From C extension to pure C

Migrating RBS

松山

Alexander Momchilov – April 17, 2025

RubyKaigi 2025

Alexander Momchilov

Staff Eng @ Ruby DX team

🔗 momchilov.ca

🇨🇦 Toronto

3 talks about Type Checking

Embracing Ruby magic:
Statically analyzing DSLs

Vinicius Stock

Yesterday. Time travel,
or wait for the video :)

From C extension to pure C:
Migrating RBS

Alexand**er** Momchilov

Right here,
right now

Inline RBS comments for
seamless type checking
with Sorbet

Alexand**re** Terrasa

Tomorrow at 14:00,
In the Sub Hall

Is your name also Alex?

Join our team!

What is RBS?

- Short for “Ruby signature”
- Standard notation for type annotations
- Originally written in separate `.rbs` files
- Inline RBS lets you write RBS in comments in your “real” `.rb` files
- Programs with RBS signatures can be type checked by *Steep*

Example RBS file

`message.rbs`

```
class Message
  attr_reader from: User
  attr_reader string: String
  attr_reader reply_to: Message?

  def initialize: (from: User, string: String) -> void

  def reply: (from: User, string: String) -> Message
end
```

Example RBS inline

message.rb

```ruby
class Message
  attr_reader from #: User
  attr_reader string #: String
  attr_reader reply_to #: Message?

  #: (from: User, string: String) -> void
  def initialize(from:, string:)
    @from = from
    # ...
  end

  #: (from: User, string: String) -> Message
  def reply
    Message.new(...)
  end
end
```

Why are we interested in RBS?

- Shopify uses Sorbet for checking our Ruby code
  - Static checker written in C++ … *Blazingly fast™*
  - Scales to our huge monolith
- Our team wanted to replace Sorbet's "sig" syntax
  - It's a Ruby DSL, which requires `sorbet-runtime` gem
  - Sigs are verbose and clunky

RBS comment vs Sorbet sig

```rb
class Point
  attr_reader :x #: Integer
  attr_reader :y #: Integer

  #: (x: Integer, y: Integer) -> void
  def initialize(x:, y:)
    @x = x
    @y = y
  end
end
```

```rb
class Point
  extend T::Sig

  sig { returns(Integer) }
  attr_reader :x
  sig { returns(Integer) }
  attr_reader :y

  sig { params(x: Integer, y: Integer
  def initialize(x:, y:)
    @x = x
    @y = y
  end
end
```


RBS

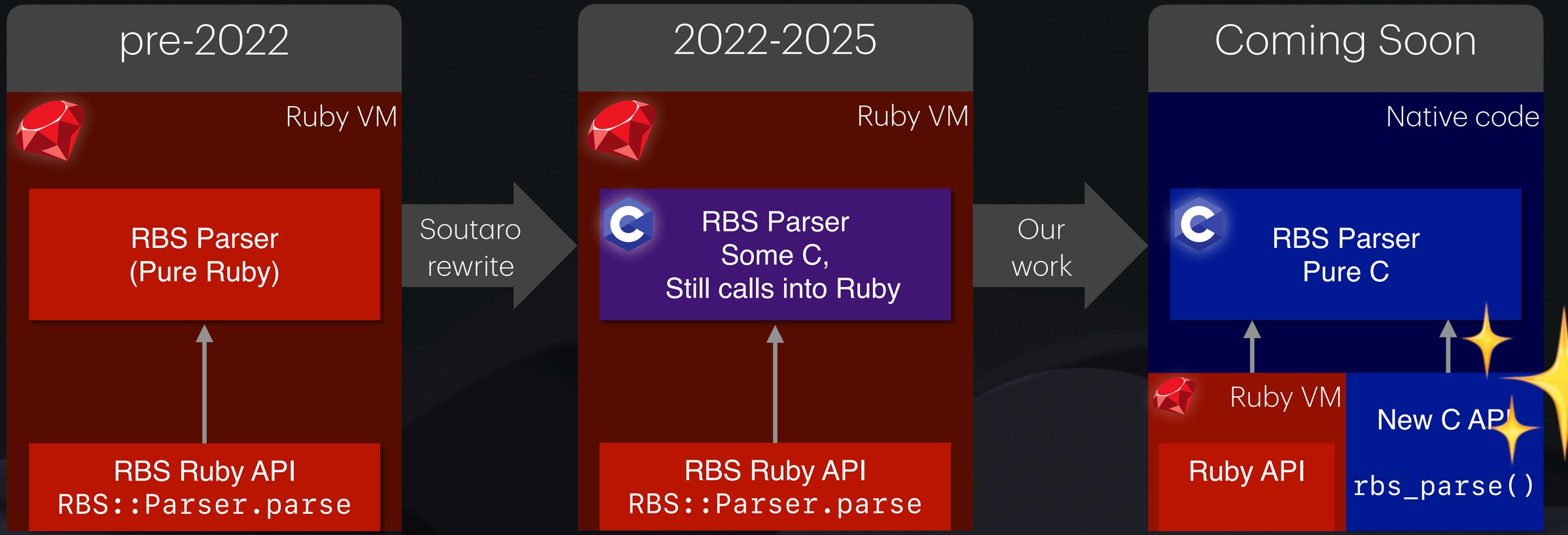
RBS + Sorbet demo

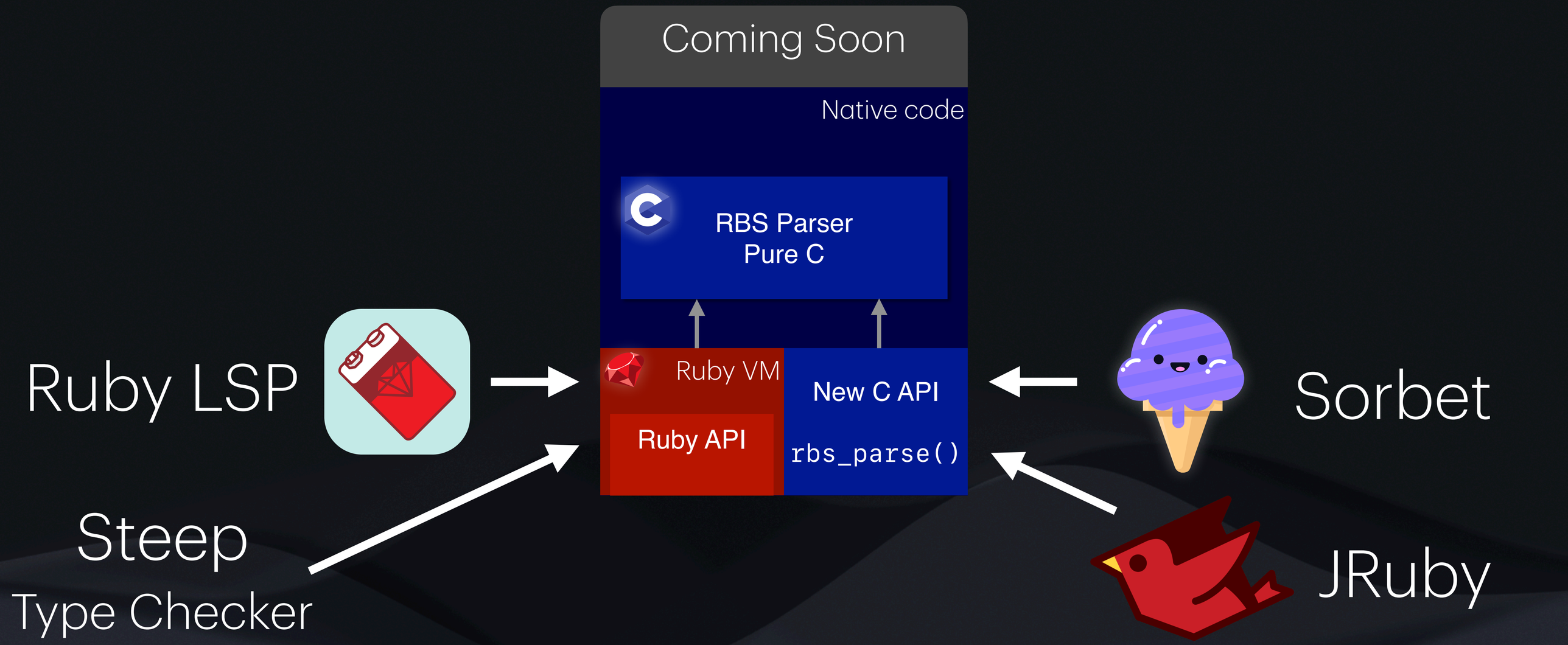
```ruby
# typed: true

require "sorbet-runtime"

class Example
  extend T::Sig

  sig { params(i: Integer).returns(String) }
  def should_return_a_string(i)
    i.to_s # This works :)
  end
end
```

https://github.com/ruby/rbs/pull/2398

So what’s so hard about that?

The Evolution of RBS




Coming Soon
Native code
RBS Parser
Pure C
Ruby VM
Ruby API
New C API
rbs_parse()
Ruby LSP
Steep
Type Checker
Sorbet
JRuby

Why is the Ruby dependency a problem?

- Using Ruby features requires the GVL
  - Limits CPU parallelism to just 1 thread
  - Example: Sorbet parses files in parallel on multiple threads
- MRI's C extension API is not portable
- Can hand-tune memory layout, so we can lower usage

Why did RBS use Ruby in the first place?

What do we have to replace?

1. Error handling via Exceptions
2. Memory management: allocation and garbage collection
3. Ruby data structures like: Objects, Array, Hash, and more
4. Polymorphism and dynamic method dispatch

Error handling

Ruby convenience 1

Error handling

- Idiomatic Ruby code handles errors via exceptions
- Raising an exception acts like a “deep return”
- Moves control flow to the most recent `rescue` block
- Ruby C API exposes `rb_raise()`, `rb_rescue()`, `rb_ensure()`, etc.

```
// Parse optional tuple like [String, Integer, bool]?
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

// Parse tuple like [String, Integer, bool]
rbs_node_t *parse_tuple() {
    if (next() != '[') {
        rb_raise(rb_eSyntaxError,
            "Expected tuple to start with '['");
    }

    // ...
}
```

```
// Parse optional tuple like [String, Integer, bool]?
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

// Parse tuple like [String, Integer, bool]
rbs_node_t *parse_tuple() {
    if (next() != '[') {
        rb_raise(rb_eSyntaxError,
            "Expected tuple to start with '['");
    }

    // ...
}
```

So how do we throw exceptions without Ruby?

You don’t. 😄

```c
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

rbs_node_t *parse_tuple() {
    if (next() != '[') {
        rb_raise(rb_eSyntaxError,
            "Expected tuple to start with '['");
    }

    // ...
}
```

```
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

rbs_node_t *parse_tuple() {
    if (next() != '[') {
        return false;
    }

    // ...
}
```

false = “operation failed 👎”

```
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

rbs_node_t *parse_tuple() {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

true = “operation succeeded 👍”

We need to change the signatures!

```c
rbs_node_t *parse_optional_tuple() {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

rbs_node_t *parse_tuple() {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple = parse_tuple();

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    parse_tuple(&tuple);

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    parse_tuple(&tuple);

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

Did this succeed?

