Constexpr STL Containers:
Challenges and a Limitless Allocator Implementation

Presented by Sergey Dobychin
Telegram: @sssersh

Agenda

1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
4. Additional Applications
5. Results

`constexpr`

`constexpr`

`constexpr` - A value of a variable or function can appear in a constant expression.

https://en.cppreference.com/w/cpp/language/constexpr

`constexpr`

`constexpr` - A value of a variable or function can appear in a constant expression.

Constant Expression - An expression that can be evaluated at compile time.

https://en.cppreference.com/w/cpp/language/constexpr

`constexpr`

- Before **C++11**: Basic evaluations

`constexpr`

- Before **C++11**: Basic evaluations
- **C++11**: Introduction of the `constexpr` keyword and `constexpr` functions

`constexpr`

- Before **C++11**: Basic evaluations
- **C++11**: Introduction of the `constexpr` keyword and `constexpr` functions
- **C++20**: Support for `new` and `delete` in `constexpr` functions

`constexpr`

- Before **C++11**: Basic evaluations
- **C++11**: Introduction of the `constexpr` keyword and `constexpr` functions
- **C++20**: Support for `new` and `delete` in `constexpr` functions
- **C++26 (expected)**: Making all standard containers `constexpr`

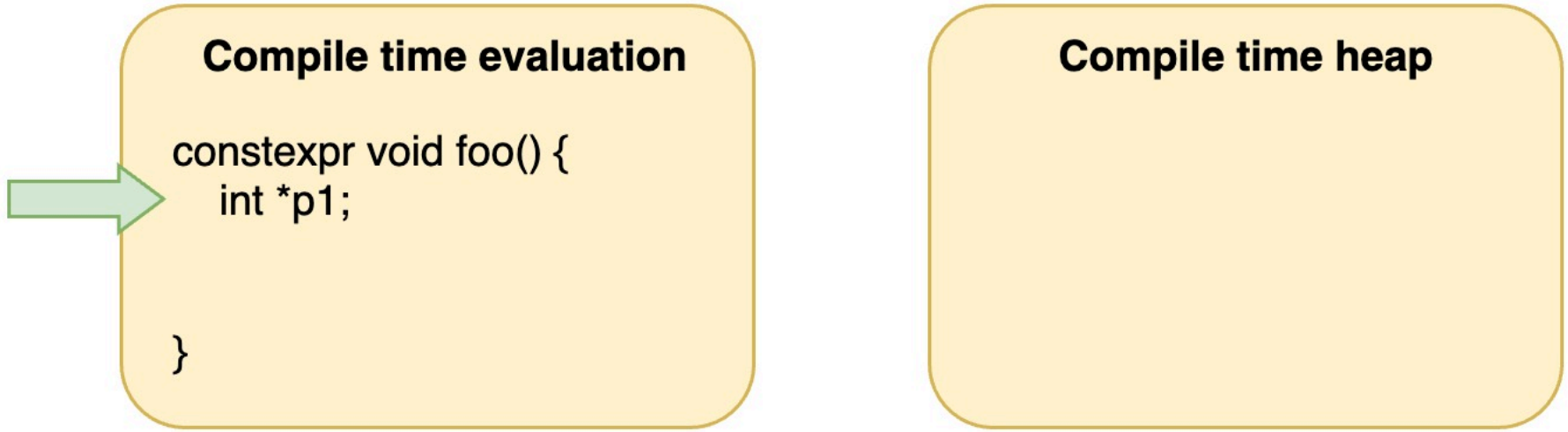
Memory Allocations in `constexpr`

Memory Allocations in `constexpr`

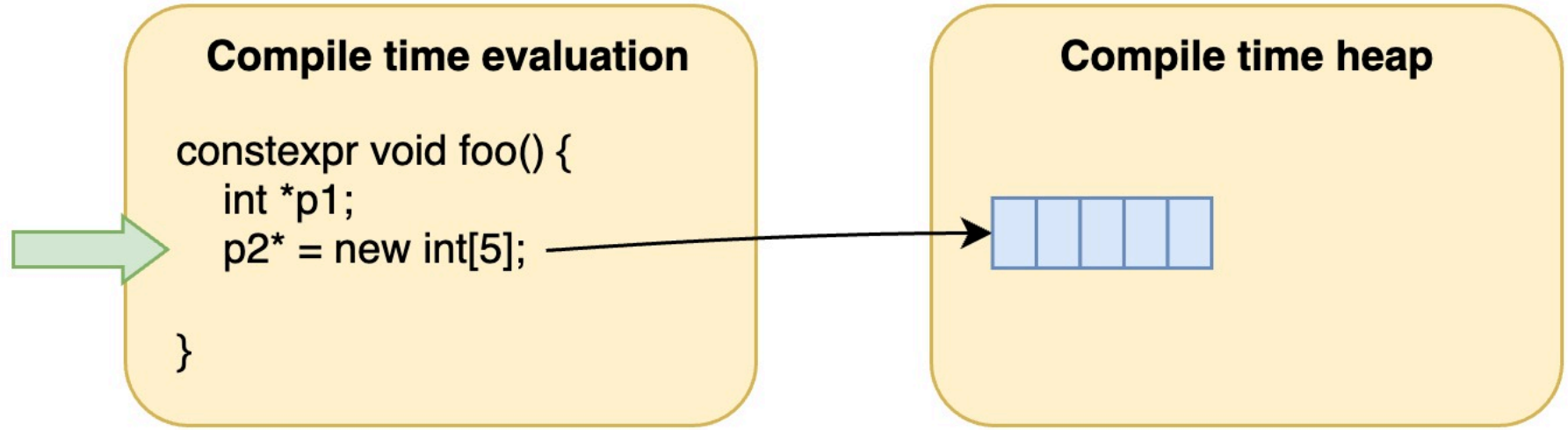
- Transient
- Non-transient

Transient `constexpr` Allocations

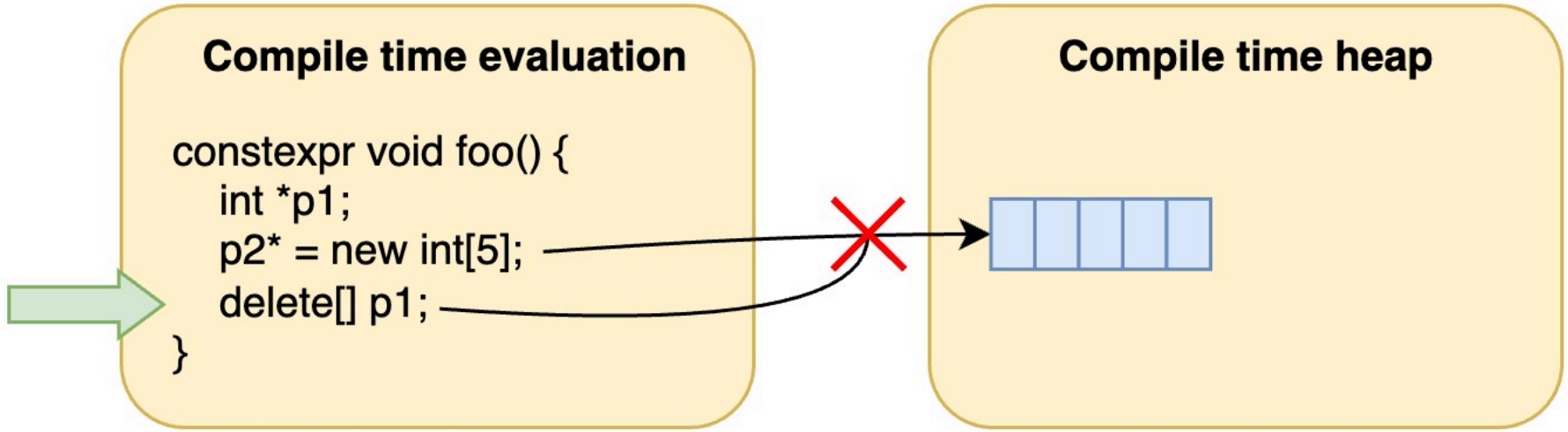
Transient `constexpr` Allocations



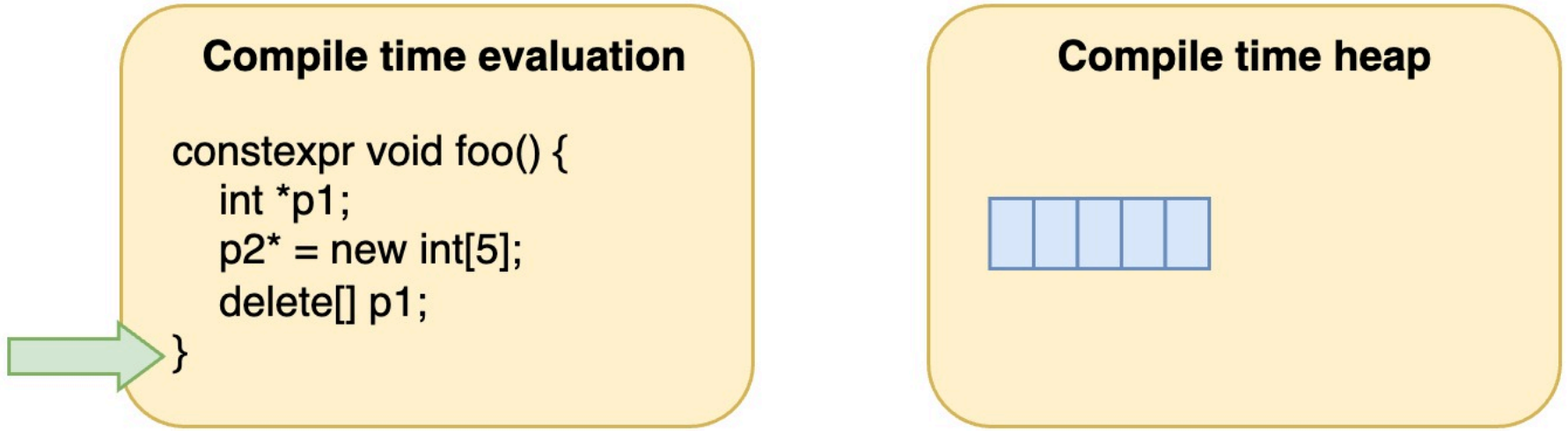
Transient `constexpr` Allocations



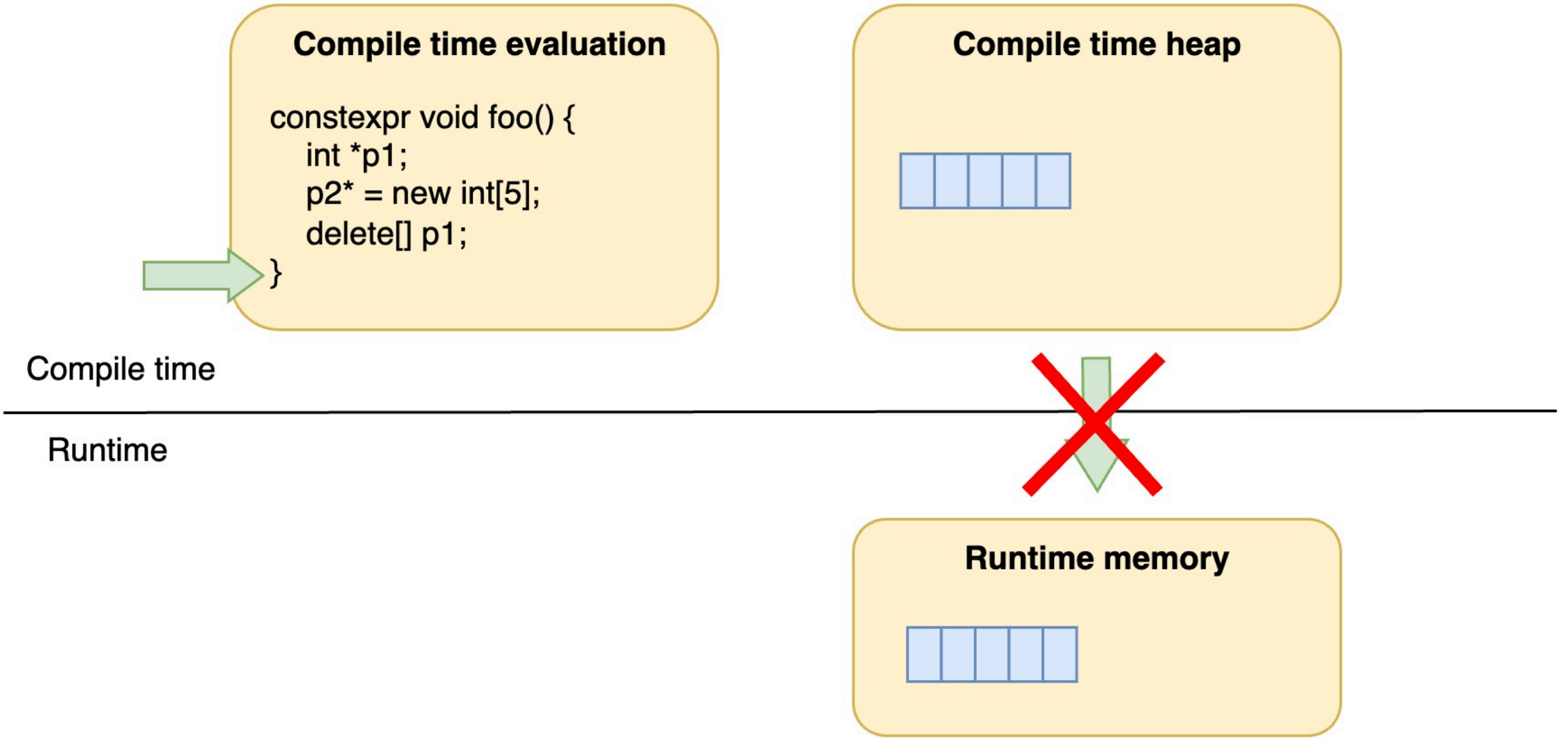
Transient `constexpr` Allocations



Transient `constexpr` Allocations

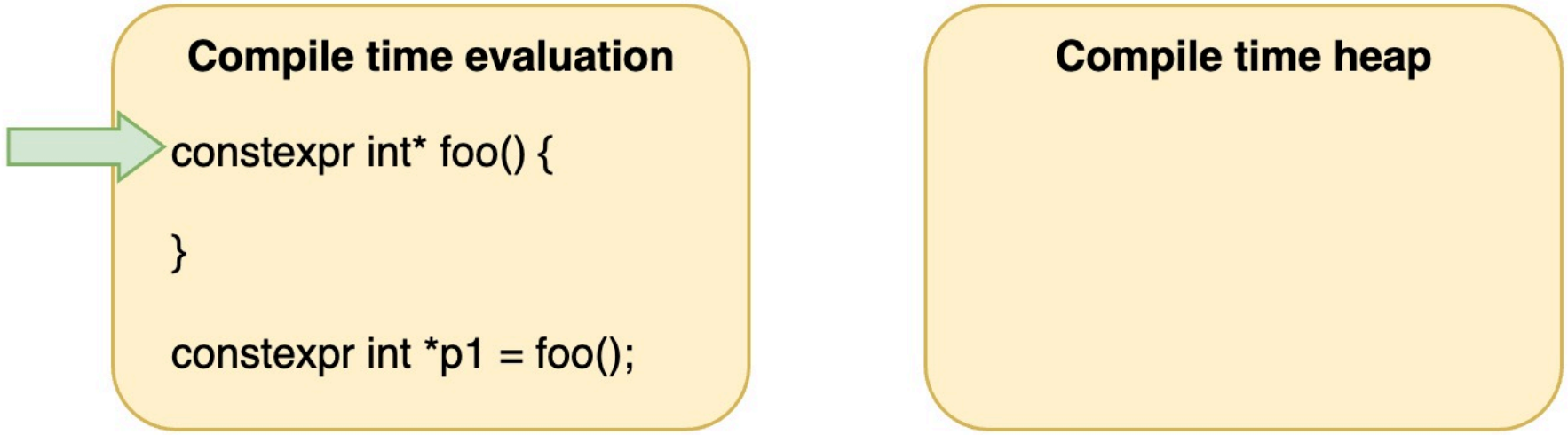


Transient `constexpr` Allocations

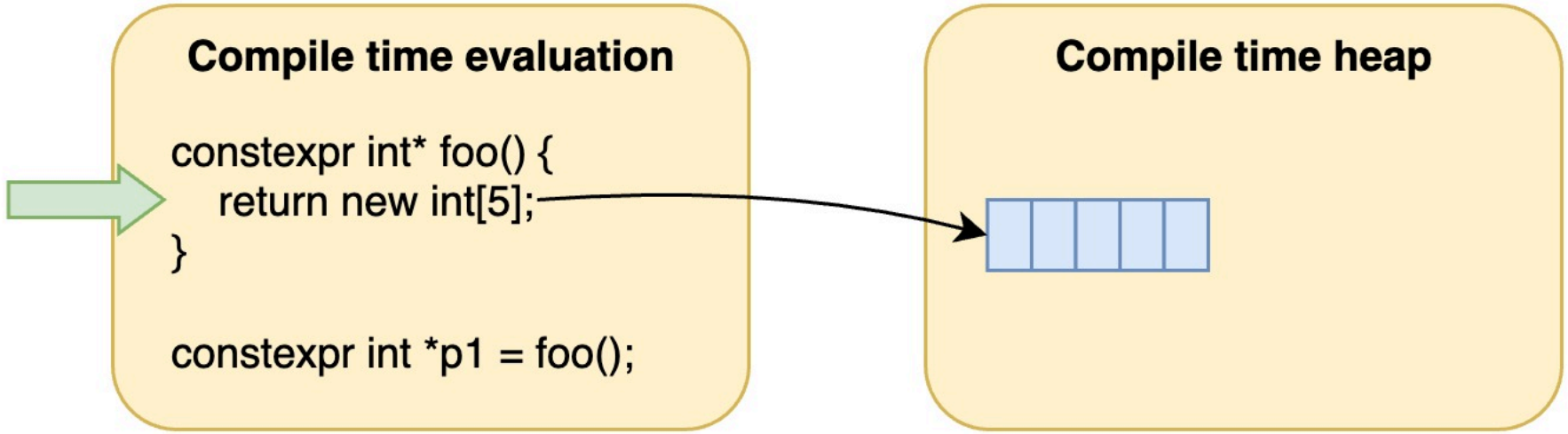


Non-transient `constexpr` Allocations

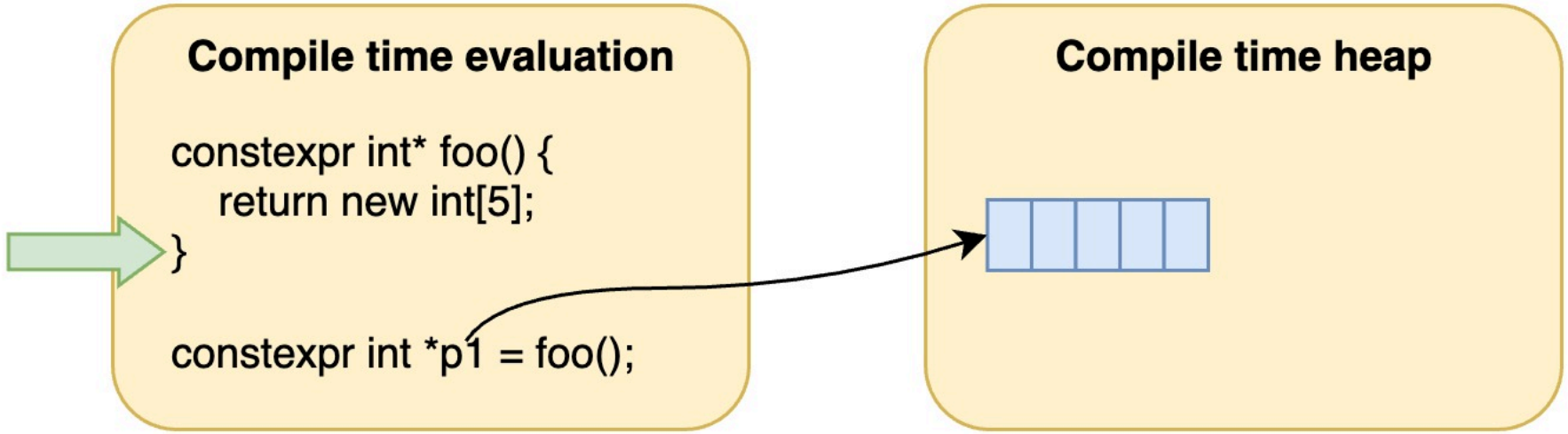
Non-transient `constexpr` Allocations



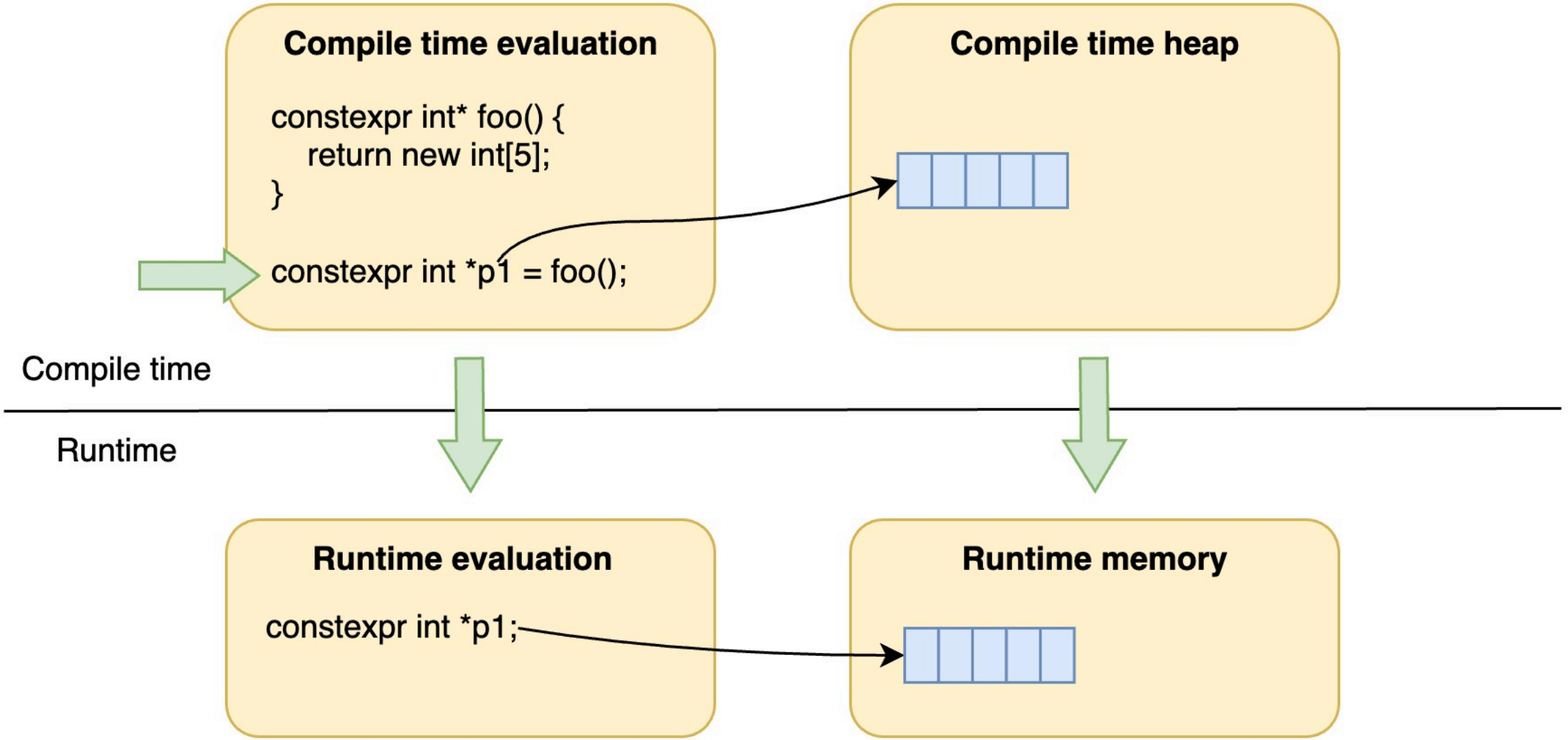
Non-transient `constexpr` Allocations



Non-transient `constexpr` Allocations



Non-transient `constexpr` Allocations



`constexpr` Allocations in C++20

`constexpr` Allocations in C++20

Transient: Possible

```
constexpr void foo() {
    std::vector<int> v = { 1, 2, 3 };
    auto p* = std::allocator<int>().allocate(3);
    std::allocator<int>().deallocate(p);
}
```

`constexpr` Allocations in C++20

Transient: Possible

```
constexpr void foo() {
    std::vector<int> v = { 1, 2, 3 };
    auto p* = std::allocator<int>().allocate(3);
    std::allocator<int>().deallocate(p);
}
```

Non-transient: Impossible

```
constexpr std::vector<int> foo() {
    return { 1, 2, 3 };
}

constexpr std::vector<int> v1 = foo(); // Compile-time Error
constexpr std::vector<int> v2 = { 1, 2, 3, 4, 5, 6 }; // Compile-time Error
constexpr int* data = std::allocator<int>().allocate(3); // Compile-time Error
```

`constexpr` Allocations in C++20

Transient: Possible

```
constexpr void foo() {
    std::vector<int> v = { 1, 2, 3 };
    auto p* = std::allocator<int>().allocate(3);
    std::allocator<int>().deallocate(p);
}
```

Non-transient: Impossible

```
constexpr std::vector<int> foo() {
    return { 1, 2, 3 };
}
// Top-level constexpr Objects
constexpr std::vector<int> v1 = foo(); // Compile-time Error
constexpr std::vector<int> v2 = { 1, 2, 3, 4, 5, 6 }; // Compile-time Error
constexpr int* data = std::allocator<int>().allocate(3); // Compile-time Error
```

1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 **Proposals**
   1.3 Data Structures
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
4. Additional Applications
5. Results

Accepted Proposals for C++20

- P0784
  - `constexpr` new/delete and `std::allocator` (transient)
  - `constexpr` destructors
  - `constexpr` placement new (via `std::construct_at`)

Accepted Proposals for C++20

- **P0784**
  - `constexpr` new/delete and `std::allocator` (transient)
  - `constexpr` destructors
  - `constexpr` placement new (via `std::construct_at`)
- **P1004**: `constexpr` support for `std::vector`
- **P0980**: `constexpr` support for `std::string`

Accepted Proposals for C++20

- **P0784**
  - `constexpr` new/delete and `std::allocator` (transient)
  - `constexpr` destructors
  - `constexpr` placement new (via `std::construct_at`)
- **P1004**: `constexpr` support for `std::vector`
- **P0980**: `constexpr` support for `std::string`
- **P1002**: Allow try/catch in `constexpr`
- **P1006**: `constexpr std::pointer_traits`
- **P2273**: `constexpr std::unique_ptr` (C++23)

Accepted Proposals (C++26)

- **P3372**
  - `constexpr` containers and adaptors
- **P2747**
  - `constexpr` placement new
- **P2738**
  - `constexpr` cast from `void*`
- **P2686**
  - `constexpr` structured bindings and references to `constexpr` variables
- **P3491**
  - `define_static_{string,object,array}`

P3491: `define_static_{string,object,array}`

- only since C++26
- requires separate built-in functions for each container
- doesn't work with custom containers

Work-in-Progress Proposals

- **P3094**
  - `std::basic_fixed_string`
- **P3032**
  - Less transient `constexpr` allocation
- **P1974**
  - Non-transient `constexpr` allocation using `propconst`
- **P2670**
  - Non-transient `constexpr` allocation

P1974

```
constexpr unique_ptr<unique_ptr<int>> uui
    = make_unique<unique_ptr<int>>(
        make_unique<int>());

int main() {
    unique_ptr<int>& ui = *uui;
    ui.reset();
}
```

Rejected Proposals

- P0784

Rejected Proposals

- **P0784**
  - Rejected for non-transient allocations

Rejected Proposals

- **P0784**
  - Rejected for non-transient allocations
- **P0574**
  - `std::constexpr_vector`

Rejected Proposals

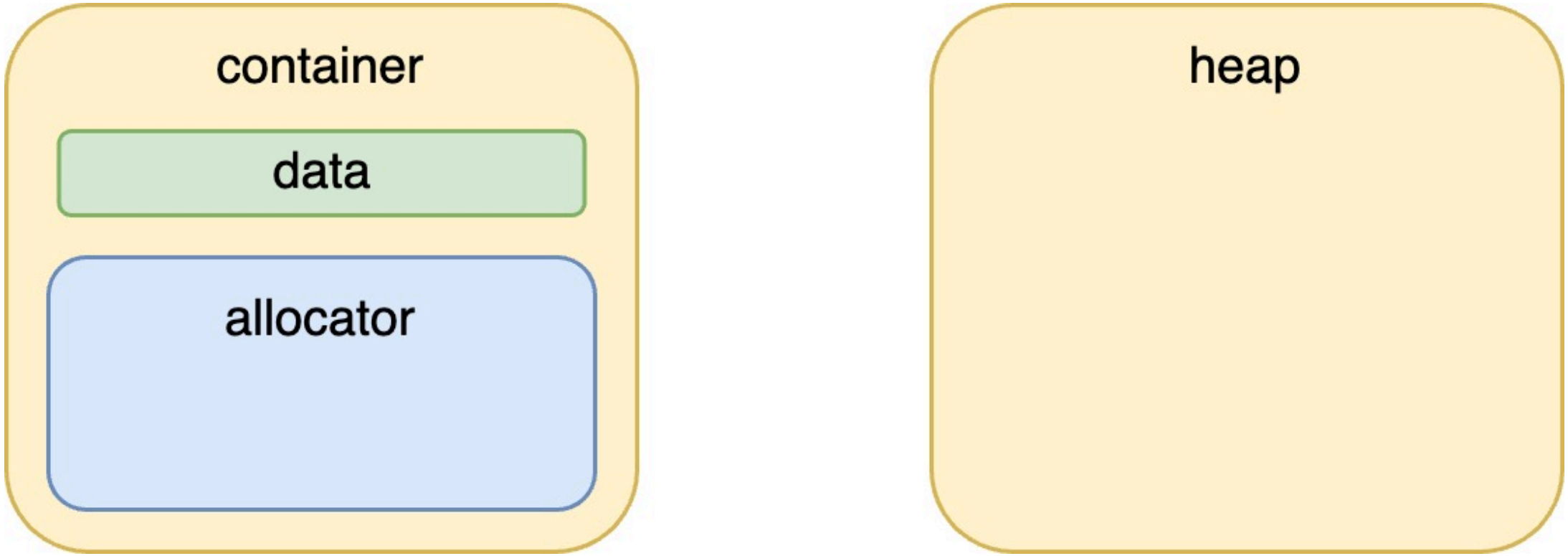
- **P0784**
  - Rejected for non-transient allocations
- **P0574**
  - `std::constexpr_vector`
    - **P0639** - Changing attack vector of the `constexpr_vector`

Proposal P0639

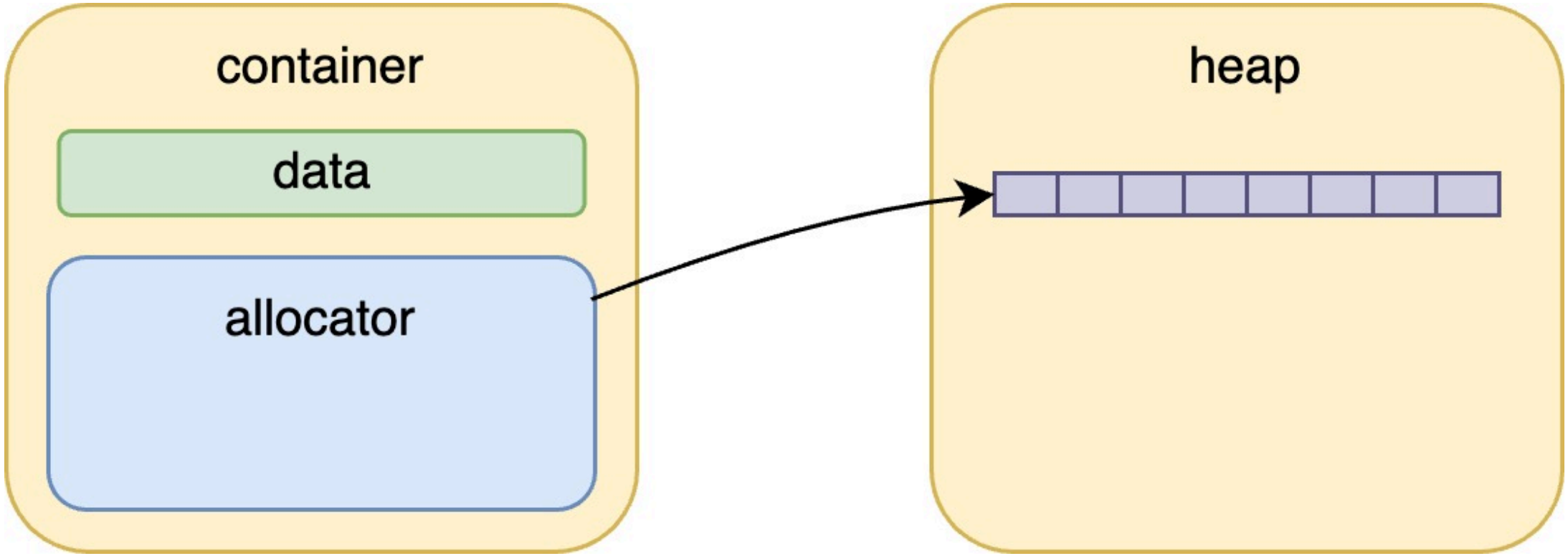
```
template <class T, unsigned N = 100>
struct constexpr_allocator {
    T data[N] = {};
    ...
}
```

https://github.com/zamazan4ik/constexpr_allocator/blob/master/test.cpp

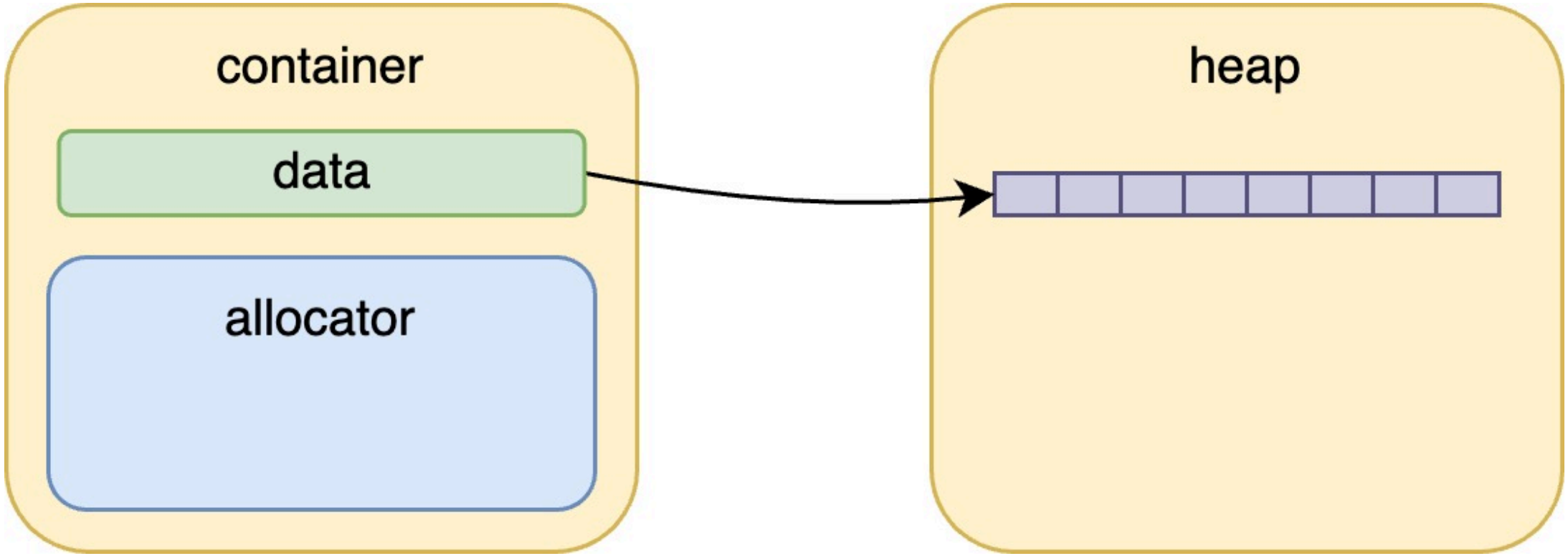
Proposal P0639



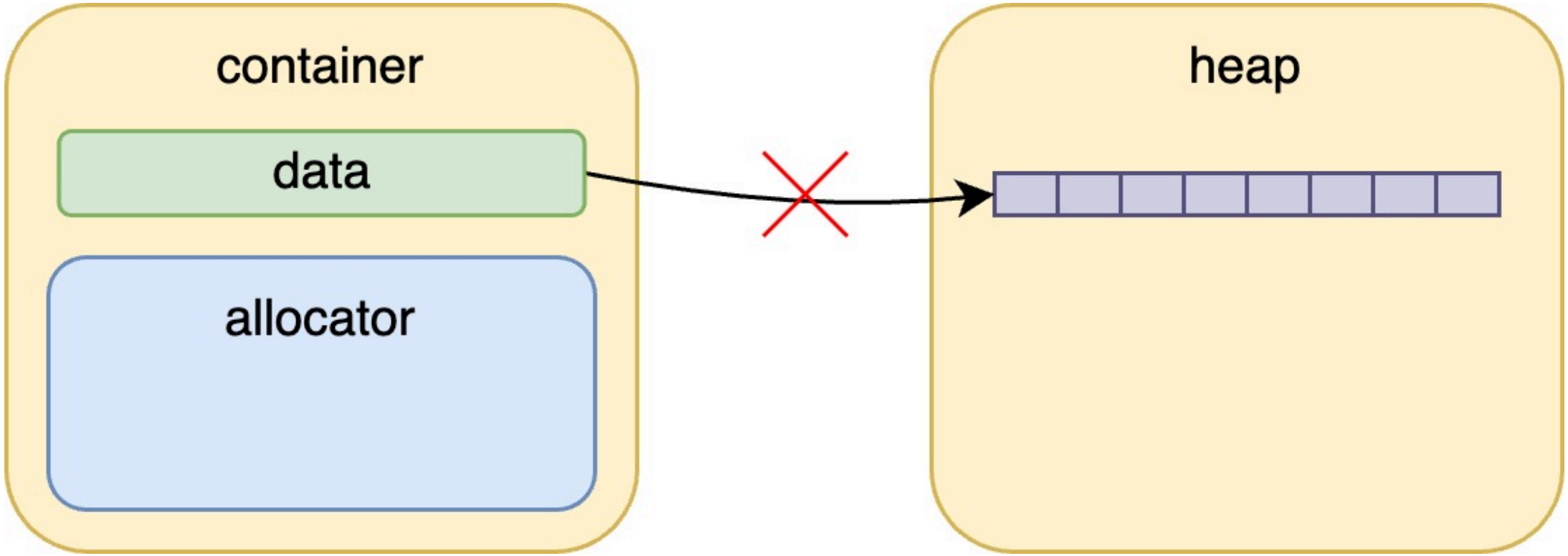
Proposal P0639



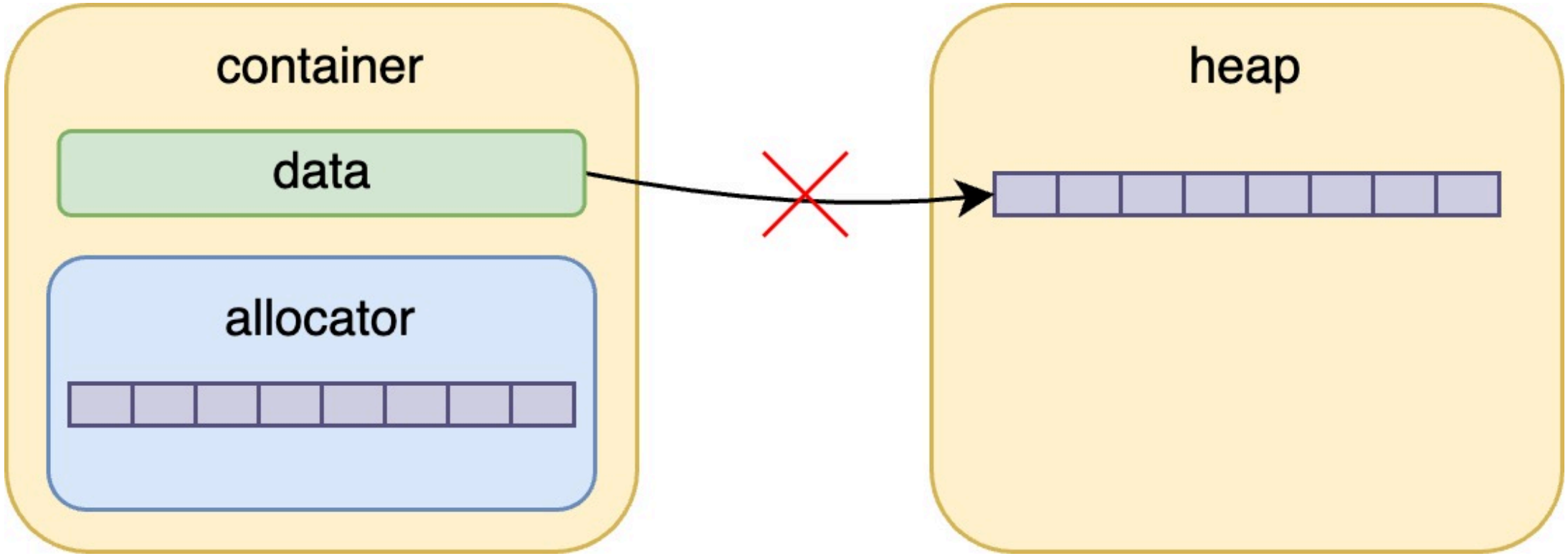
Proposal P0639



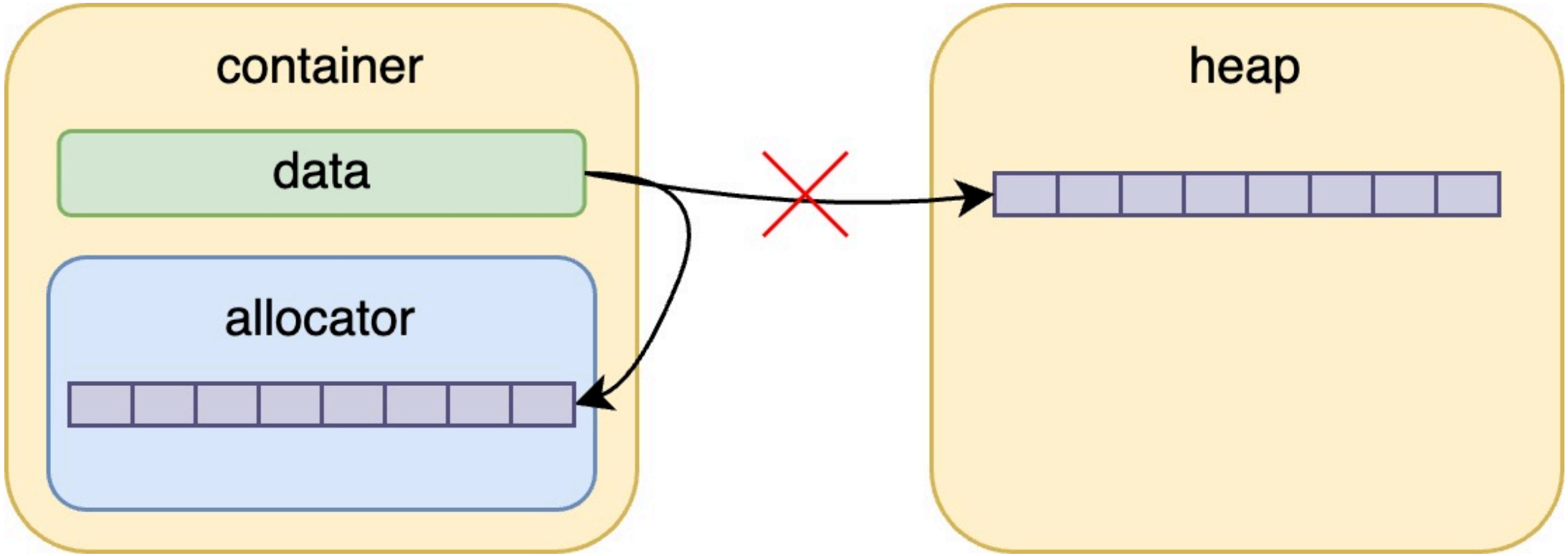
Proposal P0639



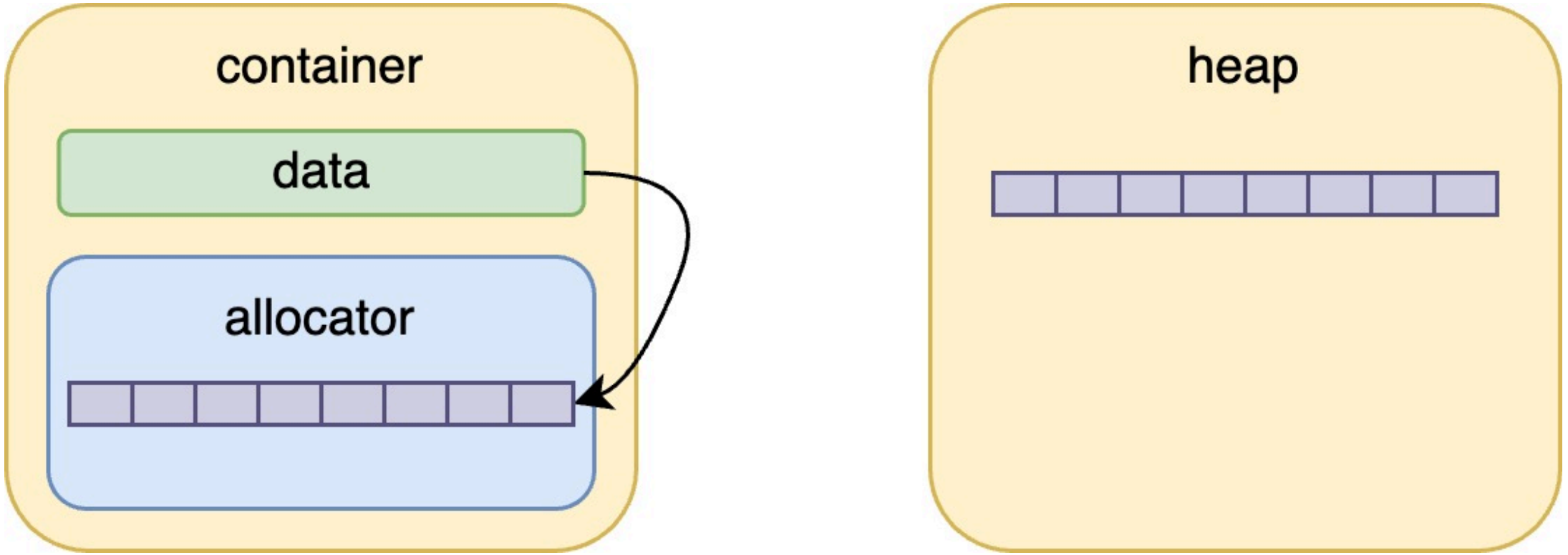
Proposal P0639



Proposal P0639



Proposal P0639



1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 **Data Structures**
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
4. Additional Applications
5. Results

Containers with Local Buffer

- `constexpr` containers:
  - `std::array`
  - library **frozen**
  - library **constainer**
  - `std::string`*
- Non-`constexpr` containers
  - static_vector, small_vector, and others from Boost
  - FastPimpl from the **userver** library
- etc.

