
- Resource management is based on analysis and versatile profiling
- Analysis and profiling data is collected and stored by the platform
- Fine-grained resource isolation is implemented at programming language level
  - No containers needed (like in Docker)
  - No need to have separate process per service
- Still system-level resource management is available for isolating services at production cluster as well building isolated development environment for unsafe activity such as developing binding modules and coding with infinite loops enabled

## More platform features

- Lightweight deployment:
  - deploying only changes
  - no data and function duplication
- Complex deployment schemes are available «out-of-the-box» for any service
- Centralized service configuration
- Initial deployment at cloud service requires no configuration,
  *same 2 clicks as forking on GitHub*
- Extra configuration is minimalistic
  due to absence of intermediate unnecessary entities and concepts
- Untrusted code is properly runnable «out-of-the-box»

# Next gen IDE

IT content delivery platform



# Next gen IDE

## Service as a Service IDE

- Service development IDE
- VCS integration
- Strict semantic analysis, testability, refactoring and debug
- UI for deployment and CI/CD configuration
- IDE is the only tool developer needs

- High reliability and efficiency
  of development, debug, deployment and incident resolution

# Next gen IDE

## Enterprise toolset

- Tightly integrated issue tracker (akin: Jira)
- HR portal (enterprise solutions)/social network (akin: Linkedin)
- Messenger (akin: Slack)
- Notification system (akin: e-mail)

- «Out-of-the-box» enterprise toolset
- Software-driven workflow instead of manager-driven
- Unified time management
- Effective HR/career management

# Next gen IDE

## Streaming and education platform

- Video streaming coding and debugging
- Team-work online software development
- Built-in tutorial framework
- Gamified education
- Built-in Q&A service
  with guaranteed compilable and runnable snippets (akin: Stack Overflow)

- Effective staff education and expertise sharing
- Convenient and controllable staff hiring

# Next gen IDE

## News feed

- Project suggestions
- IDE extension suggestions
- Educational material suggestions
- Vacancy suggestions
- HardCode and infrastructure publications
- IT-industry news

- Recent and relevant information for developer
- Wide terms for advertising

# Next gen IDE

## Online shop: core and partner solutions

- Service as a Service cloud platform services
- IDE extensions: debuggers, analyzers, editors, linters, etc.
- Enterprise toolset: multiple stuff
- Educational content, reviews, hackathones, steam subscriptions (akin: YouTube, Twitch)
- Libraries and services (akin: GitHub)
- Client applications (akin: Steam)
- Game engines (akin: Unreal)
- Games
- Browser platform