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    if (!parse_tuple(&tuple)) {

    }

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    if (!parse_tuple(&t
        return false;
    }

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

Bubble up

Error propagation is manual

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    if (!parse_tuple(&tuple)) {
        return false;
    }

    return (next() == '?') ? rbs_optional_new(tuple) : tuple;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    // ...

    return true;
}
```

```
bool parse_optional_tuple(rbs_node_t **result) {
    rbs_node_t *tuple;
    if (!parse_tuple(&tuple)) {
        return false;
    }

    result = (next() == '?') ? rbs_optional_new(tuple) : tuple;
    return true;
}

bool parse_tuple(rbs_node_t **result) {
    if (next() != '[') {
        return false;
    }

    result = // ...

    return true;
}
```

Memory management

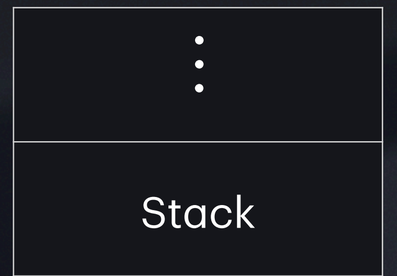
Ruby convenience 2

```
Foo *parse_foo() {
    A *a = parse_a();
    B *b = parse_b();
    C *c = parse_c();

    return new_foo(a, b, c);
}
```

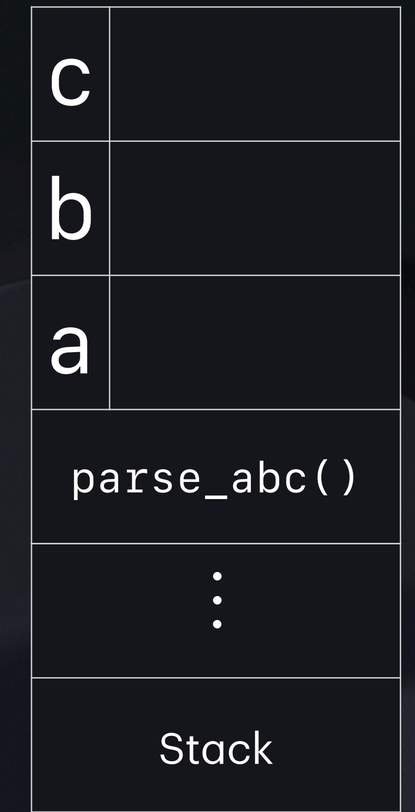
```
Foo *parse_foo() {
    A *a = parse_a();

    B *b = parse_b();

    C *c = parse_c();

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

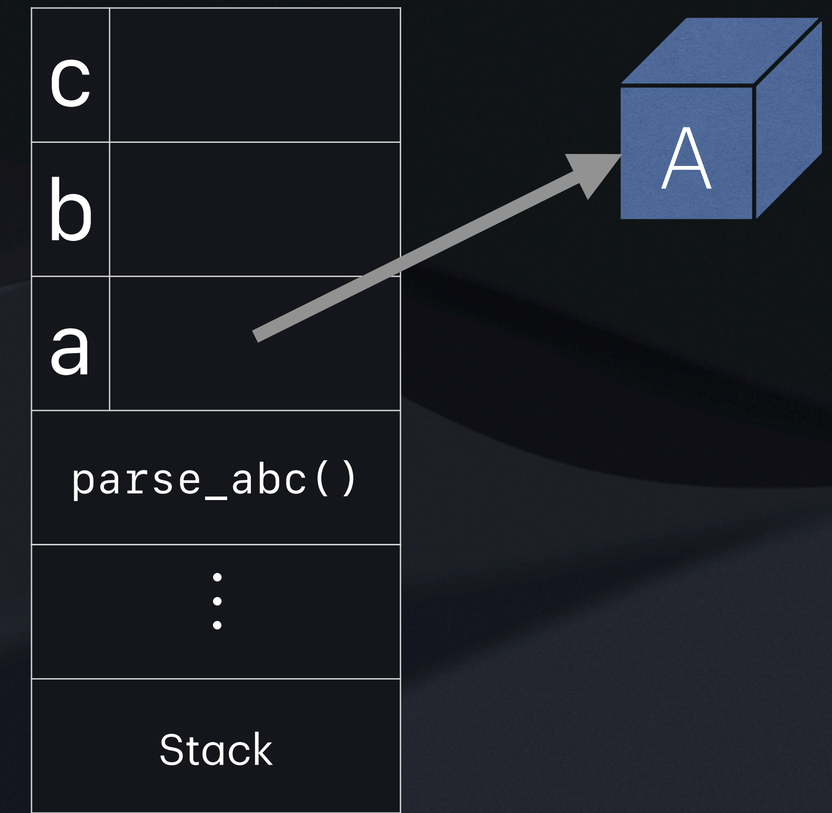
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



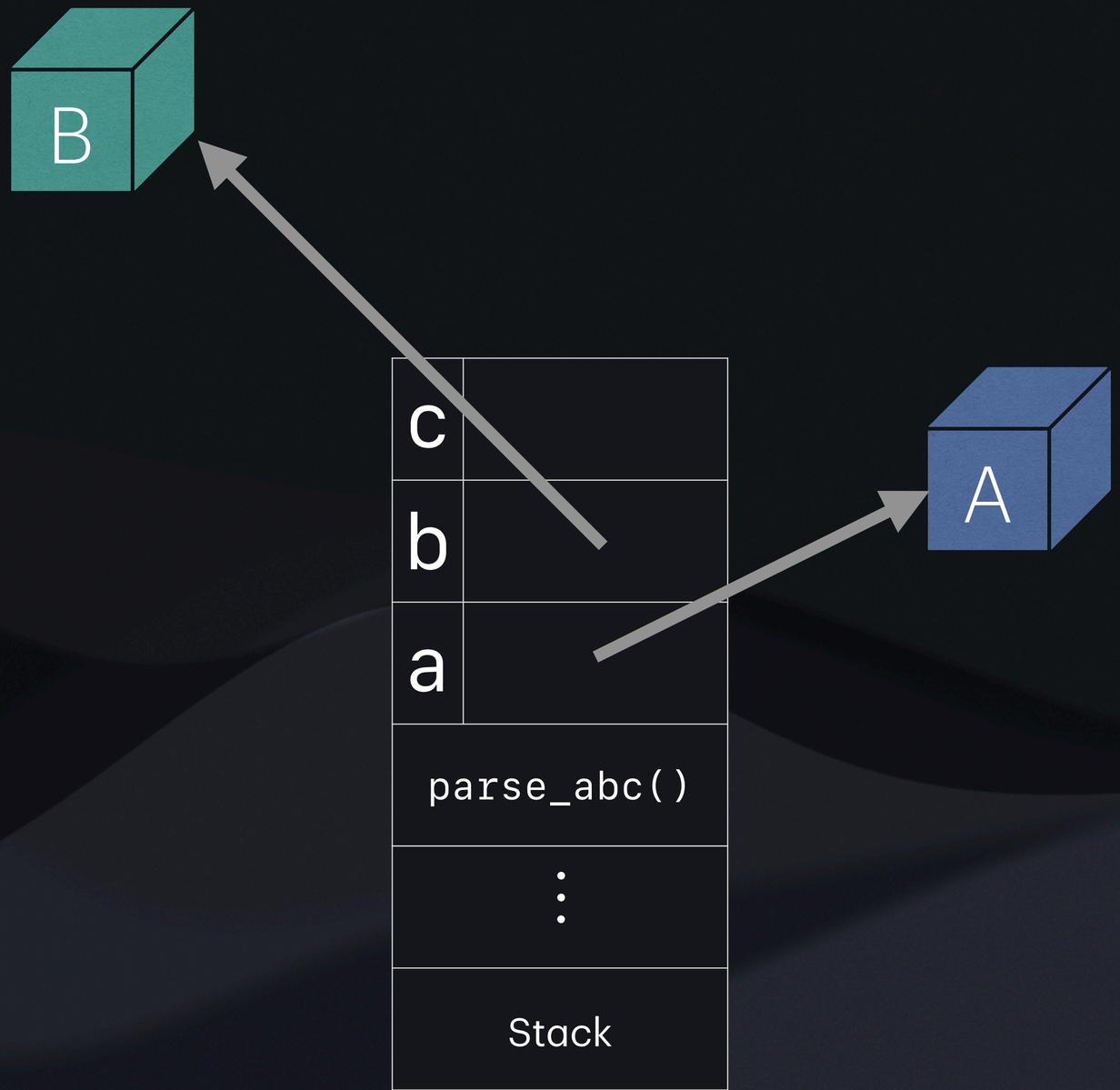
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



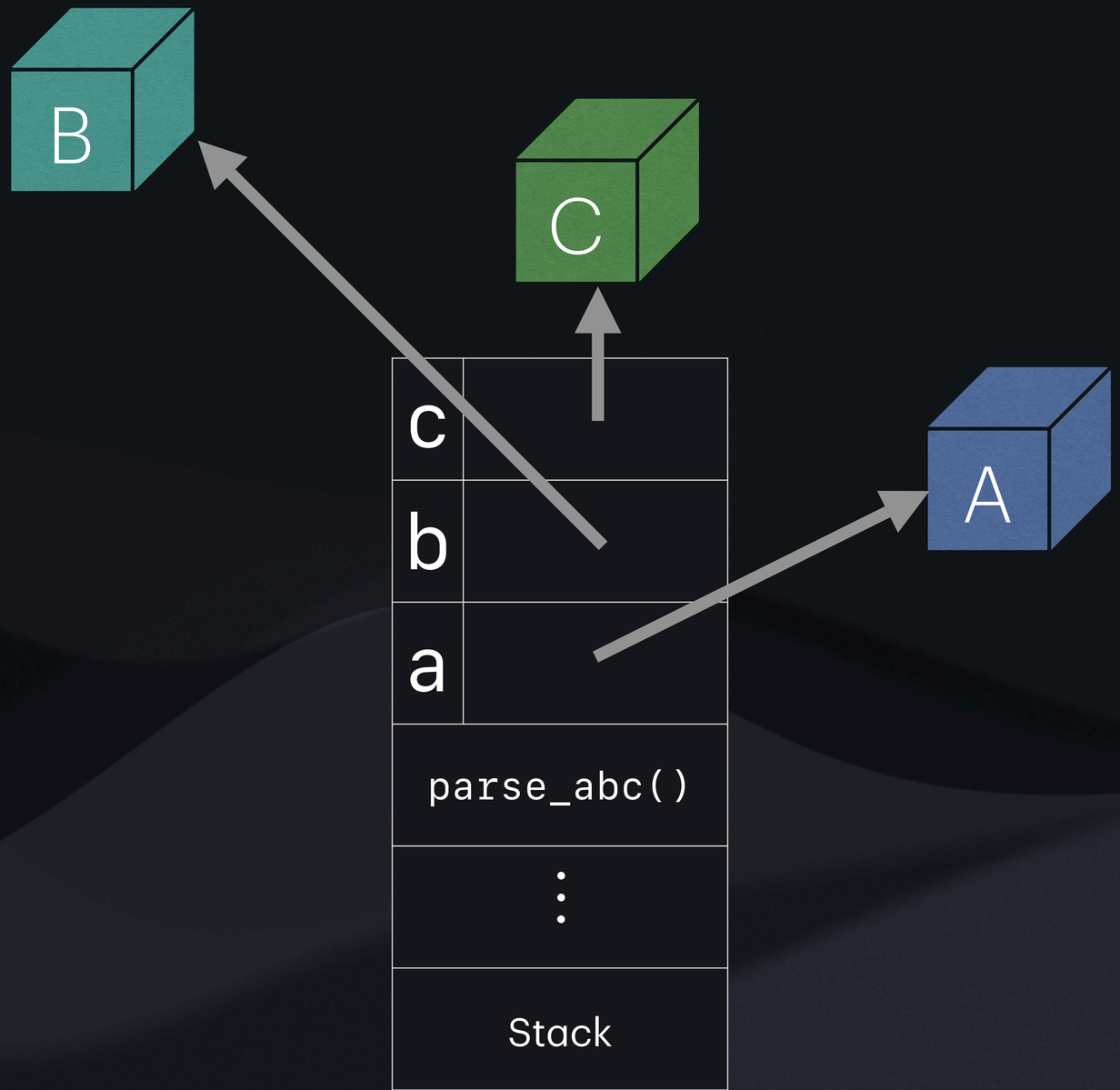
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