Containers with Local Buffer

- `constexpr` containers:
  - `std::array`
  - library **frozen**
  - library **constainer**
  - **`std::string`** *
- Non-`constexpr` containers
  - static_vector, small_vector, and others from Boost
  - FastPimpl from the **userver** library
- etc.

`std::string` and `constexpr`

```
static constexpr std::string s1 = "string for example";
```

`std::string` and `constexpr`

```
// compilation error
static constexpr std::string s1 = "string for example";
```

`std::string` and `constexpr`

```
// compilation error
static constexpr std::string s1 = "string for example";

// compiled sucessfully due to SSO
static constexpr std::string s2 = "small";
```

Overview: Conclusion

Overview: Conclusion

- `constexpr` **allocations**:
  - Currently, only transient allocations are supported.

Overview: Conclusion

- **`constexpr` allocations**:
  - Currently, only transient allocations are supported.
- **top-level `constexpr` containers**:
  - Containers with *local buffer* are supported.

Overview: Conclusion

- **`constexpr` allocations**:
  - Currently, only transient allocations are supported.
- **top-level `constexpr` containers**:
  - Containers with *local buffer* are supported.
  - In the standard library, only `std::array` qualifies.

1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
2. Analysis
   2.1 **Challenges and Solutions**
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
4. Additional Applications
5. Results

Challenges

- How to calculate the buffer size?

Buffer Size: Explicit Definition

```
constexpr auto array = std::array<int, 5>({ 1, 2, 3, 4, 5 });

constexpr frozen::unordered_map<frozen::string, int, 2> olaf = {
    {"19", 19},
    {"31", 31},
};
```

Buffer Size: Deduce from Initializer

Buffer Size: Deduce from Initializer

`std::array`:

```
template<class T, std::size_t N>
constexpr std::array<std::remove_cv_t<T>, N> to_array( T (&a)[N] );
```

Frozen library:

```
template <typename T, typename U, std::size_t N>
constexpr auto make_unordered_map( std::pair<T, U> const (&items)[N] ) {
  return unordered_map<T, U, N>{items};
}
```

Buffer Size: Deduce from Initializer

`std::array`:

```
template<class T, std::size_t N>
constexpr std::array<std::remove_cv_t<T>, N> to_array( T (&a)[N] );
```

Frozen library:

```
template <typename T, typename U, std::size_t N>
constexpr auto make_unordered_map( std::pair<T, U> const (&items)[N] ) {
  return unordered_map<T, U, N>{items};
}
```

- **works only in simple cases**

Buffer Size: Calculate in Two Steps

Buffer Size: Calculate in Two Steps

Approach: Create the object in two compilation steps.

- FastPimpl by Antony Polukhin
  https://www.youtube.com/watch?v=mkPTreWiglk
  First compilation triggers an error where the compiler outputs the deduced buffer size.
  This size is then substituted in the second compilation step.

Buffer Size: Calculate in Two Steps

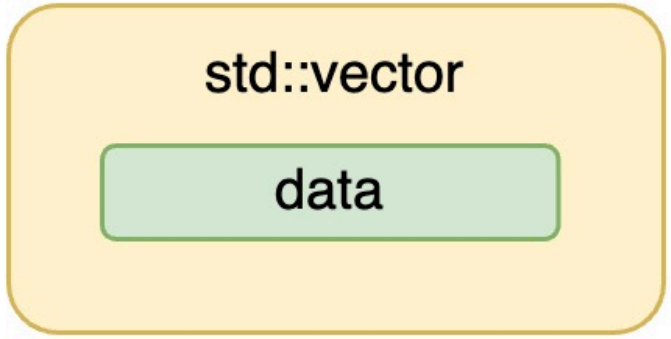
Approach: Create the object in two compilation steps.

- FastPimpl by Antony Polukhin
  https://www.youtube.com/watch?v=mkPTreWiglk
  First compilation triggers an error where the compiler outputs the deduced buffer size.
  This size is then substituted in the second compilation step.
- Bartłomiej Filipek's method:
  https://www.cppstories.com/2021/constexpr-vecstr-cpp20/

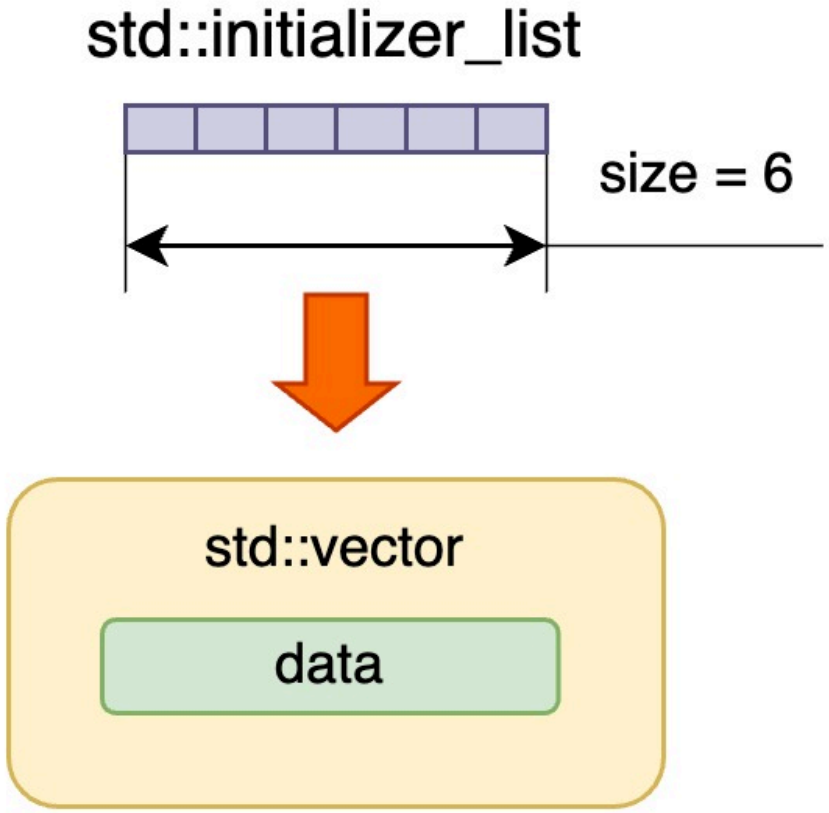
```
constexpr auto str = "hello world abc programming";
constexpr auto word = longestWord<longestWordSize(str)>(str);
```

1. Dynamic memory and containers in constexpr
2. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
3. Analysis
   2.1 Challenges and Solutions
   2.2 **Non-Trivial Cases**
4. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
5. Additional Applications
6. Results

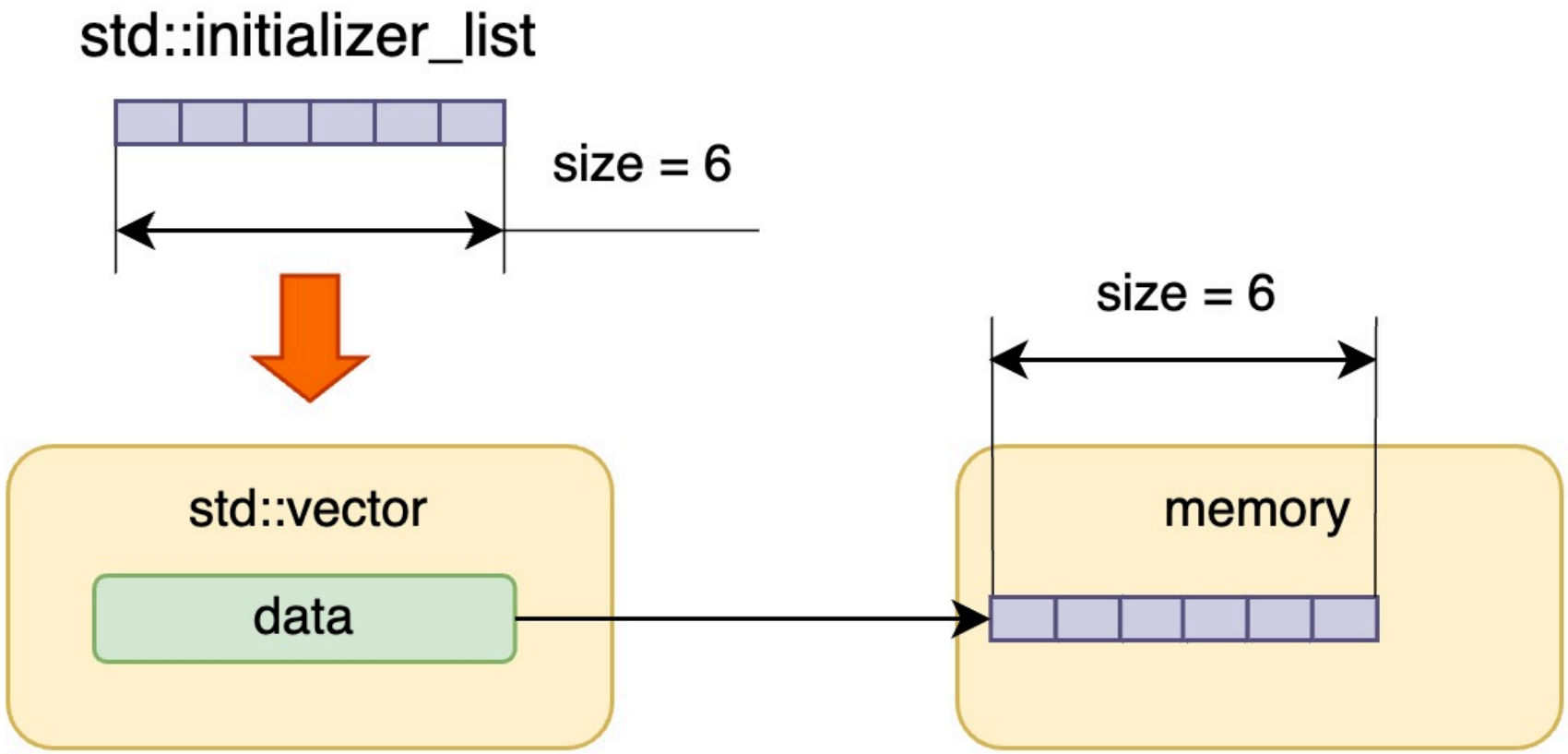
Non-Trivial Buffer Size: Example 1



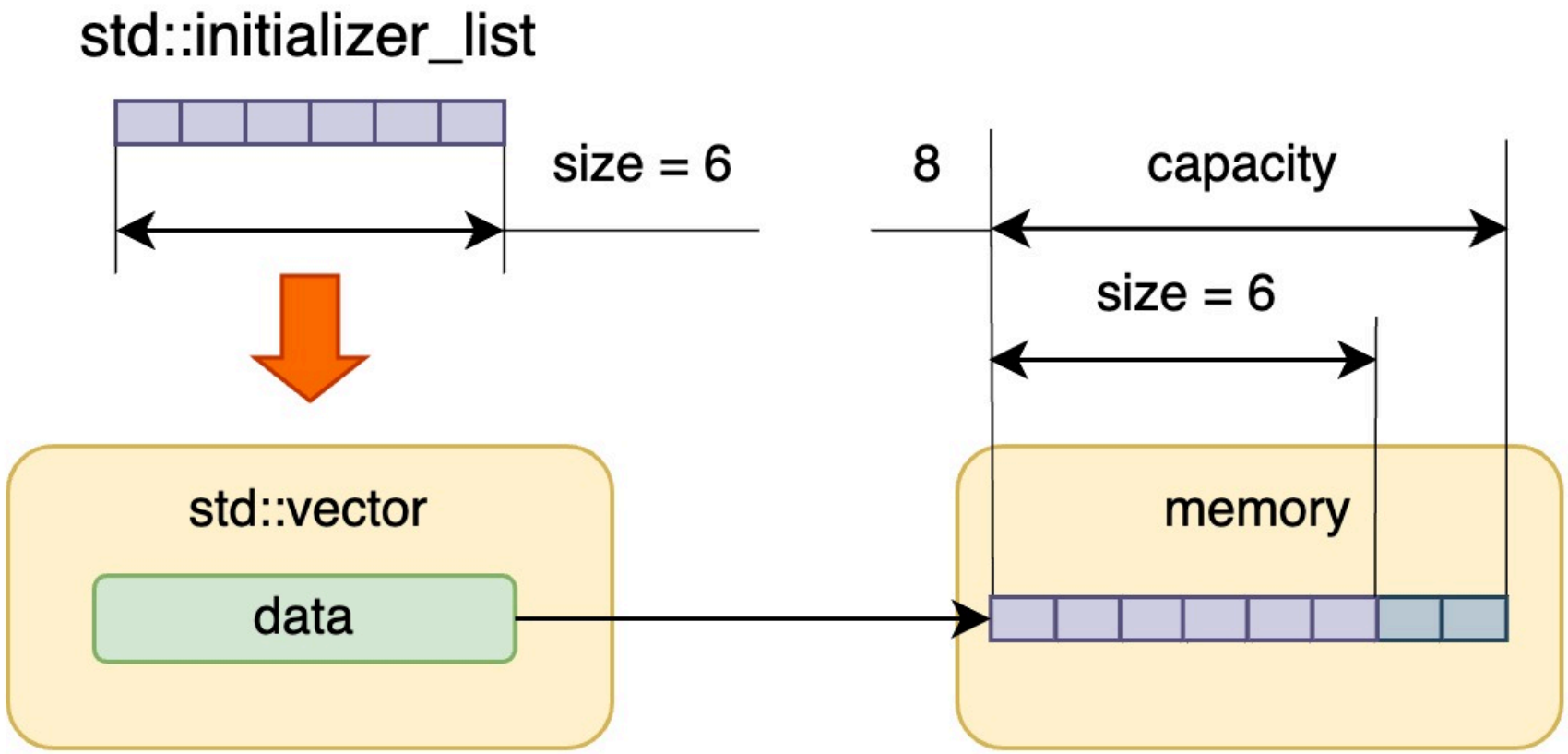
Non-Trivial Buffer Size: Example 1



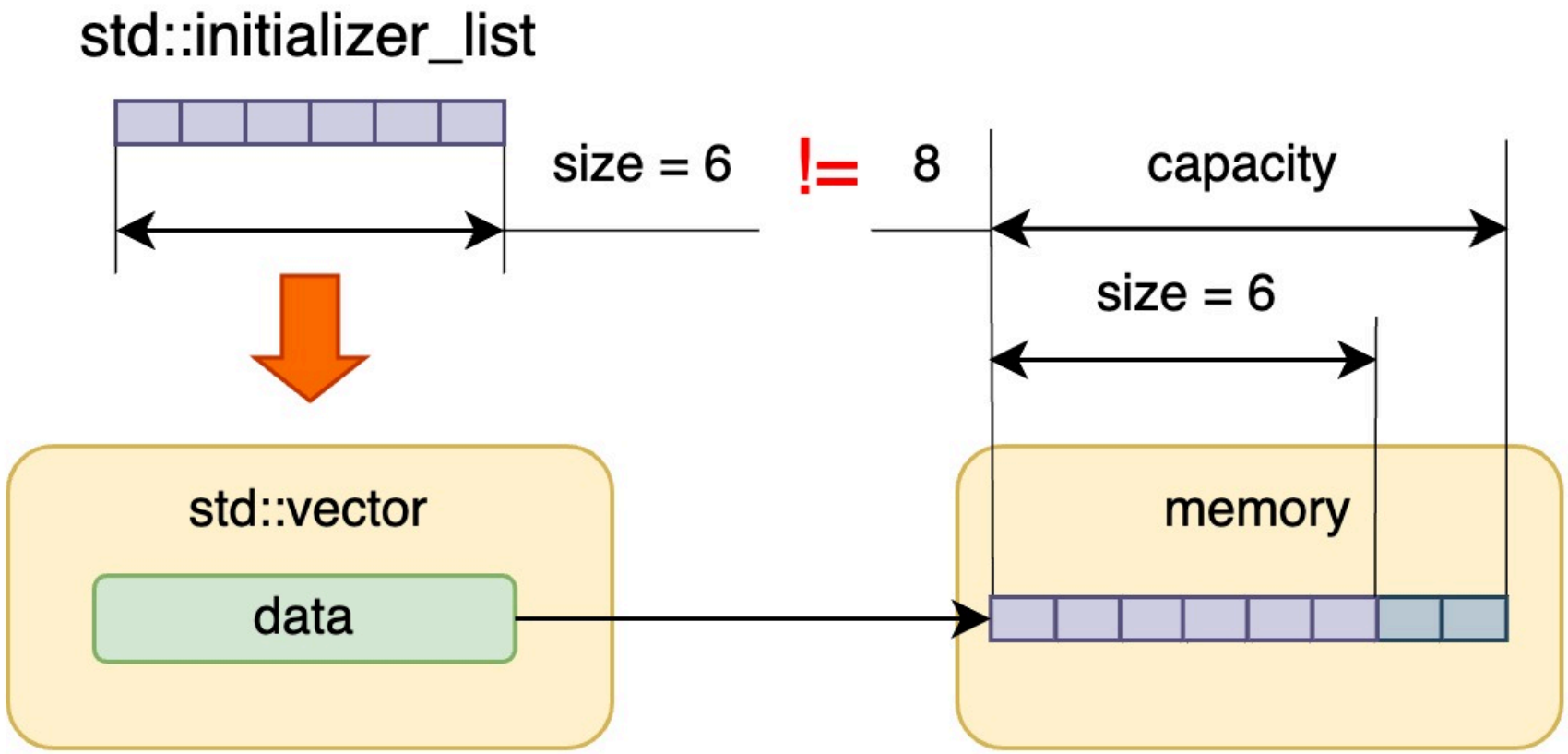
Non-Trivial Buffer Size: Example 1



Non-Trivial Buffer Size: Example 1



Non-Trivial Buffer Size: Example 1



Non-Trivial Buffer Size: Example 2

```
std::vector<int> v = { 1, 2, 3 };
std::unordered_map<int, int> v = { { 1, 1 },  { 2, 3 },  { 3, 3 } };
```

Non-Trivial Buffer Size: Example 2

```
std::vector<int> v = { 1, 2, 3 };
std::unordered_map<int, int> v = { { 1, 1 },  { 2, 3 },  { 3, 3 } };
```

What is *asymptotic complexity*?

Non-Trivial Buffer Size: Example 2

```
vector( std::initializer_list<T> init,                                  (since C++11)
        const Allocator& alloc = Allocator() );                         (until C++20)
constexpr vector( std::initializer_list<T> init,                (10)
                  const Allocator& alloc = Allocator() );               (since C++20)
```

https://en.cppreference.com/w/cpp/container/vector/vector

Non-Trivial Buffer Size: Example 2

```
vector( std::initializer_list<T> init,                                  (since C++11)
        const Allocator& alloc = Allocator() );                         (until C++20)
constexpr vector( std::initializer_list<T> init,                (10)
                  const Allocator& alloc = Allocator() );               (since C++20)
```

...

Complexity

...

https://en.cppreference.com/w/cpp/container/vector/vector

Non-Trivial Buffer Size: Example 2

```
vector( std::initializer_list<T> init,                                  (since C++11)
        const Allocator& alloc = Allocator() );                         (until C++20)
constexpr vector( std::initializer_list<T> init,                (10)
                  const Allocator& alloc = Allocator() );               (since C++20)
```

...

Complexity

...

10) Linear in size of `init`.

https://en.cppreference.com/w/cpp/container/vector/vector

Non-Trivial Buffer Size: Example 2

```
unordered_map( std::initializer_list<value_type> init,
               size_type bucket_count = /* implementation-defined */,
               const Hash& hash = Hash(),                              (13)   (since C++11)
               const key_equal& equal = key_equal(),
               const Allocator& alloc = Allocator() );
```

https://en.cppreference.com/w/cpp/container/unordered_map/unordered_map

Non-Trivial Buffer Size: Example 2

```
unordered_map( std::initializer_list<value_type> init,
               size_type bucket_count = /* implementation-defined */,
               const Hash& hash = Hash(),                              (13)   (since C++11)
               const key_equal& equal = key_equal(),
               const Allocator& alloc = Allocator() );
```

...

Complexity

...

https://en.cppreference.com/w/cpp/container/unordered_map/unordered_map

Non-Trivial Buffer Size: Example 2

```
unordered_map( std::initializer_list<value_type> init,
               size_type bucket_count = /* implementation-defined */,
               const Hash& hash = Hash(),                              (13)   (since C++11)
               const key_equal& equal = key_equal(),
               const Allocator& alloc = Allocator() );
```

...

Complexity

...

13-15) Average case $O(N)$ (N is `std::size(init)`), worst case $O(N^2)$.

https://en.cppreference.com/w/cpp/container/unordered_map/unordered_map

Non-Trivial Buffer Size: Example 2

```
unordered_map(initializer_list<value_type> __il)
{
    insert(__il.begin(), __il.end());
}
```

https://github.com/llvm-mirror/libcxx/blob/master/include/unordered_map

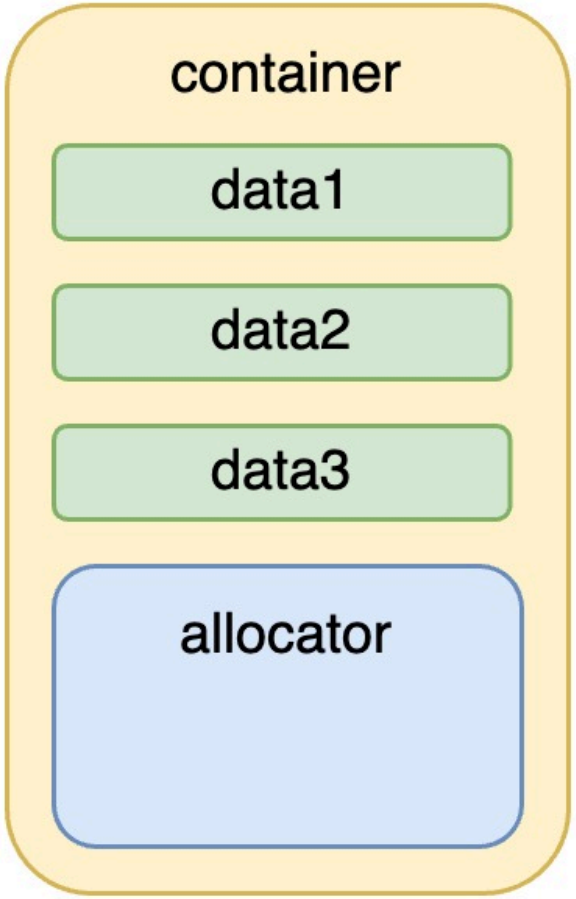
Non-Trivial Buffer Size: Example 2

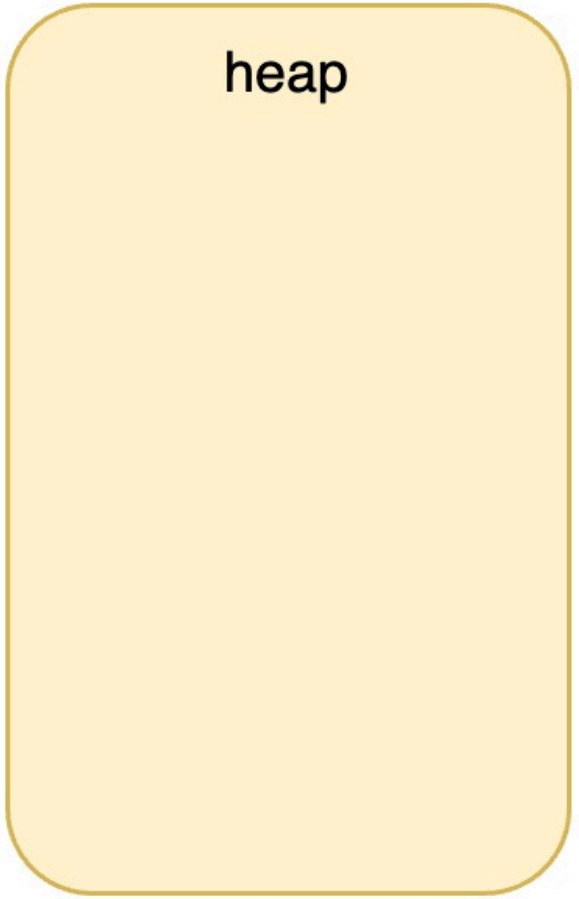
```
unordered_map(initializer_list<value_type> __il)
{
    insert(__il.begin(), __il.end());
}

void insert(_InputIterator __first, _InputIterator __last)
{
    for (; __first != __last; ++__first)
        __table_.__insert_unique(*__first);
}
```

https://github.com/llvm-mirror/libcxx/blob/master/include/unordered_map

Non-Trivial Buffer Size: Example 2

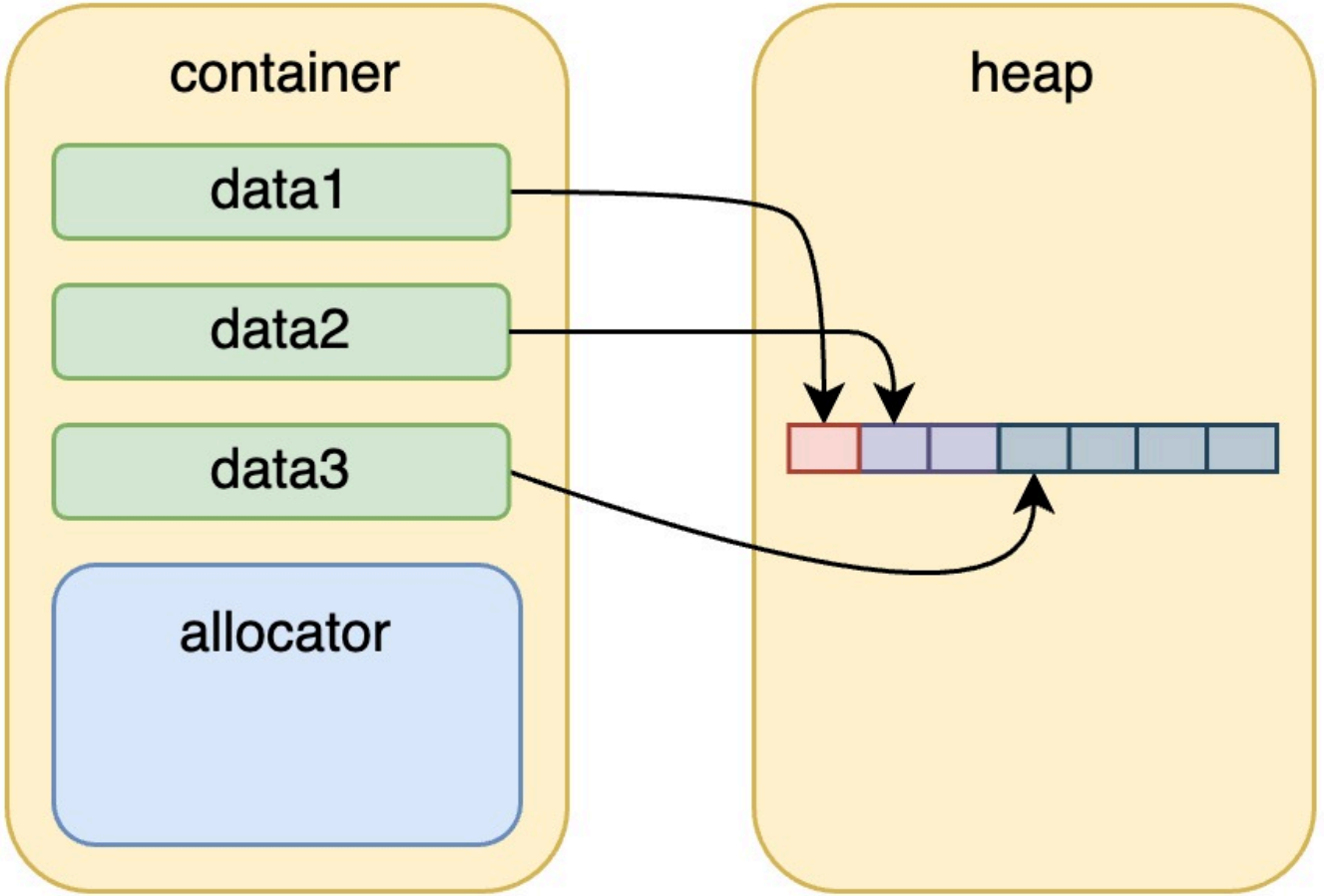
```
container(init_type init)
{




}
```





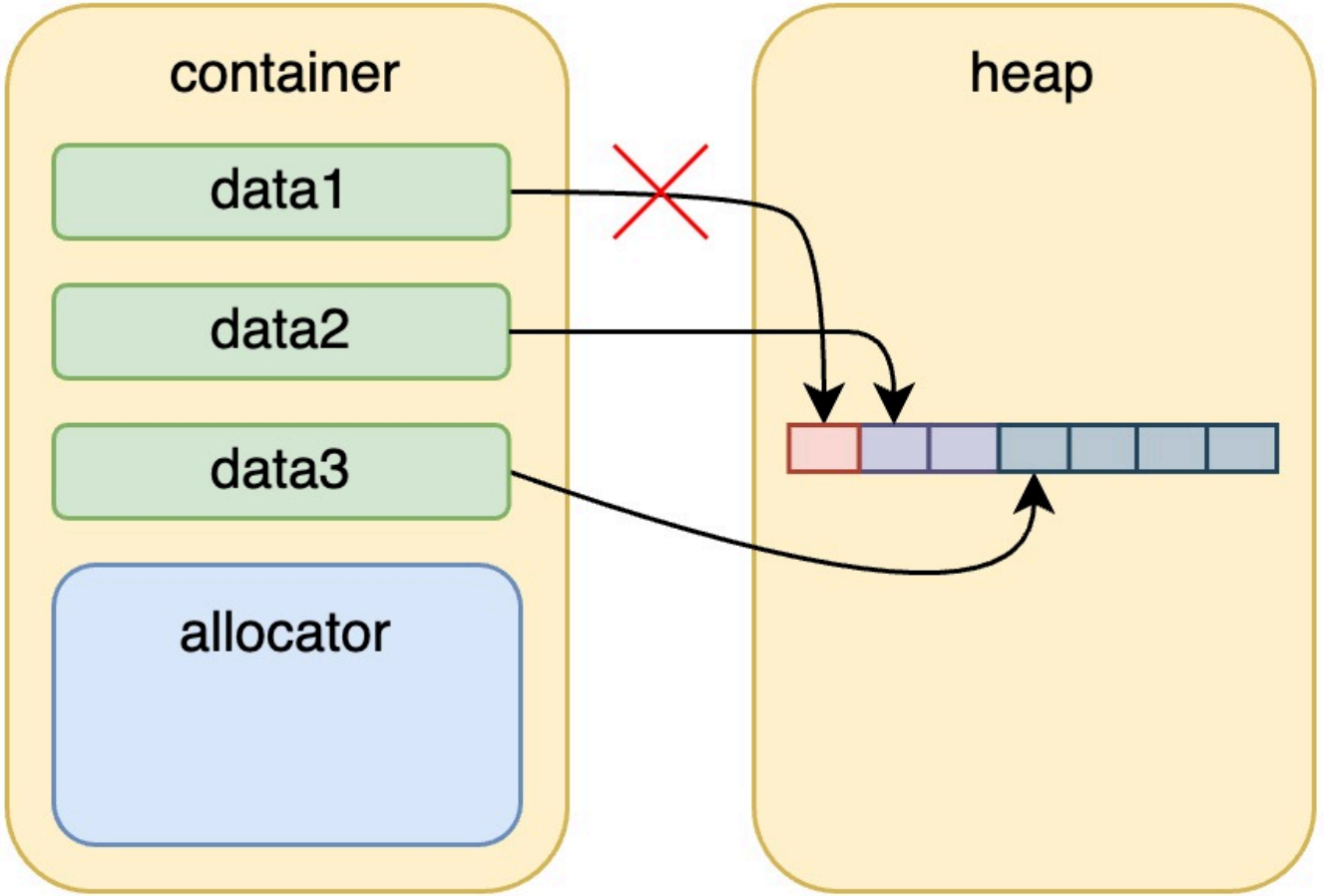
Non-Trivial Buffer Size: Example 2

```
container(init_type init)
{
...
    data1 = a.allocate(1);
    data2 = a.allocate(2);
    data3 = a.allocate(4);

...
}
```



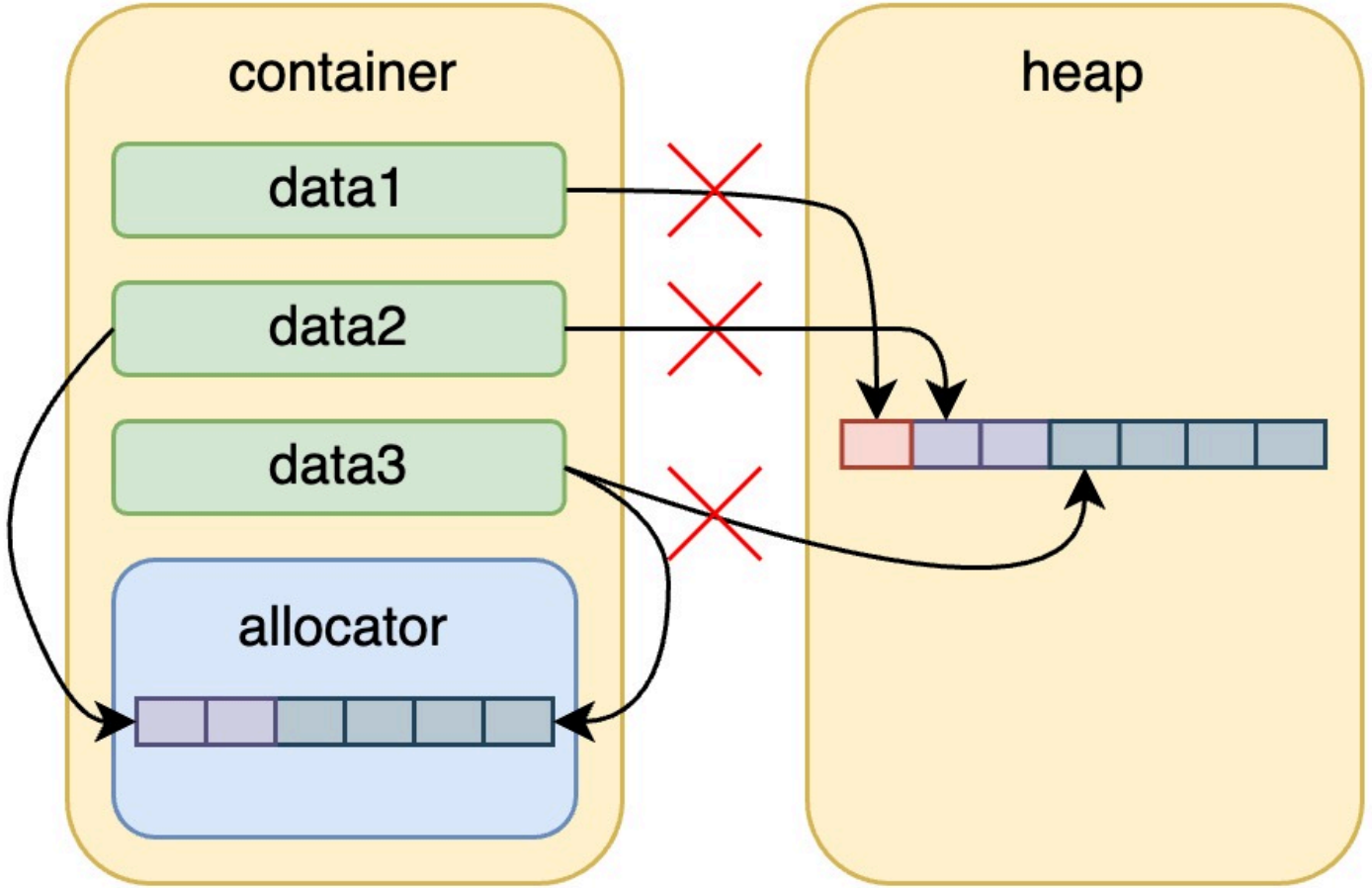
Non-Trivial Buffer Size: Example 2

```
container(init_type init)
{
...
    data1 = a.allocate(1);
    data2 = a.allocate(2);
    data3 = a.allocate(4);
    a.deallocate(data1, 1);
...
}
```



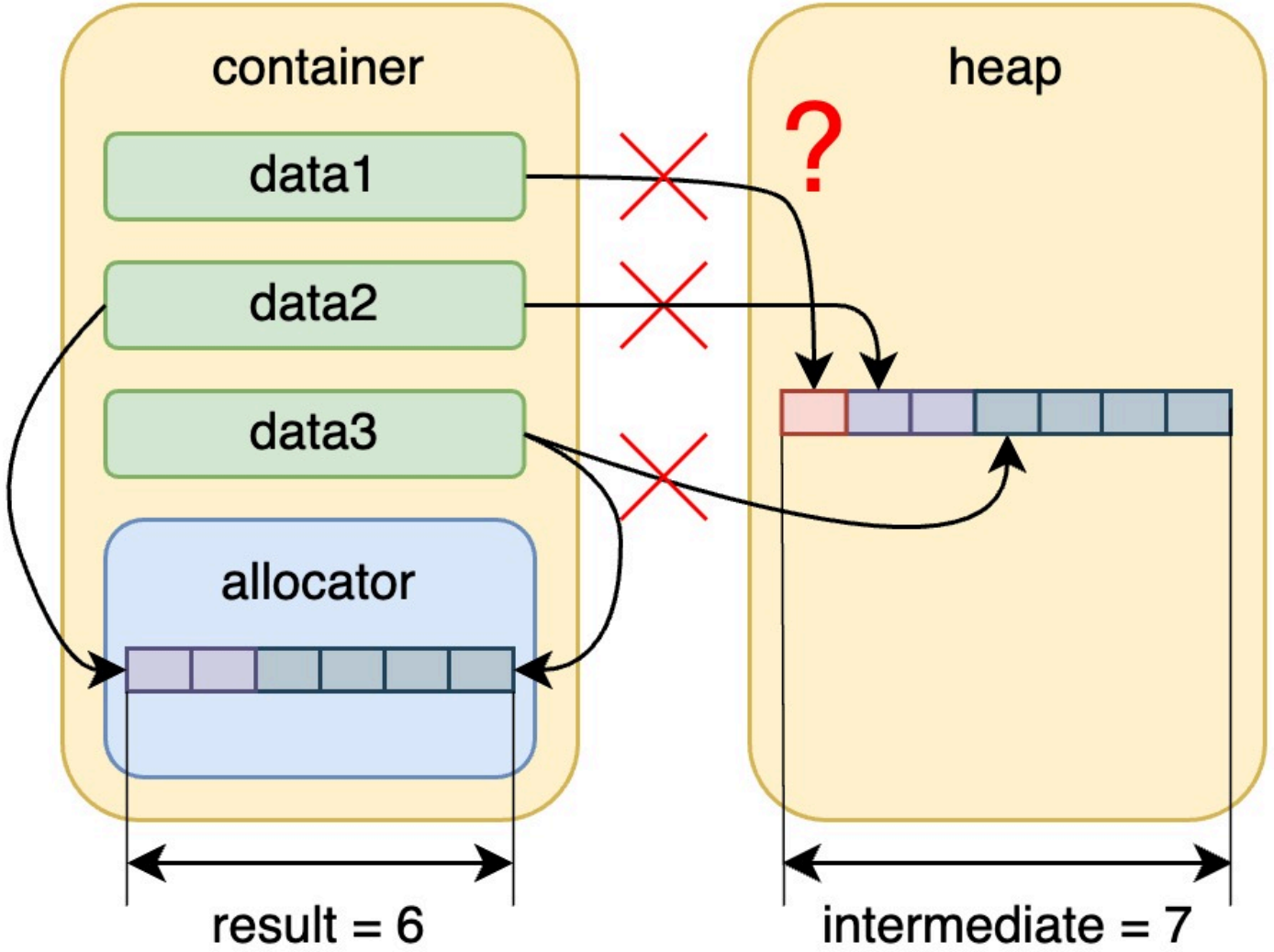
Non-Trivial Buffer Size: Example 2

```
container(init_type init)
{
...
    data1 = a.allocate(1);
    data2 = a.allocate(2);
    data3 = a.allocate(4);
    a.deallocate(data1, 1);
...
}
```