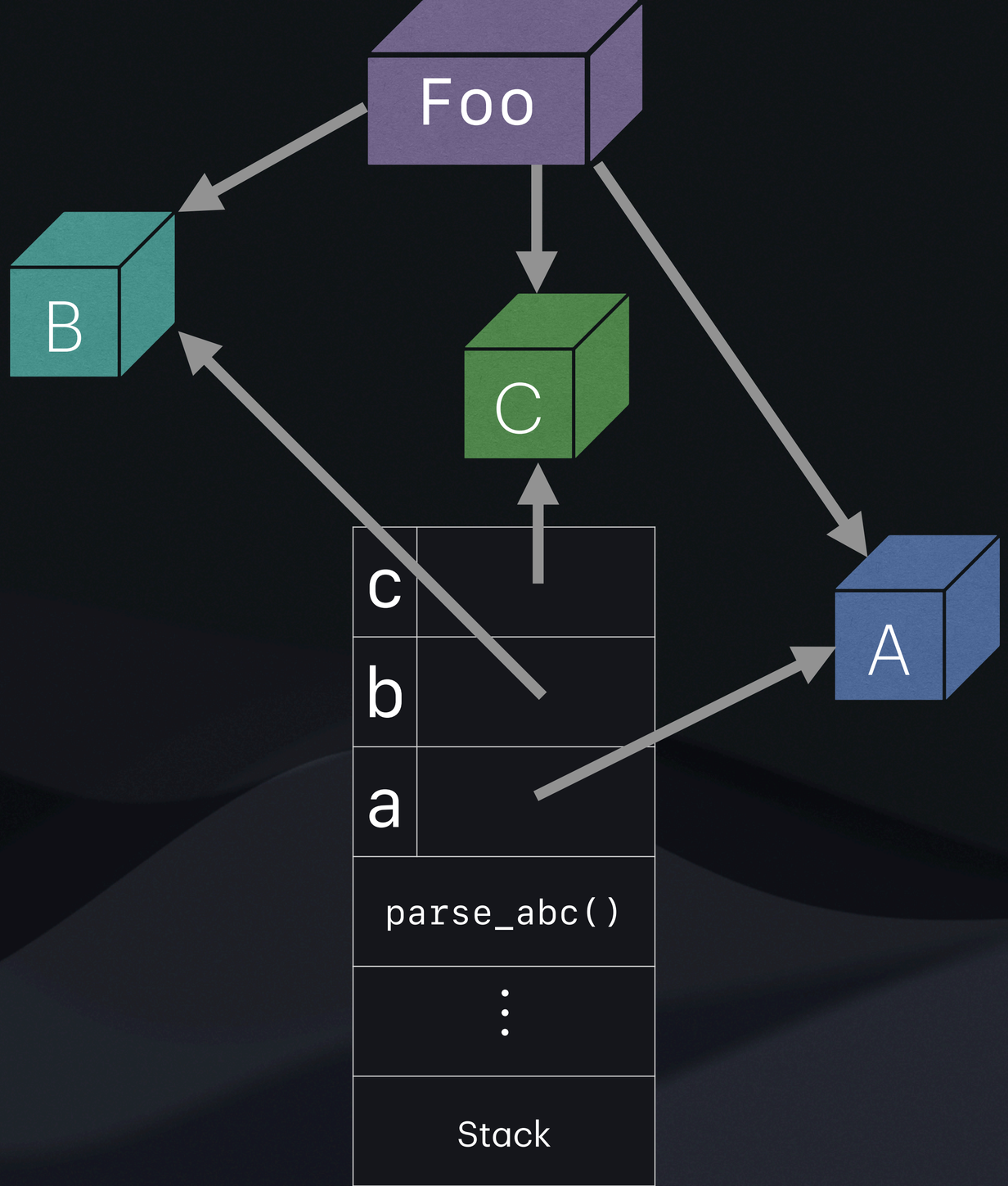
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



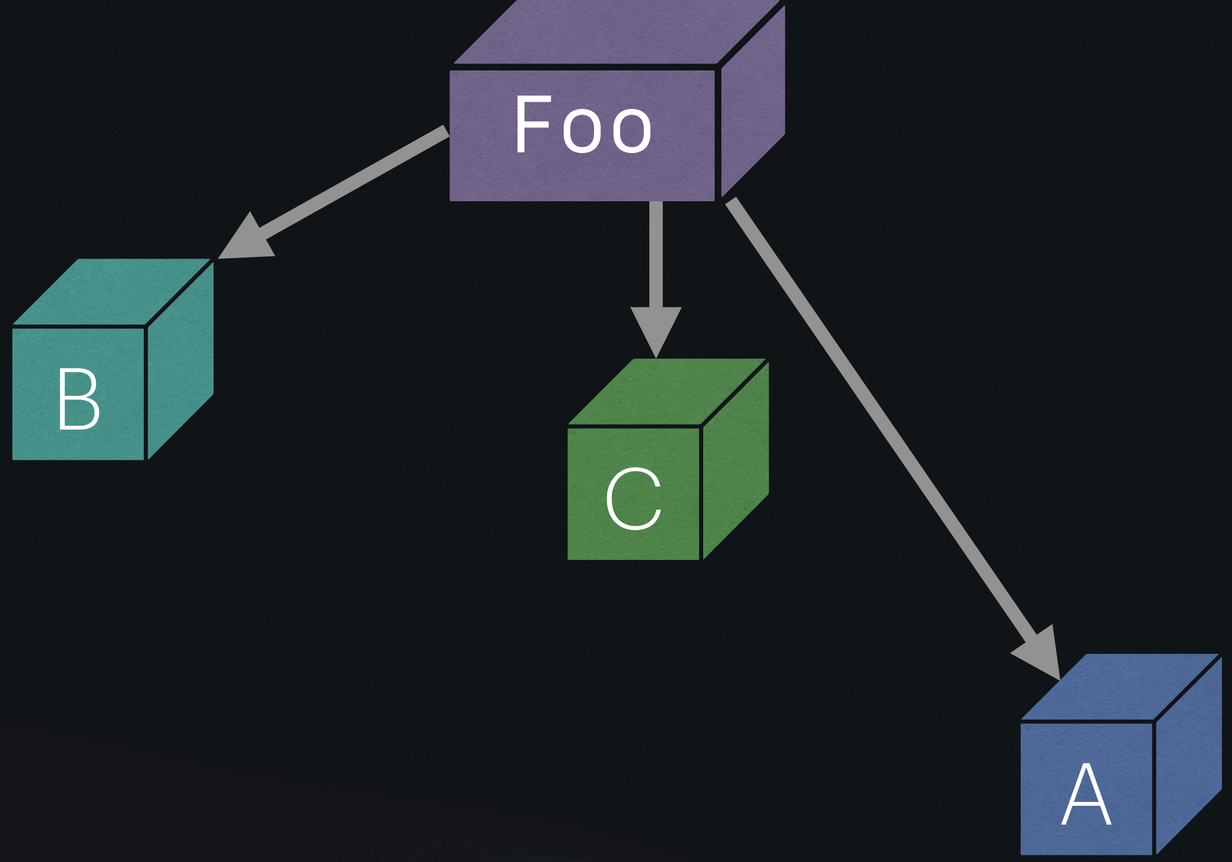
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

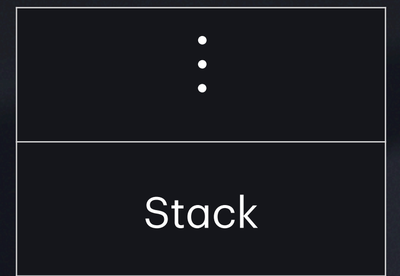


```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

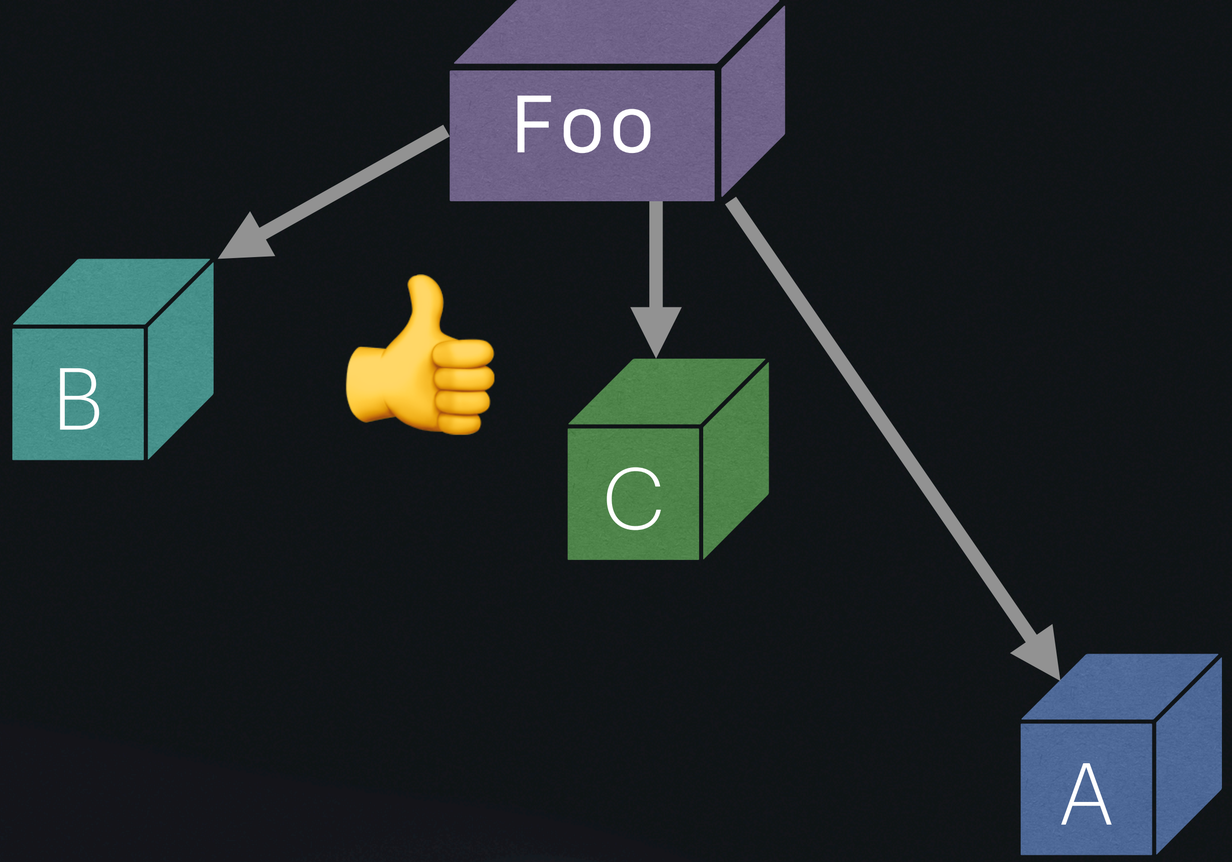


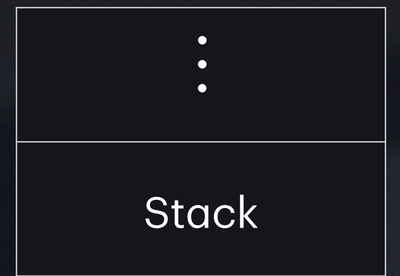
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



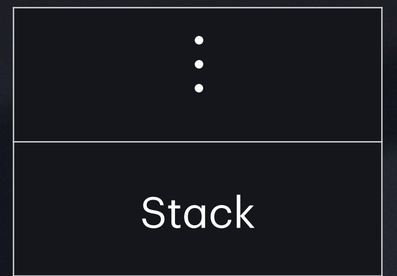


```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

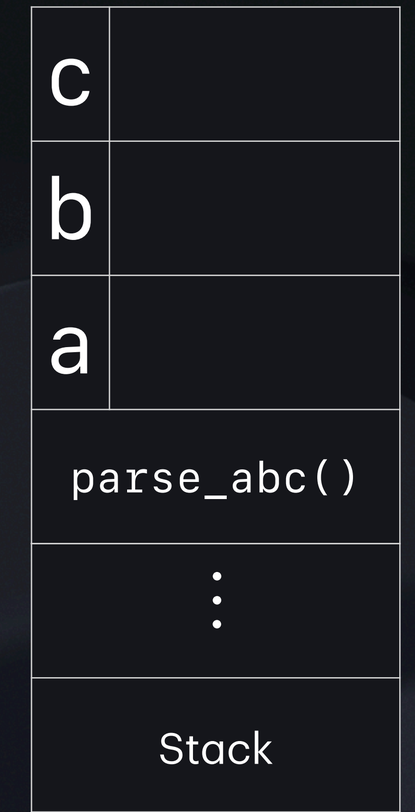




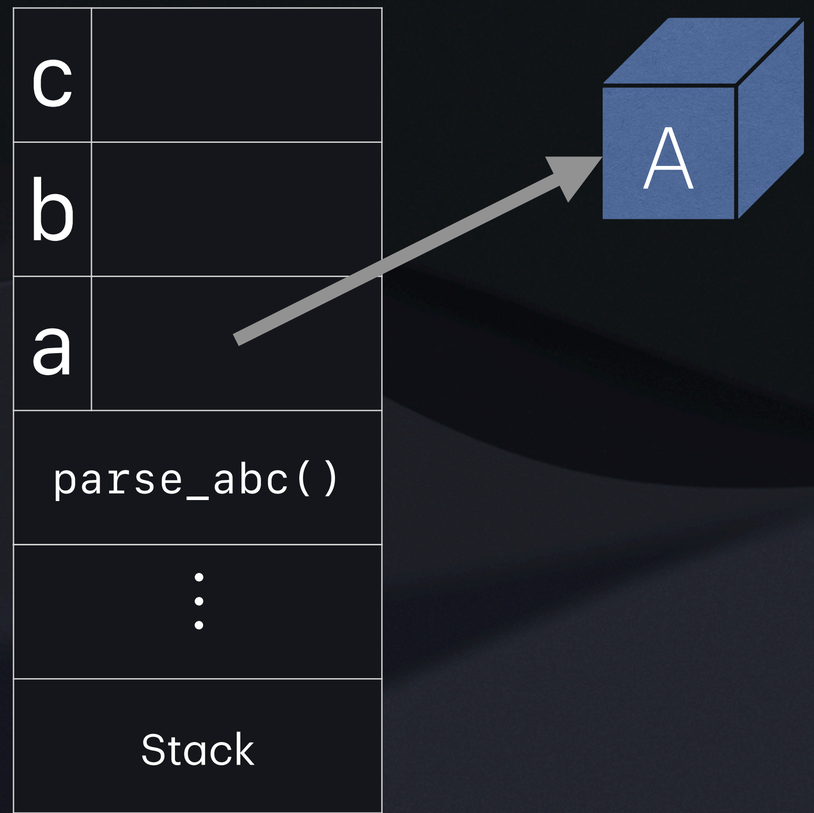
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



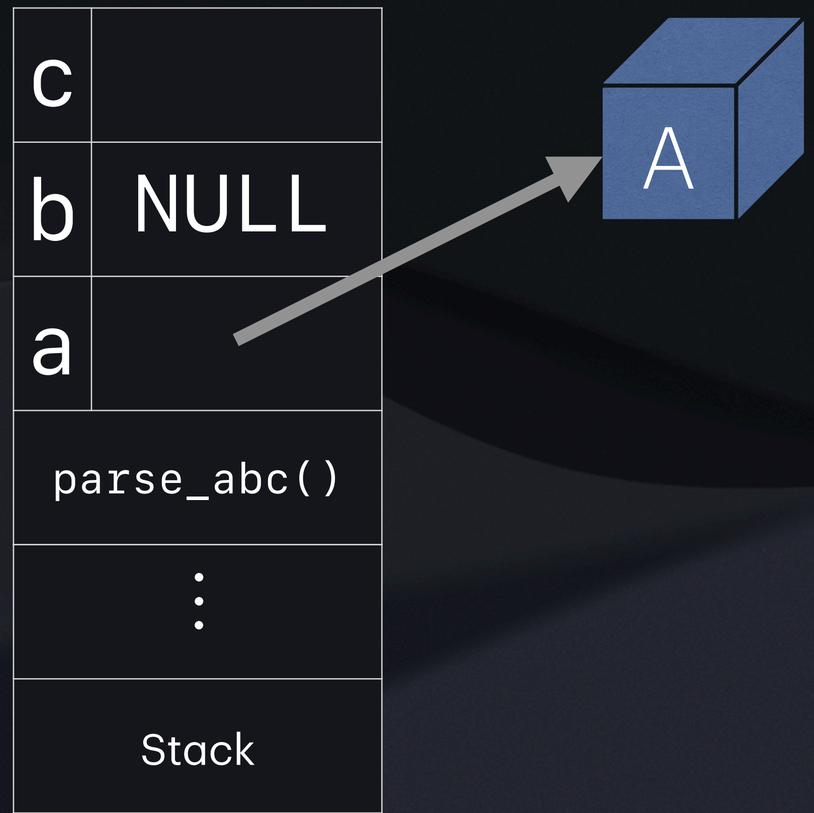
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



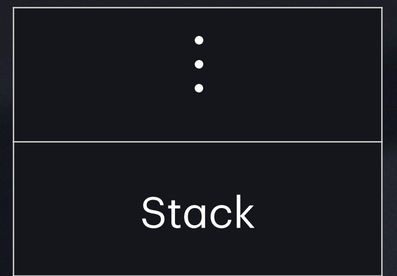
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```



```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

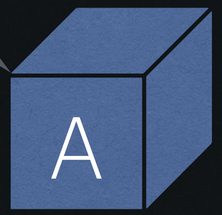


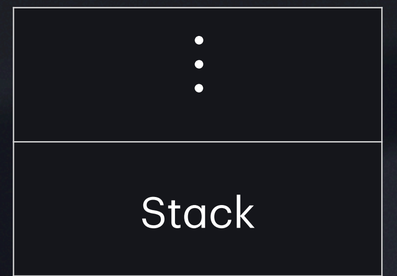
```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

❌

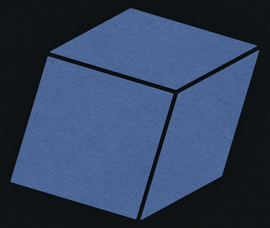
⚠️

Object A leaked!

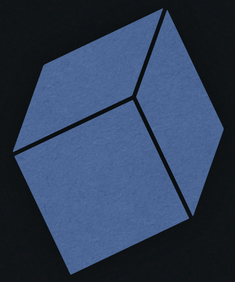


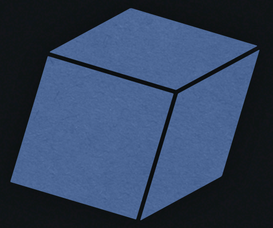


#:

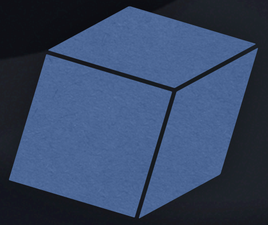


#: (

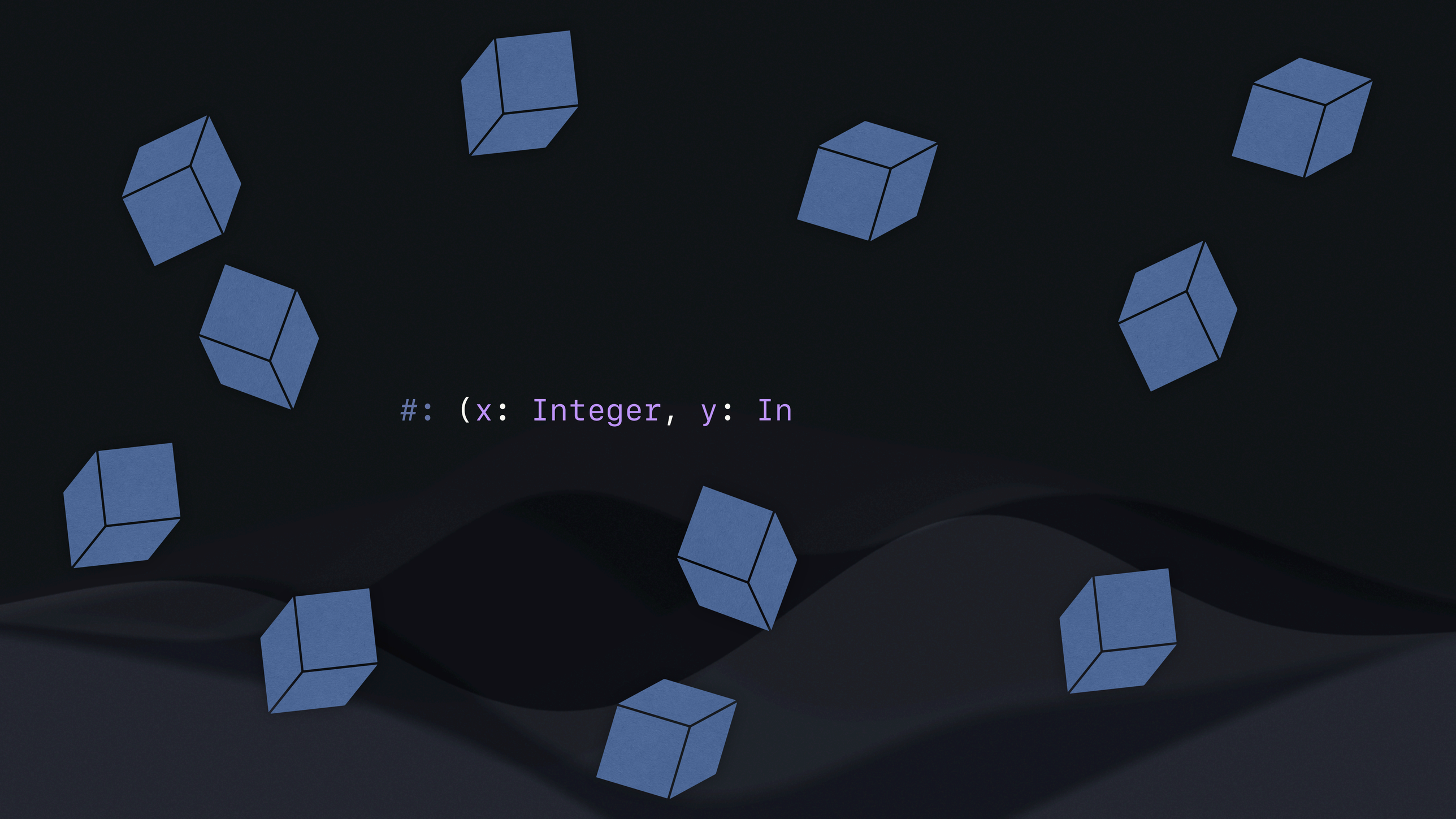


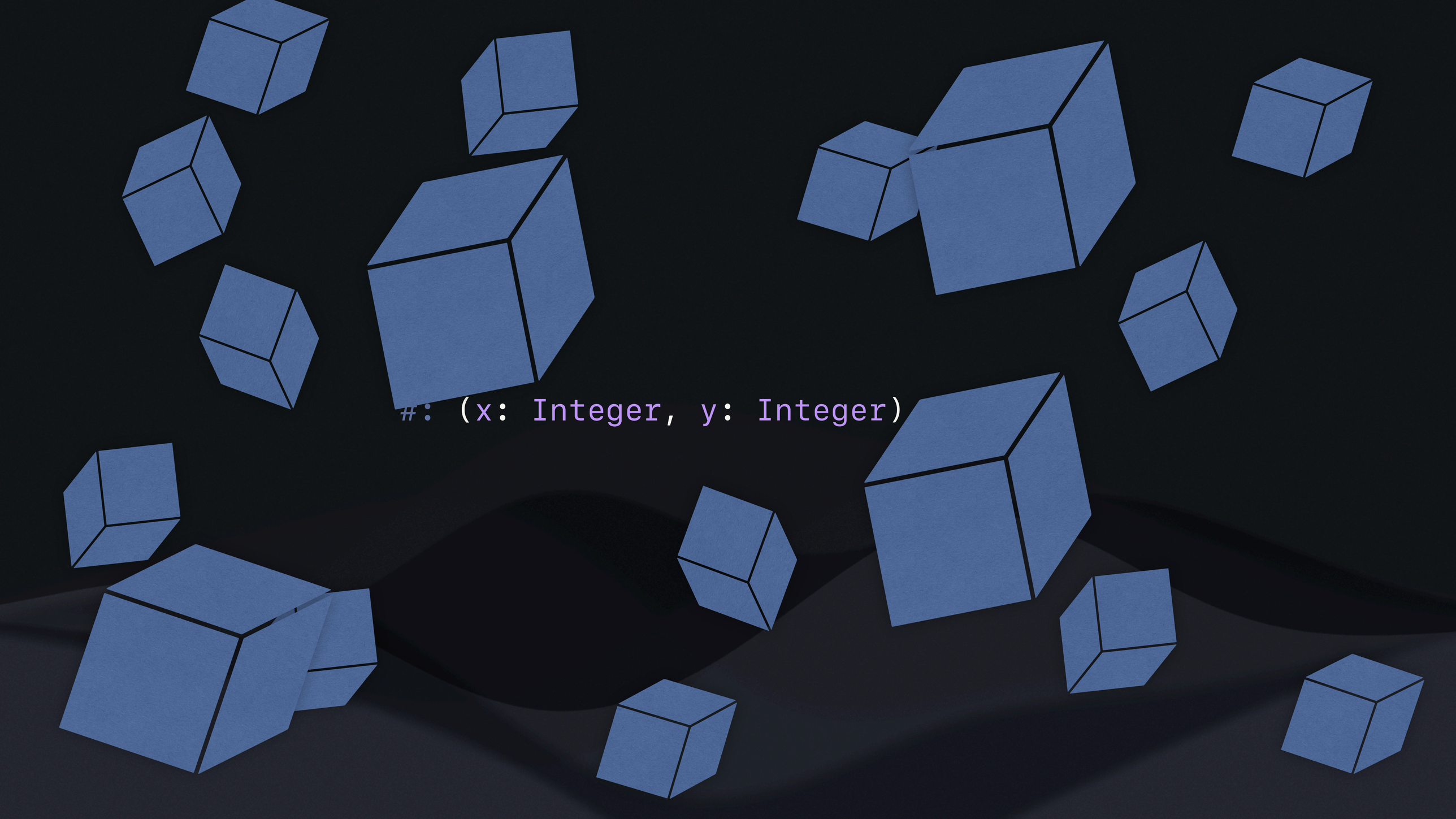


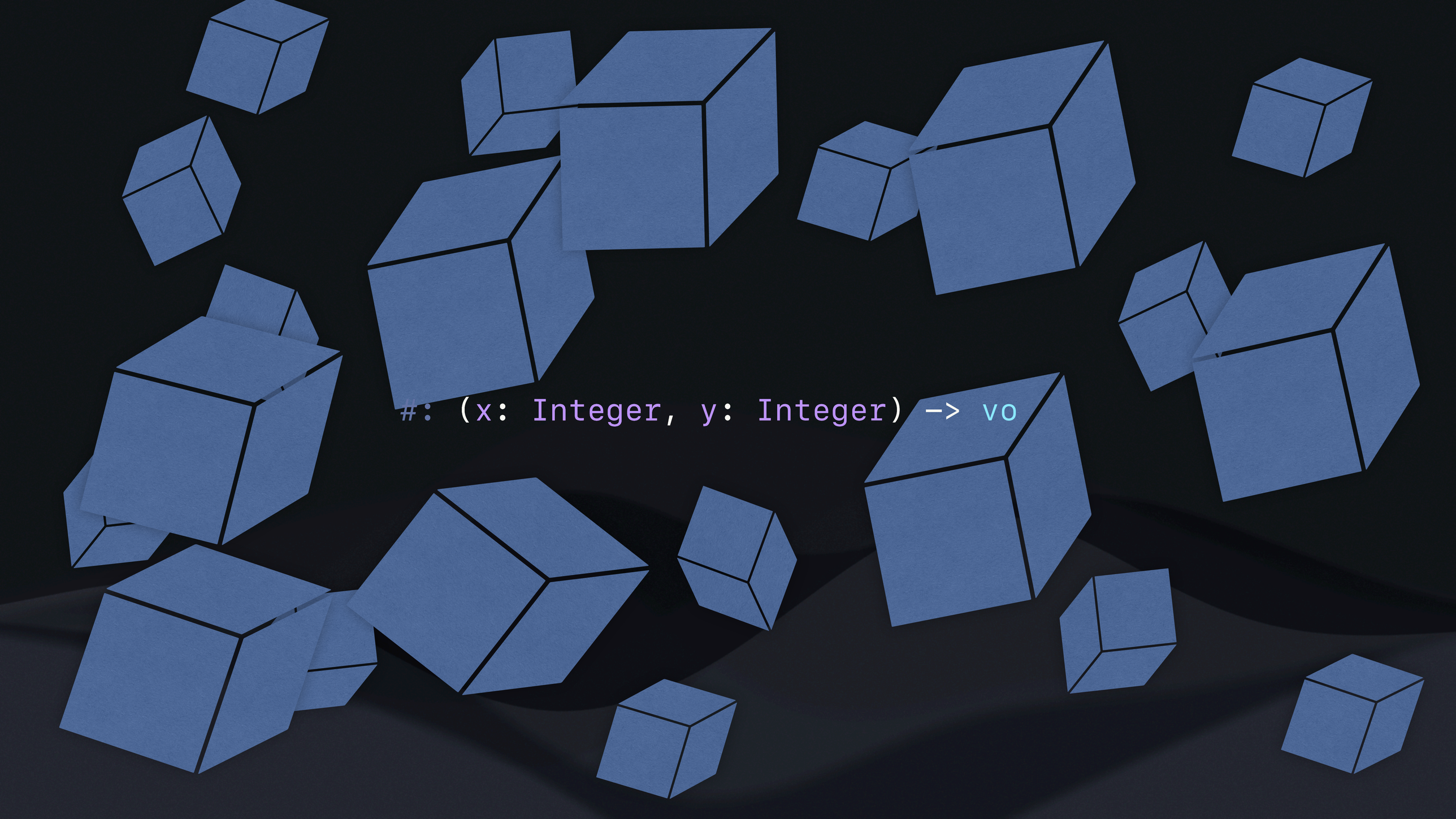
```
#: (x: I
```

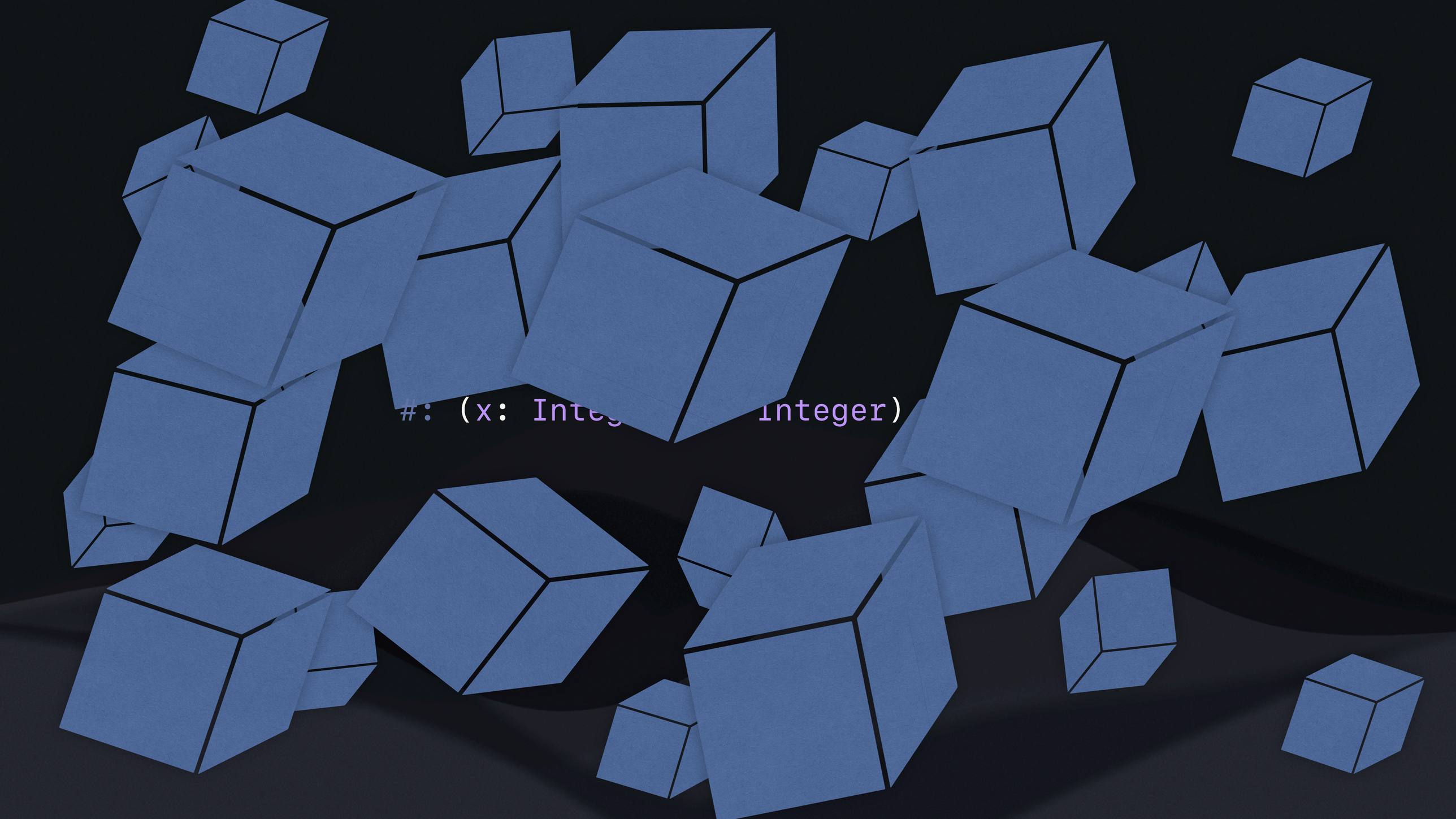


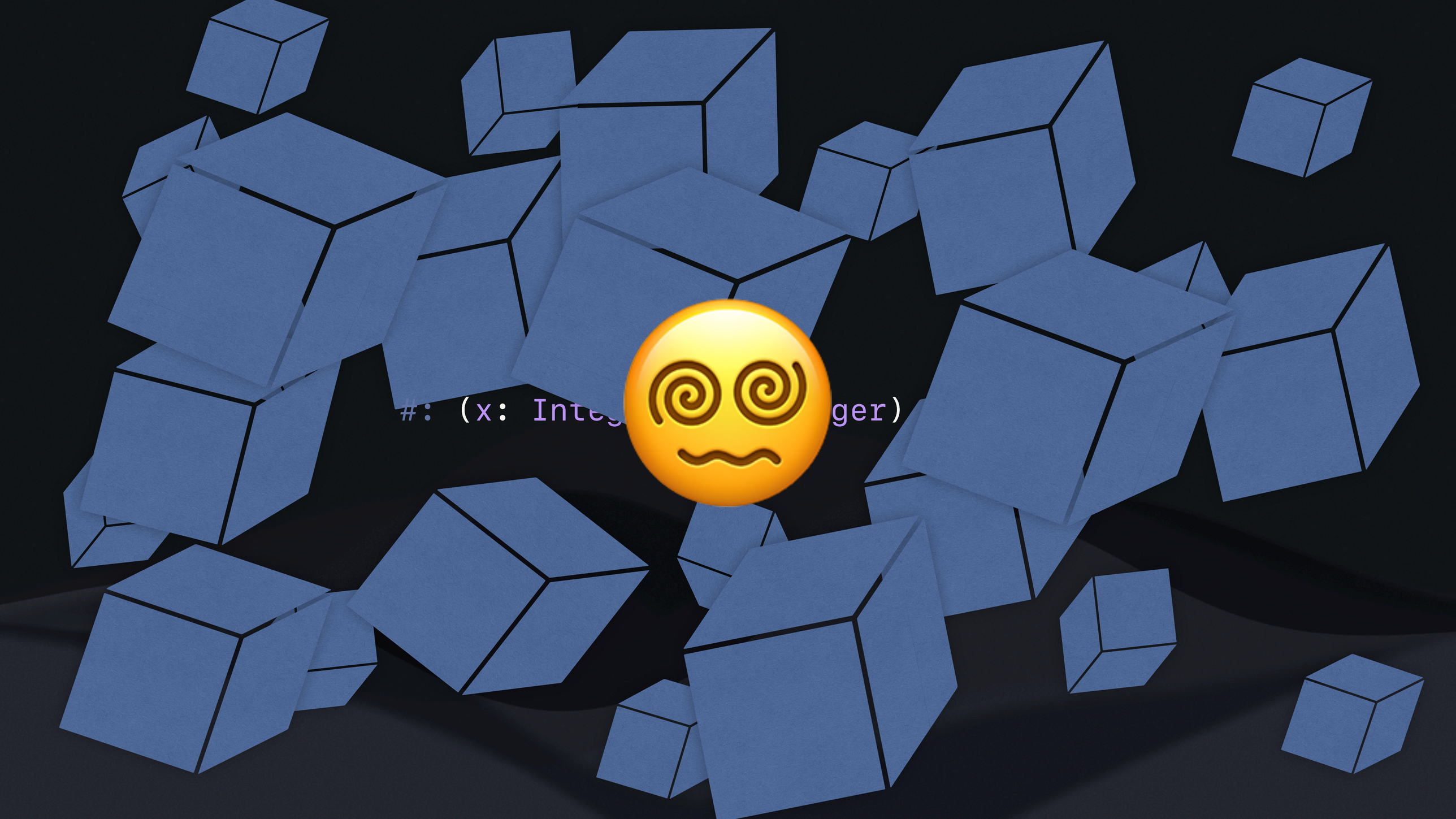

#: (x: Integer,


#: (x: Integer, y: In


#: (x: Integer, y: Integer)


#: (x: Integer, y: Integer) -> vo


#: (x: Inte nteger)


#: (x: Inte ger)

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {

        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {


        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) {
        return NULL;
    }

    B *b = parse_b();
    if (b == NULL) {
        free(a);
        return NULL;
    }

    C *c = parse_c();
    if (c == NULL) {
        free(a);
        free(b);
        return NULL;