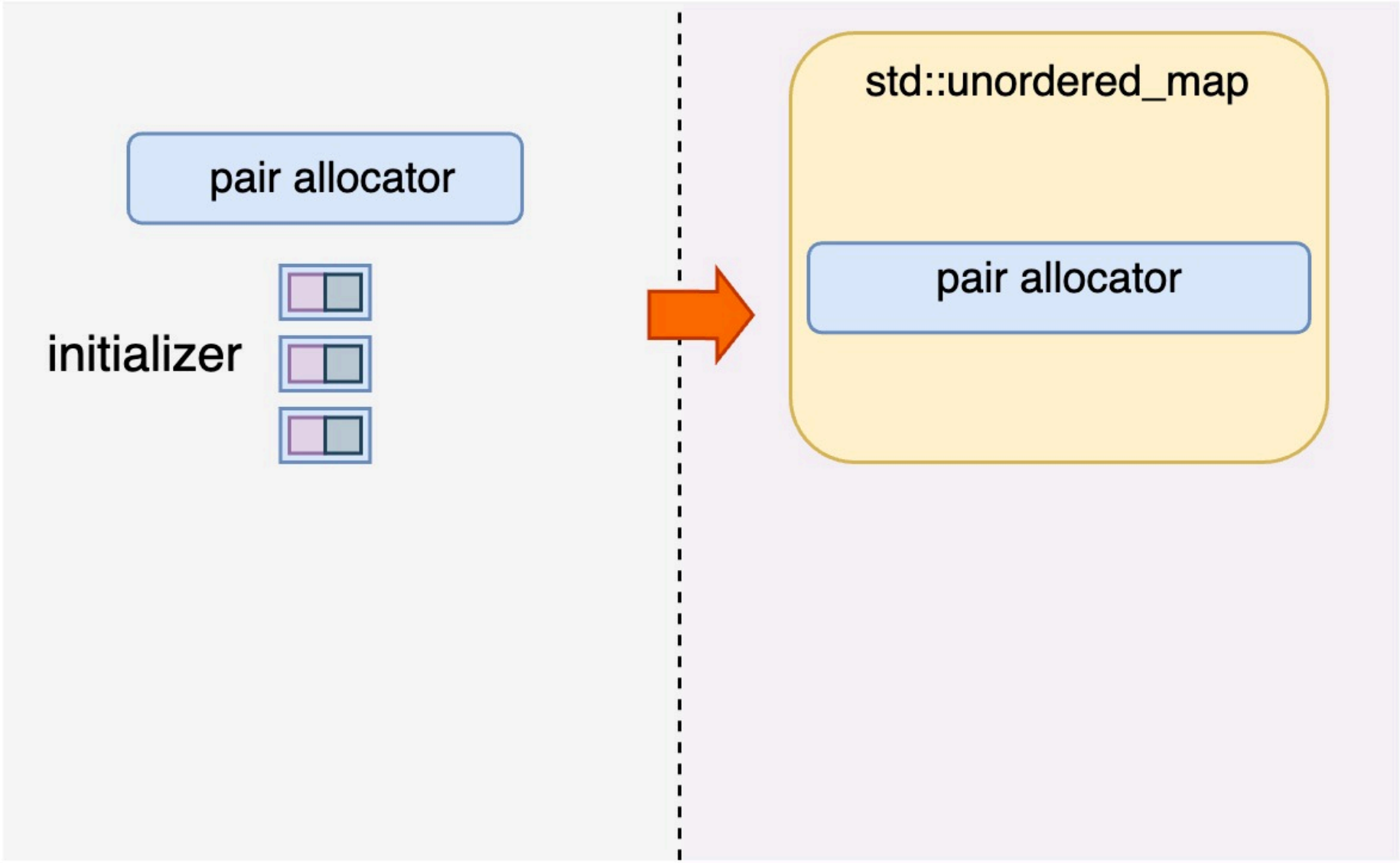


Non-Trivial Buffer Size: Example 2

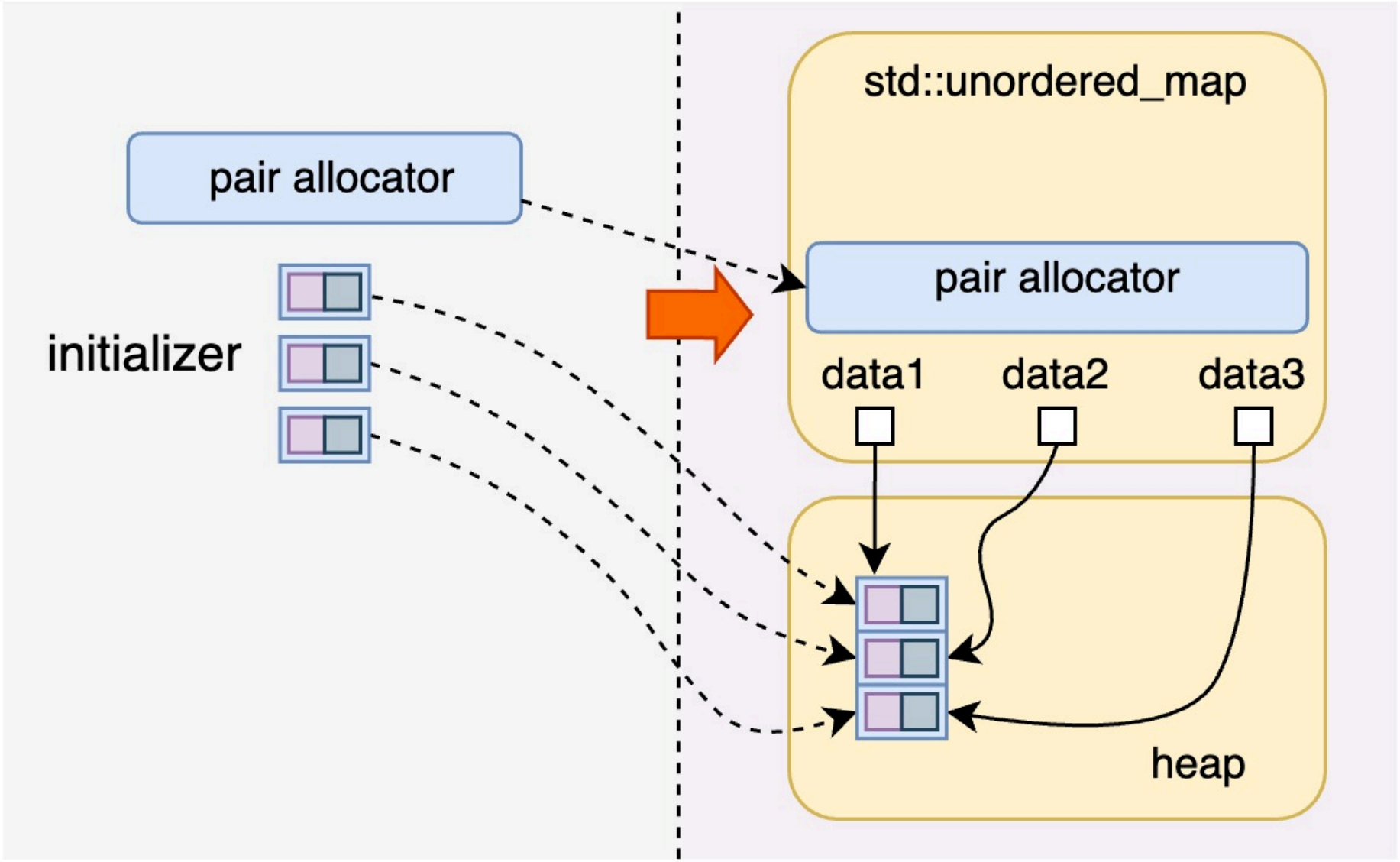
```
container(init_type init)
{
...
    data1 = a.allocate(1);
    data2 = a.allocate(2);
    data3 = a.allocate(4);
    a.deallocate(data1, 1);
...
}
```



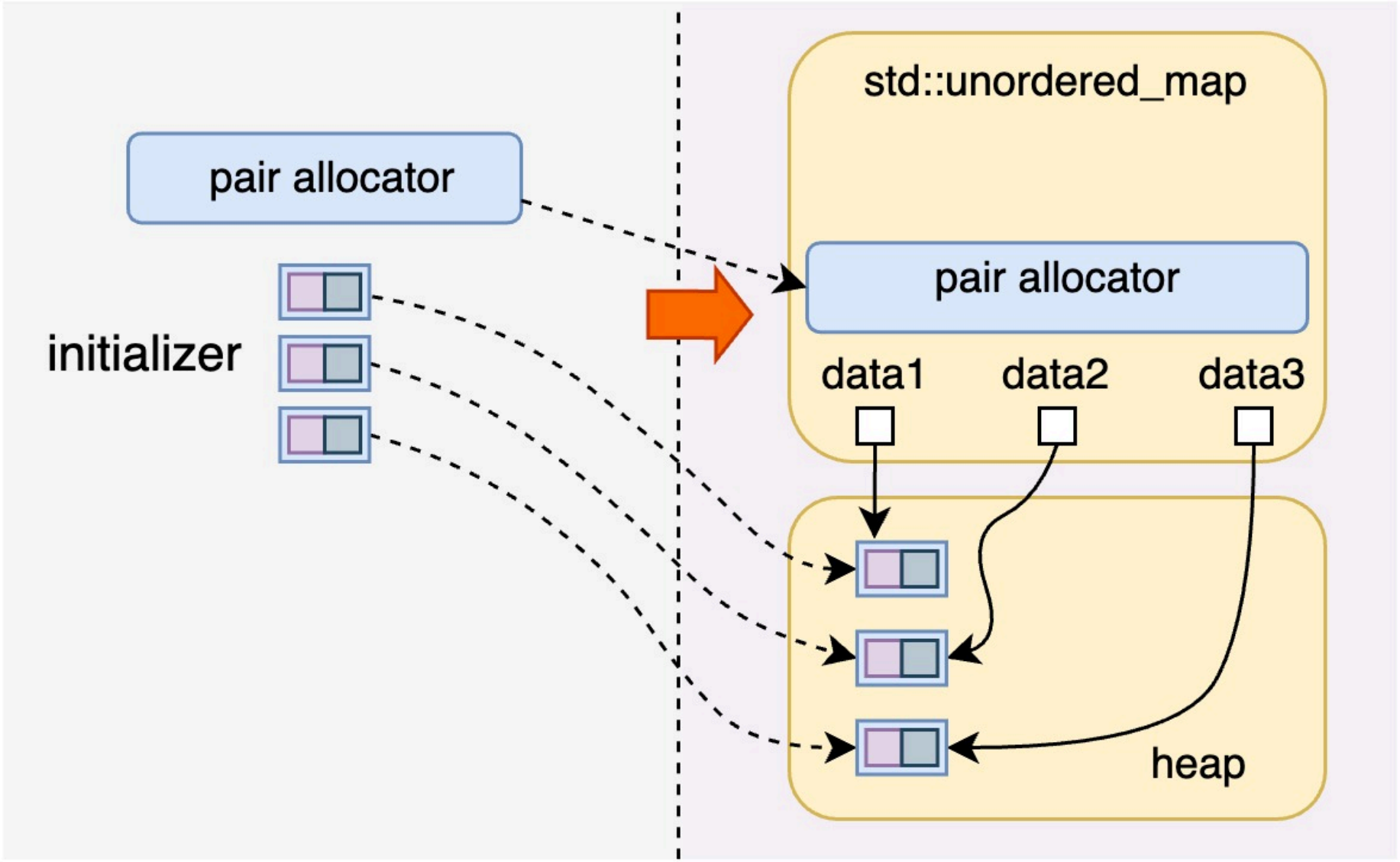
Non-Trivial Buffer Size: Example 3



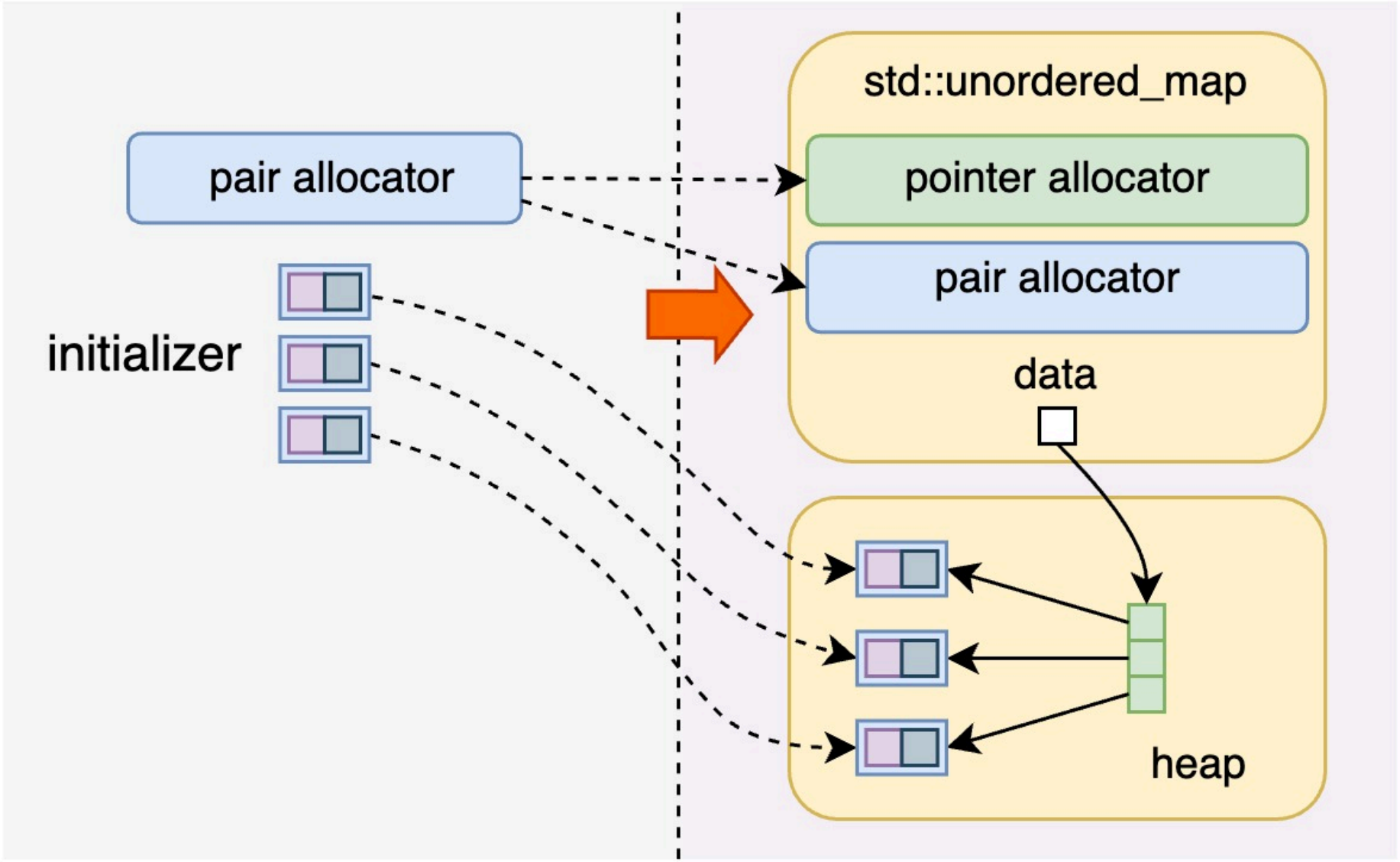
Non-Trivial Buffer Size: Example 3



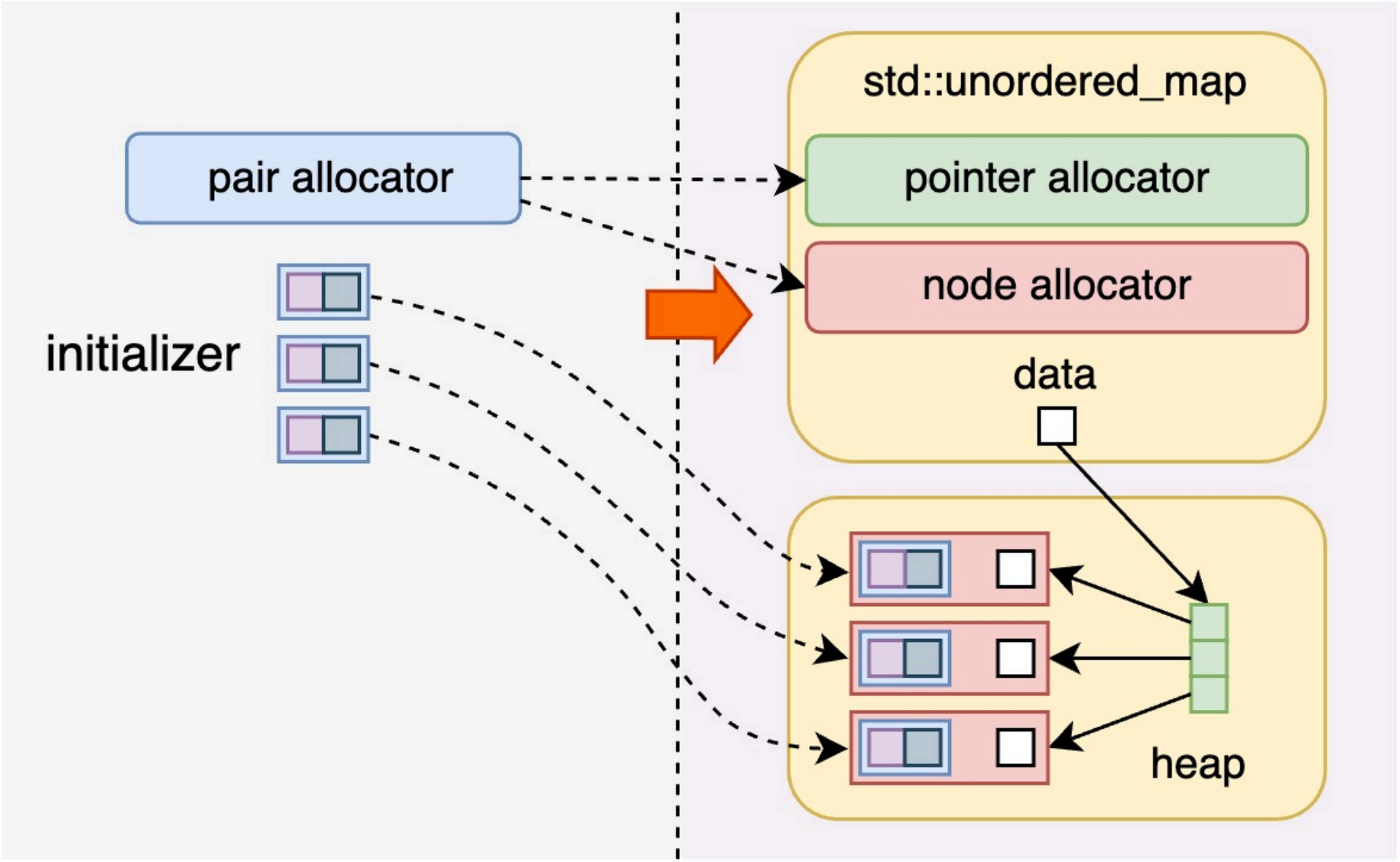
Non-Trivial Buffer Size: Example 3



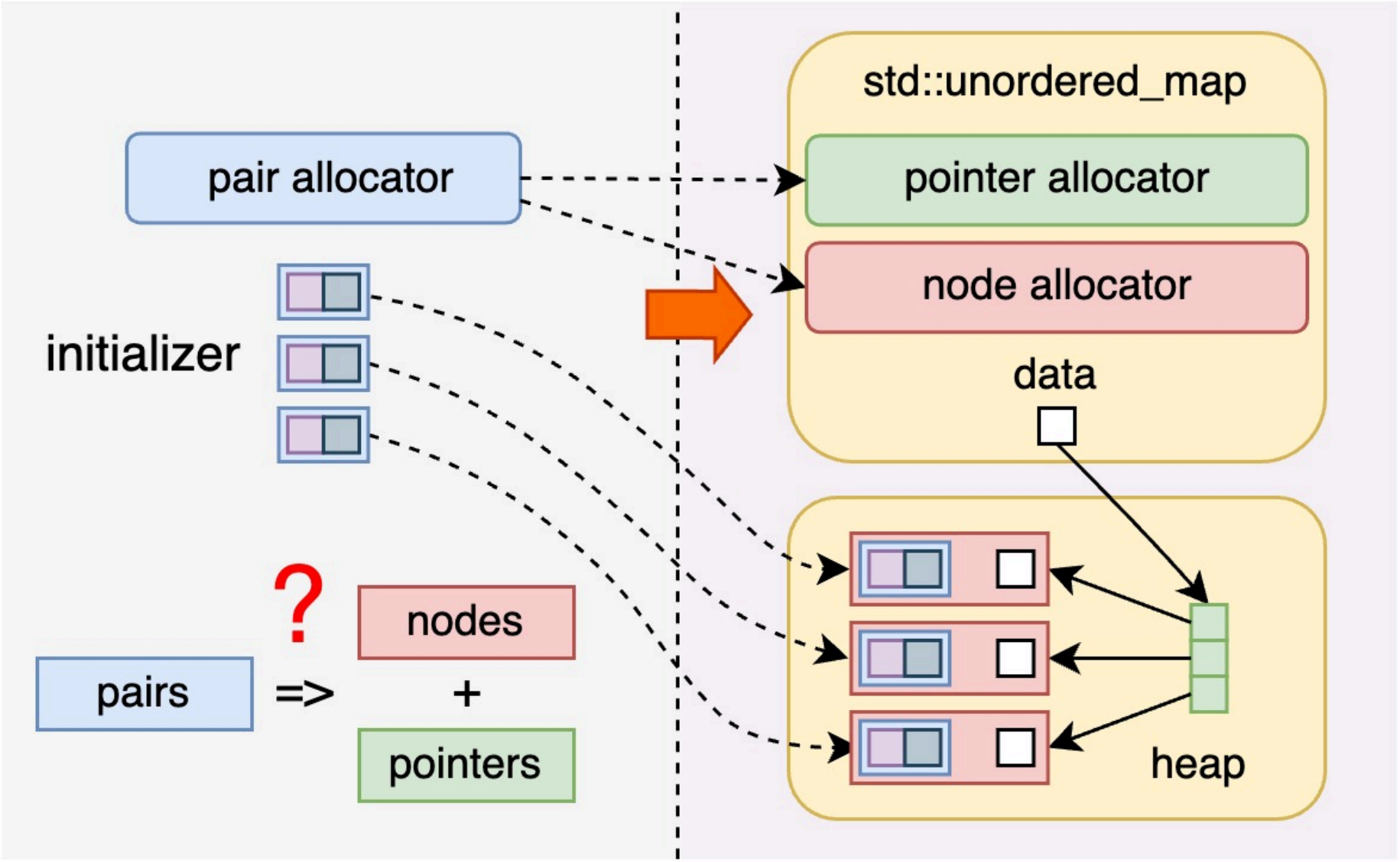
Non-Trivial Buffer Size: Example 3



Non-Trivial Buffer Size: Example 3



Non-Trivial Buffer Size: Example 3



Analysis: conclusions

Analysis: conclusions

- **Where to store** container's elements?

Analysis: conclusions

- **Where to store** container's elements?
  - in a *local buffer*

Analysis: Conclusions

- **Where to store** container's elements?

  - in a *local buffer*
- **Determining of buffer** size in the general case - **is not trivial**

Analysis: Conclusions

- **Where to store** container's elements?
  - in a *local buffer*.
- **Determining of buffer** size in the general case - **is not trivial**.
- **How to define buffer size automatically?**

Analysis: Conclusions

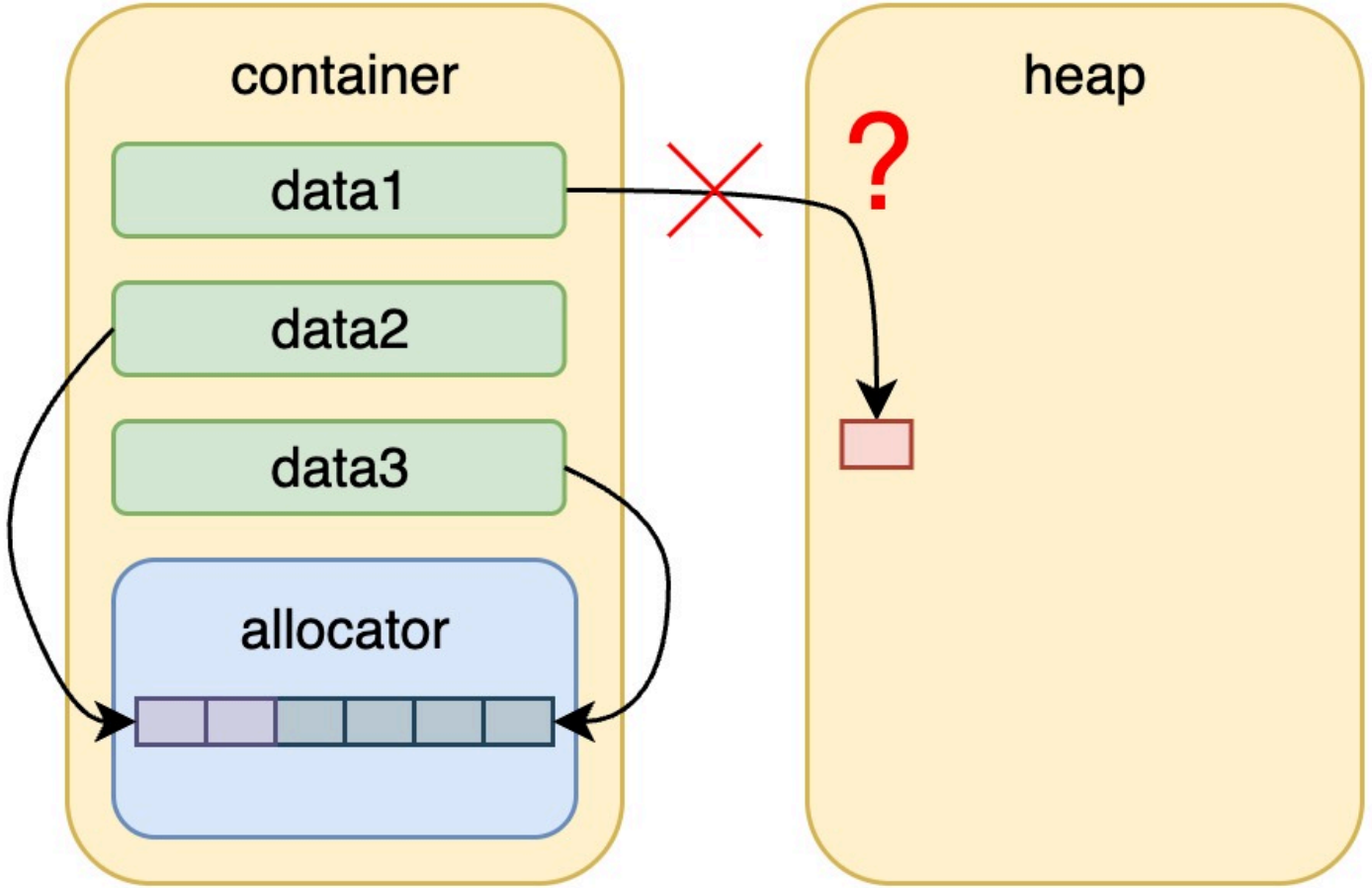
- **Where to store** container's elements?
  - in a *local buffer*.
- **Determining of buffer** size in the general case - **is not trivial**.
- **How to define buffer size automatically?**
  - Create the object in two steps:
    - 1: Define the buffer size.
    - 2: Create the target object using the defined size.

1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 **Common Solution Overview**
   2.2 Implementation Details
4. Additional Applications
5. Results

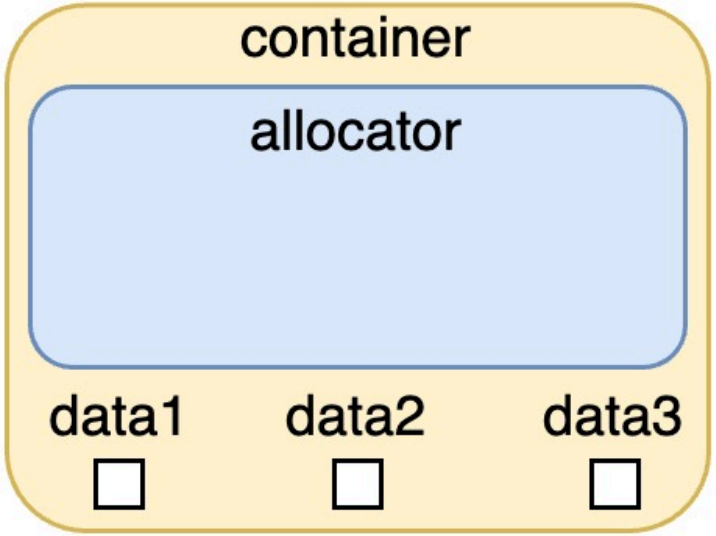
Solution: Common View

Solution: Common View

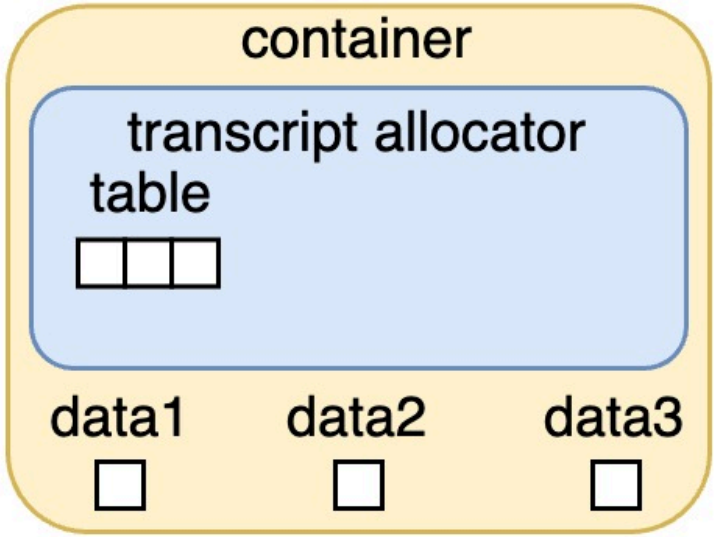
```
container(init_type init)
{
...
    data1 = a.allocate(1);
    data2 = a.allocate(2);
    data3 = a.allocate(4);
    a.deallocate(data1, 1);
...
}
```