    }

    return new_foo(a, b, c);
}
```

Memory allocators

- libc provides `malloc()` and `free()`
- Problem: object lifetimes are too granular
- Core idea: what if we group objects with similar lifetimes?
  - Allocate objects at different times
  - Freed all at once

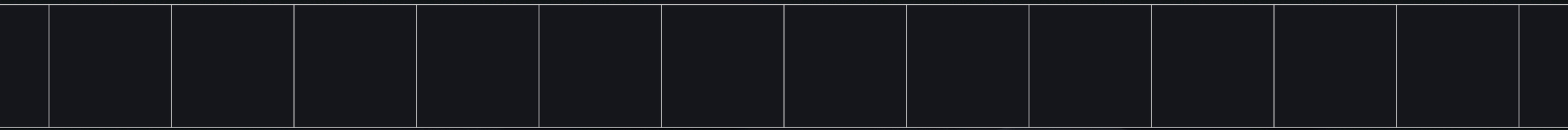
Standard library allocator

```
int   *my_int  = malloc(sizeof(  int));
char  *my_char = malloc(sizeof( char));
foo_t *my_foo  = malloc(sizeof(foo_t));

do_thing(my_int, my_char, my_foo);

free(my_int);
free(my_char);
free(my_foo);
```

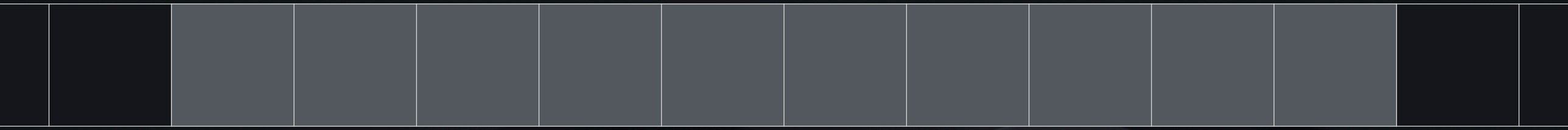
Custom allocator

```
rbs_allocator a;
rbs_allocator_init(&a, 4096);

int   *my_int  = rbs_alloc(a,   int);
char  *my_char = rbs_alloc(a,  char);
foo_t *my_foo  = rbs_alloc(a, foo_t);

do_thing(my_int, my_char, my_foo);