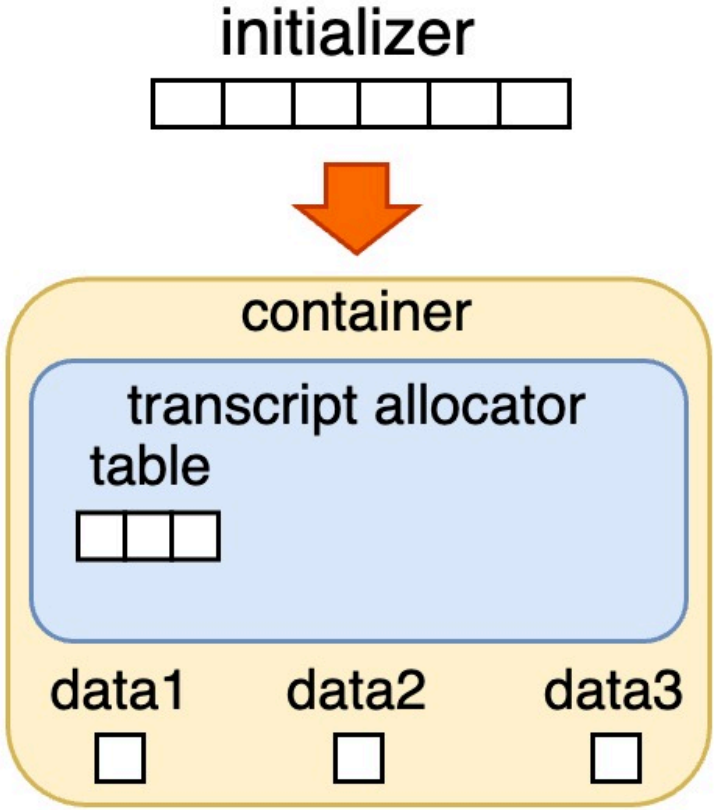
Solution: Common View



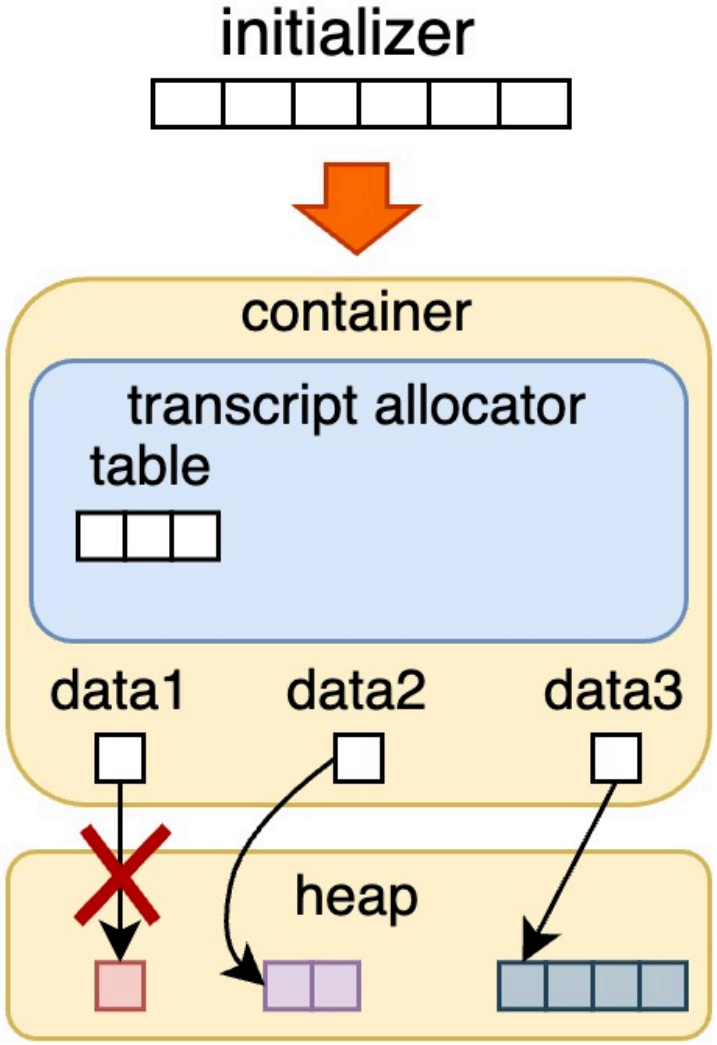
Solution: Common View



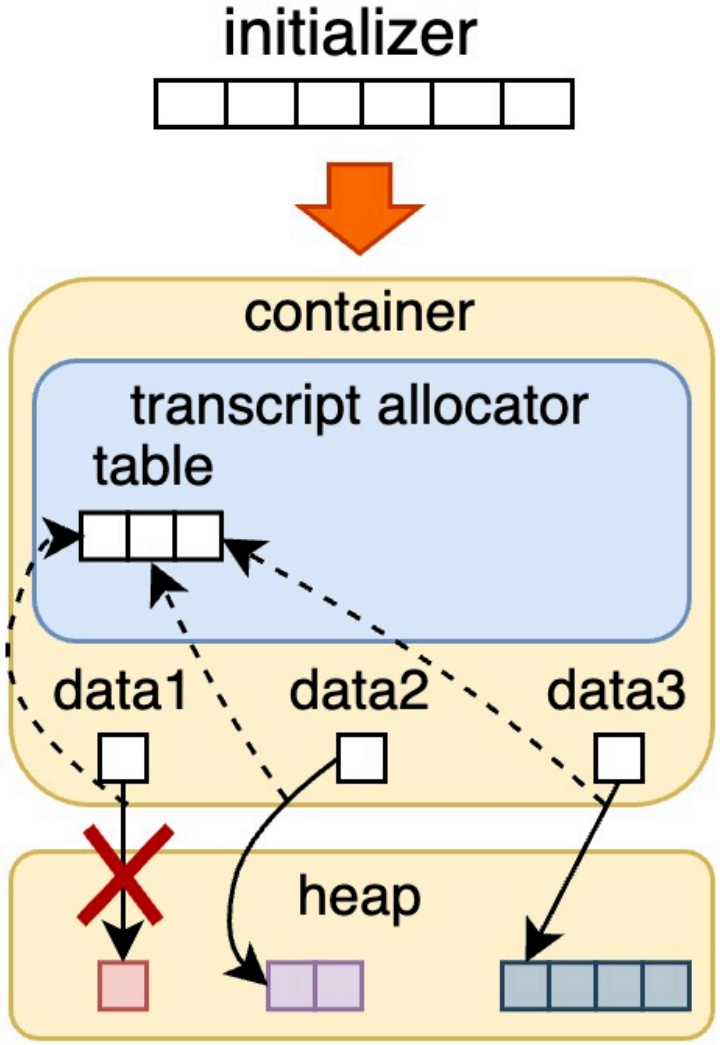
Solution: Common View



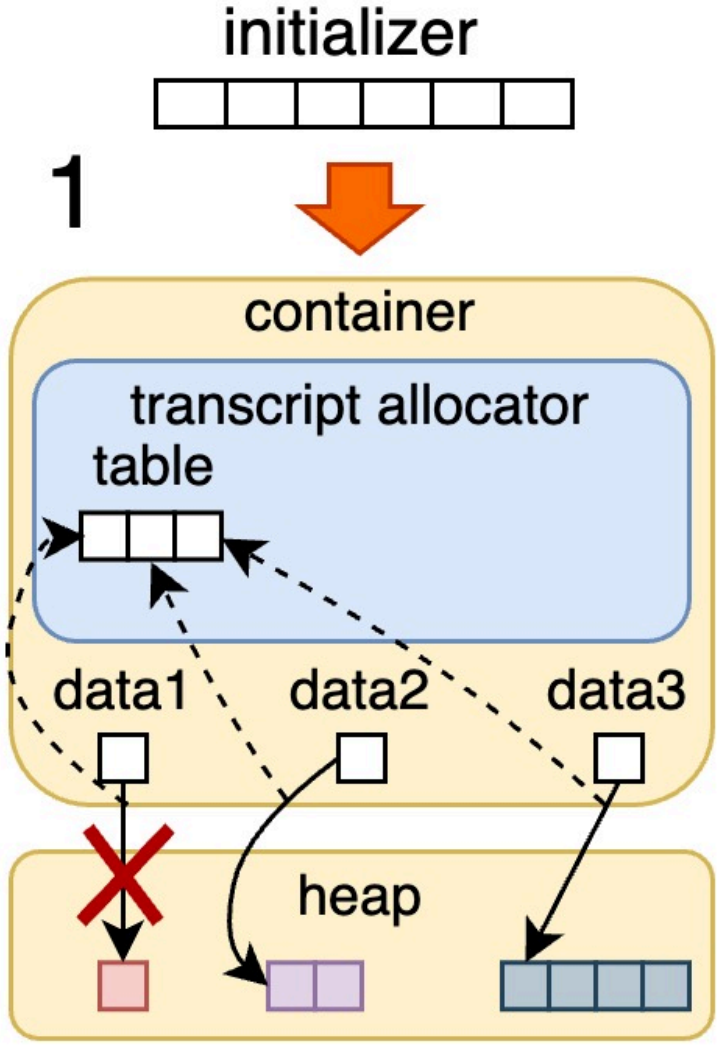
Solution: Common View



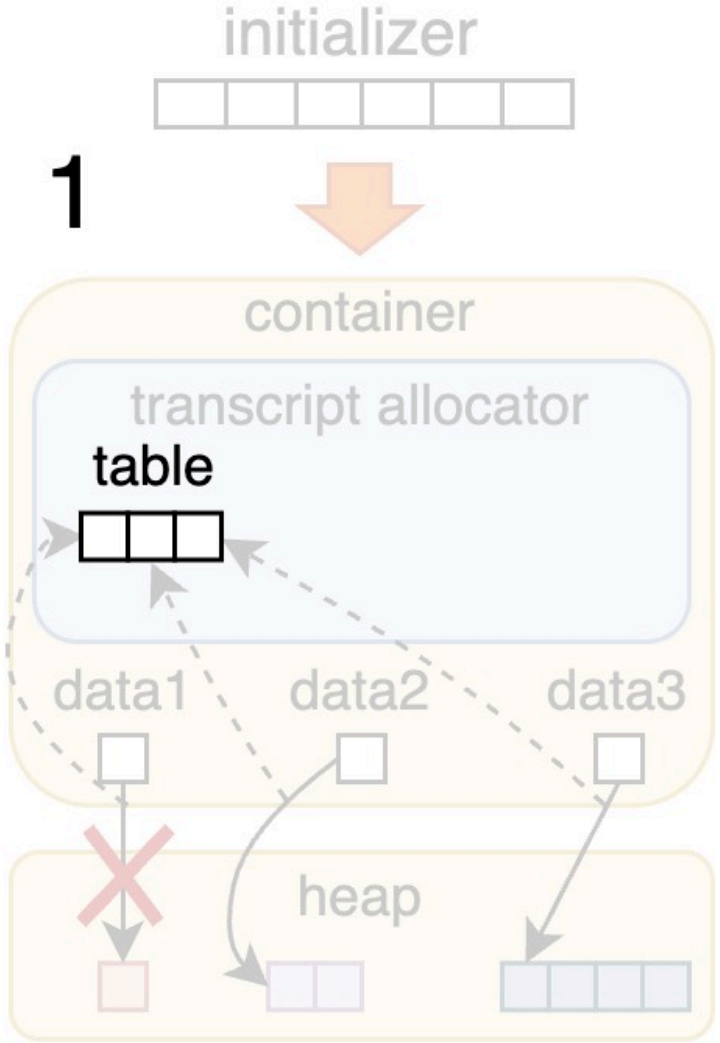
Solution: Common View



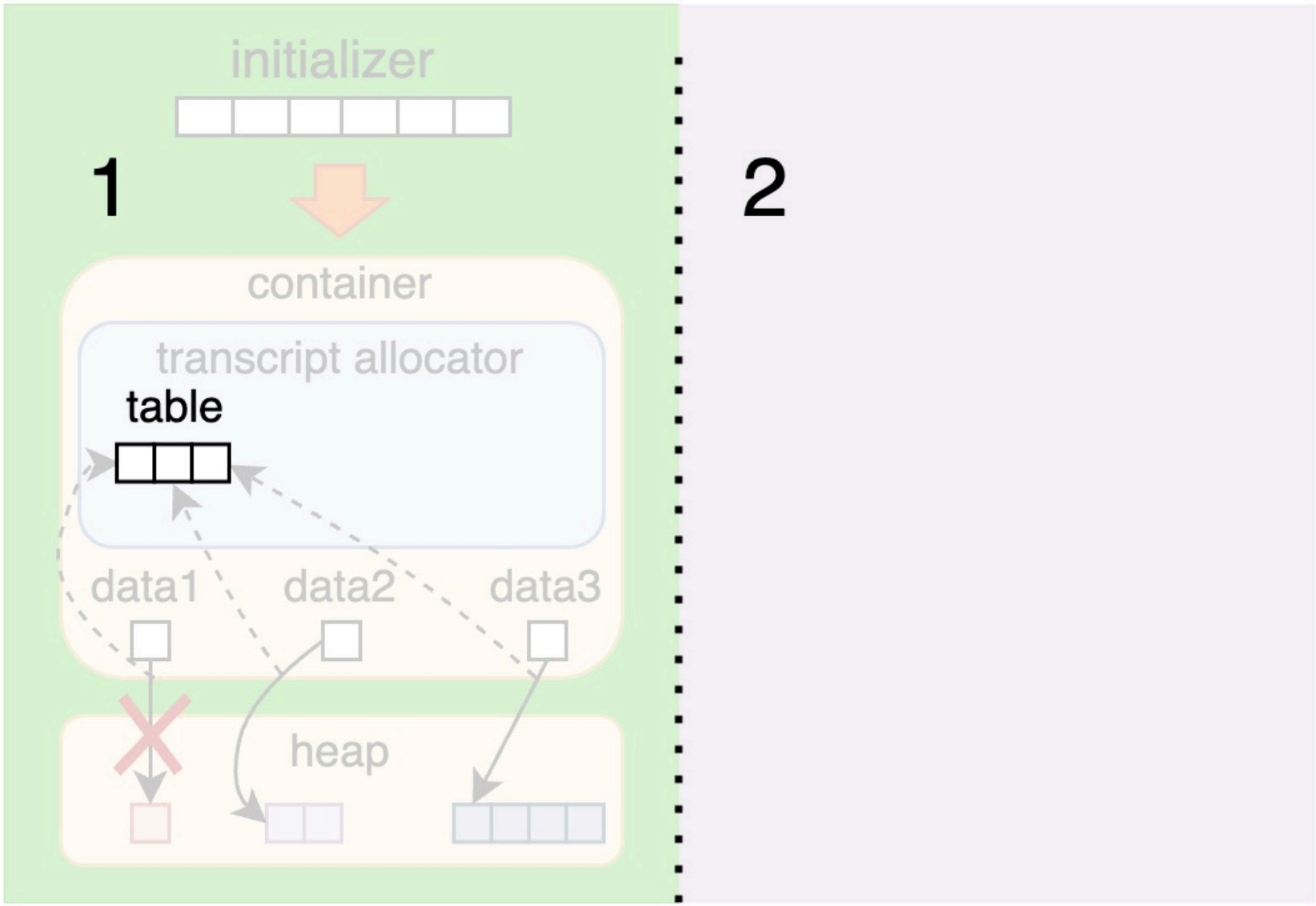
Solution: Common View



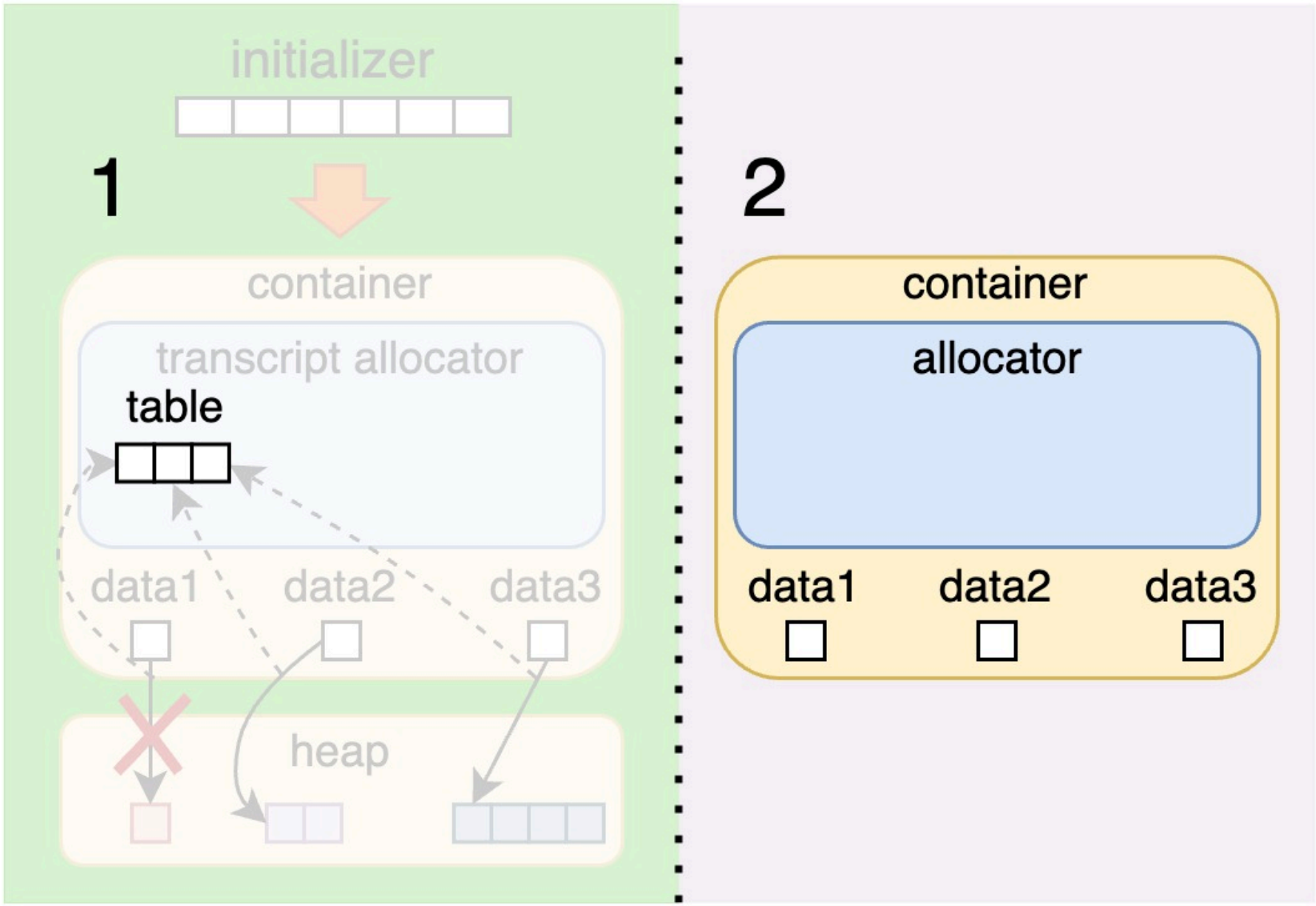
Solution: Common View



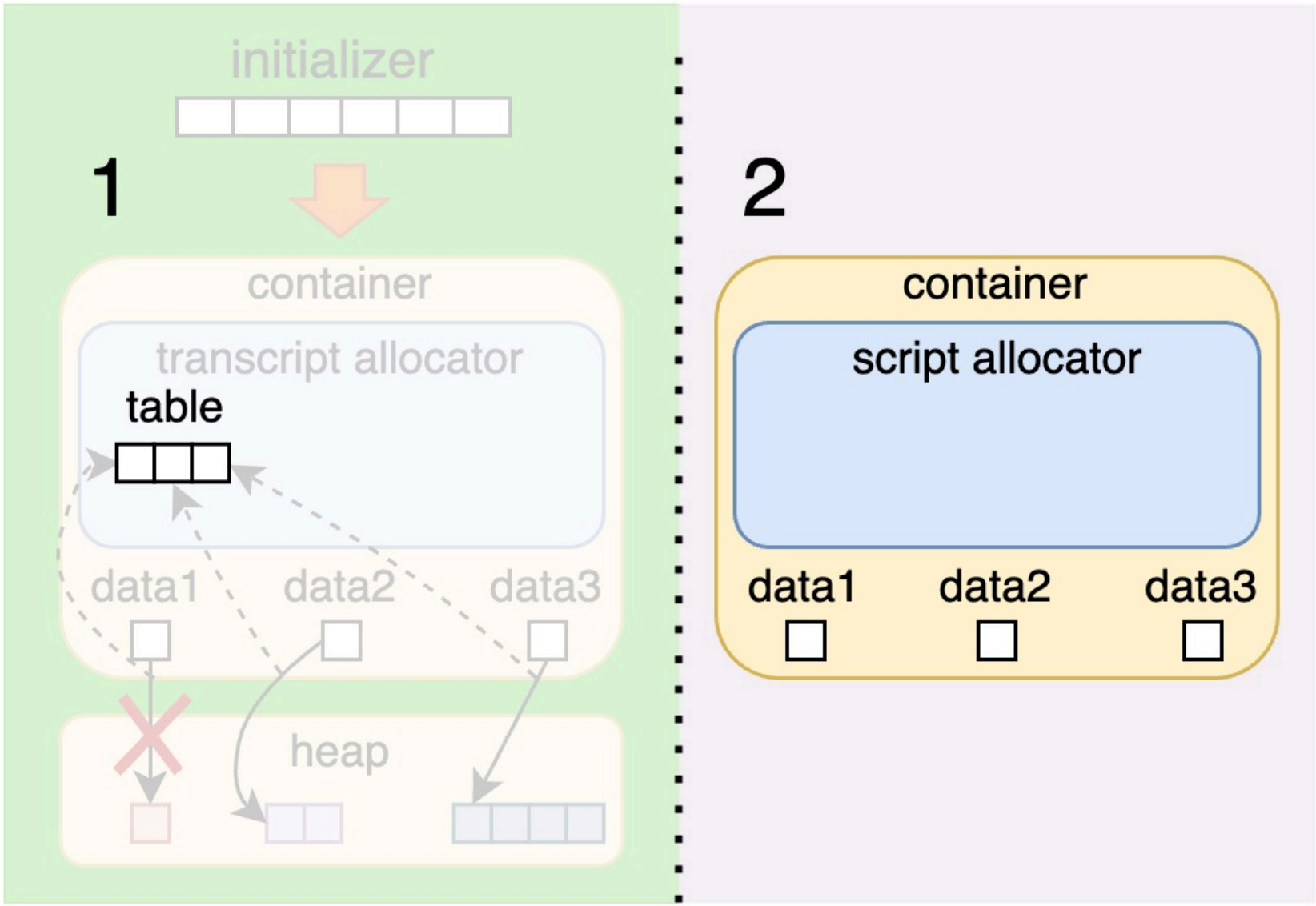
Solution: Common View



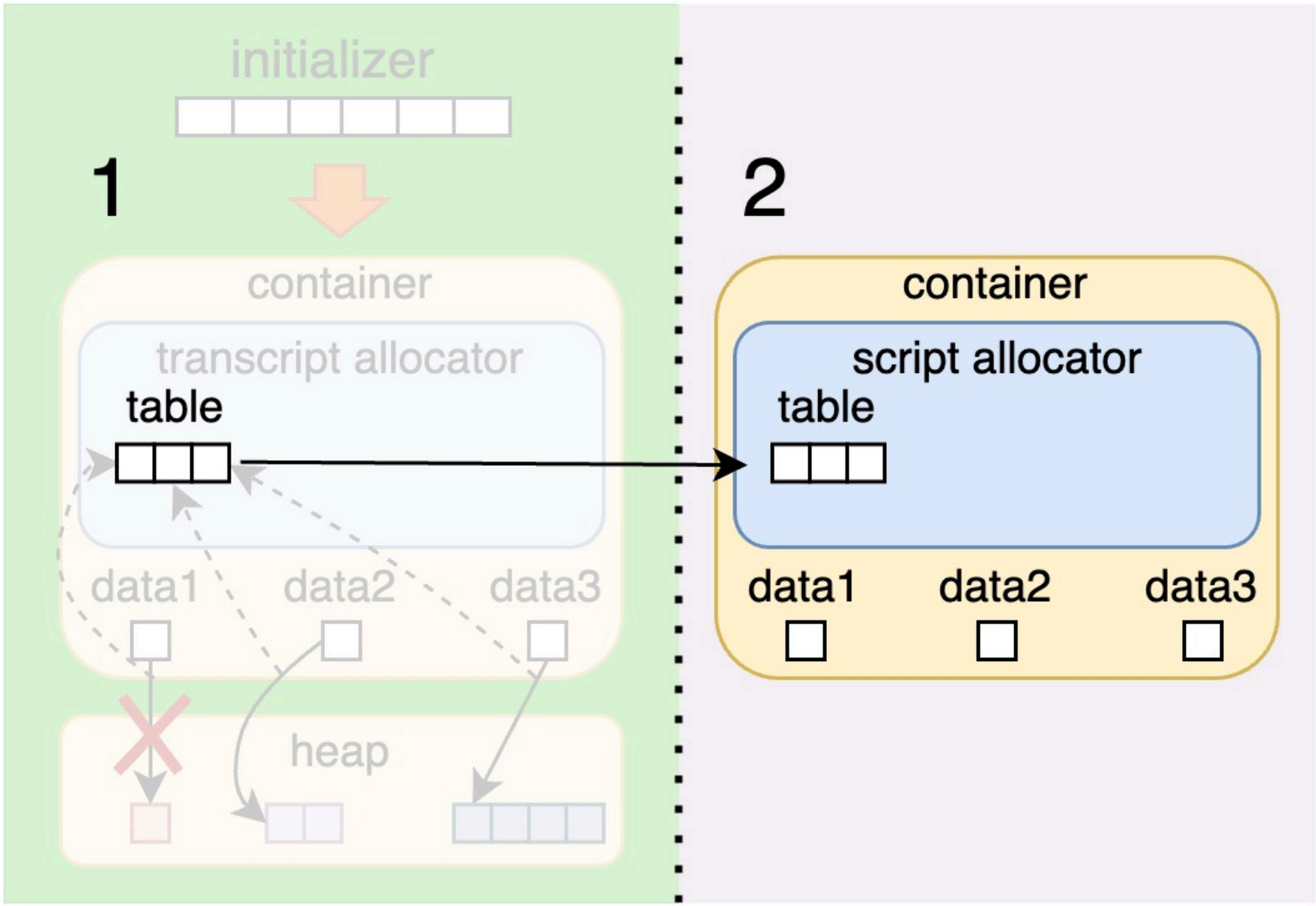
Solution: Common View



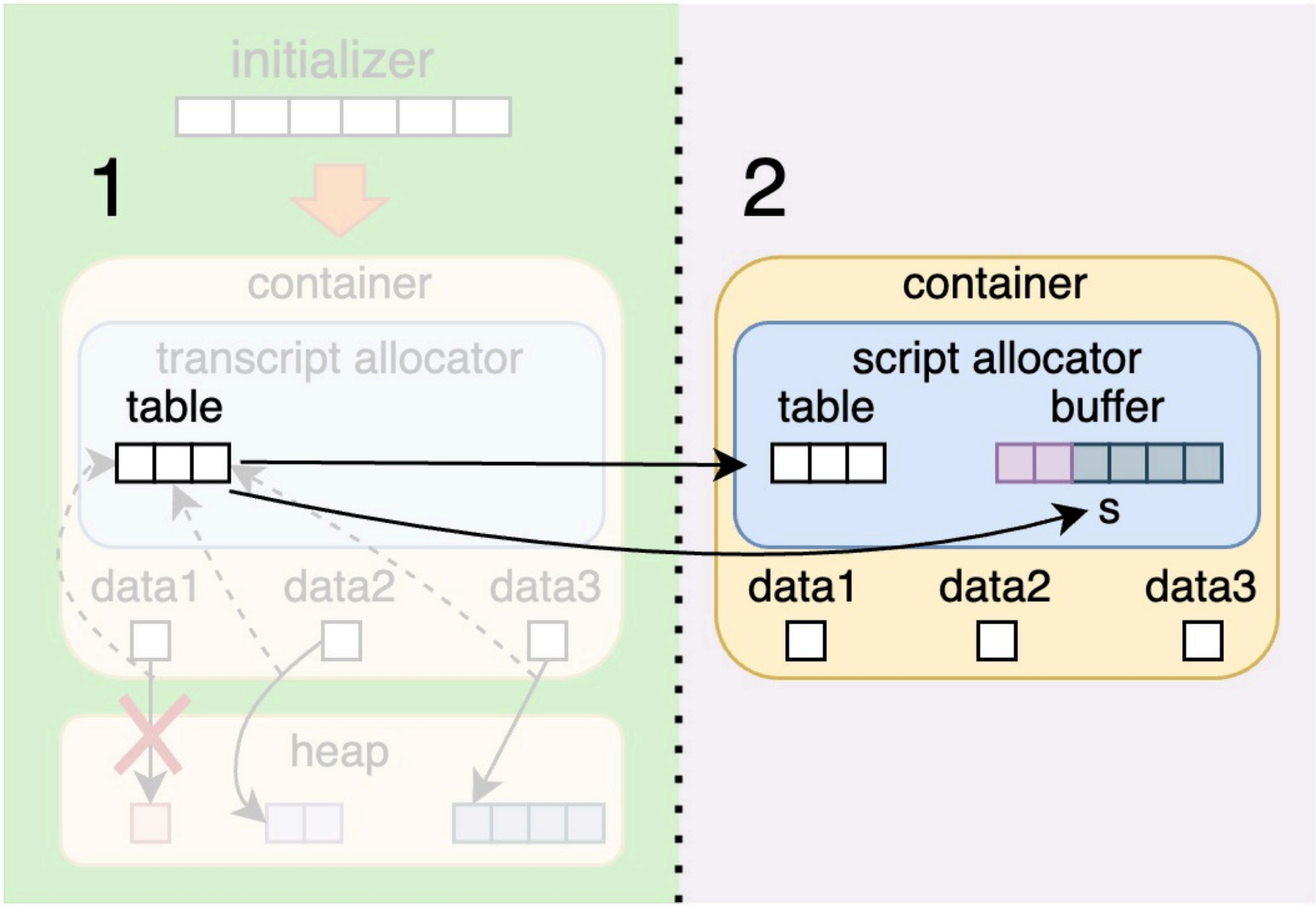
Solution: Common View



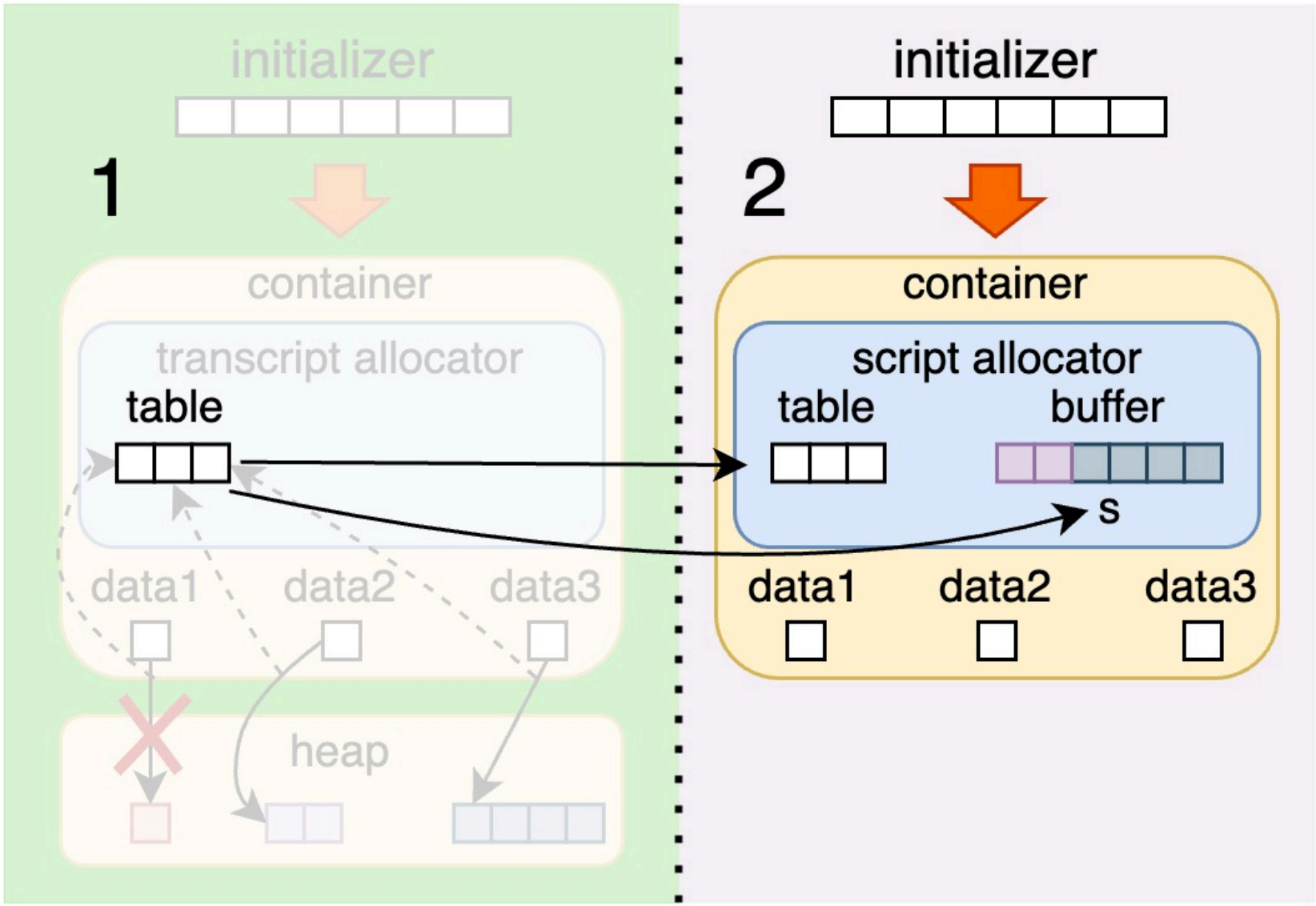
Solution: Common View



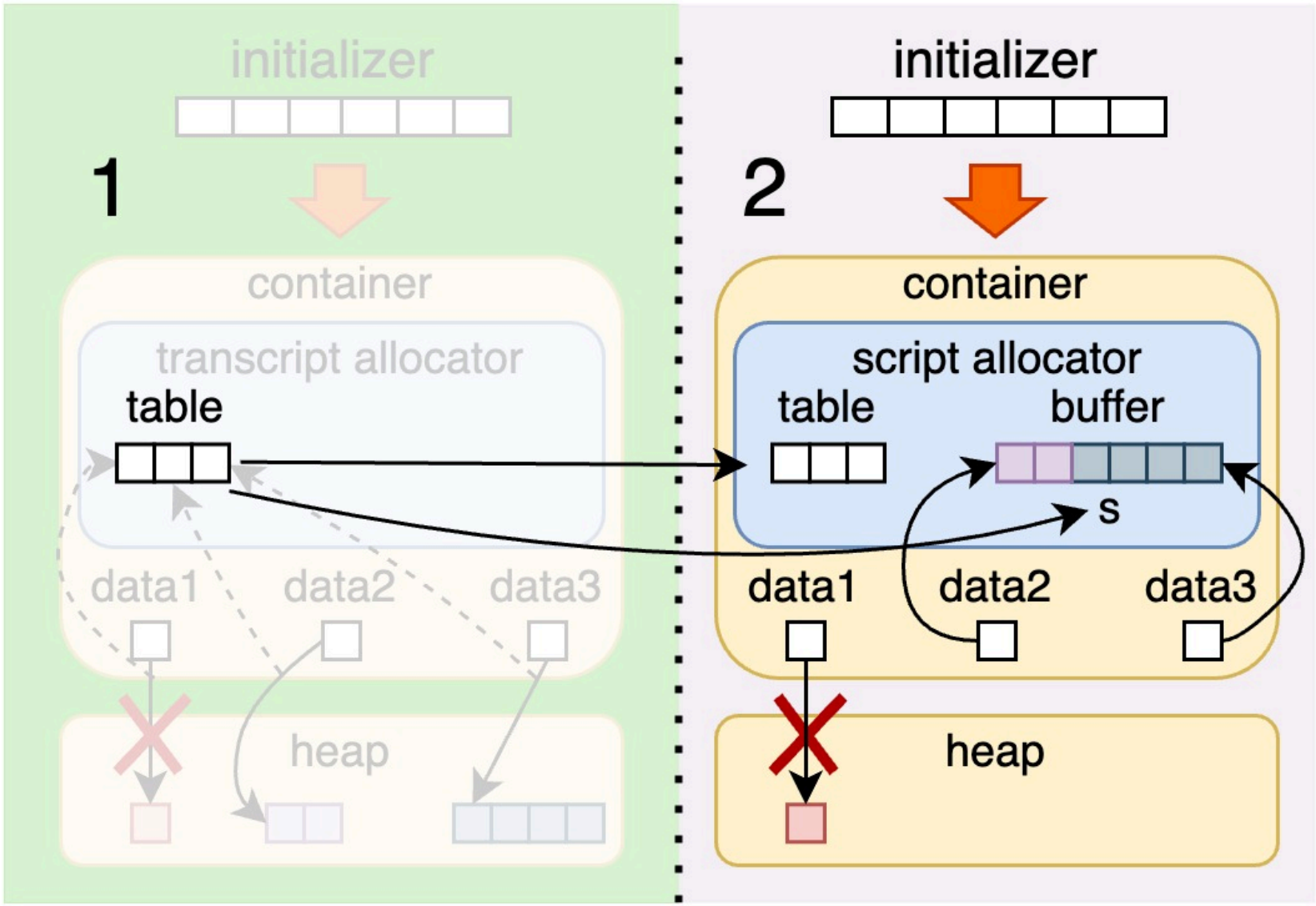
Solution: Common View



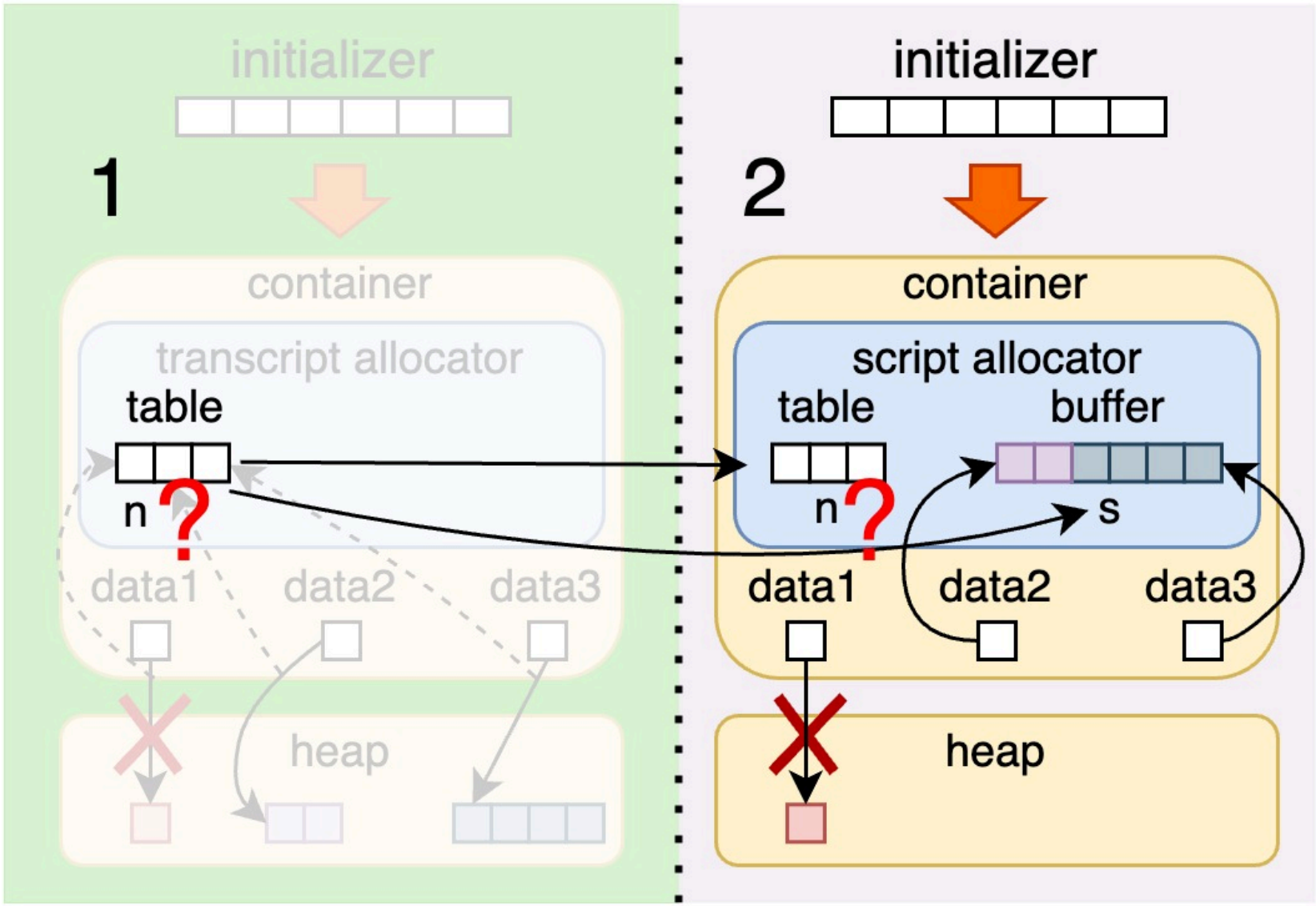
Solution: Common View



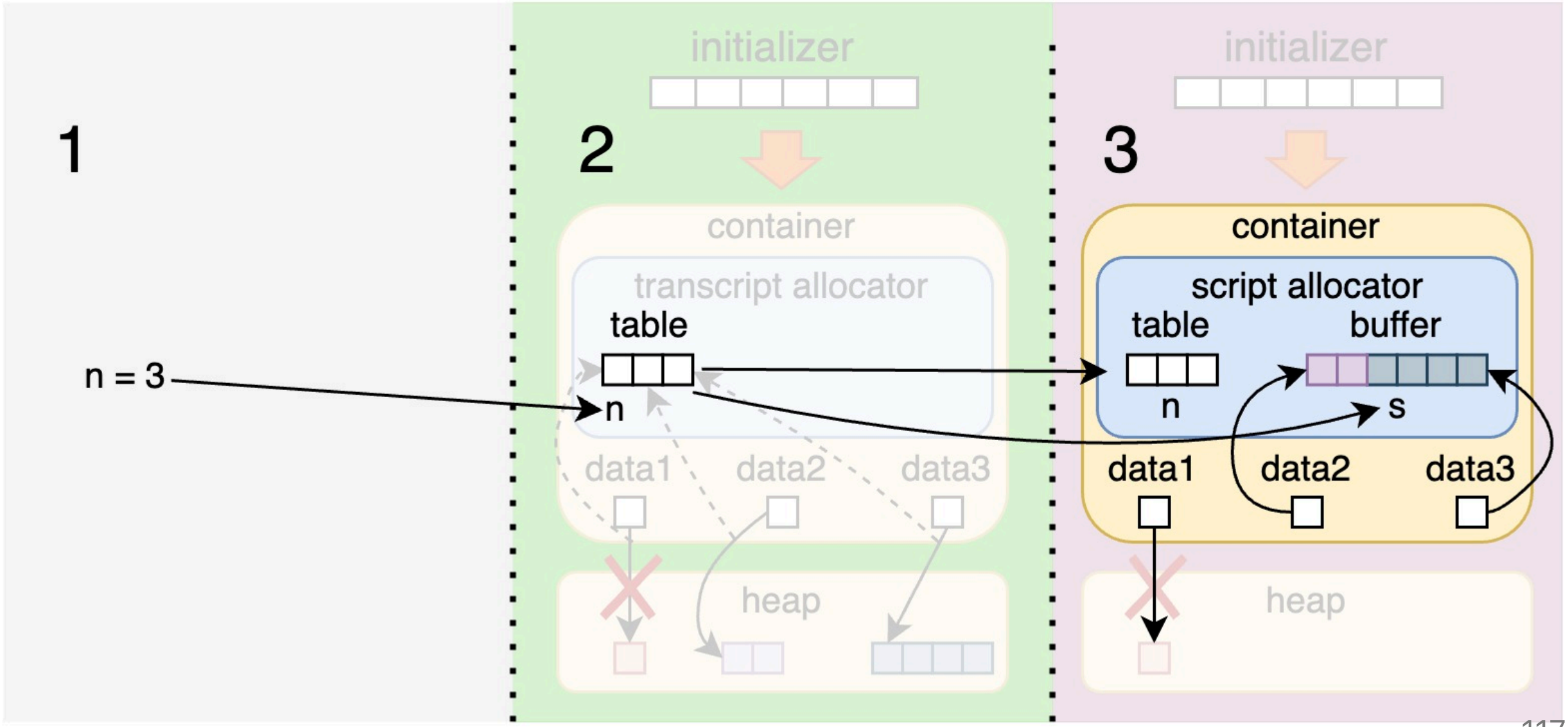
Solution: Common View



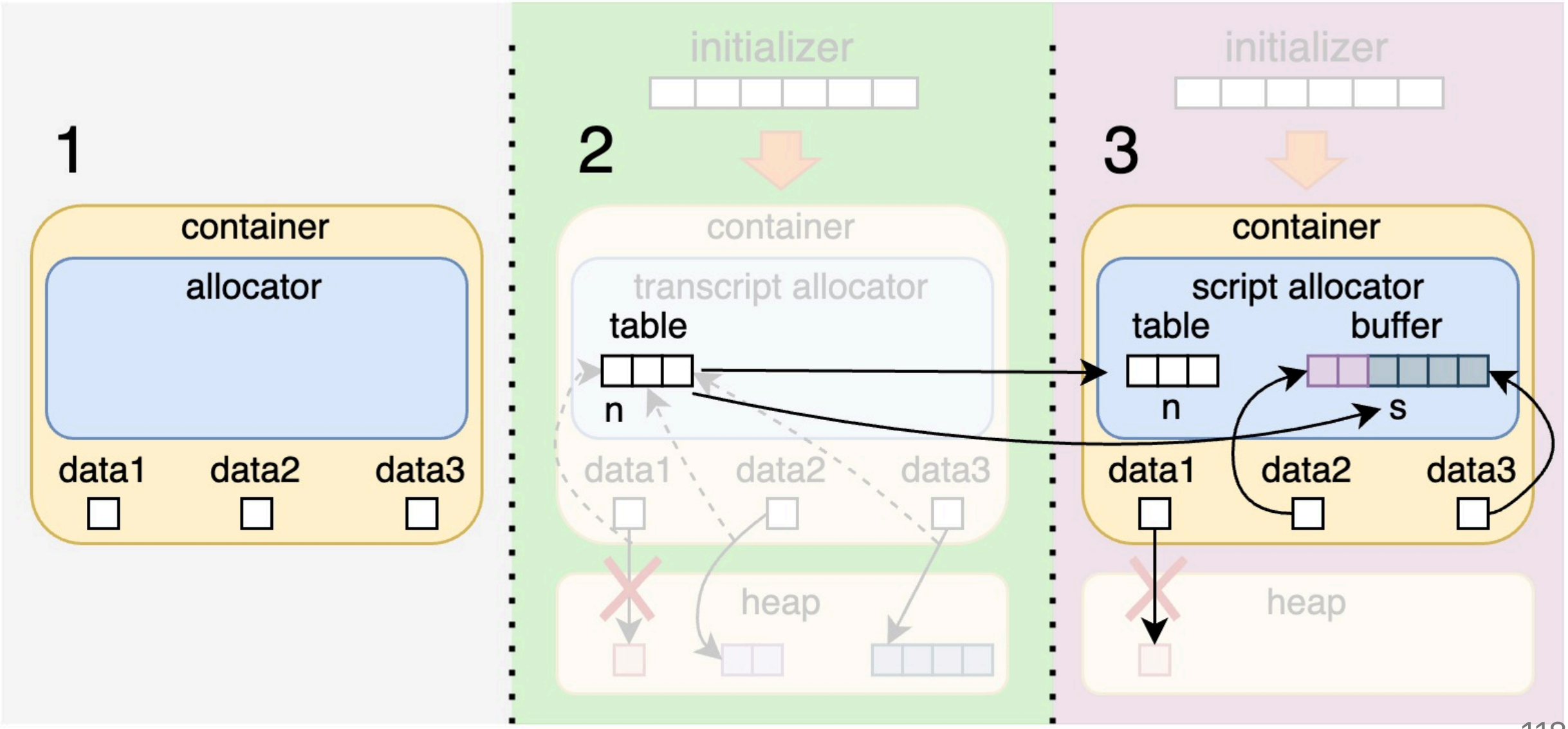
Solution: Common View



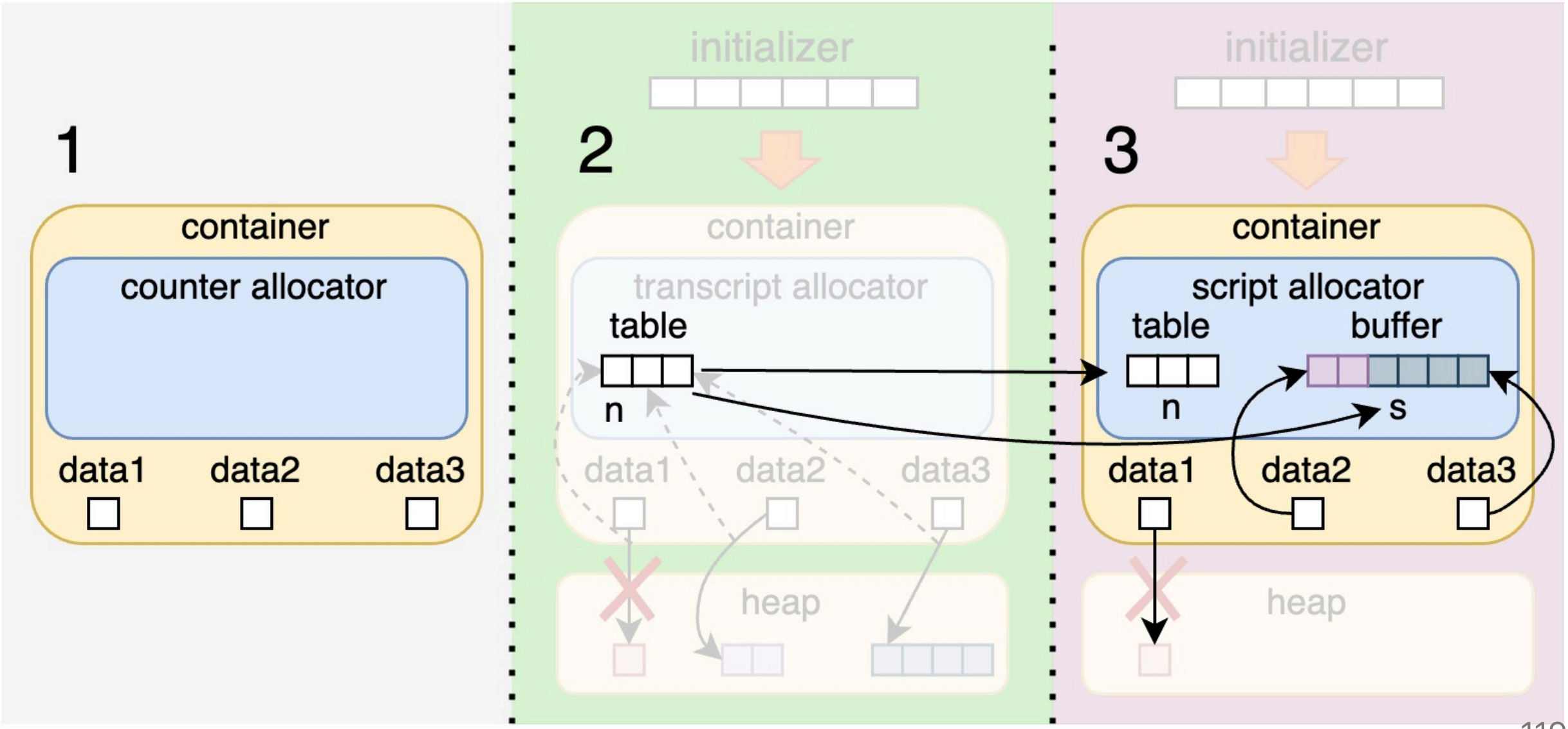
Solution: Common View



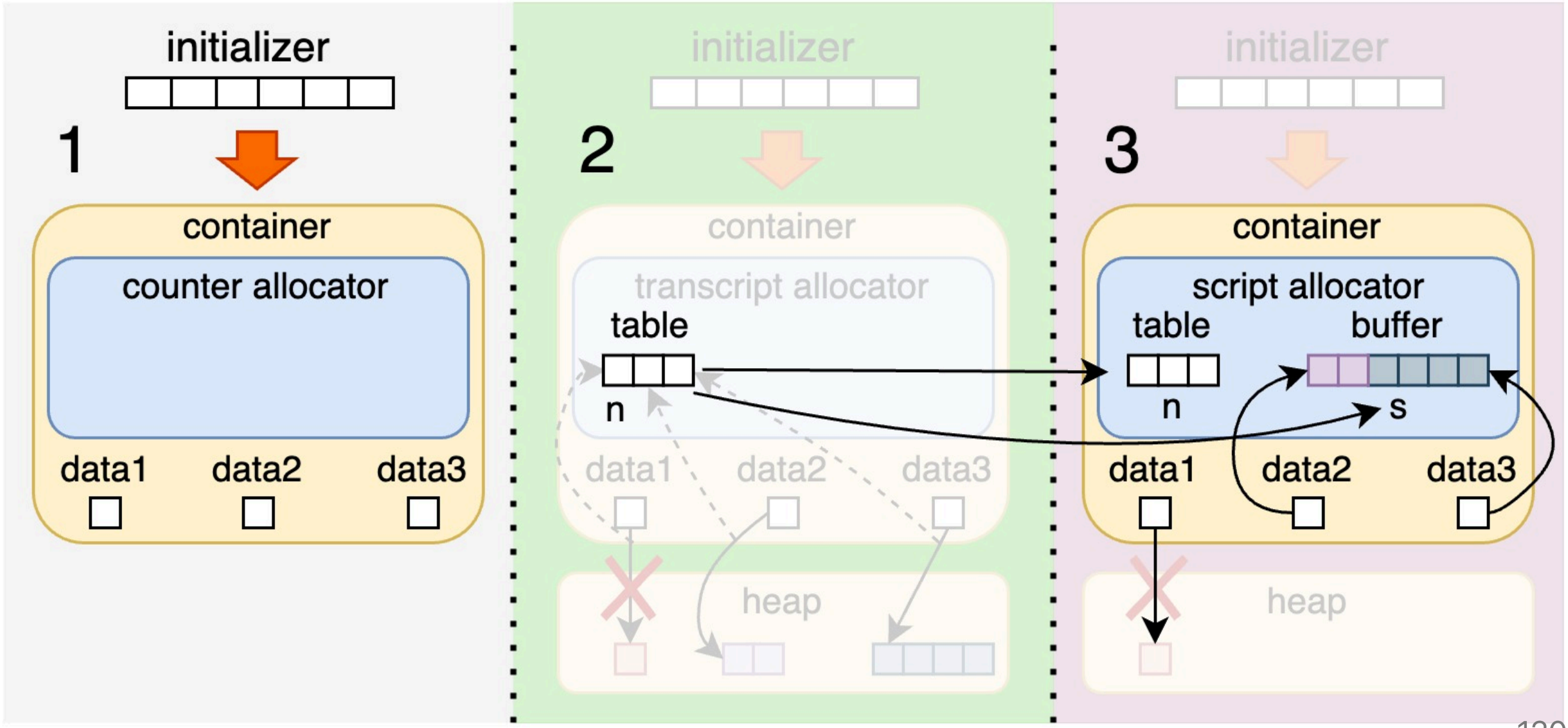
Solution: Common View



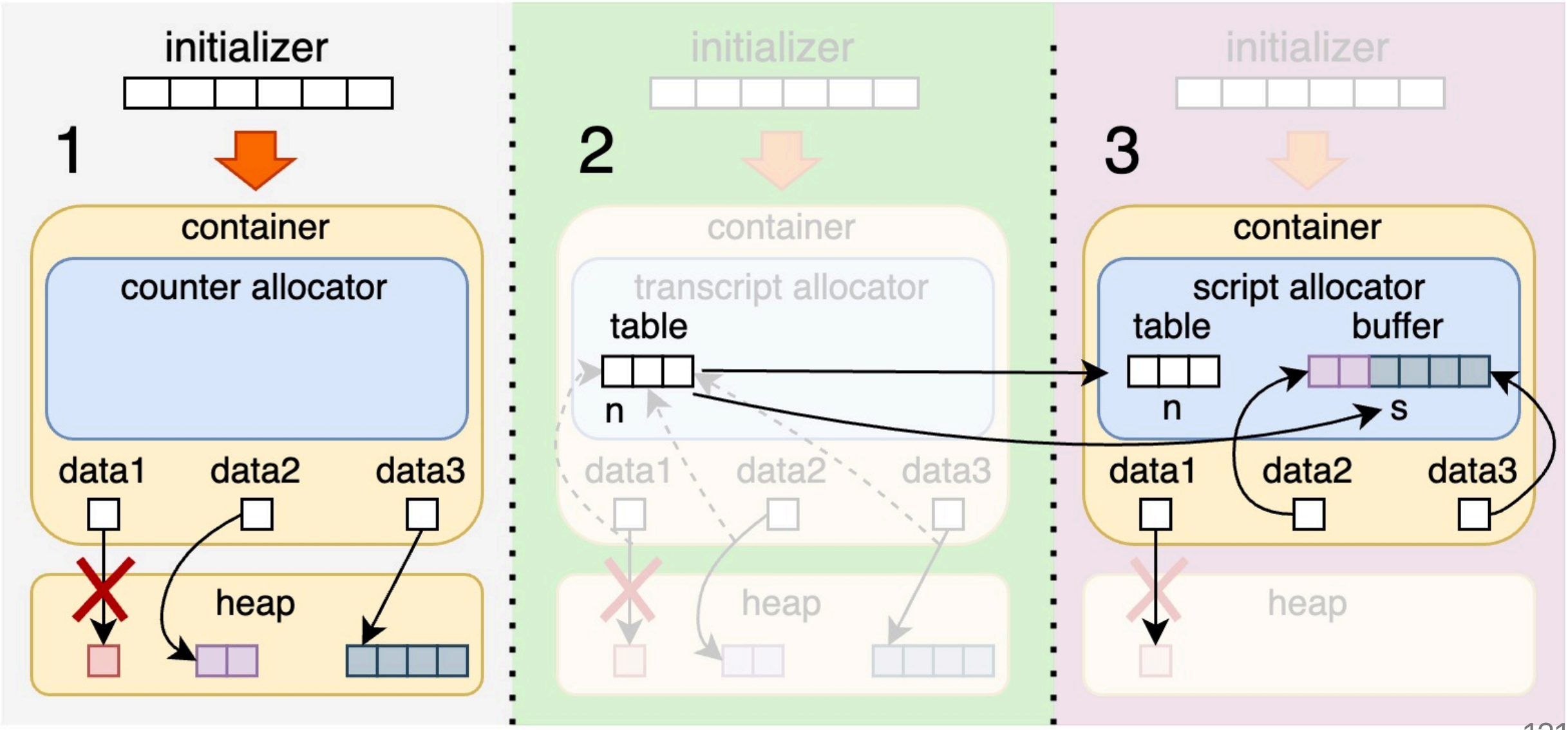
Solution: Common View



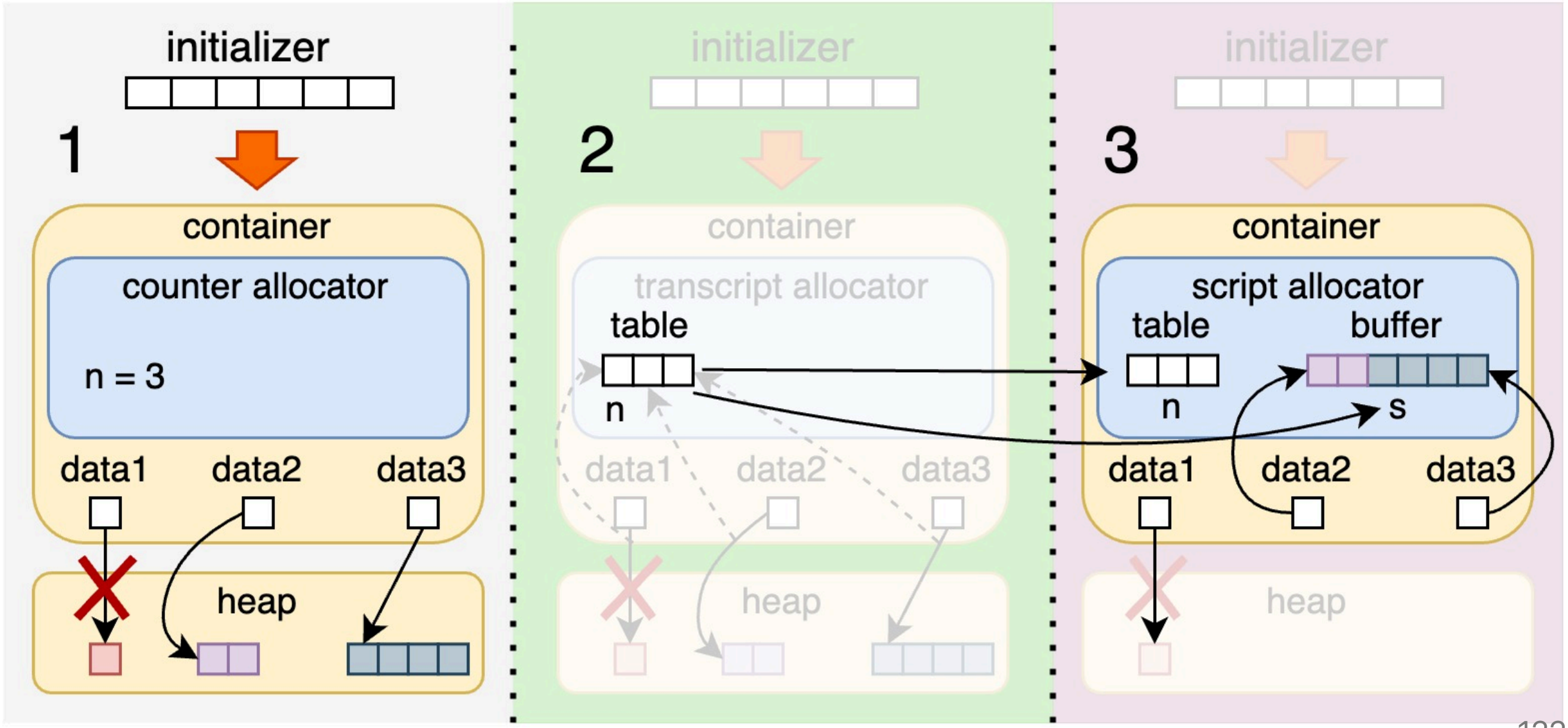
Solution: Common View



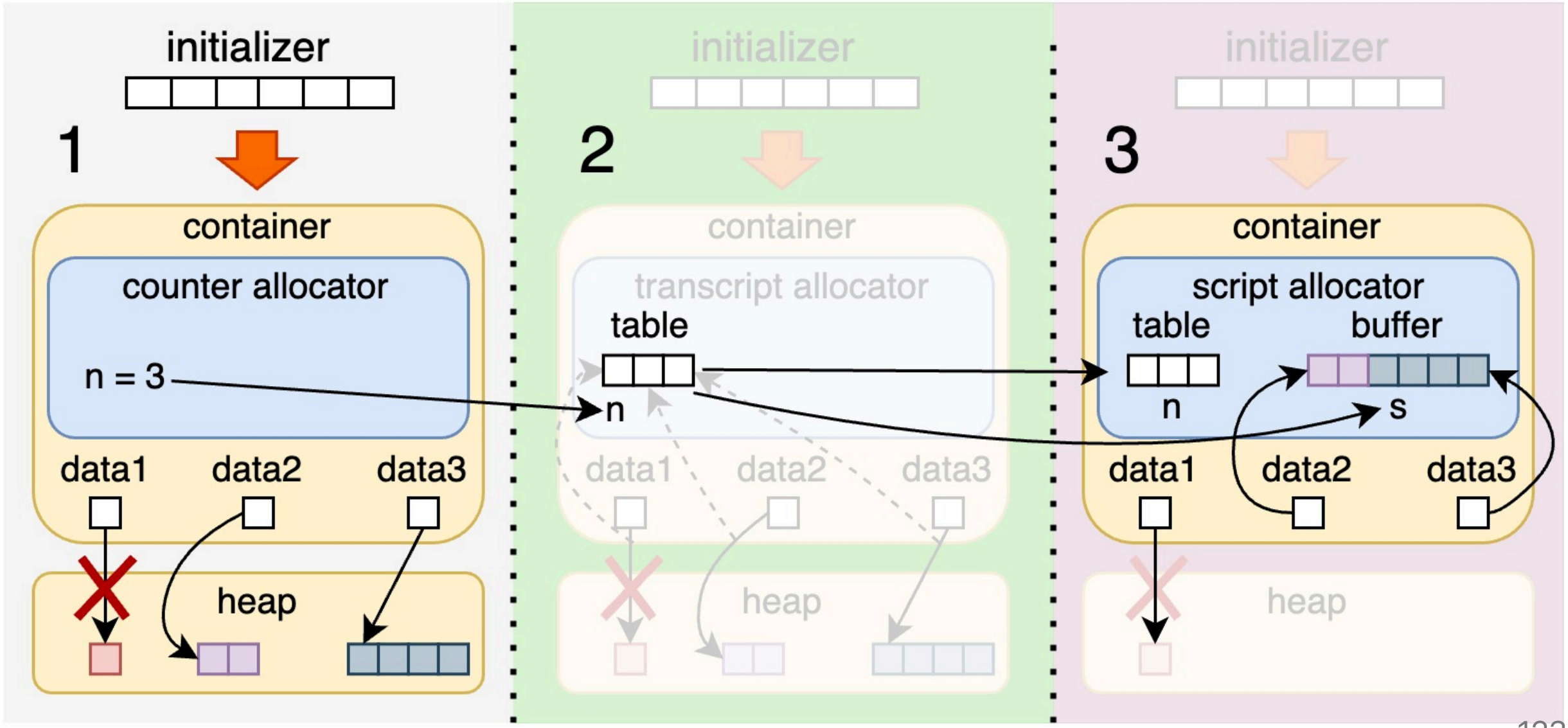
Solution: Common View



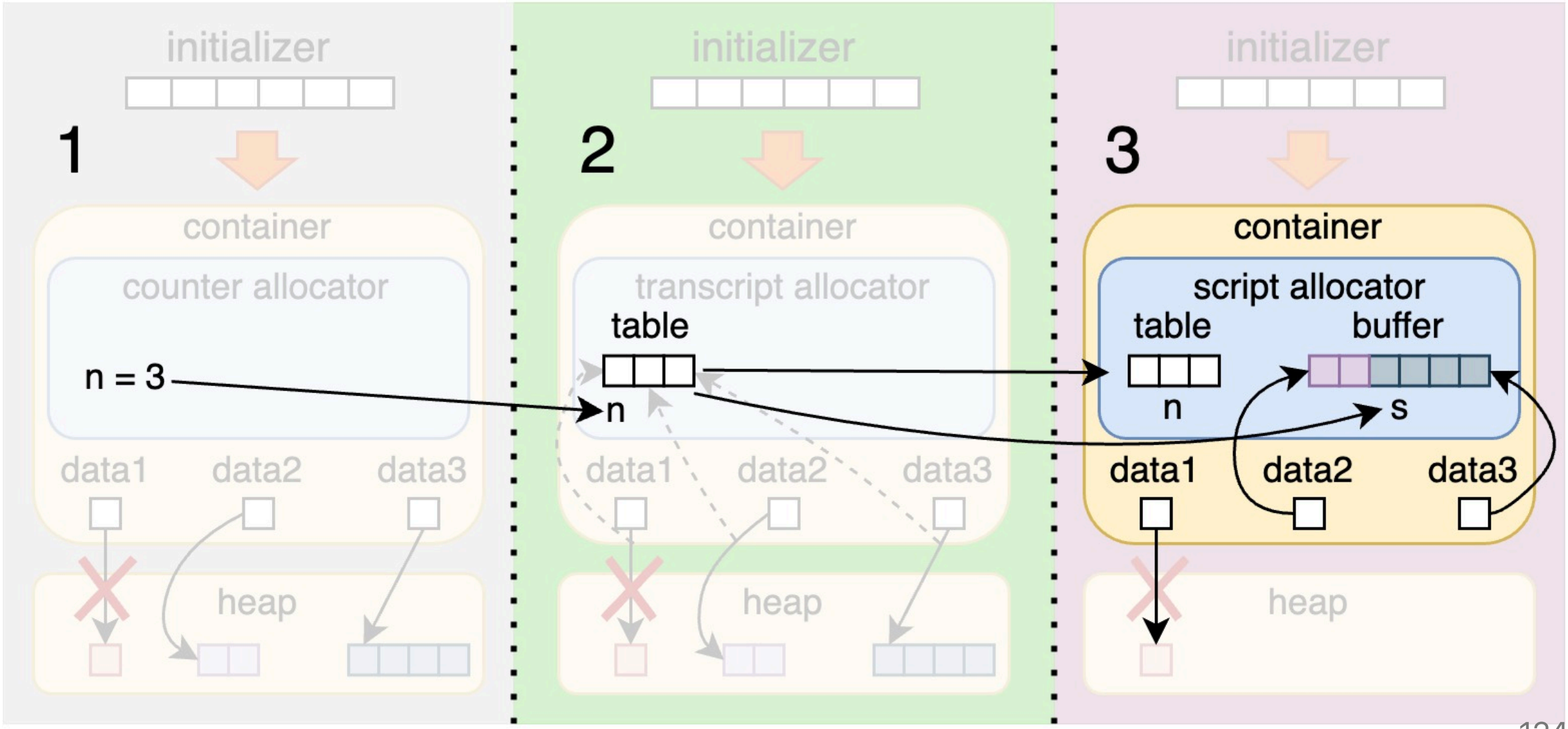
Solution: Common View



Solution: Common View



Solution: Common View



1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 **Implementation Details**
4. Additional Applications
5. Results

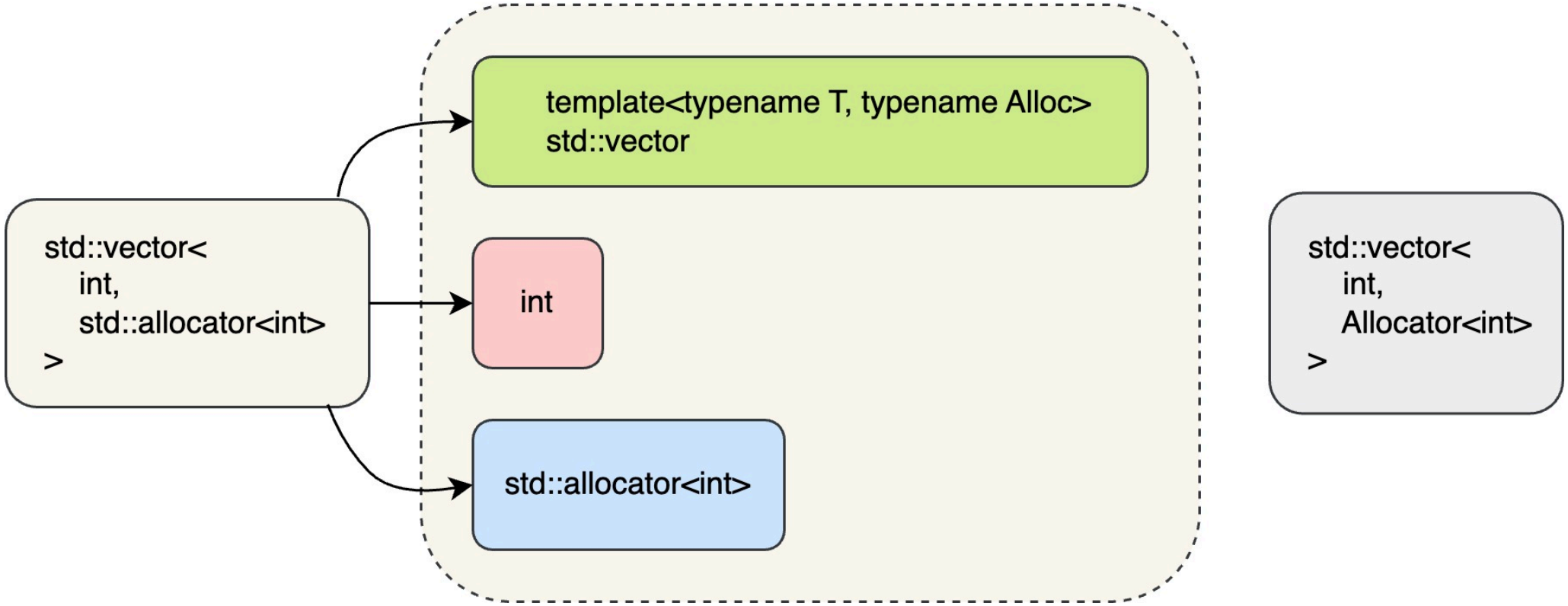
Allocator Replacement

Allocator Replacement

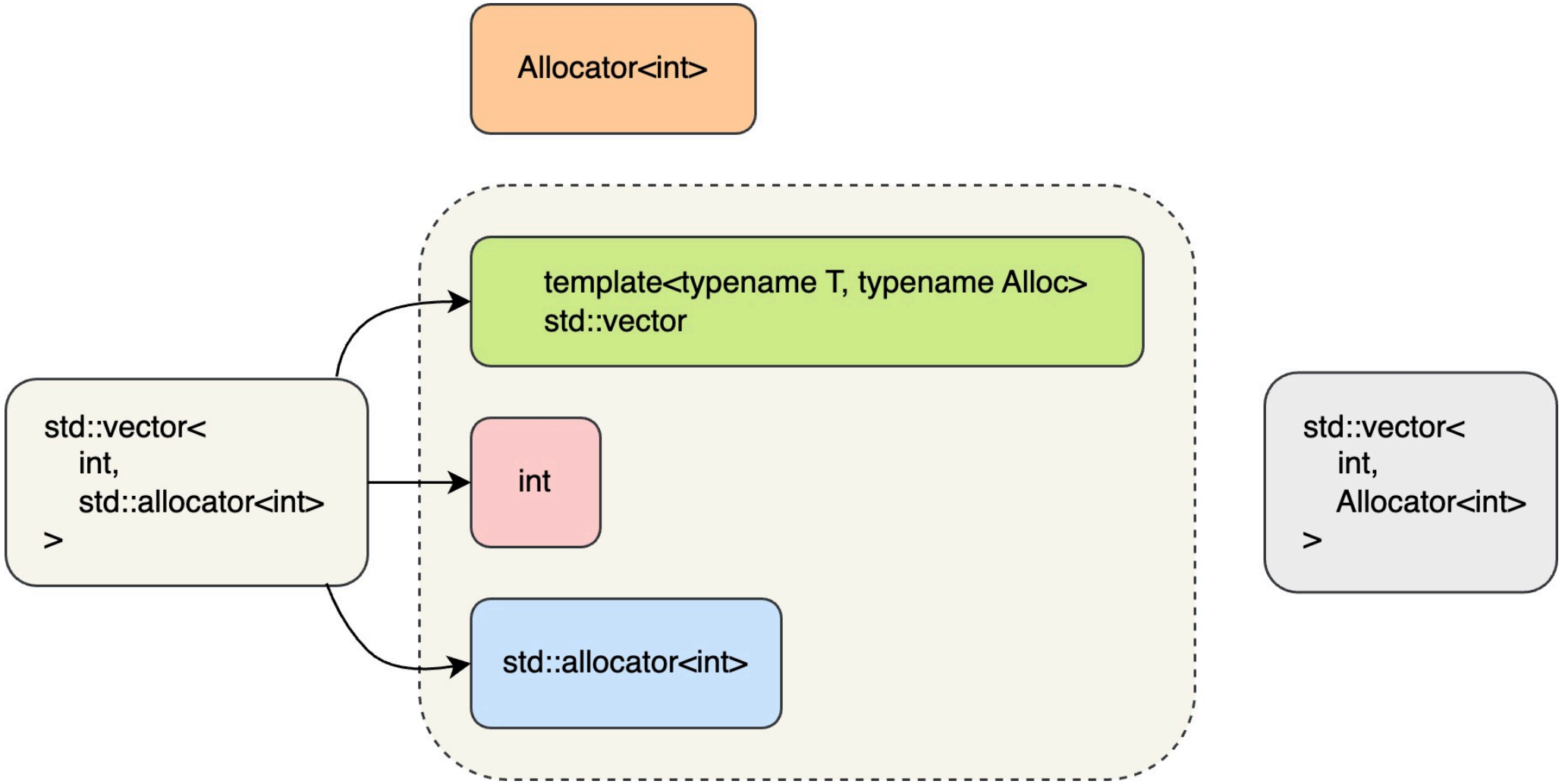
```
std::vector<
    int,
    std::allocator<int>
>
```

```
std::vector<
    int,
    Allocator<int>
>
```

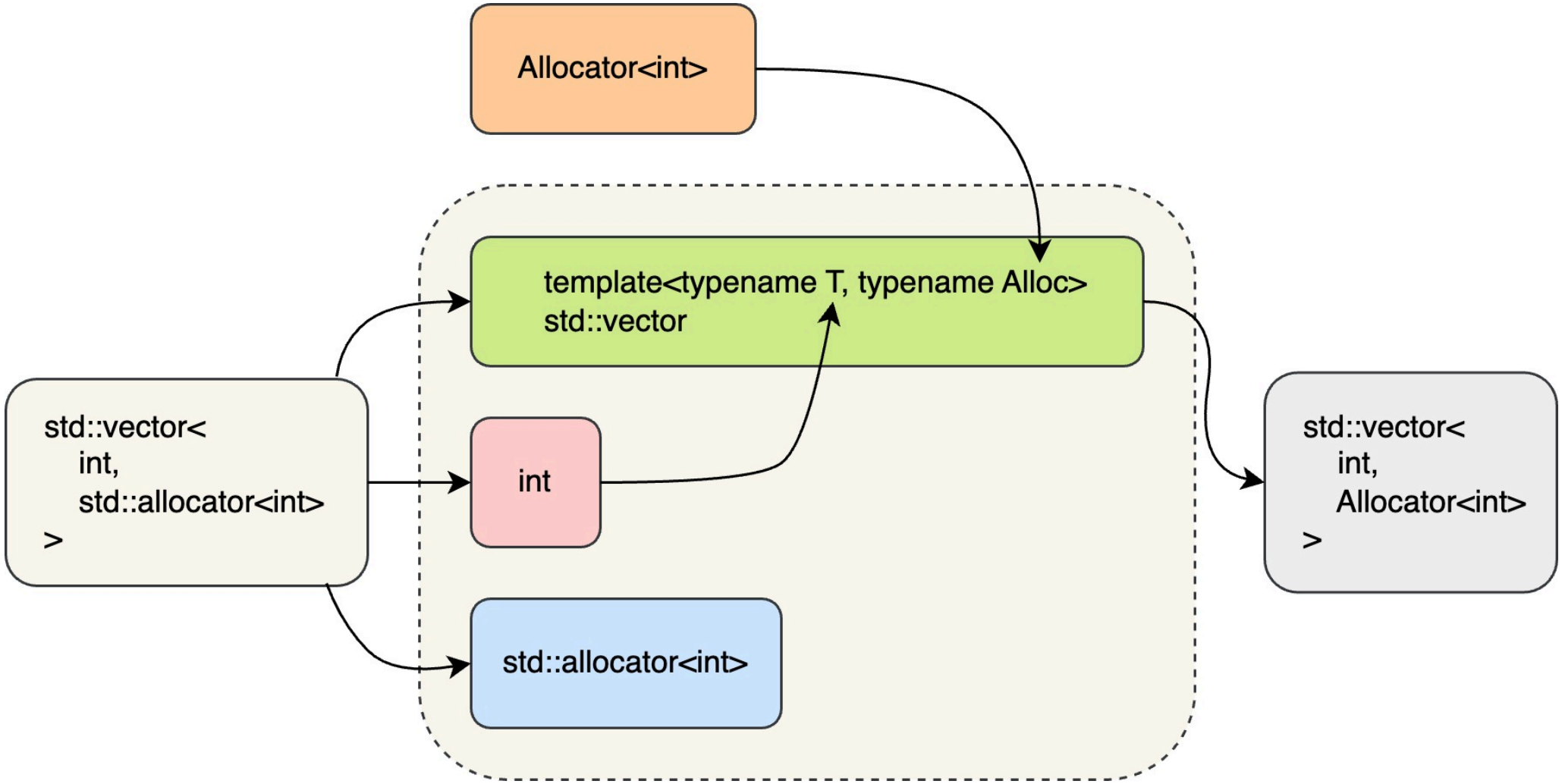
Allocator Replacement



Allocator Replacement



Allocator Replacement



Allocator Replacement

```
template<typename T, typename Allocator>
class vector
{
  ...
    template<class Other>
    struct rebind
    {
        using other = vector<T, Other> ;
    };
  ...
};

using origin_vector = std::vector<int, Allocator>;
using other_vector =
    origin_vector::rebind
        ::other<OtherAllocator>;
```

Allocator Replacement

```
template<typename T>
struct rebind_allocator;

template<
    template<typename, typename> typename Container,
    typename T,
    typename Allocator
>
struct rebind_allocator<Container<T, Allocator>>
{};

using r = rebind_allocator<std::vector<int, std::allocator<int>>>;