rbs_allocator_free(allocator);
```

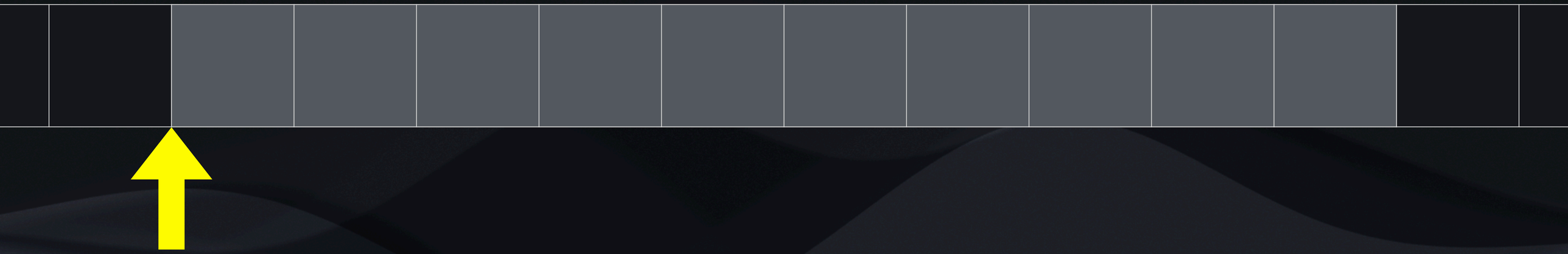
Arena allocation



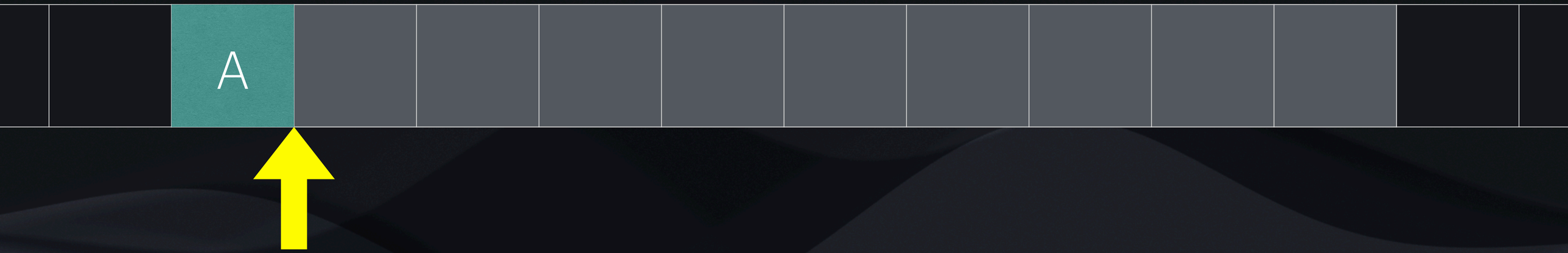
Arena allocation



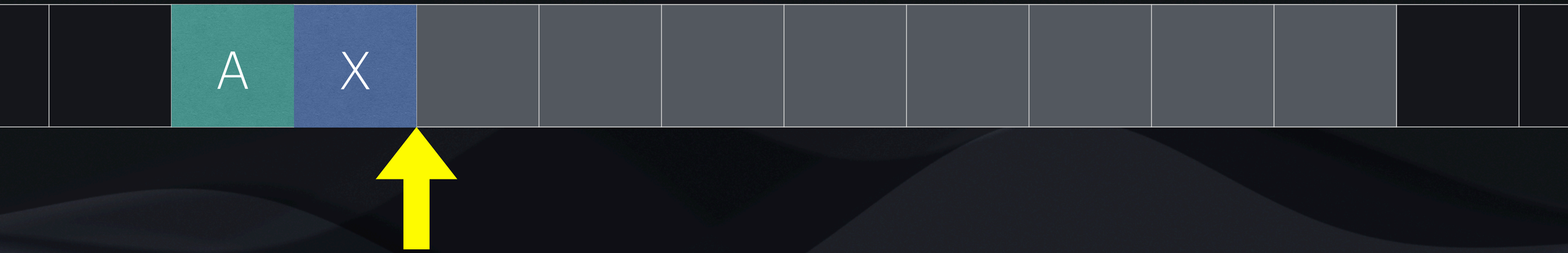
Arena allocation



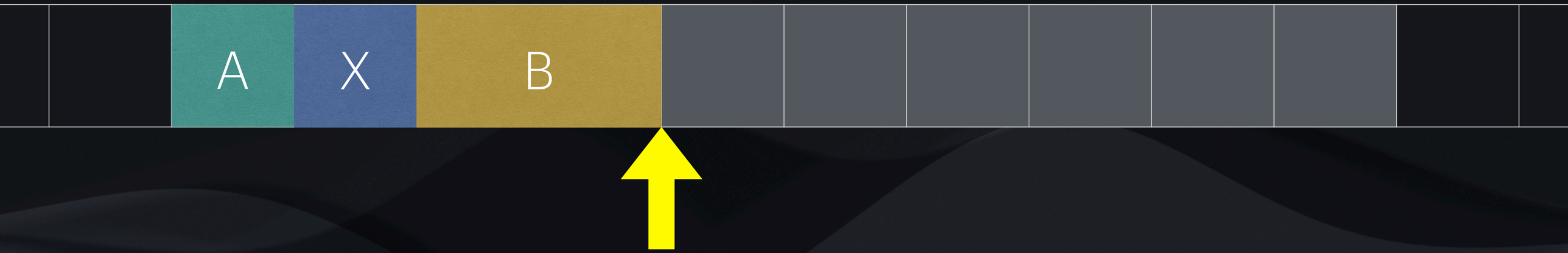
Arena allocation



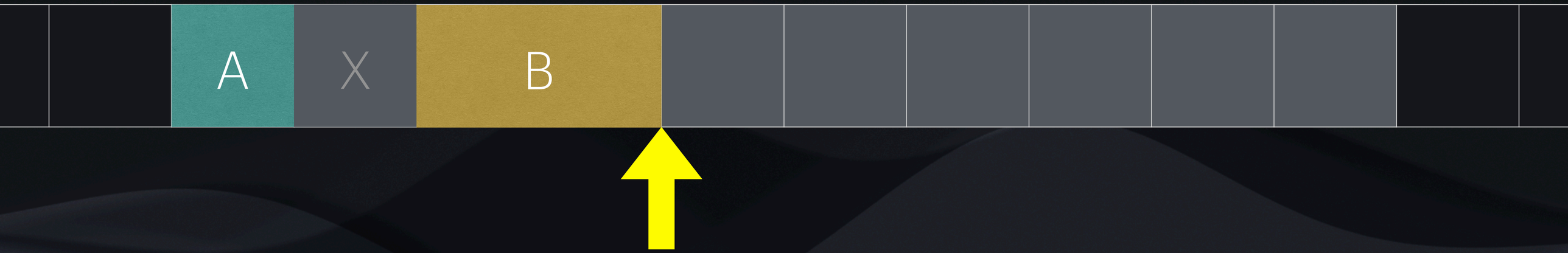
Arena allocation



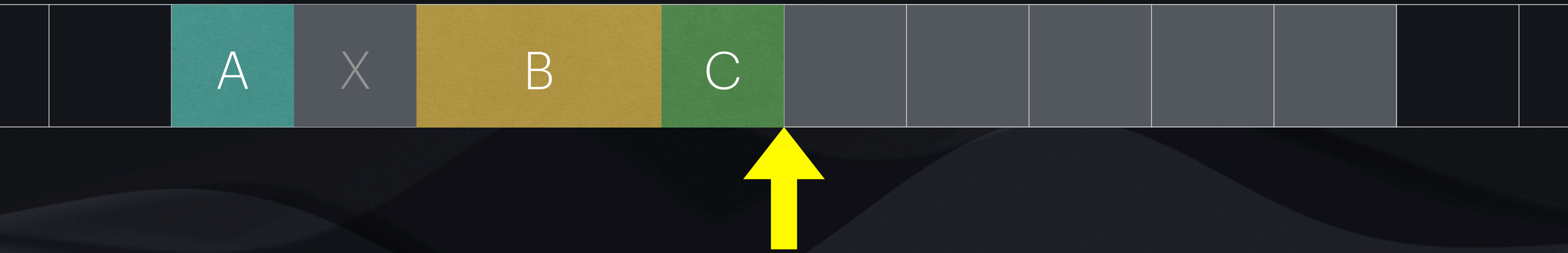
Arena allocation



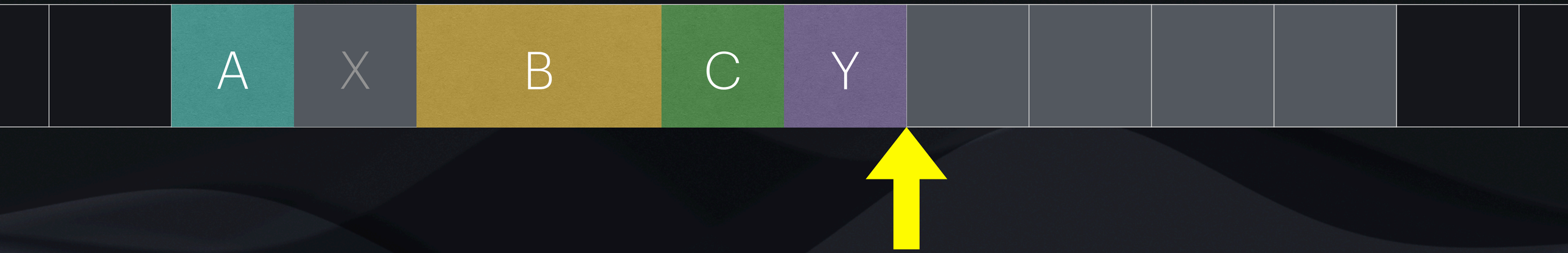
Arena allocation



Arena allocation



Arena allocation



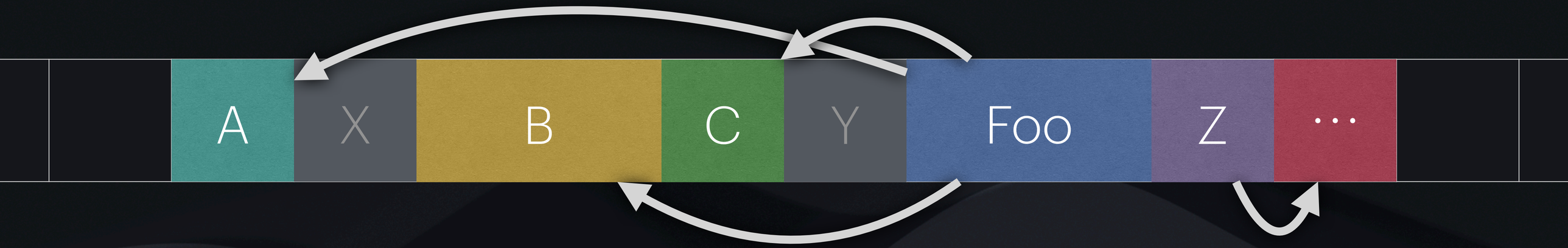
Arena allocation

Arena allocation

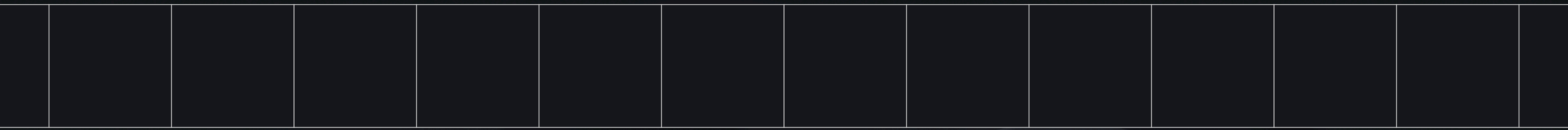
Arena allocation

Arena allocation

Arena allocation

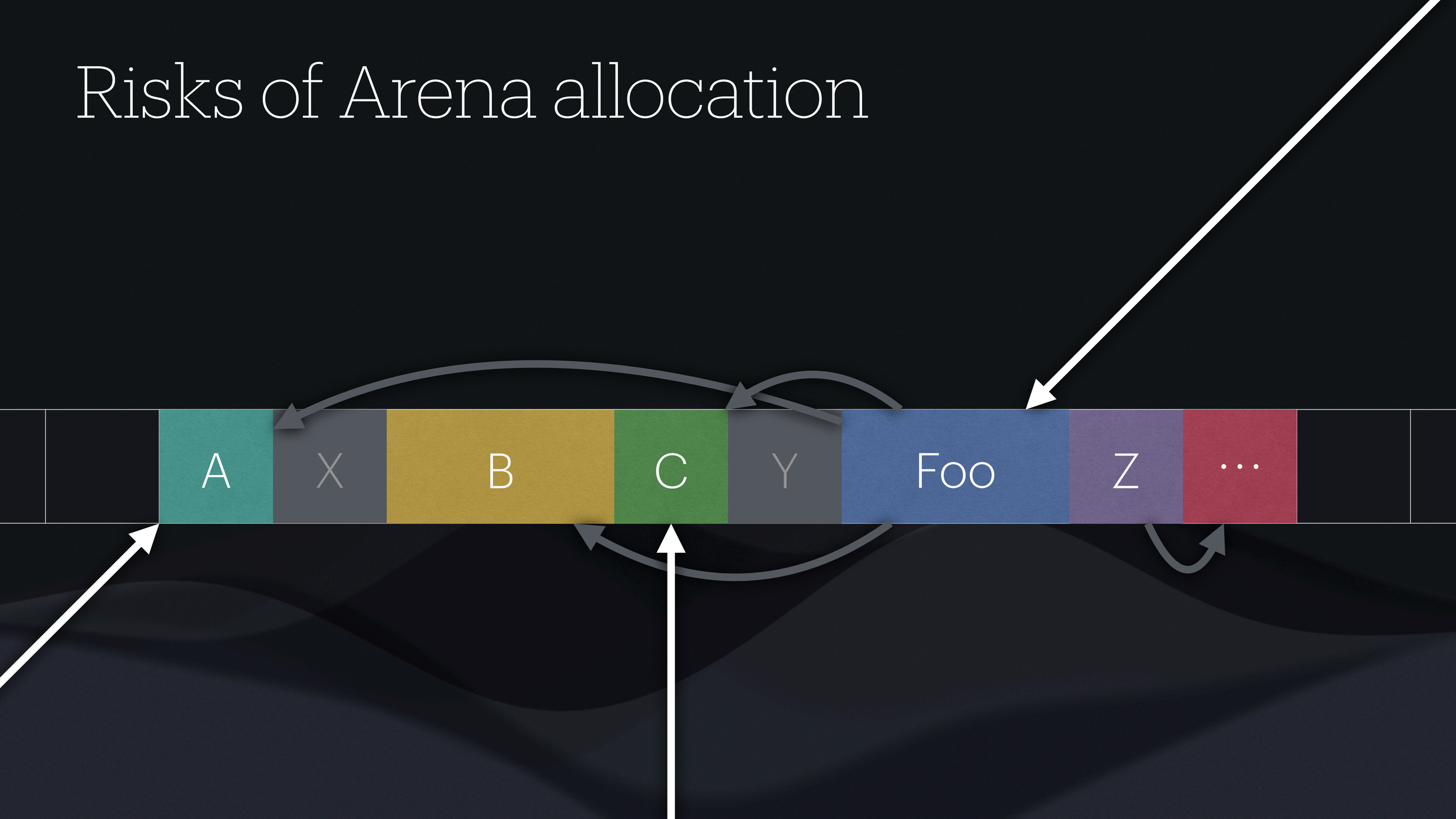


Arena allocation

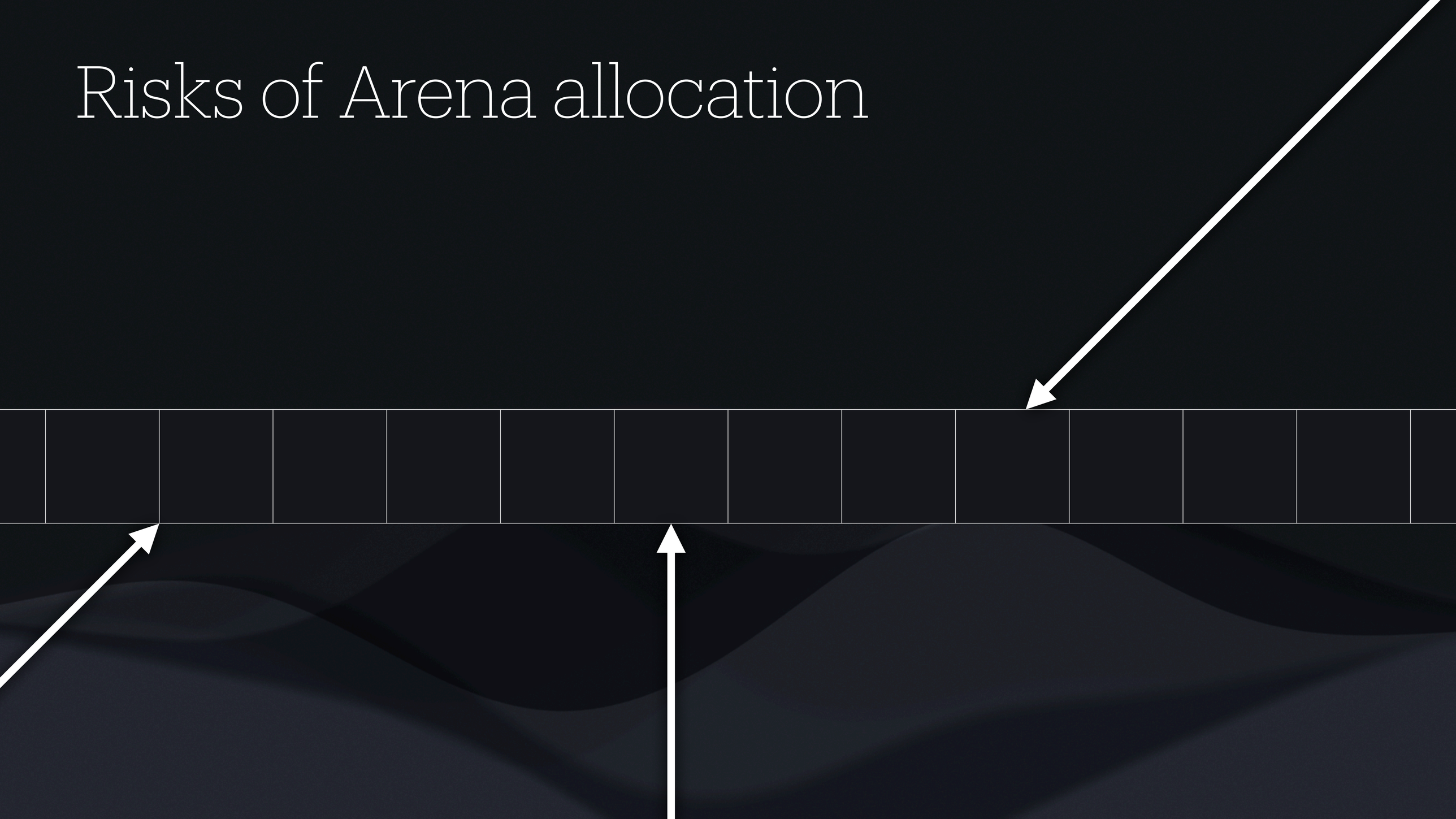


Risks of Arena allocation

Risks of Arena allocation

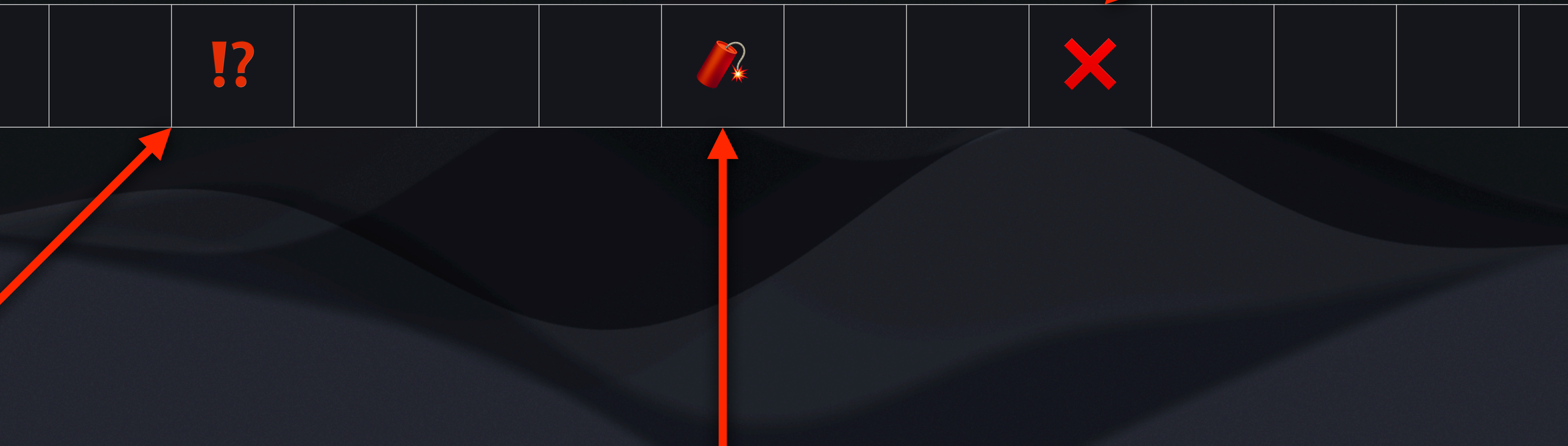


Risks of Arena allocation



Risks of Arena allocation

Dangling pointers!



Now we can write the code we wanted

No more special logic for each returns!

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

Now we can write the code we wanted

No more special logic for each returns!