```

Allocator Replacement

```
template<typename T>
struct rebind_allocator;

template<
    template<typename, typename> typename Container,
    typename T,
    typename Allocator
>
struct rebind_allocator<Container<T, Allocator>>
{};

using r = rebind_allocator<std::vector<int, std::allocator<int>>>;
```

Allocator Replacement

```
template<typename T>
struct rebind_allocator;

template<
    template<typename, typename> typename Container,
    typename T,
    typename Allocator>
struct rebind_allocator<Container<T, Allocator>>
{
    template <typename NewAllocator>
    using to = Container<T, NewAllocator>;
};

template<typename T> struct Alloc{};

using origin = std::vector<int, std::allocator<int>>;
using result = rebind_allocator<origin>::to<Alloc<int>>;
```

Initializer Passing

Initializer Passing

```
constexpr Result make(std::vector<int> init) {



}
```

Initializer Passing

```
constexpr Result make(std::vector<int> init) {




}
constexpr int count(std::vector<int> init) {
  int res = 0;
  // some magic with init
  return res;
}
```

Initializer Passing

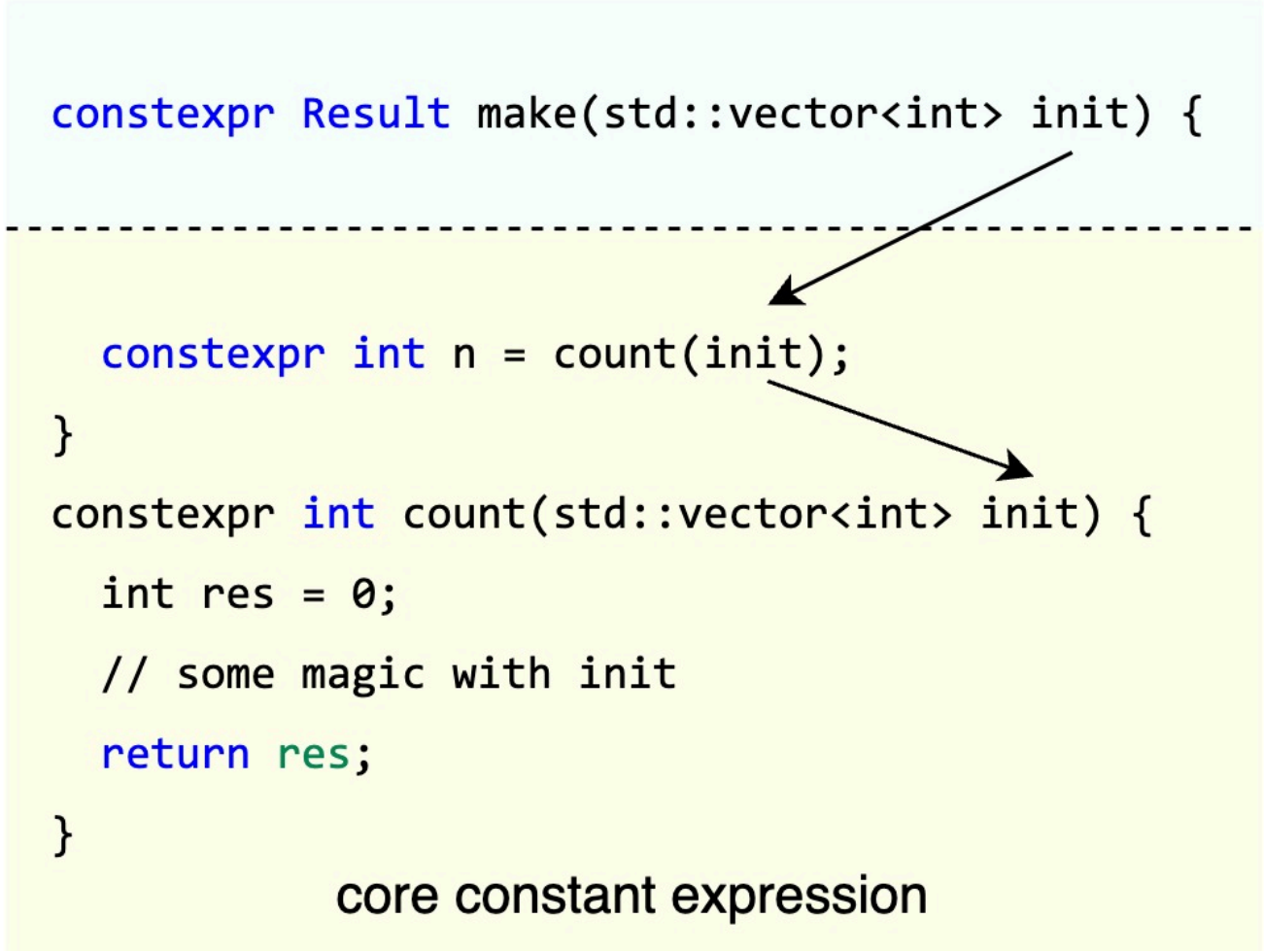
```
constexpr Result make(std::vector<int> init) {


  constexpr int n = count(init);
}
constexpr int count(std::vector<int> init) {
  int res = 0;
  // some magic with init
  return res;
}
```

Initializer Passing

```
constexpr Result make(std::vector<int> init) {
----------------------------------------------------
  constexpr int n = count(init);
}
constexpr int count(std::vector<int> init) {
  int res = 0;
  // some magic with init
  return res;
}
```

core constant expression

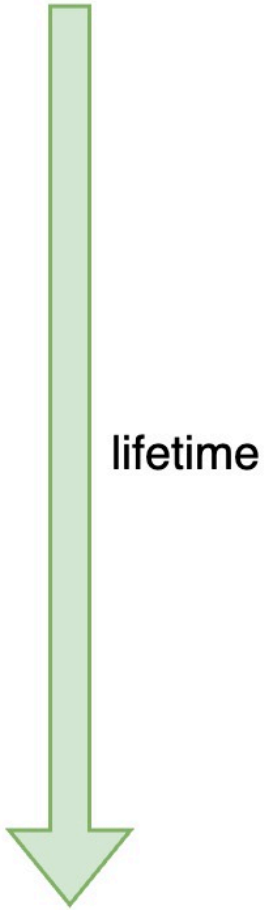
Initializer Passing



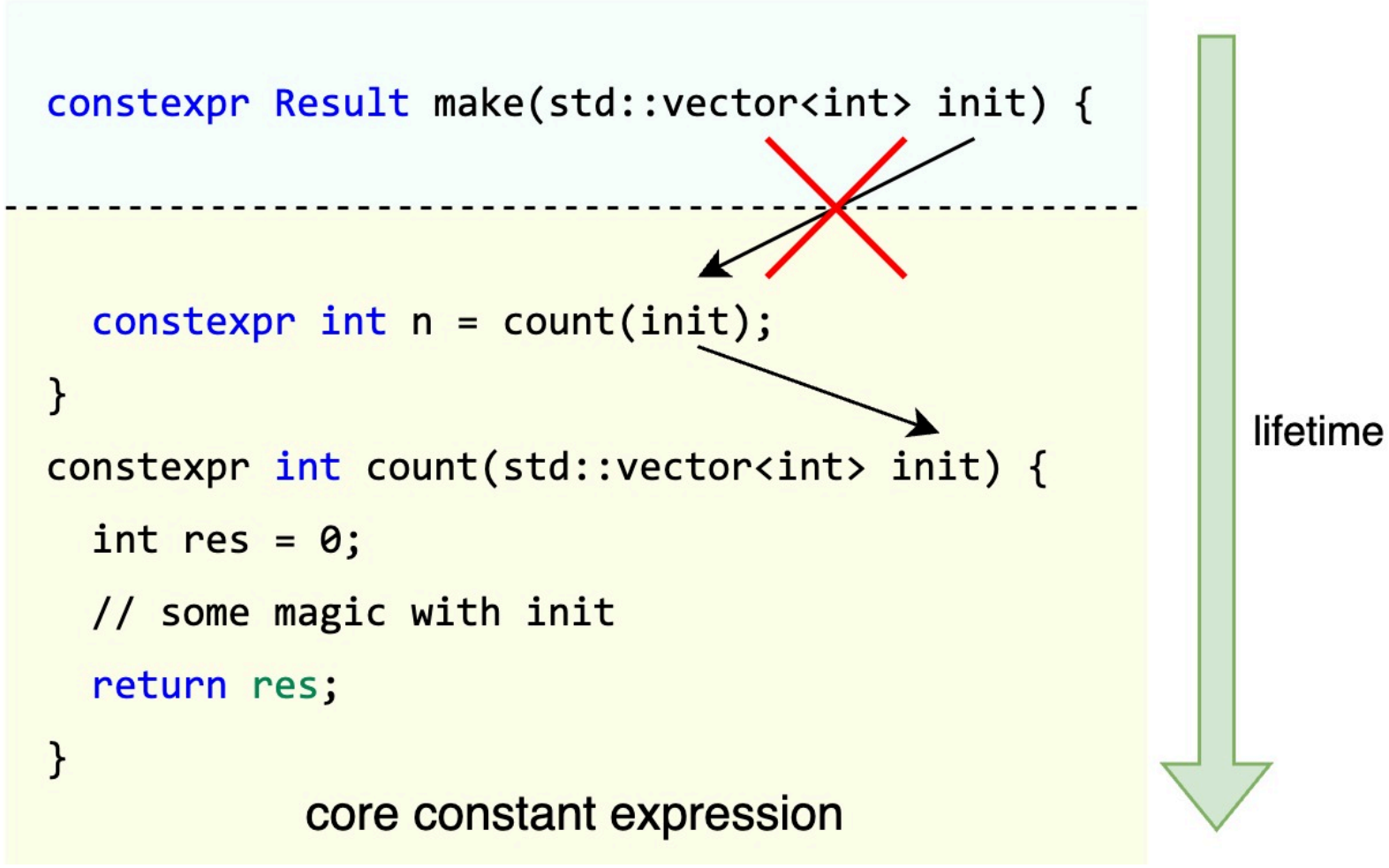
Initializer Passing

```
constexpr Result make(std::vector<int> init) {
-----------------------------------------------
  constexpr int n = count(init);
}
constexpr int count(std::vector<int> init) {
  int res = 0;
  // some magic with init
  return res;
}
```

core constant expression



Initializer Passing



Initializer Passing

Proposal 1045:

Initializer Passing

Proposal 1045: `constexpr` Function Parameters

Initializer Passing

Proposal 1045: `constexpr` Function Parameters

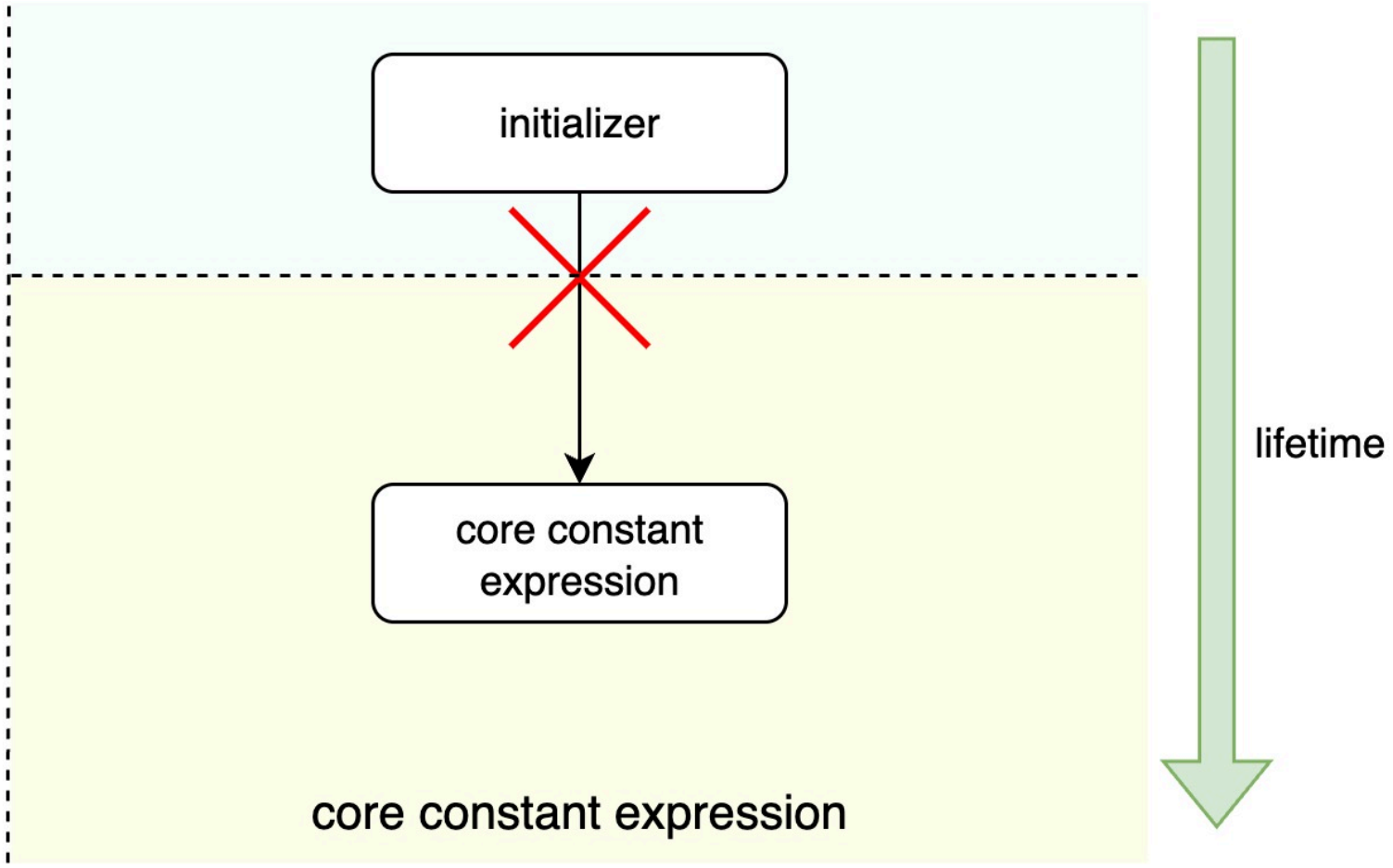
```
void f(constexpr int x)
{
    static_assert(x == 5);
}
```

Initializer Passing

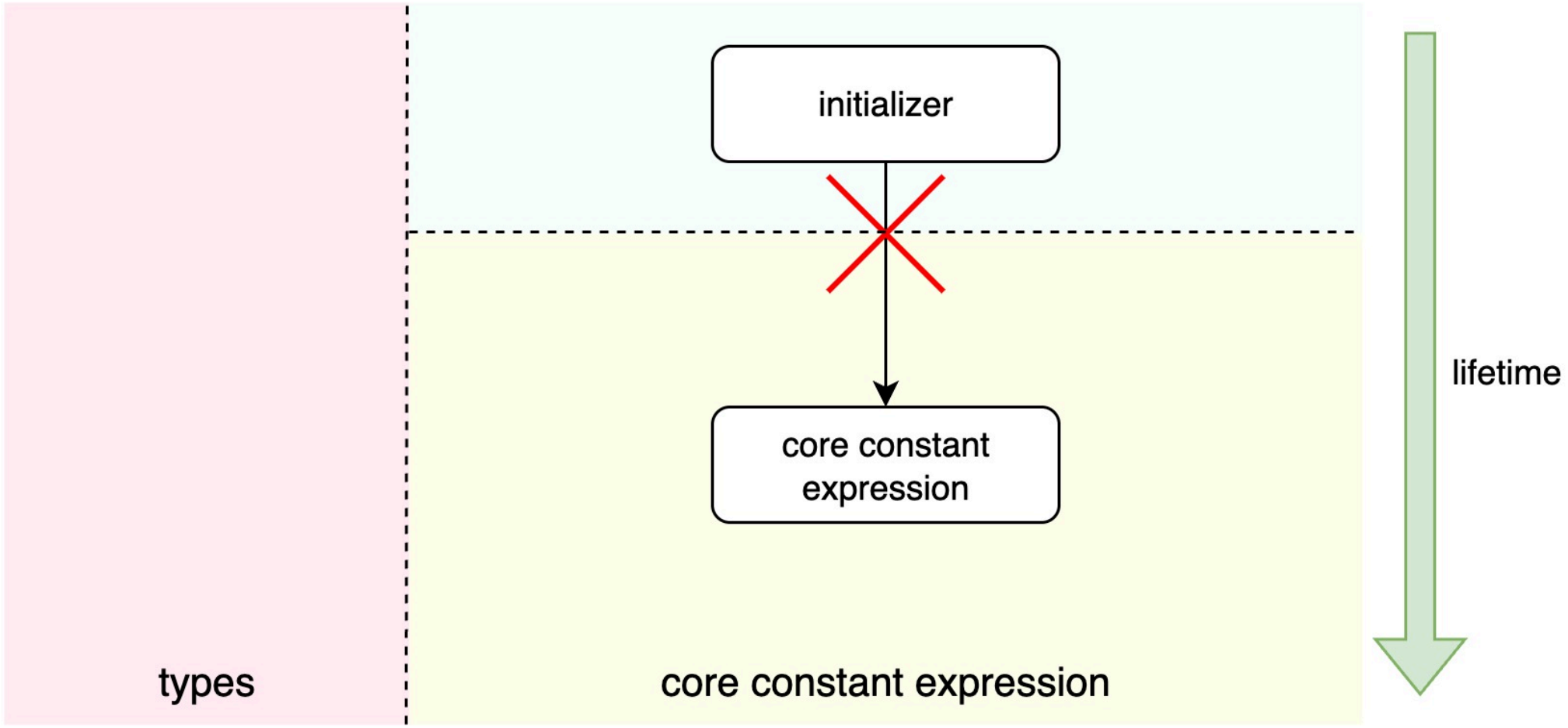
Proposal 1045: `constexpr` Function Parameters (Rejected)

```
void f(constexpr int x)
{
    static_assert(x == 5);
}
```

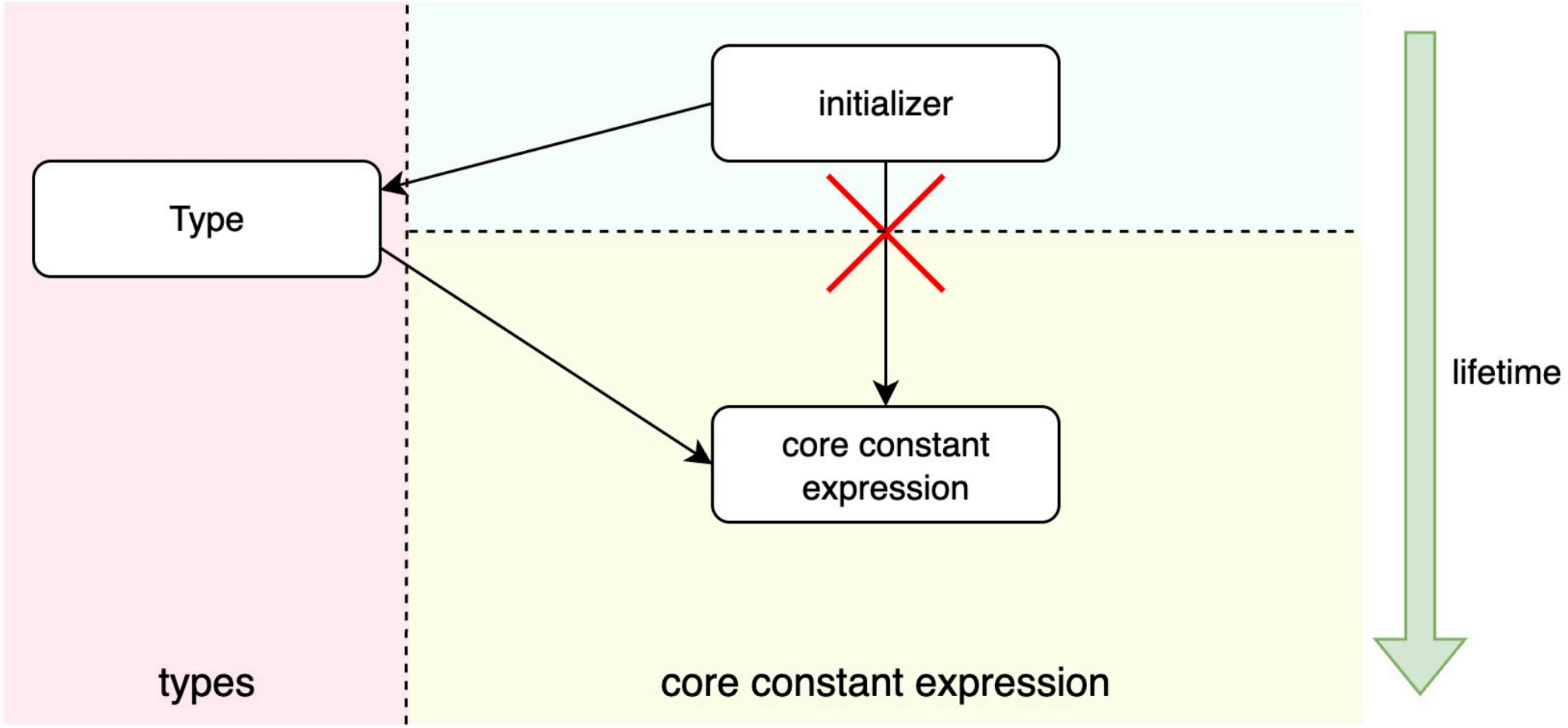
Initializer Passing



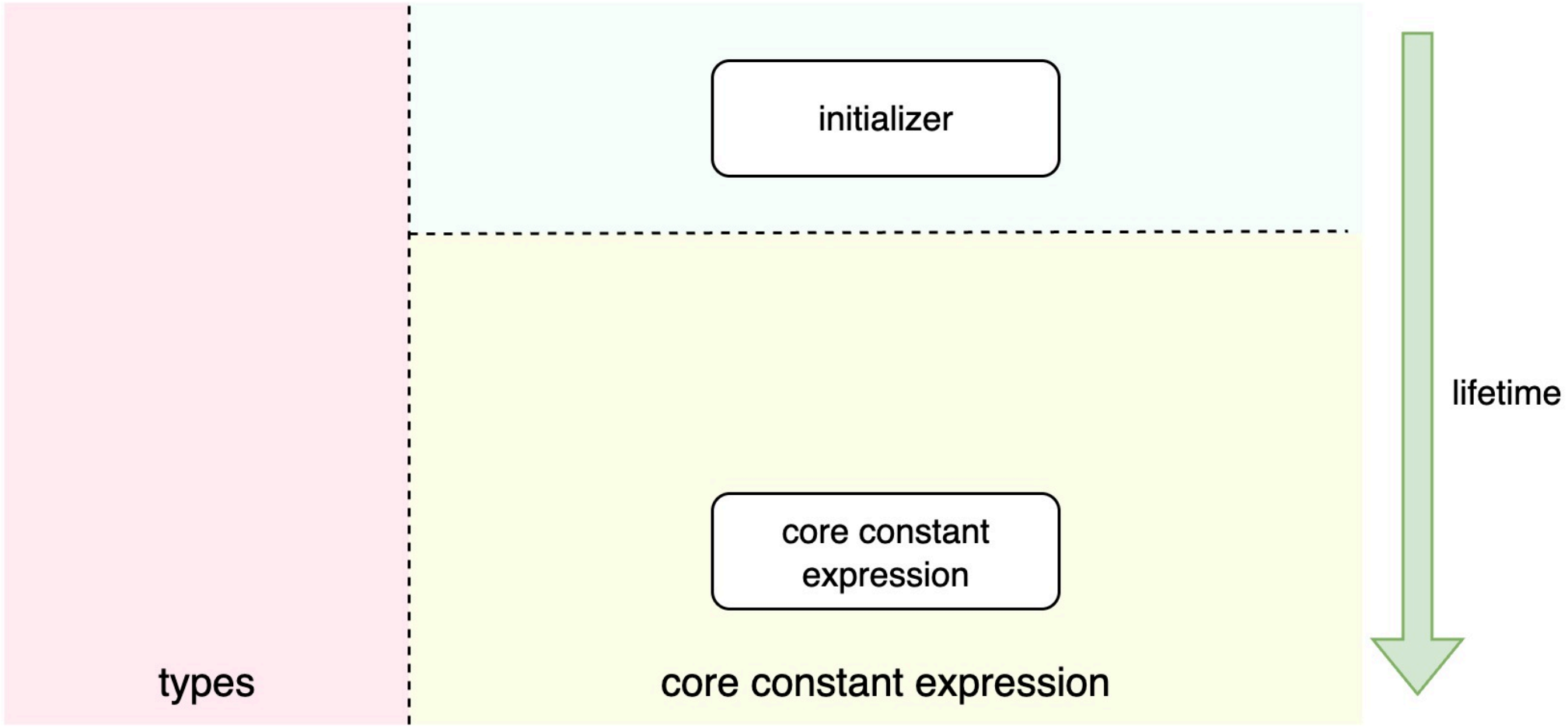
Initializer Passing



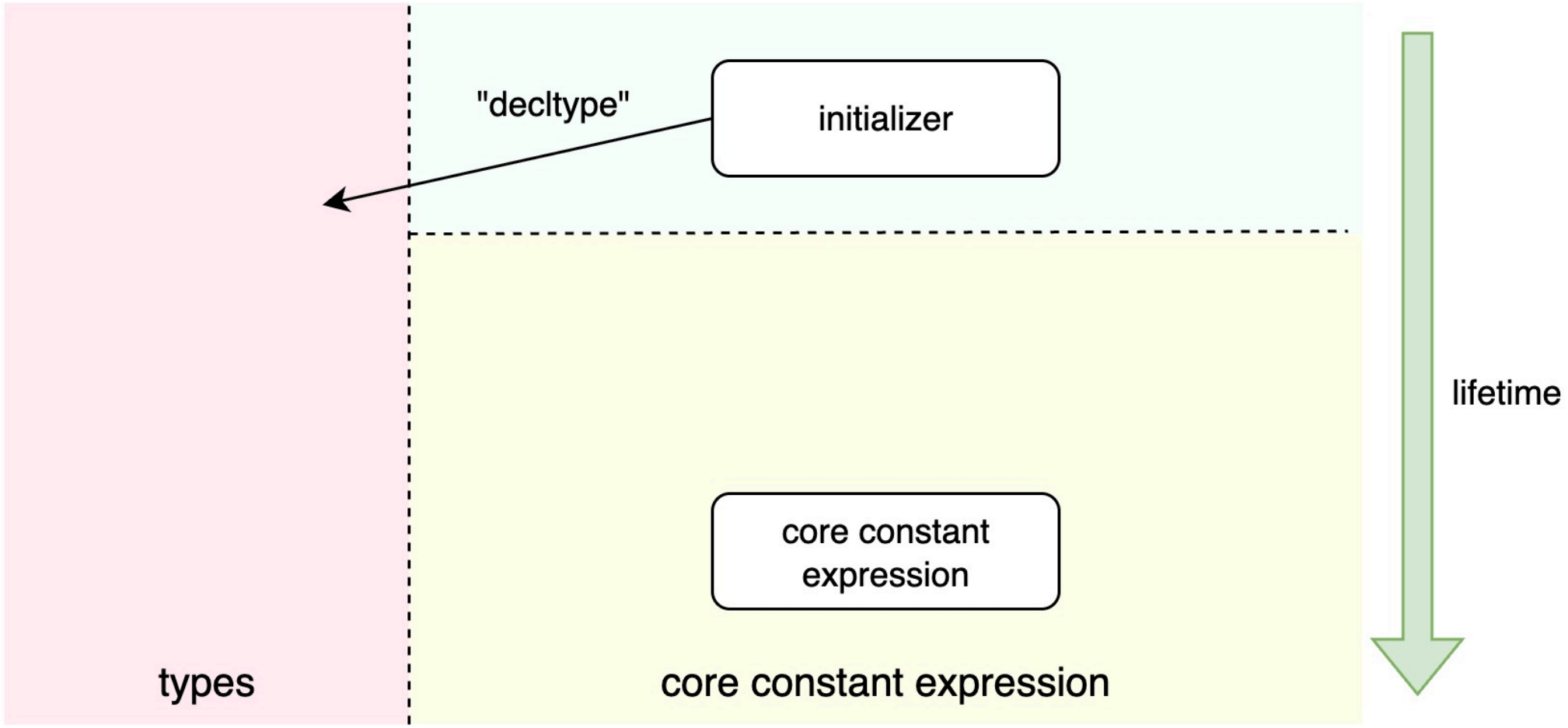
Initializer Passing



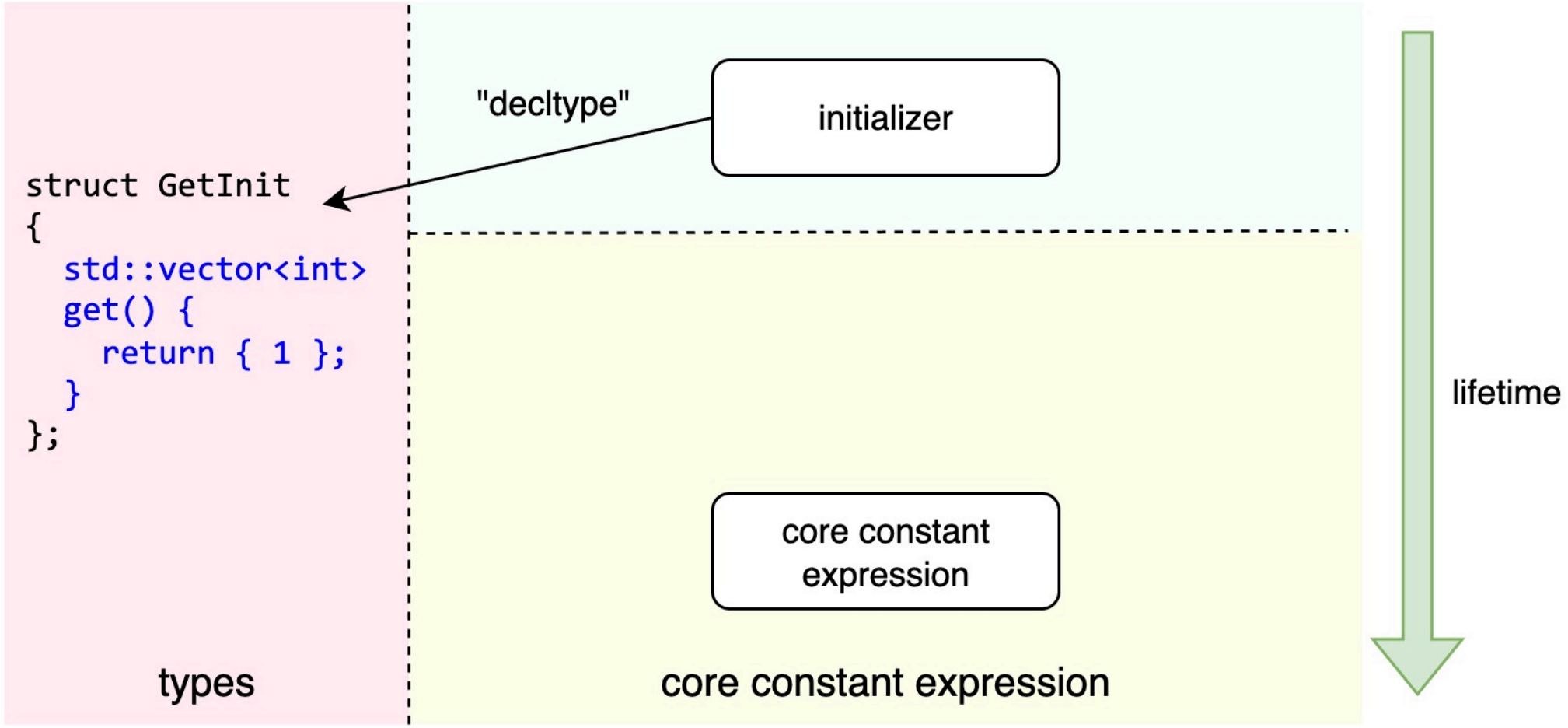
Initializer Passing



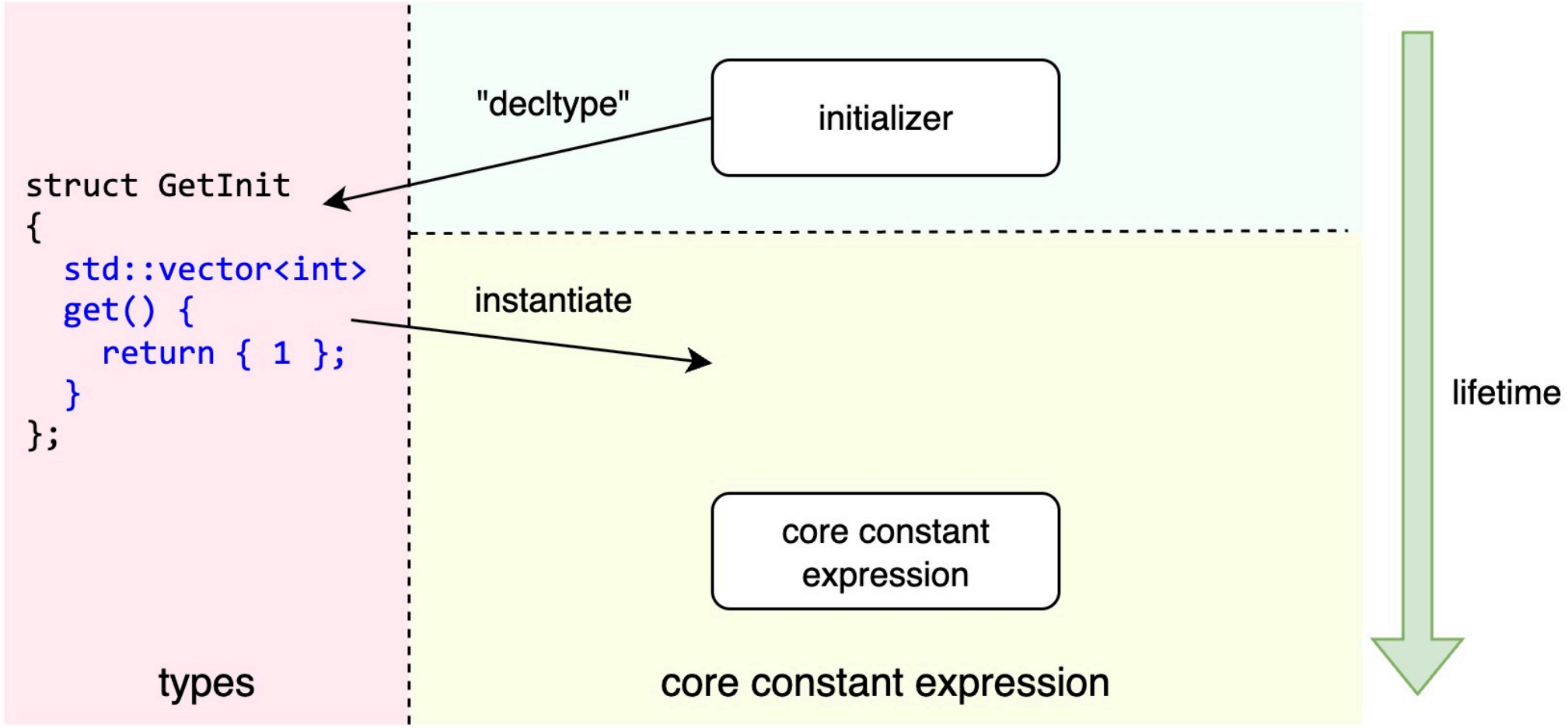
Initializer Passing



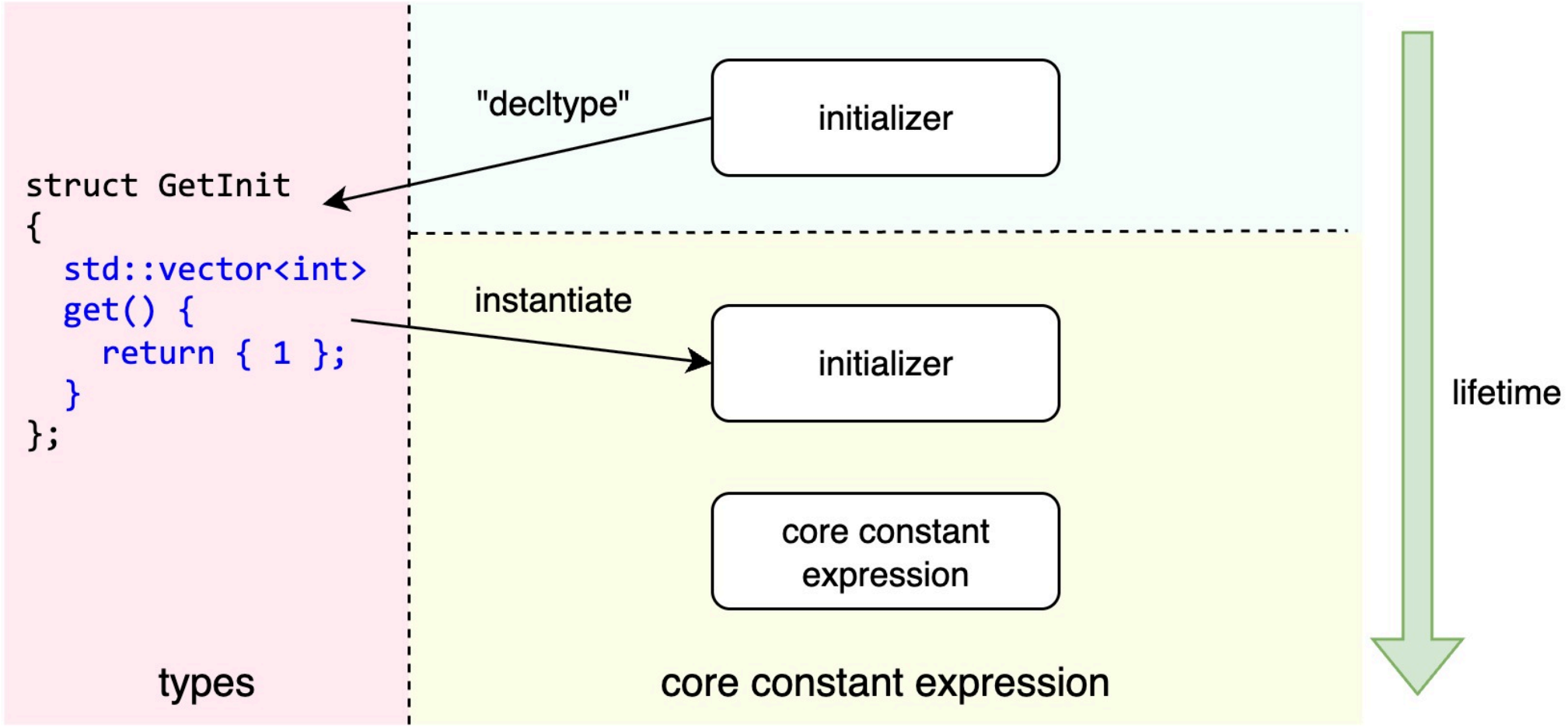
Initializer passing



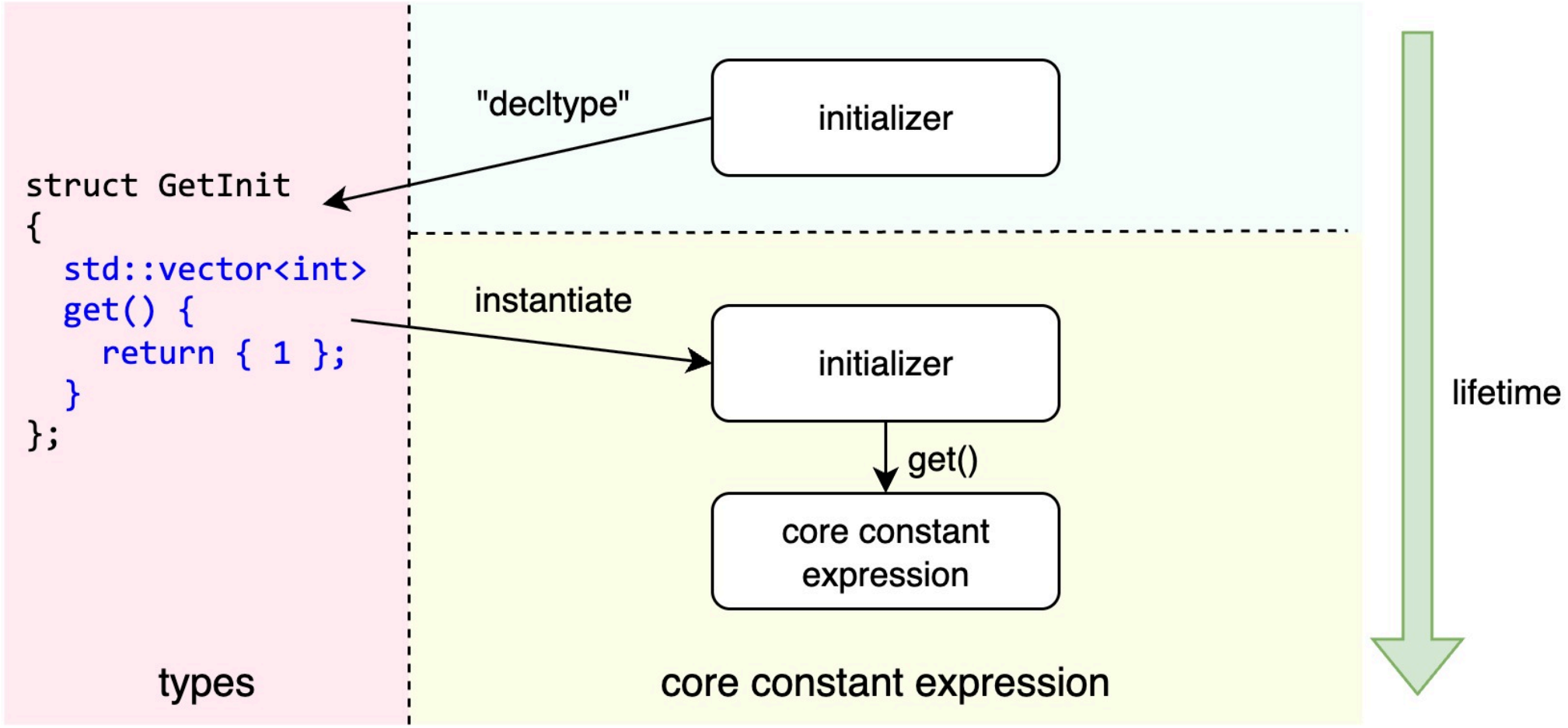
Initializer Passing



Initializer Passing



Initializer Passing



Initializer Passing

Proposal 2781: `std::constant_wrapper`

Initializer Passing

Proposal 2781: `std::constant_wrapper` (C++26)

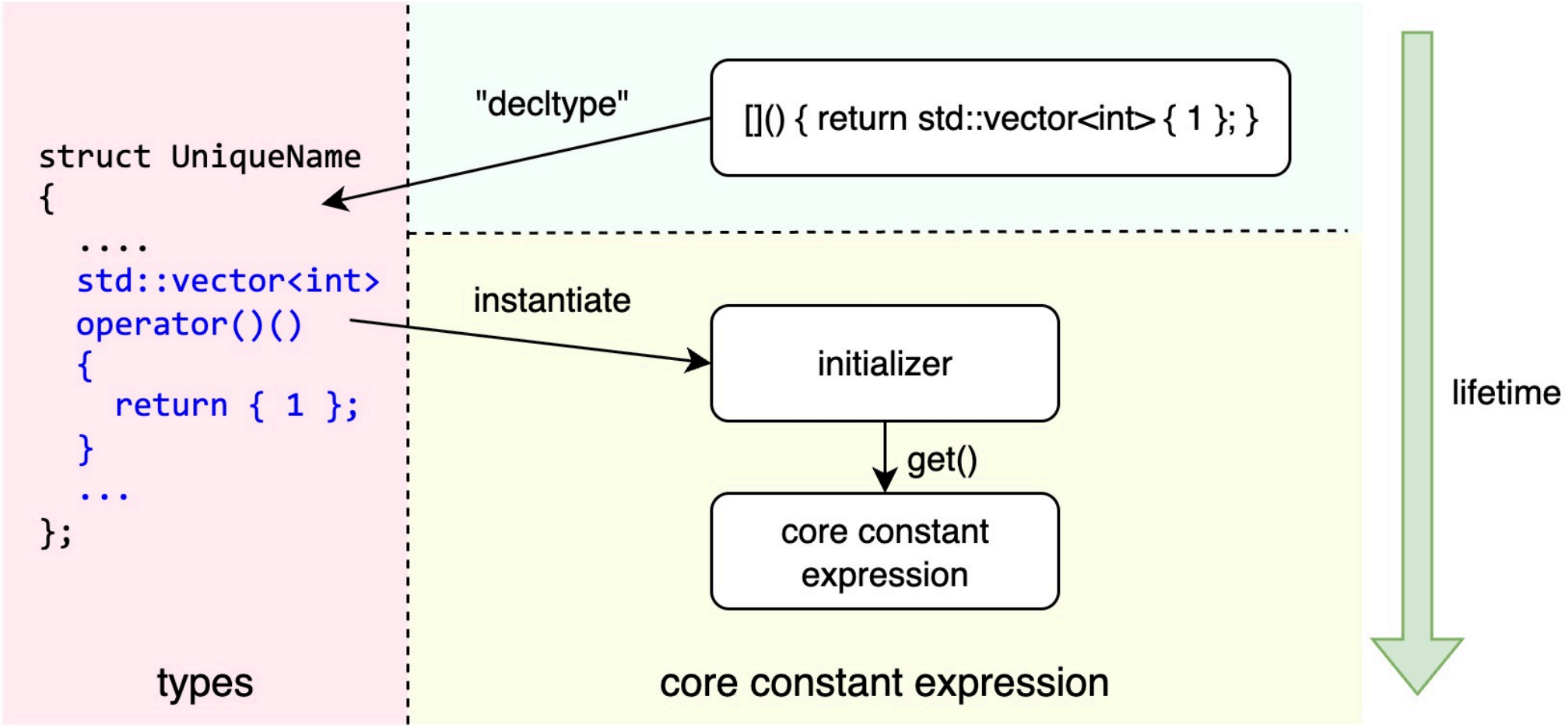
```
namespace std {
  template<auto X>
  struct constant_wrapper
  {
    using value_type = remove_cvref_t<decltype(X)>;
    using type = constant_wrapper;

    constexpr operator value_type() const { return X; }
    static constexpr value_type value = X;
  };
}

constexpr auto value = foo(std::constexpr_wrapper<1>);
```

Initializer Passing

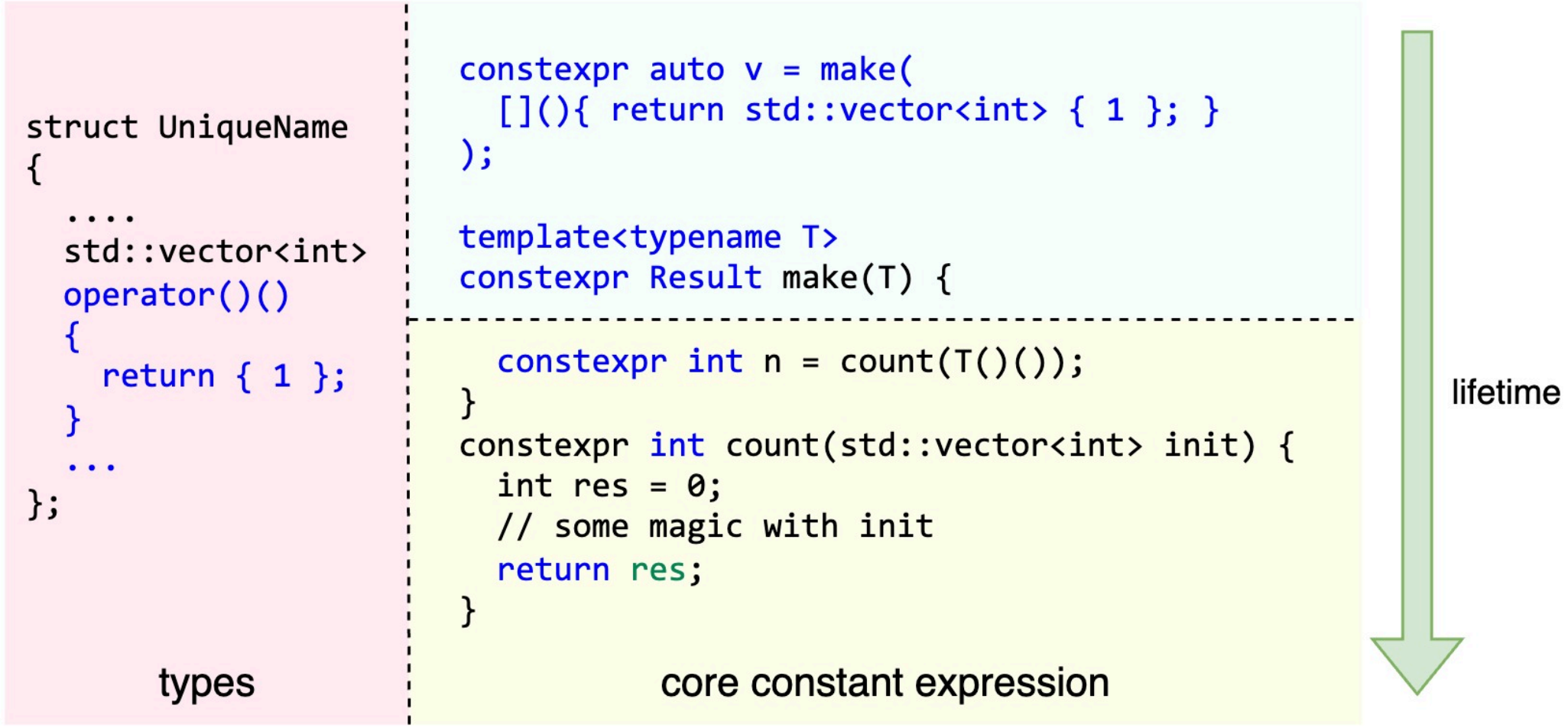
Proposal 2781: `std::constant_wrapper` (C++26)

```
namespace std {
  template<auto X>
  struct constant_wrapper
  {
    using value_type = remove_cvref_t<decltype(X)>;
    using type = constant_wrapper;

    constexpr operator()() const { return X; }
    static constexpr value_type value = X;
  };
}

constexpr auto value = foo(std::constexpr_wrapper<1>);
```