```
Foo *parse_foo() {
    A *a = parse_a();
    if (a == NULL) return NULL;

    B *b = parse_b();
    if (b == NULL) return NULL;

    C *c = parse_c();
    if (c == NULL) return NULL;

    return new_foo(a, b, c);
}
```

All objects are owned by the arena allocator, which will be freed later

The downside: passing allocators around

It's a lot of plumbing

```
rbs_hash *rbs_hash_new();

rbs_list *rbs_list_new();

rbs_string rbs_unquote_string(rbs_string self);

rbs_string rbs_string_strip_whitespace(rbs_string self);
```

The downside: passing allocators around

It's a lot of plumbing

```
rbs_hash *rbs_hash_new(                    );

rbs_list *rbs_list_new(                    );

rbs_string rbs_unquote_string(                  , rbs_string self);

rbs_string rbs_string_strip_whitespace(rbs_string self);
```

The downside: passing allocators around

It's a lot of plumbing

```
rbs_hash *rbs_hash_new(rbs_allocator *);

rbs_list *rbs_list_new(rbs_allocator *);

rbs_string rbs_unquote_string(rbs_allocator *, rbs_string self);

rbs_string rbs_string_strip_whitespace(rbs_string self);
```

Summary

Extracting a C library from a Ruby Gem

- Why: Pure C libs are more portable than Gems with C extensions
- Not calling into Ruby means GVL won’t limit parallelism
- Establish a consistent pattern around error handling
- Think about lifetimes **early** in your design and implementation
- Embrace popular libraries for fast, well-tested data structures

Join our next RBS talk

- “Inline RBS comments for seamless type checking with Sorbet”
- By Alexandre Terrasa
- Learn more about how Shopify is adopting RBS in its tooling
- Same room
- Tomorrow at 14:00


That's all Folks!