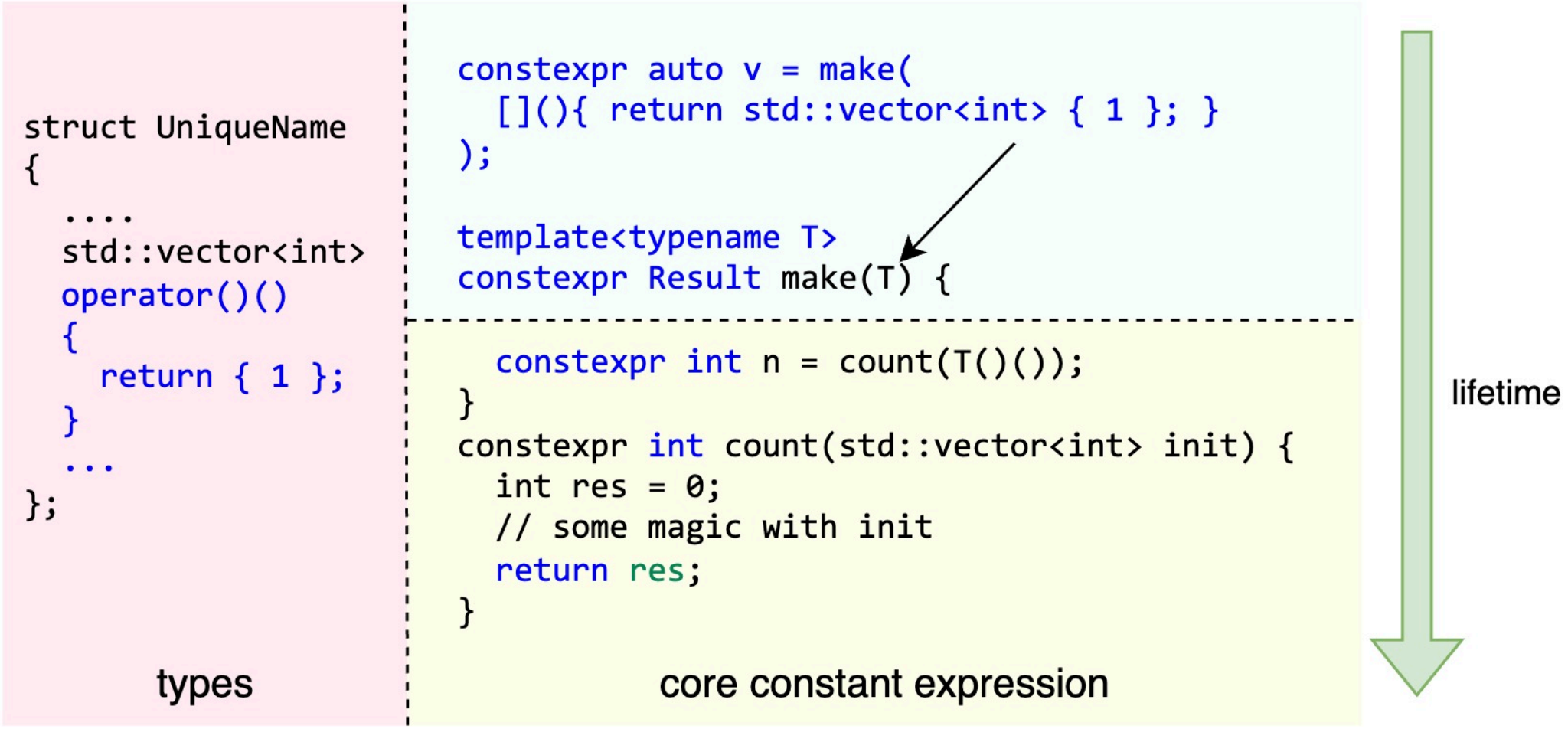
Initializer Passing



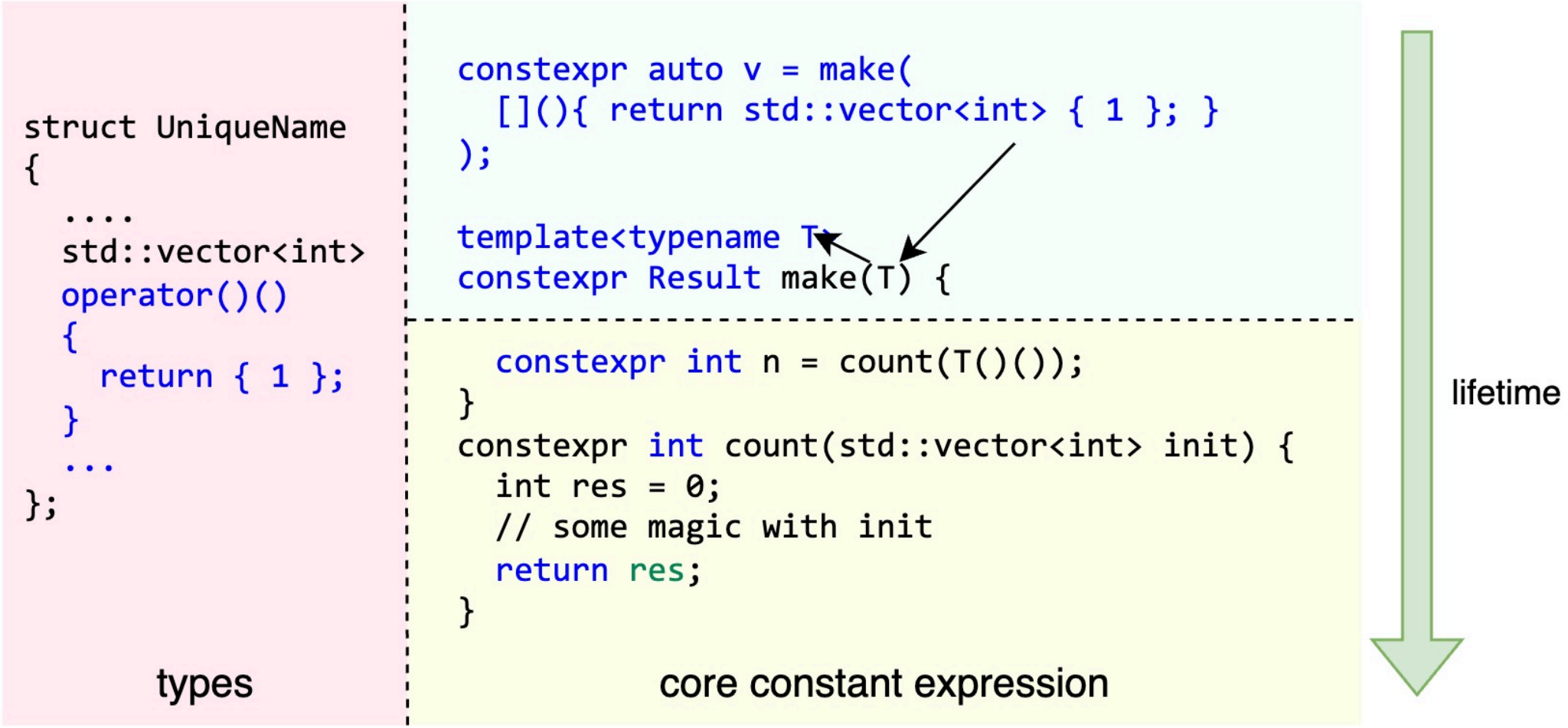
Initializer Passing



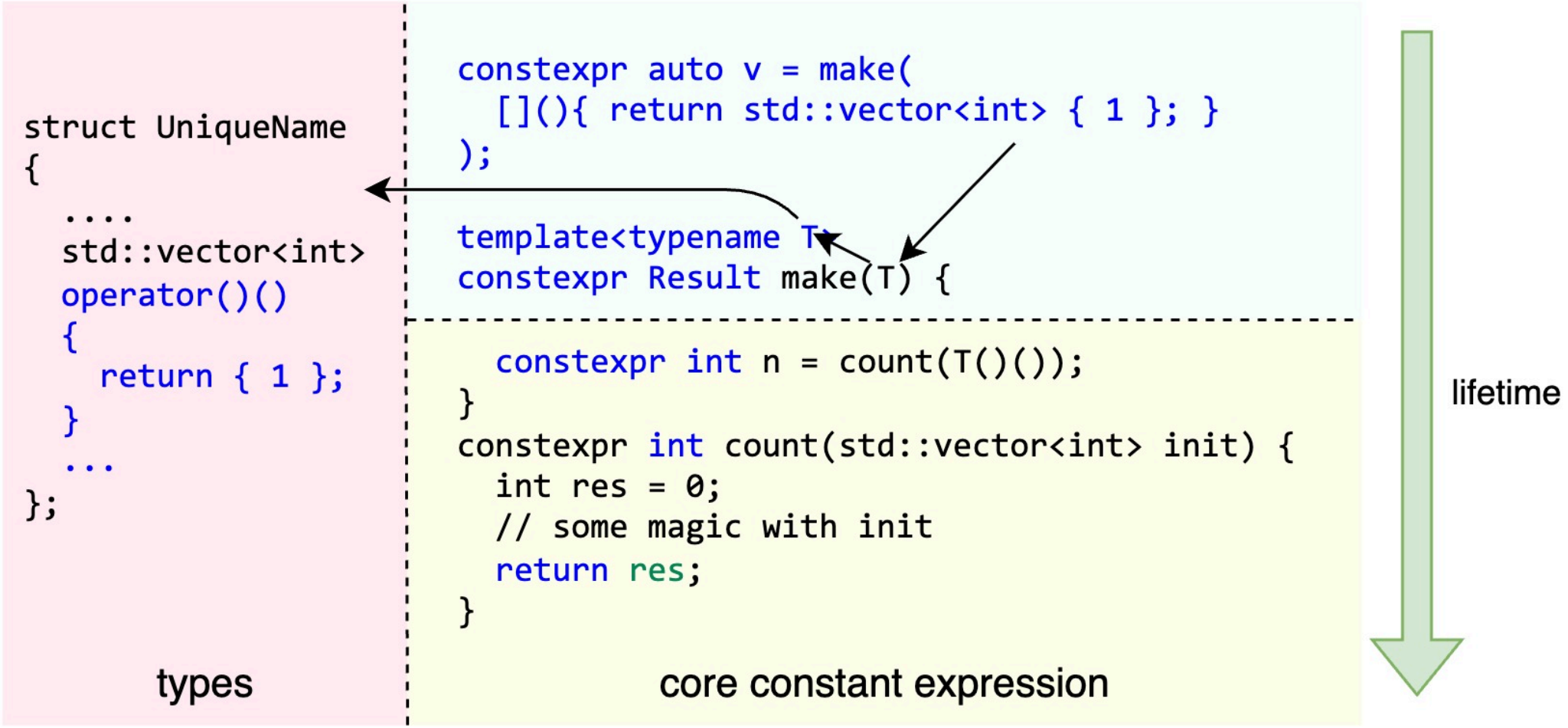
Initializer Passing



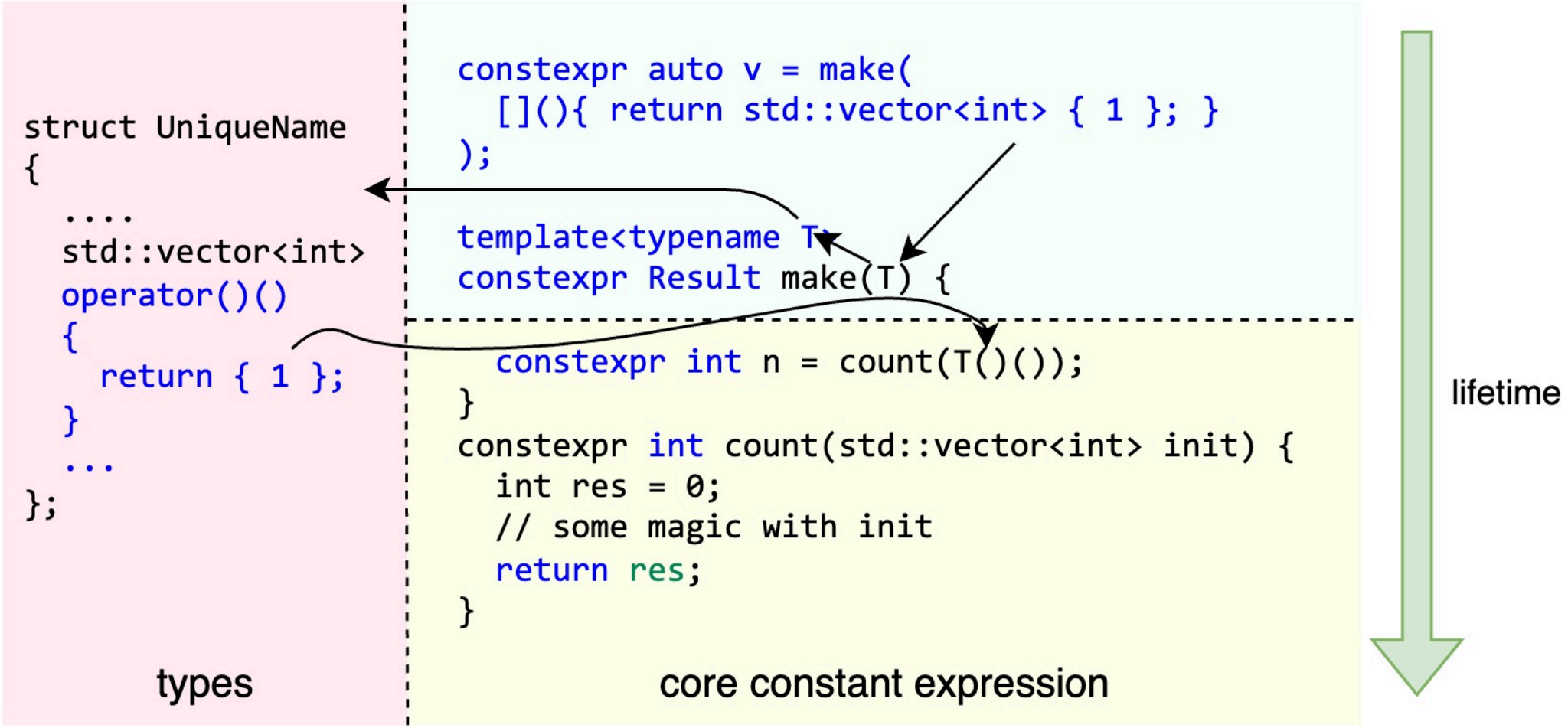
Initializer Passing



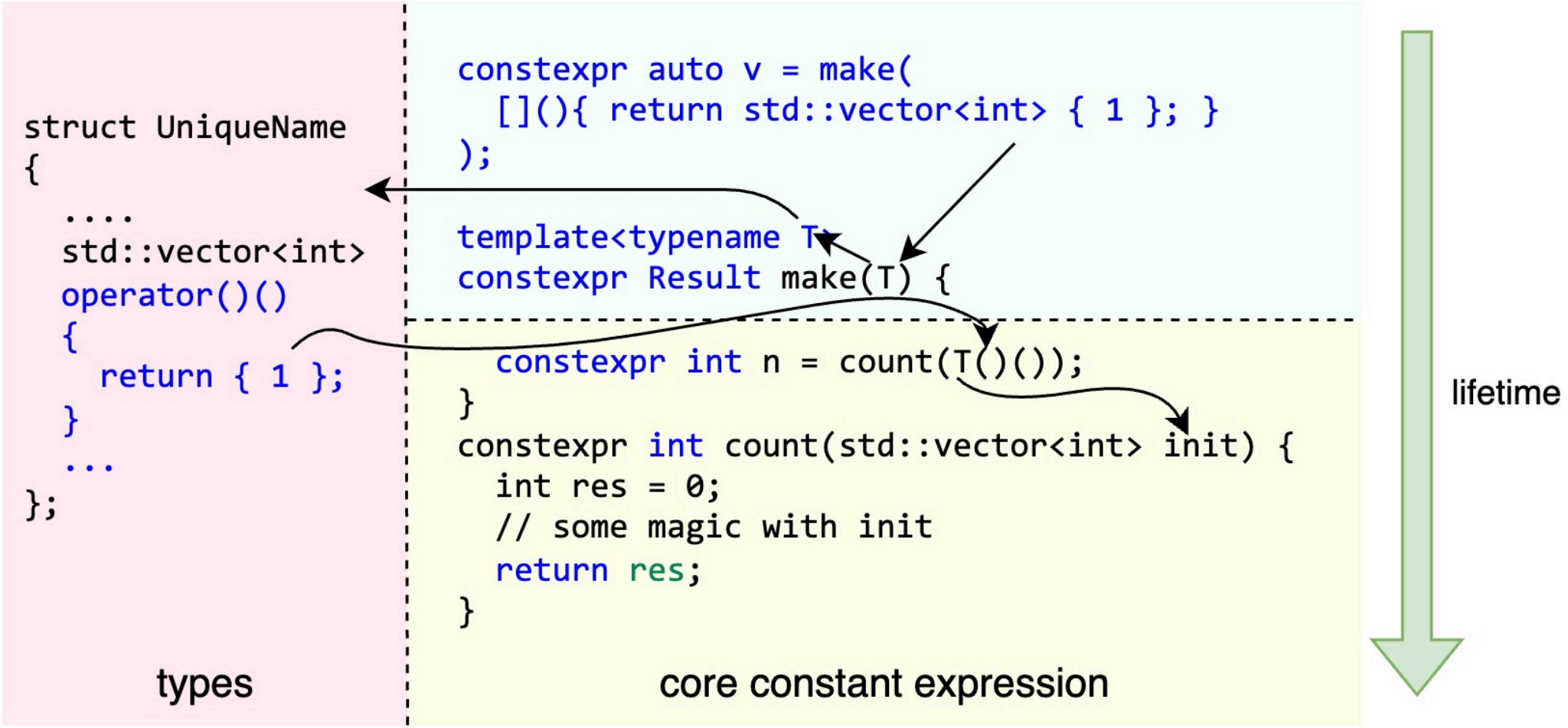
Initializer Passing



Initializer Passing



Initializer Passing



Full Implementation

https://github.com/sssersh/too_constexpr

Usage

```
constexpr std::string_view str1 = "It is string which was builded ";
constexpr std::string_view str2 = "as concatenation of two strings";

constexpr auto concat_result =
    cant::too_constexpr(
        []() -> std::string
        {
            return std::string(str1) + std::string(str2);
        }
);

// It does work!
static_assert(
  concat_result ==
  "It is string which was builded as concatenation of two strings",
  "error");
```

1. Dynamic Memory and Containers in `constexpr`
   1.1 Language Features
   1.2 Proposals
   1.3 Data Structures
2. Analysis
   2.1 Challenges and Solutions
   2.2 Non-Trivial Cases
3. `constexpr` Allocator Implementation
   2.1 Common Solution Overview
   2.2 Implementation Details
4. **Additional Applications**
5. Results

`std::basic_regex`: Creation (libc++)

`std::basic_regex`: Creation (libc++)

```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```

`std::basic_regex`: Creation (libc++)

```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```

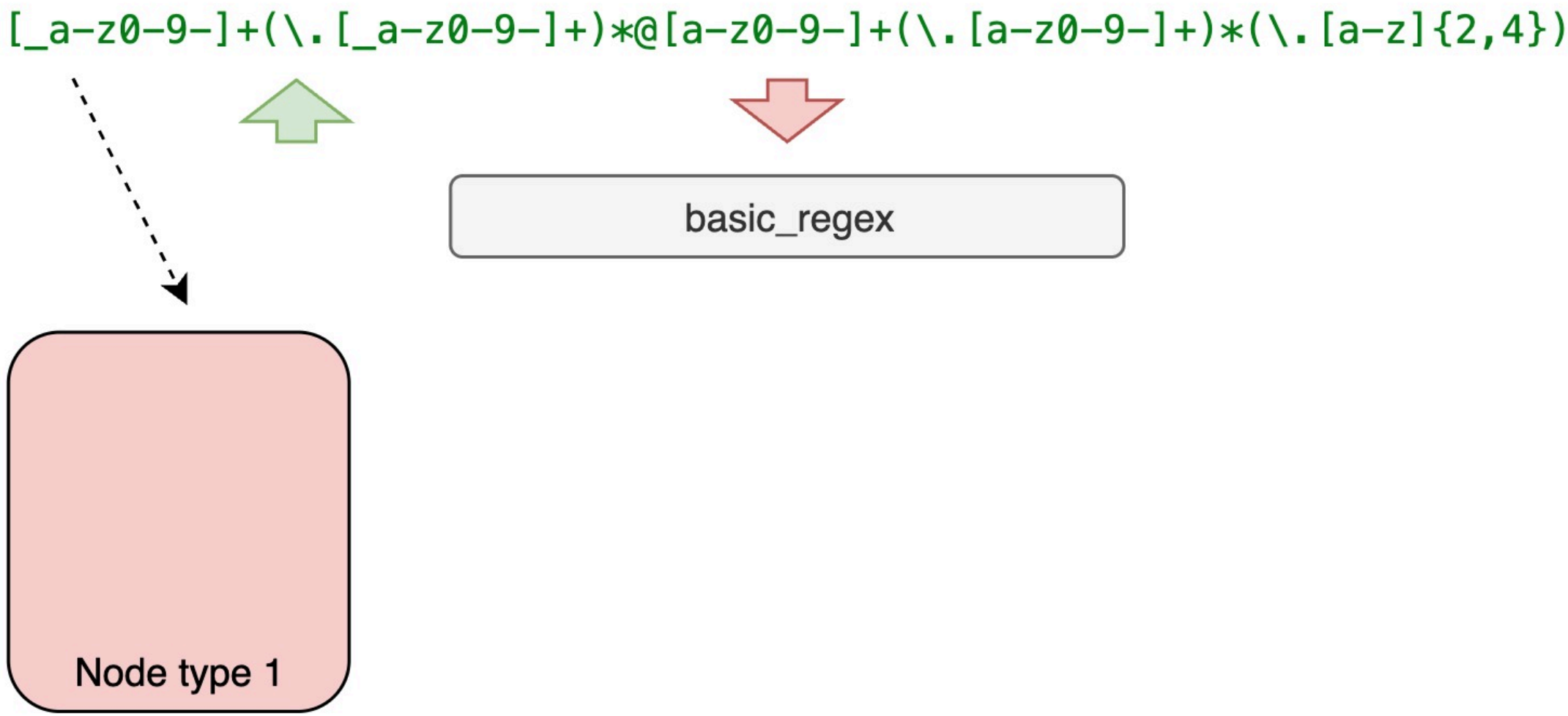
basic_regex

`std::basic_regex`: Creation (libc++)

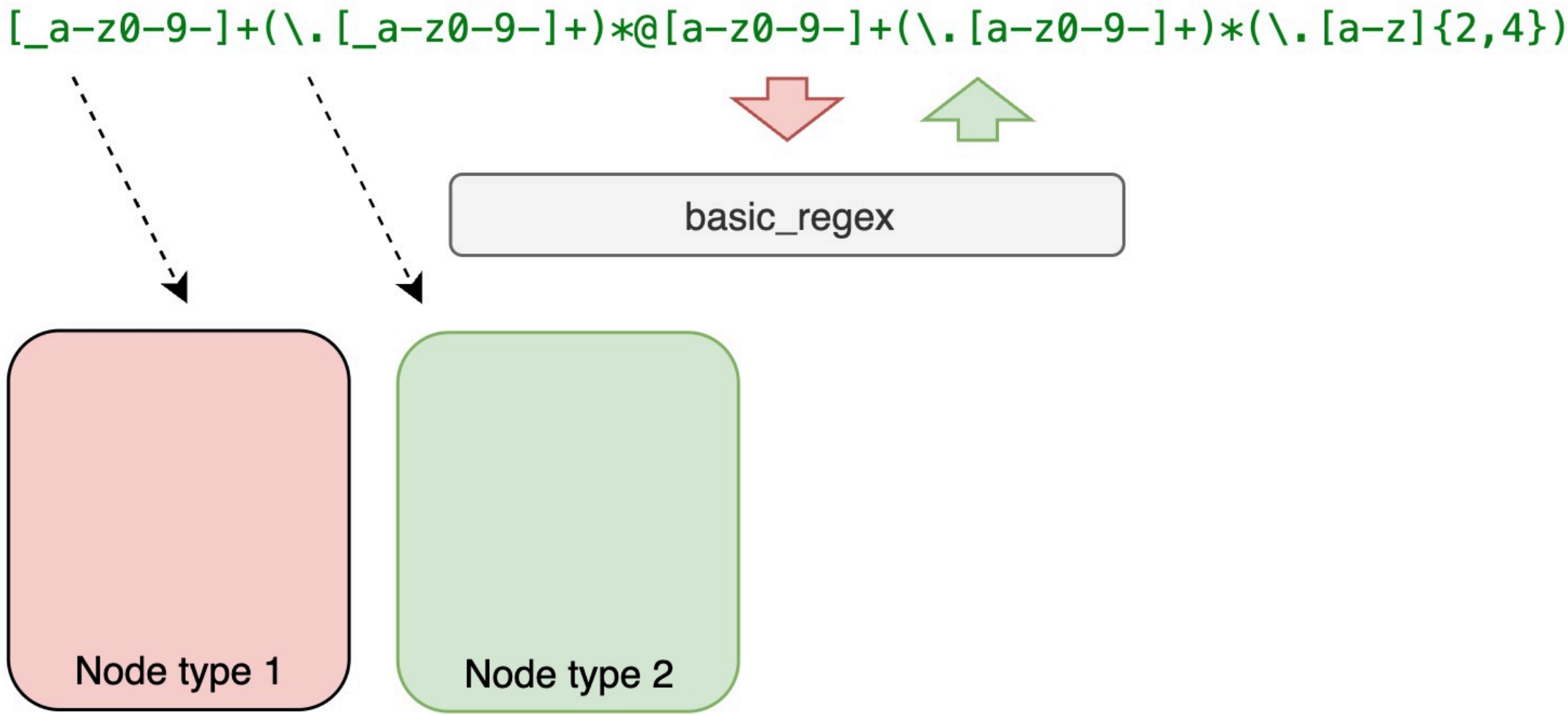
```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```

basic_regex

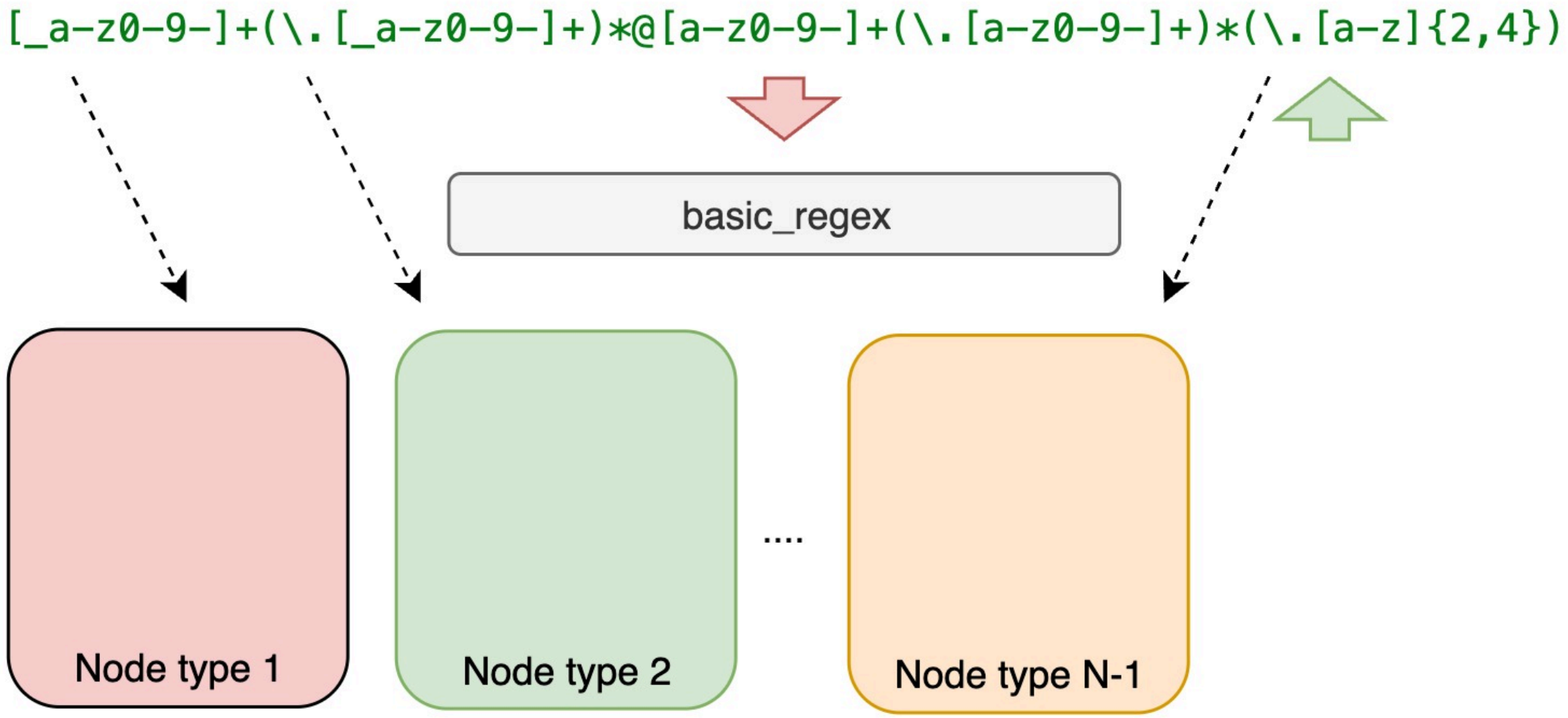
`std::basic_regex`: Creation (libc++)



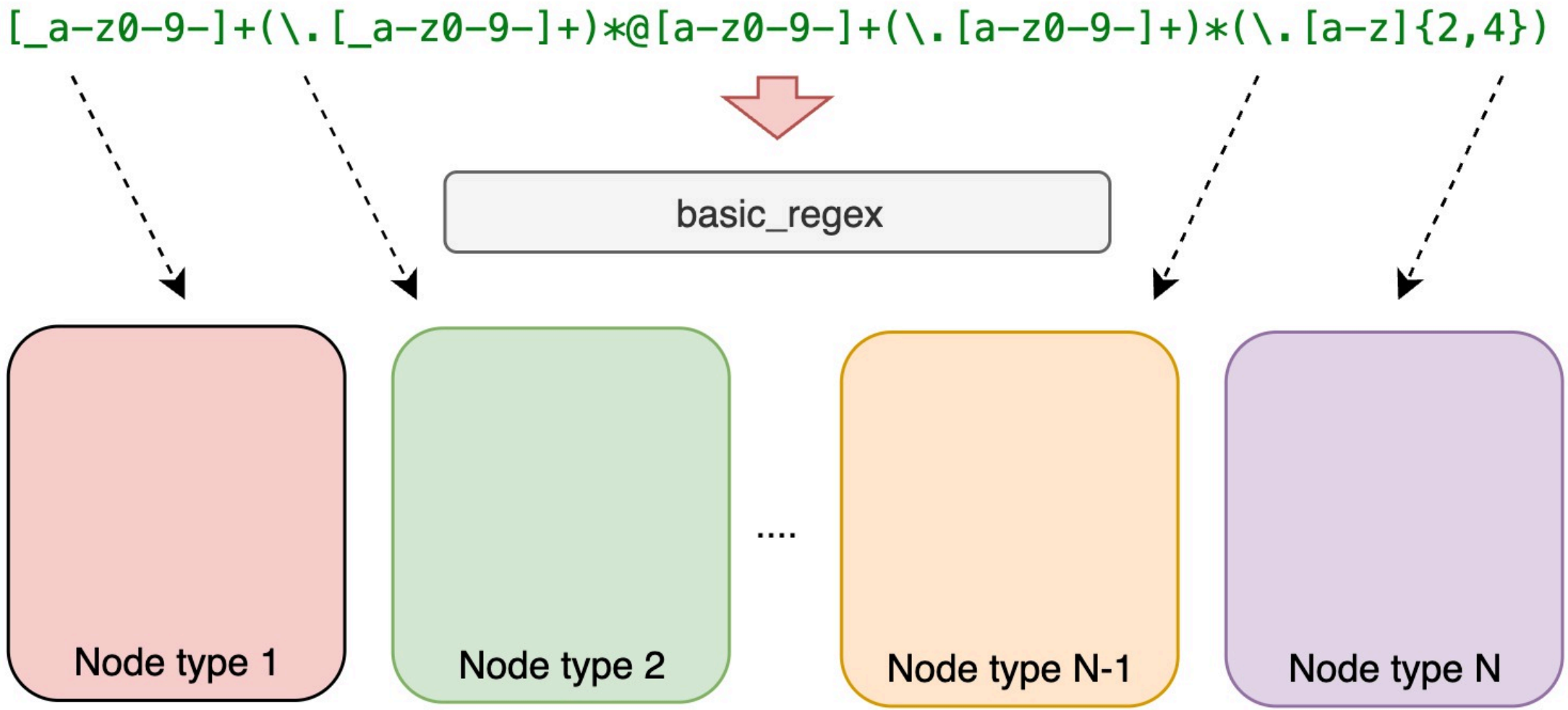
`std::basic_regex`: Creation (libc++)



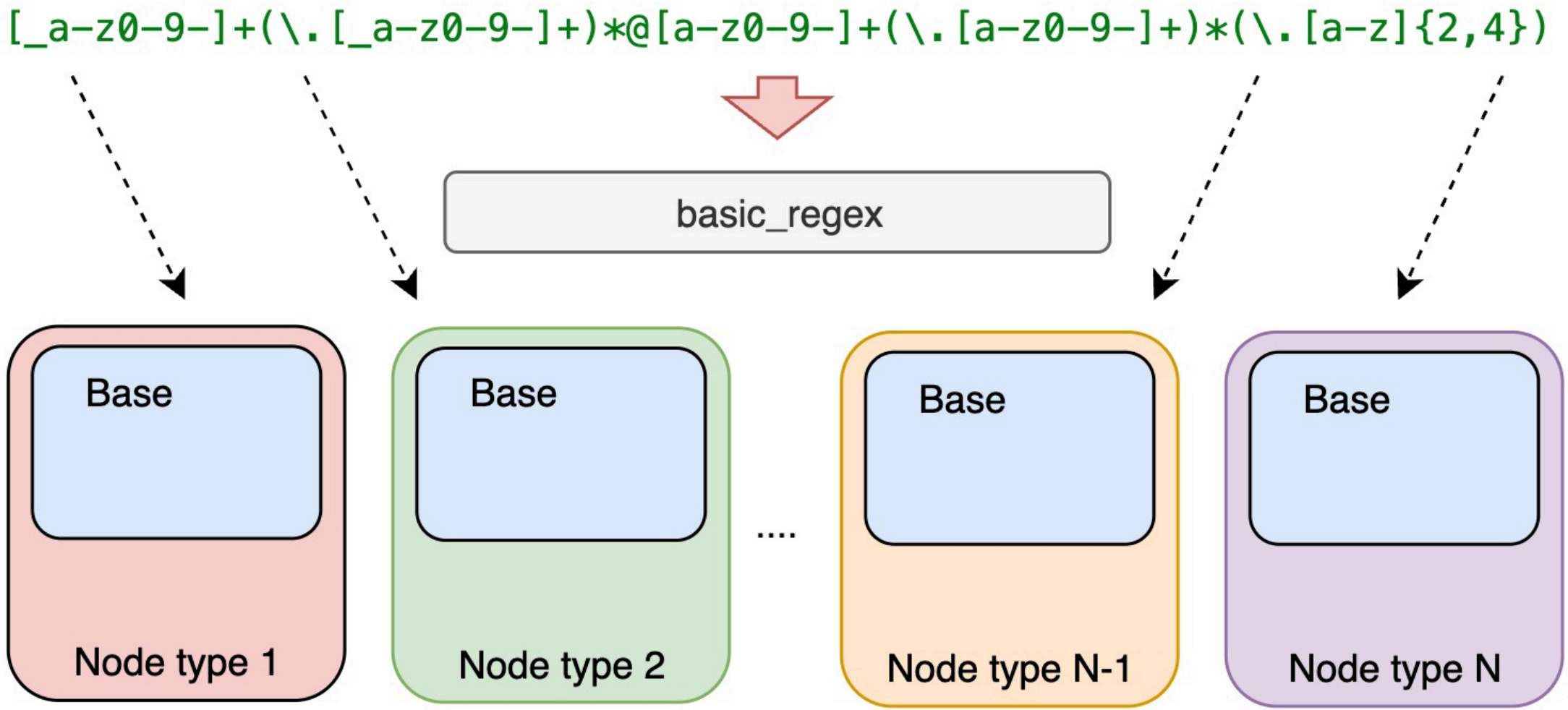
`std::basic_regex`: Creation (libc++)



`std::basic_regex`: Creation (libc++)

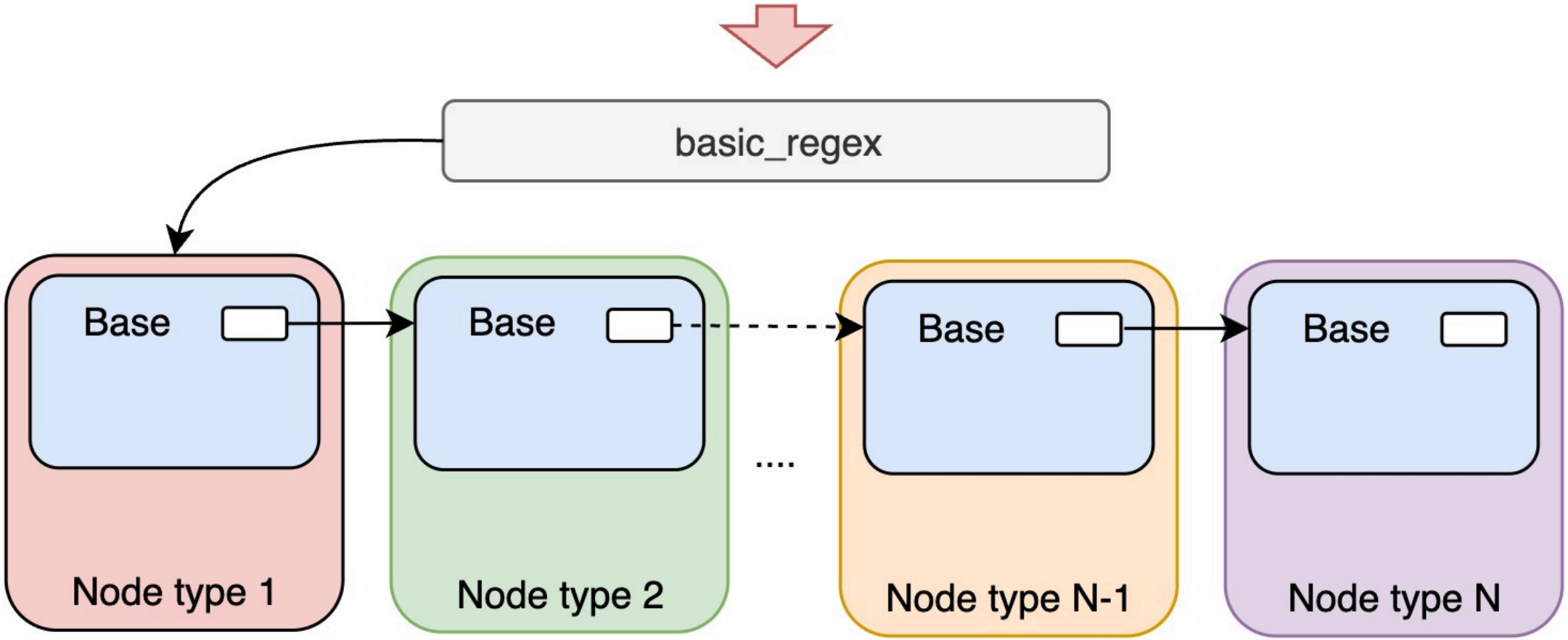


`std::basic_regex`: Creation (libc++)

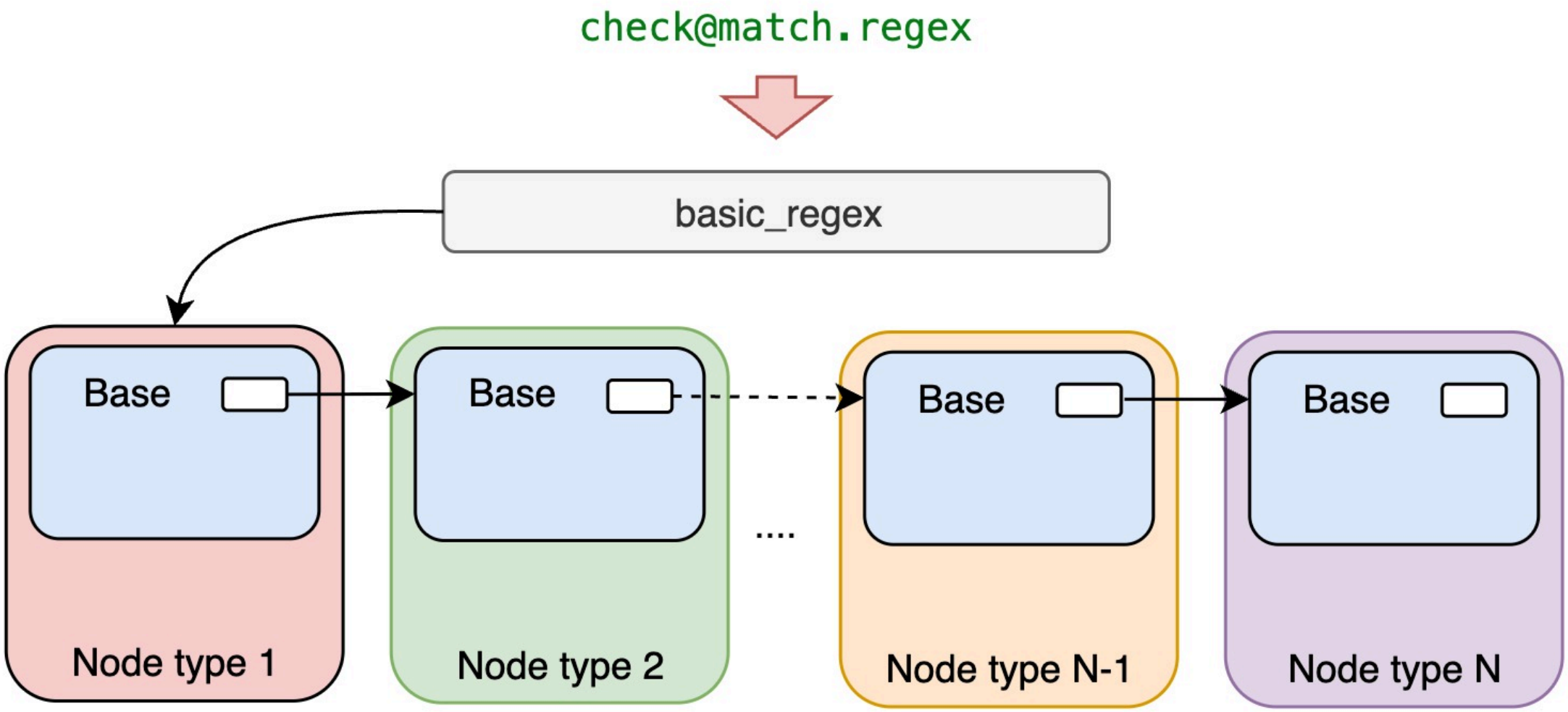


`std::basic_regex`: Creation (libc++)

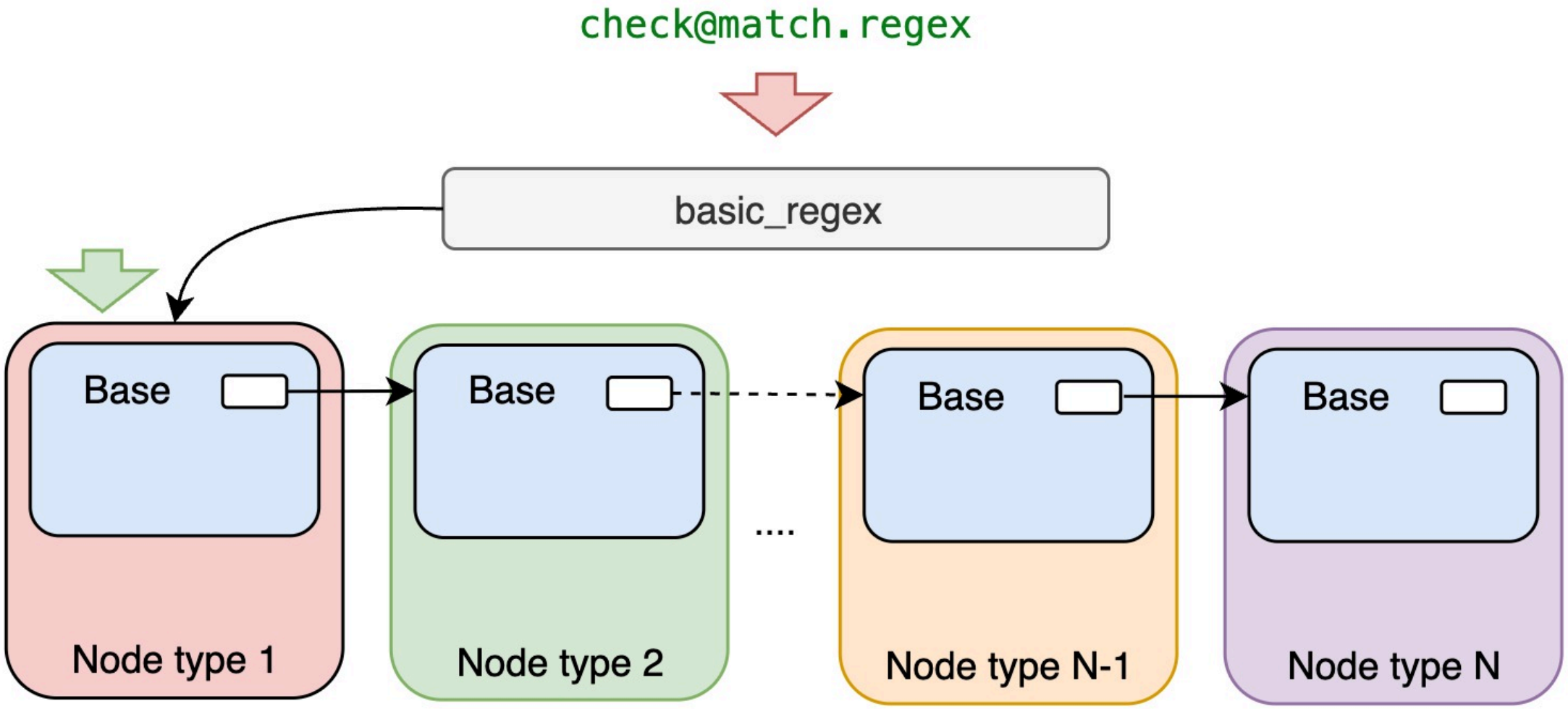
```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```



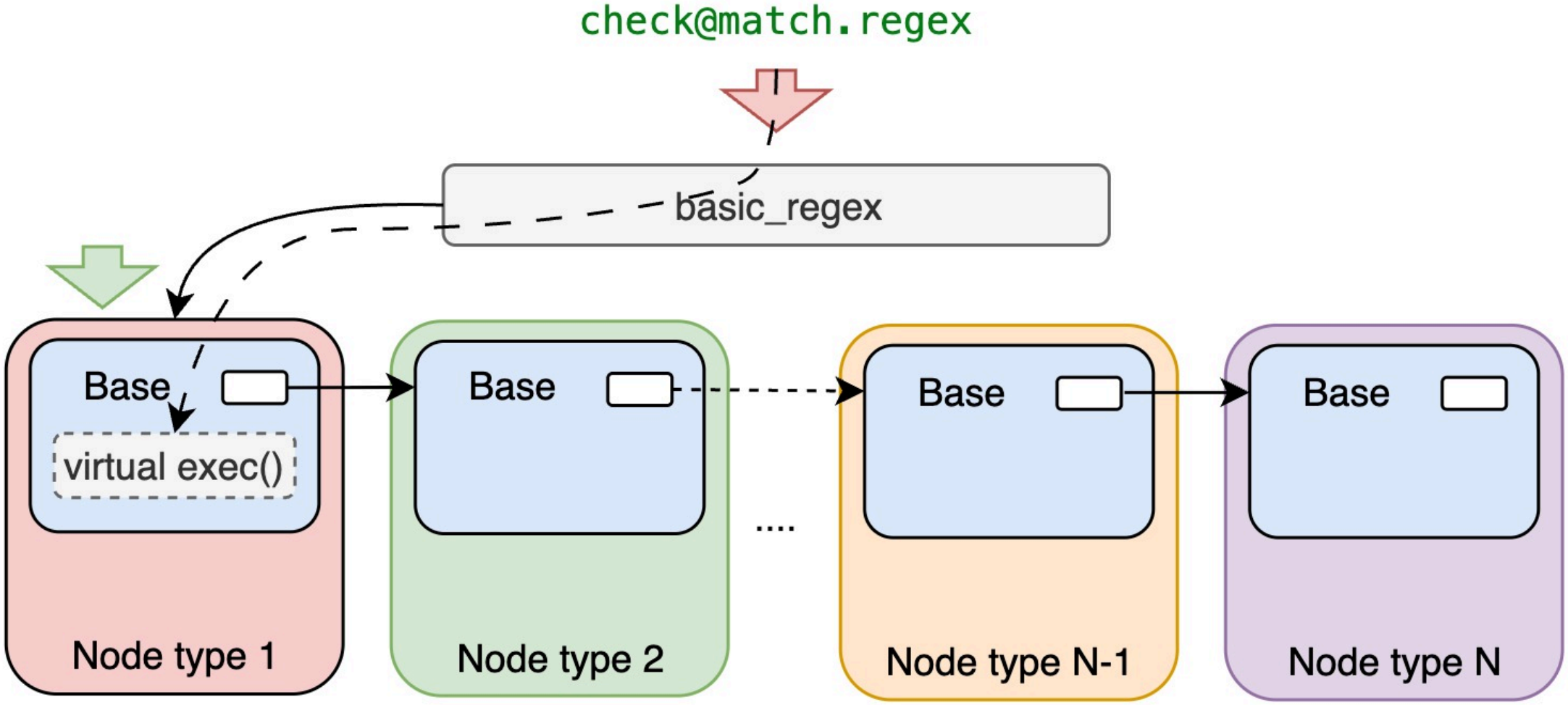
`std::basic_regex`: `regex_match` (libc++)



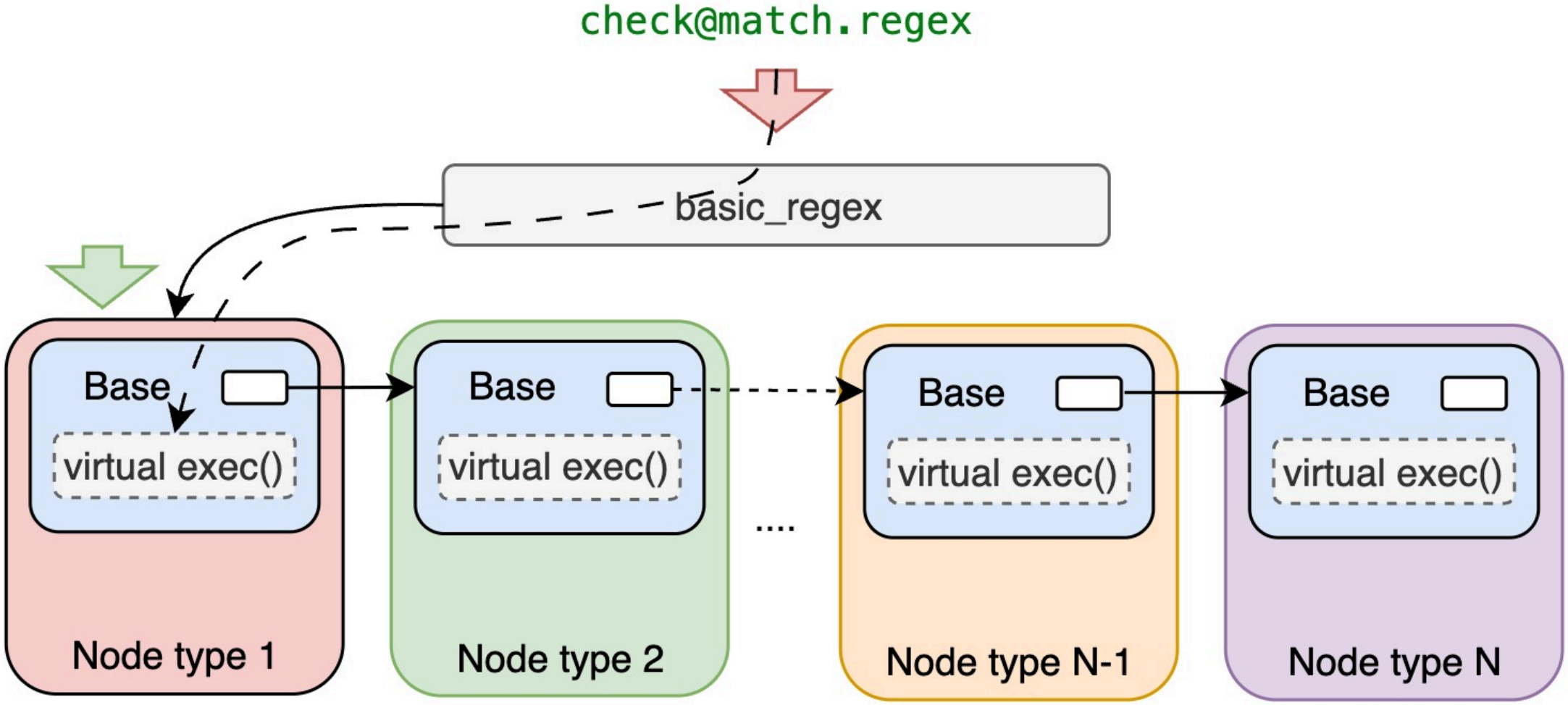
`std::basic_regex`: `regex_match` (libc++)



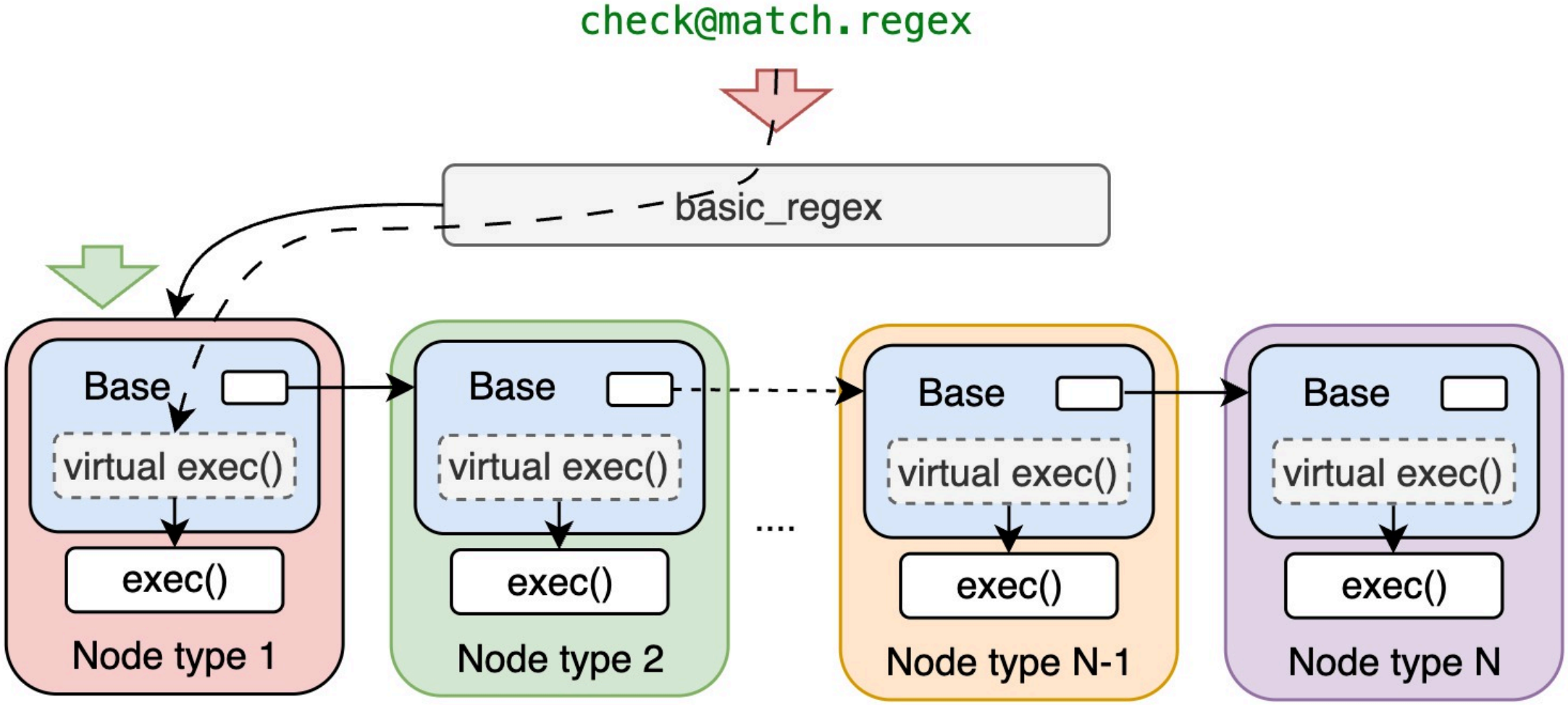
`std::basic_regex`: `regex_match` (libc++)



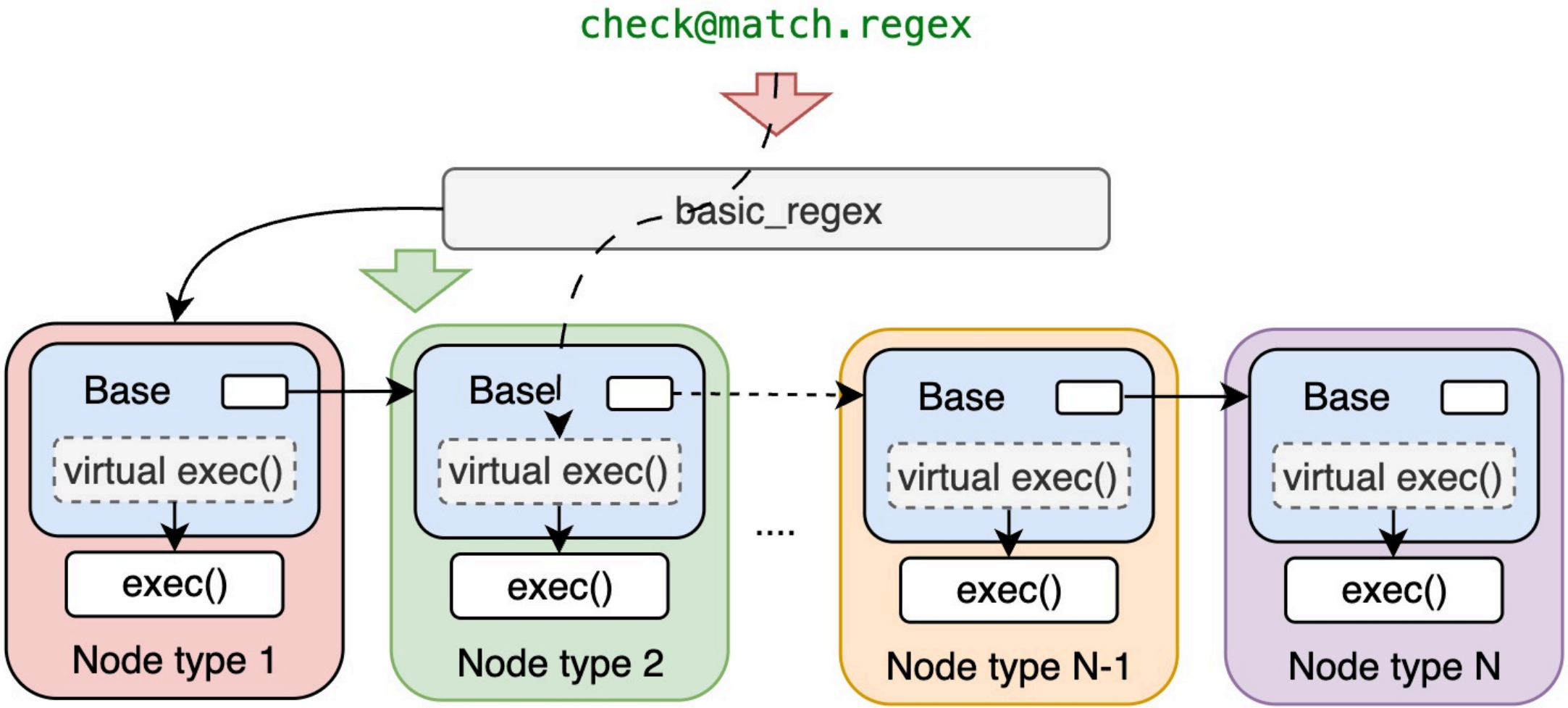
`std::basic_regex`: `regex_match` (libc++)



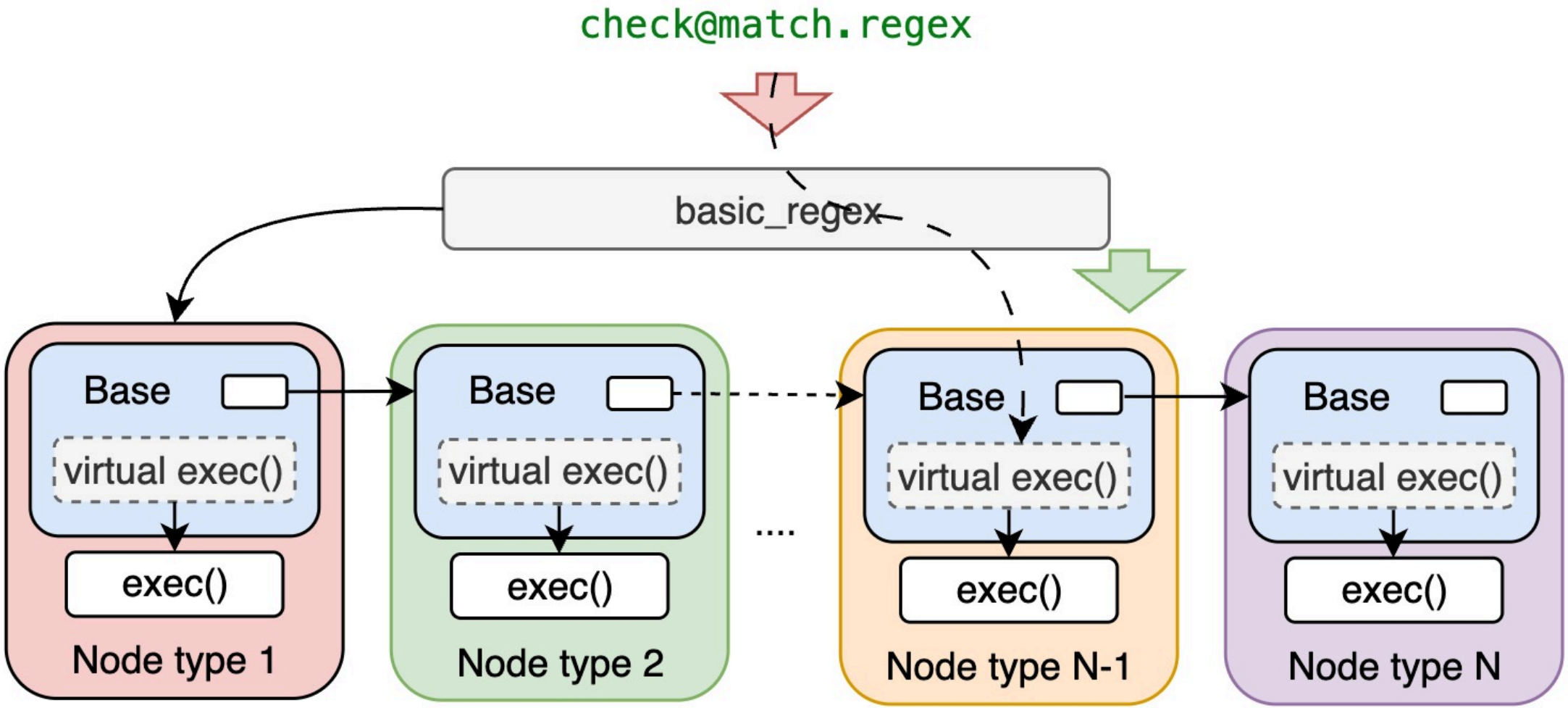
`std::basic_regex`: `regex_match` (libc++)



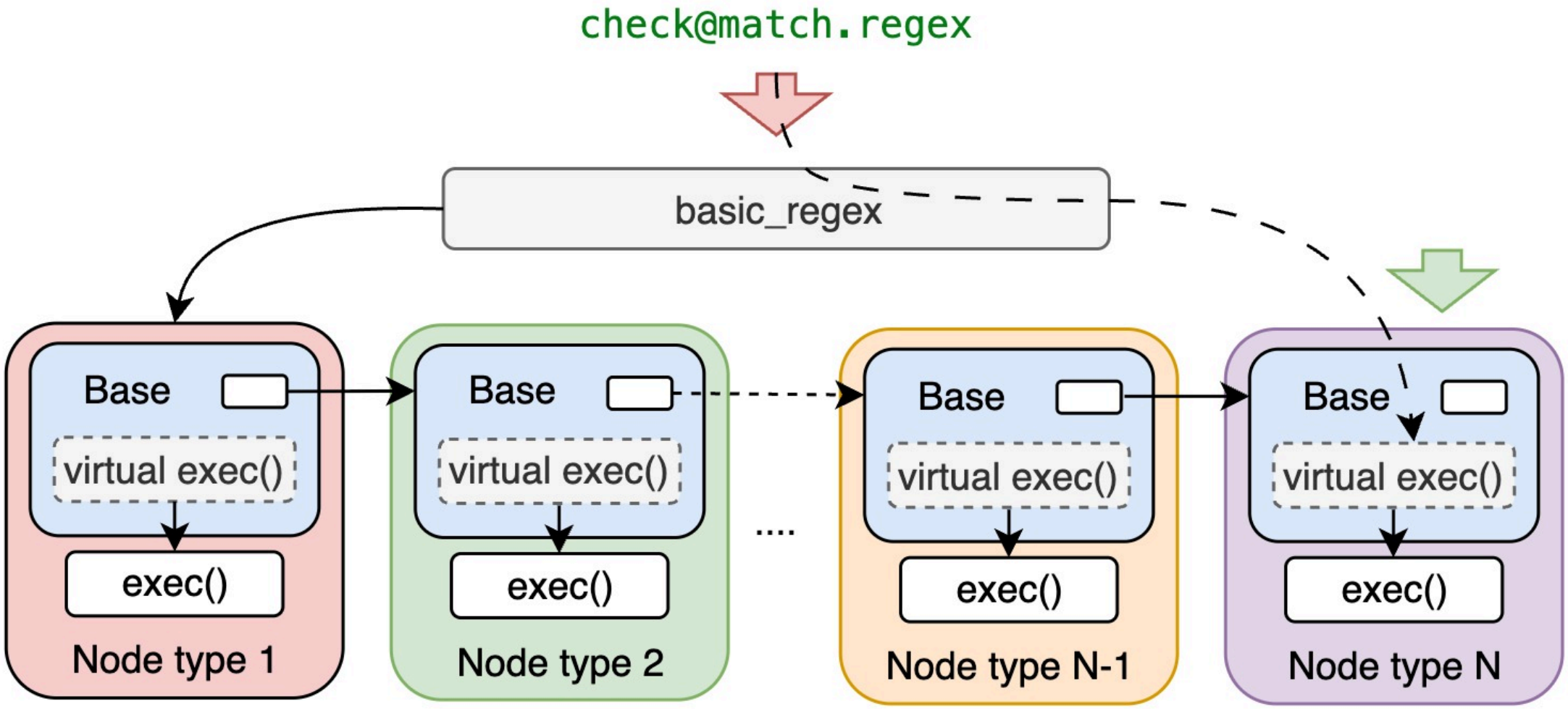
`std::basic_regex`: `regex_match` (libc++)



`std::basic_regex`: `regex_match` (libc++)



`std::basic_regex`: `regex_match` (libc++)



`constexpr` `std::basic_regex`: Requirements

`constexpr` `std::basic_regex`: Requirements

- **Avoid non-** `constexpr` **-compatible constructs**:
  - Mark **all** methods as `constexpr`.
  - Do **not** use `reinterpret_cast` or other incompatible operations.
  - Wait for `constexpr` support in `std::shared_ptr`.
- **Make locale-dependent functions** `constexpr` **(ASCII only):**
  - Implement `tolower`, `toupper` as `constexpr`
  - Provide a **table** of **caracter flags** (e.g., digit, alpha, etc.) usable at compile-time.
- Create nodes **via allocator**.

`constexpr` `std::basic_regex`: Nodes Creation

`constexpr` `std::basic_regex`: Nodes Creation

```
template <class _CharT, class _Traits>
void basic_regex<_CharT, _Traits>::__push_match_any() {




    __end_->first() = new __match_any<_CharT>(__end_->first());

    __end_          = static_cast<__owns_one_state<_CharT>*>(__end_->first());
}
```

`constexpr` `std::basic_regex`: Nodes Creation

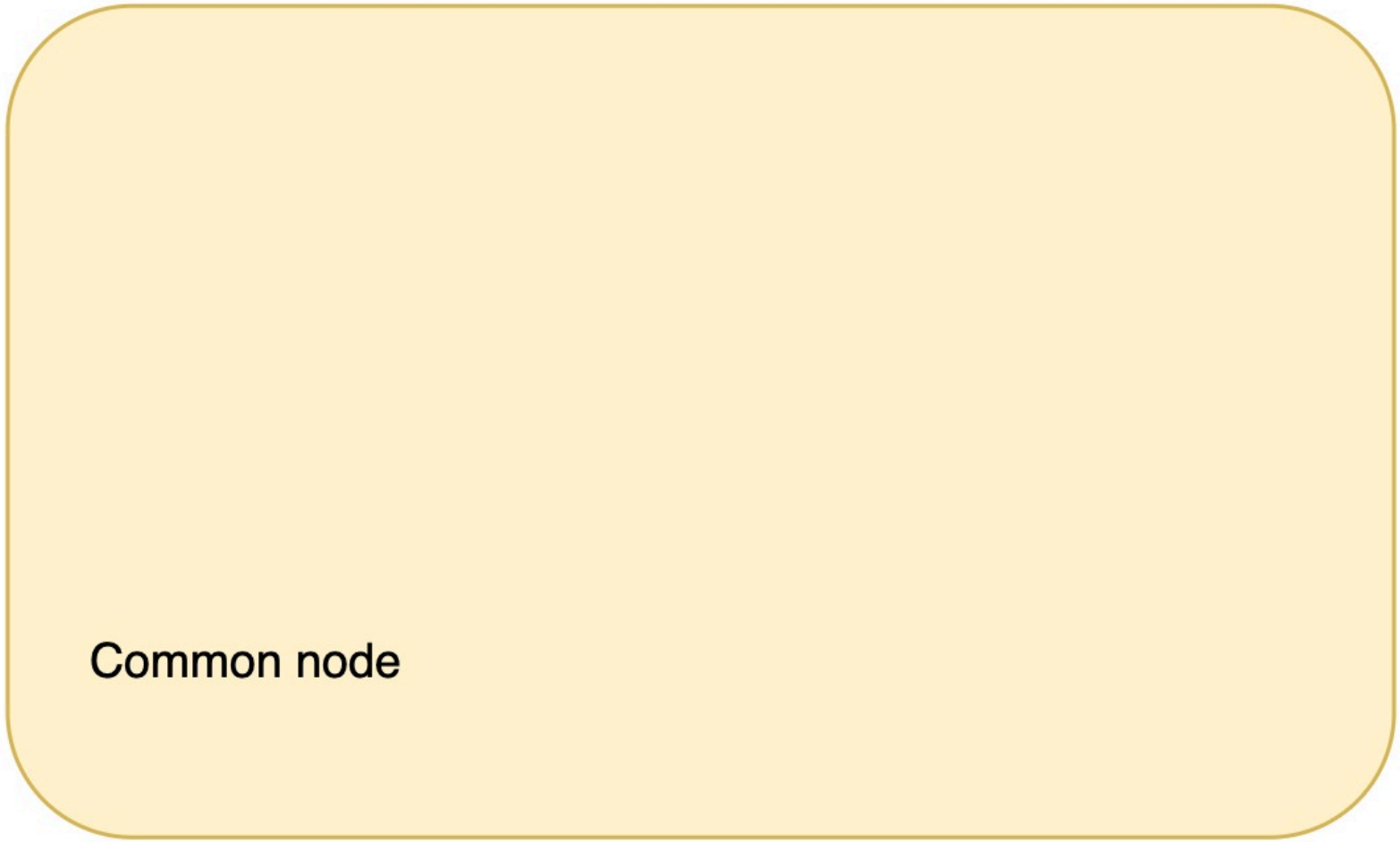
```
// template <class _CharT, class _Traits, class _Alloc> - break ABI
template <class _CharT, class _Traits>
void basic_regex<_CharT, _Traits>::__push_match_any() {




    __end_->first() = new __match_any<_CharT>(__end_->first());

    __end_          = static_cast<__owns_one_state<_CharT>*>(__end_->first());
}
```

`constexpr` `std::basic_regex`: Nodes Creation

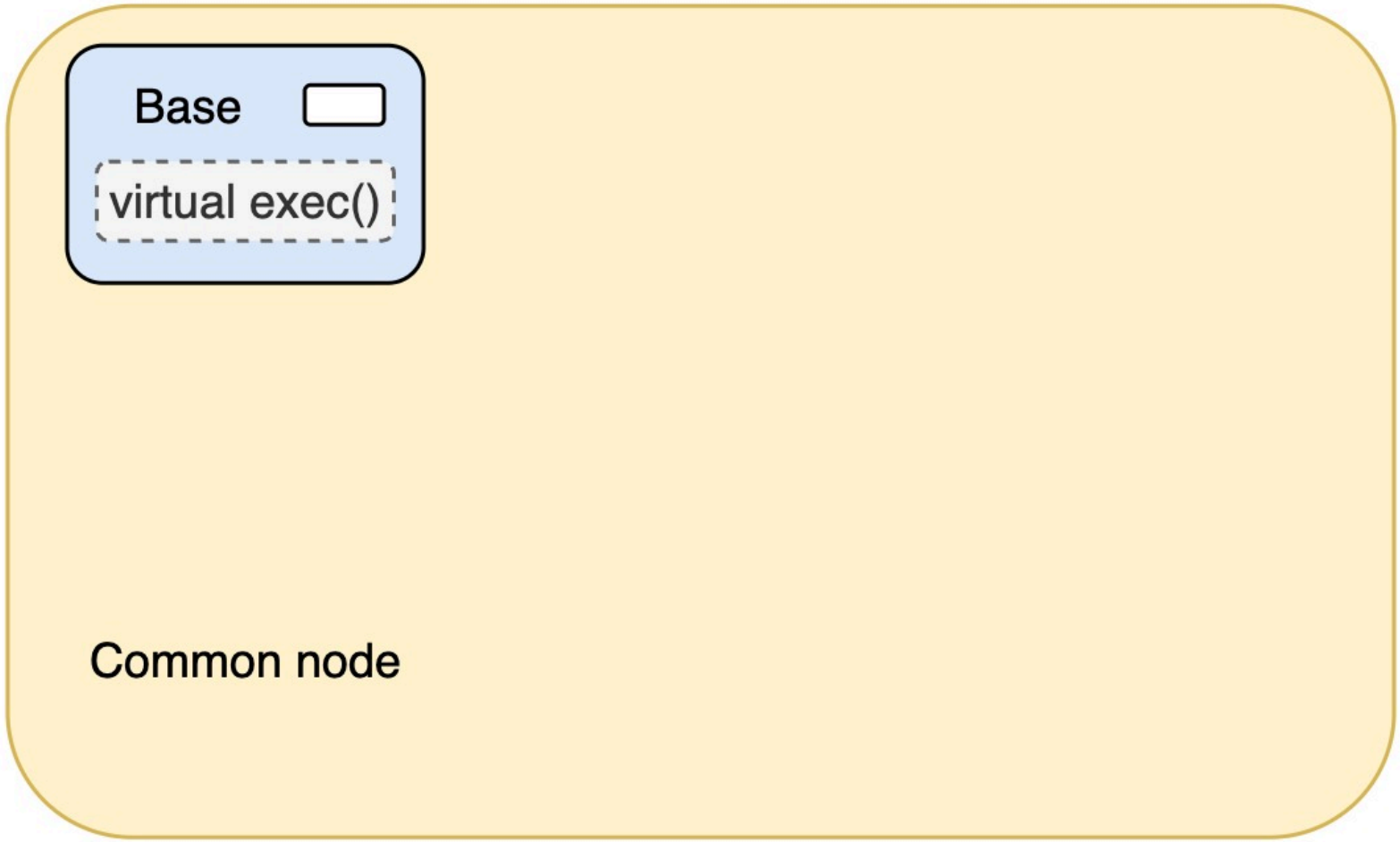
```
// template <class _CharT, class _Traits, class _Alloc> - break ABI
template <class _CharT, class _Traits>
void basic_regex<_CharT, _Traits>::__push_match_any() {
  if constexpr (requires { _Traits::node_traits; }) {
    __end_->first() =
            __traits.node_traits.create<__match_any<_CharT>>(__end_->first());
  }
  else {
    __end_->first() = new __match_any<_CharT>(__end_->first());
  }
  __end_            = static_cast<__owns_one_state<_CharT>*>(__end_->first());
}
```

`constexpr` `std::basic_regex`: Common Node

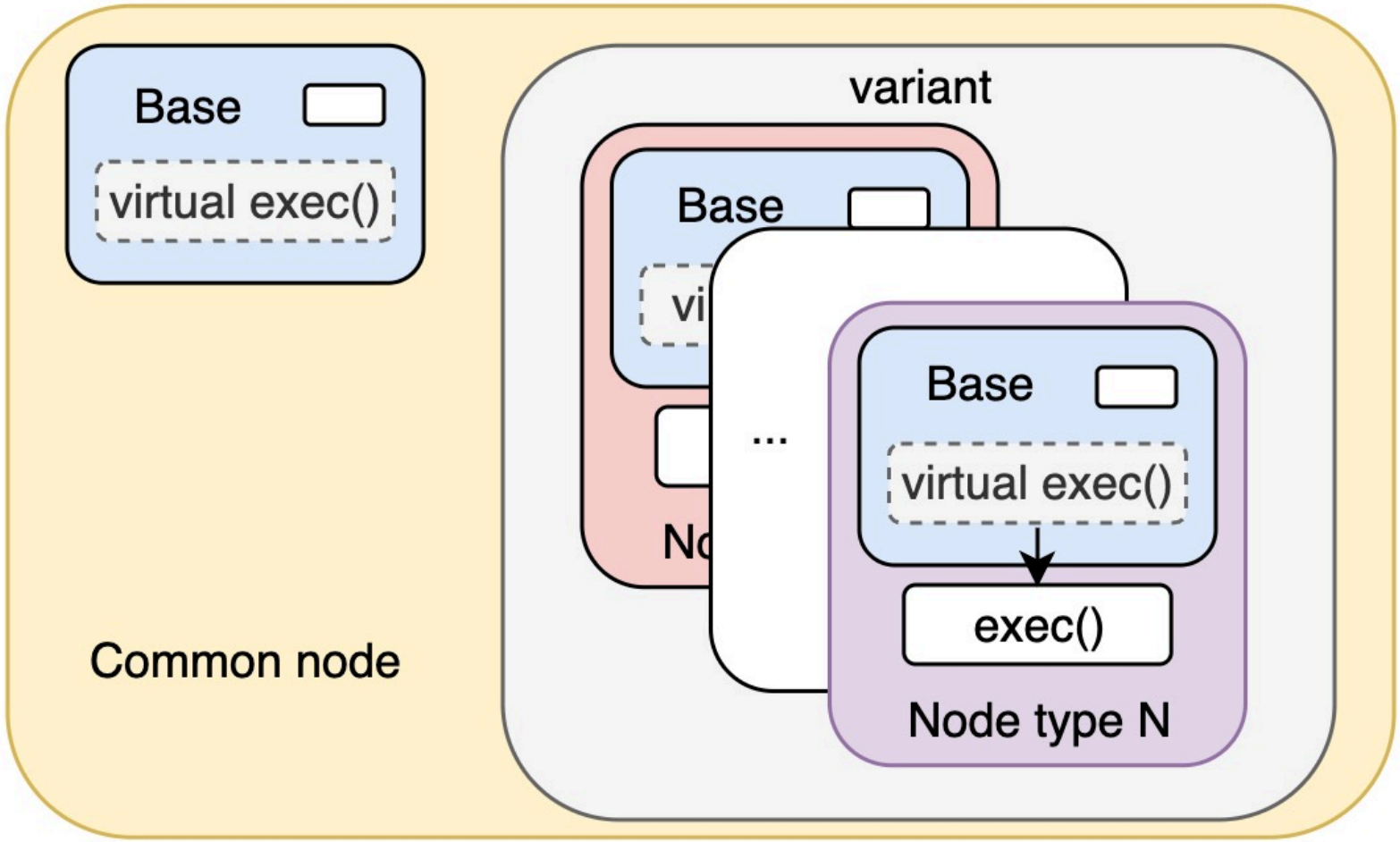
`constexpr` `std::basic_regex`: Common Node



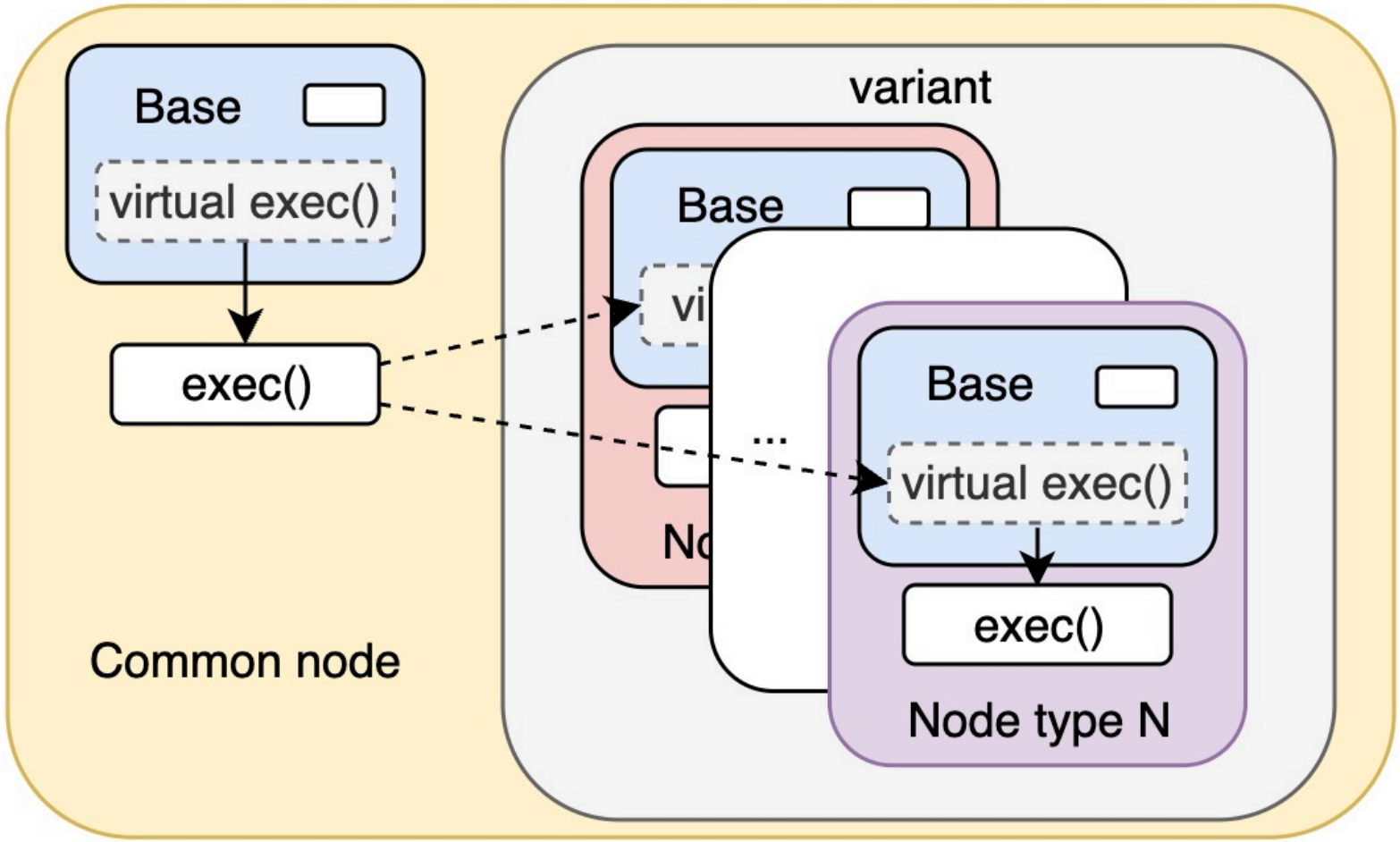
`constexpr` `std::basic_regex`: Common Node



`constexpr` `std::basic_regex`: Common Node

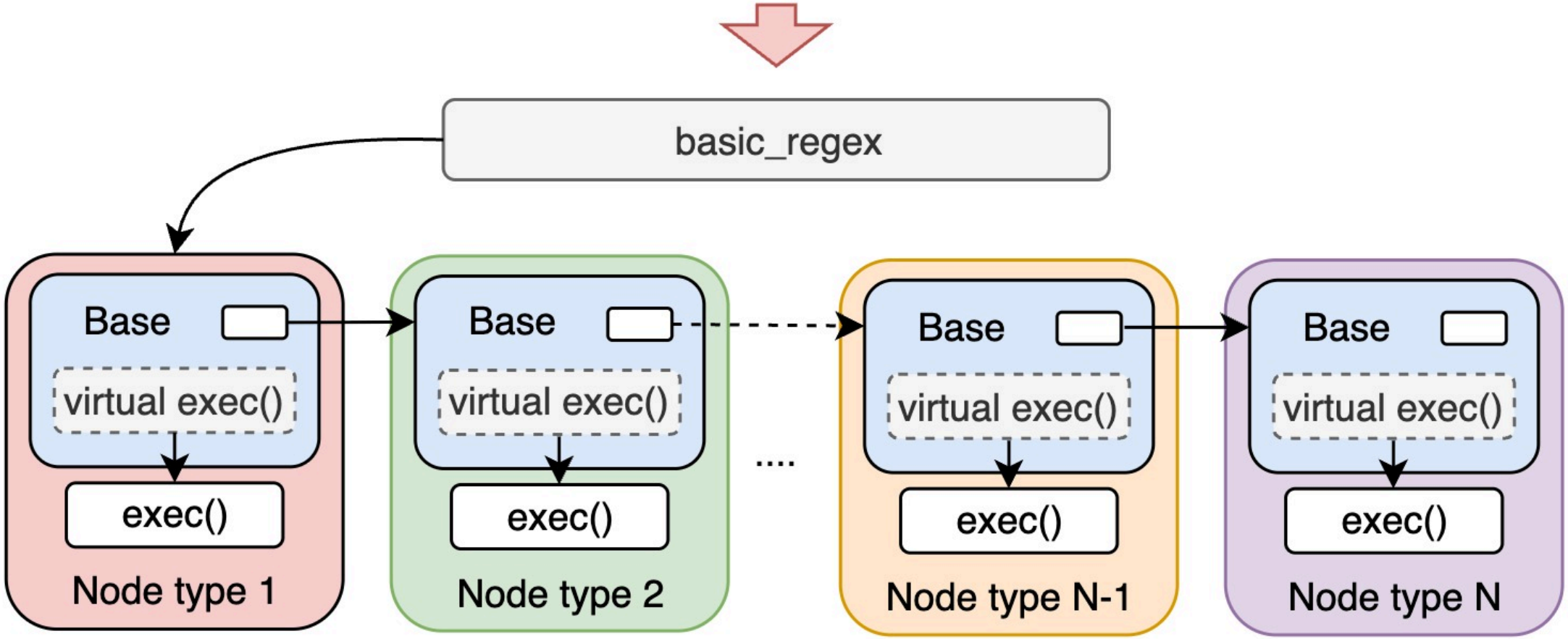


`constexpr` `std::basic_regex`: Common Node



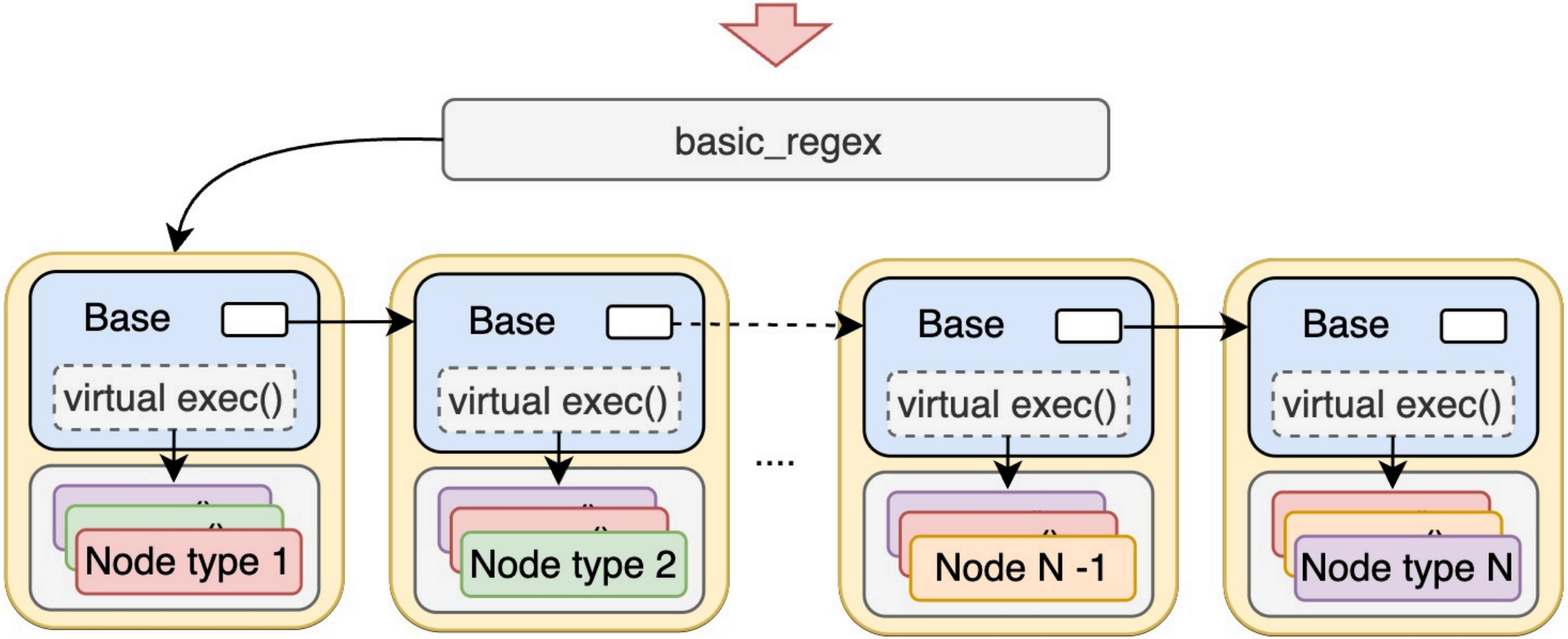
`constexpr` `std::basic_regex`: Common Node

```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```



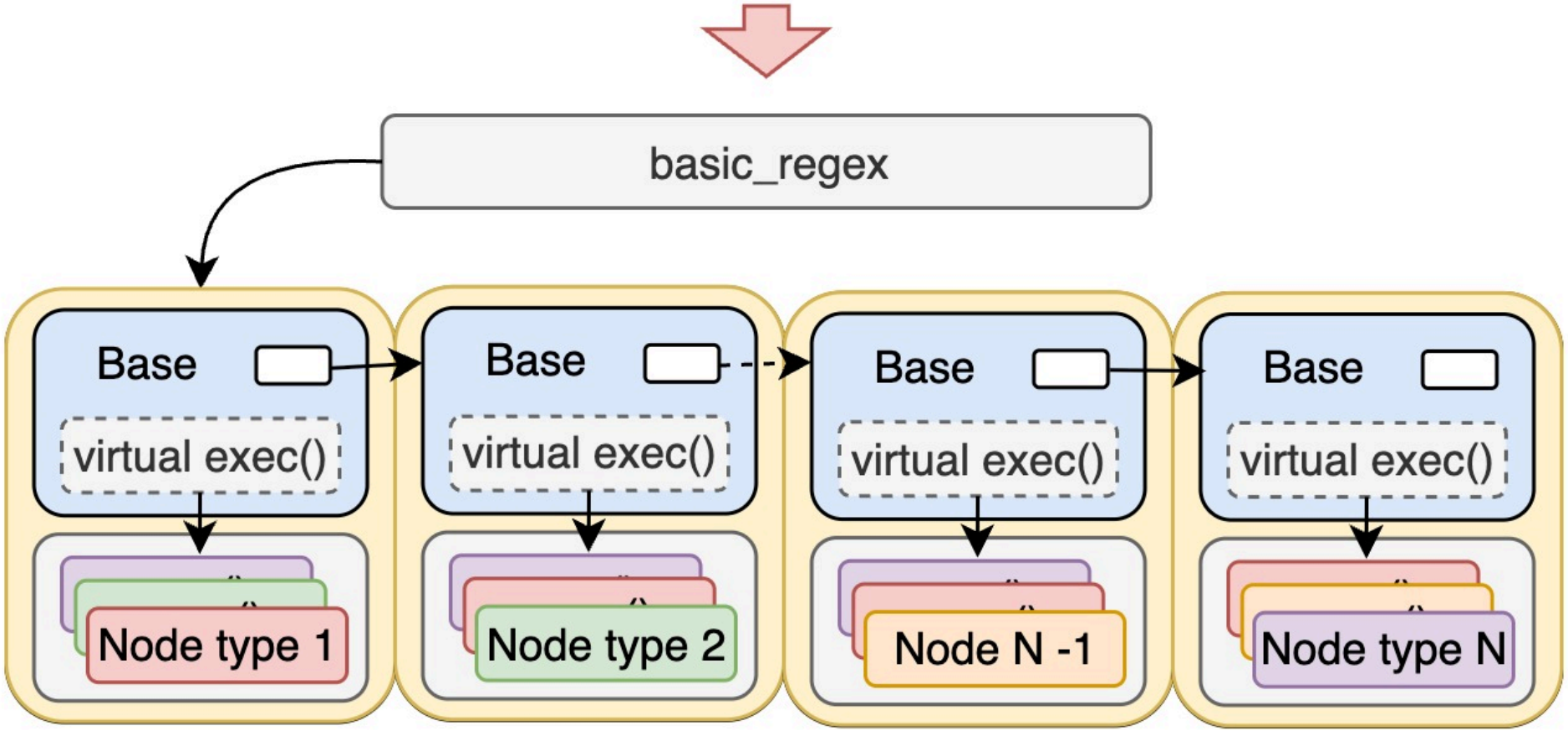
`constexpr` `std::basic_regex`: Common Node

```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```



`constexpr` `std::basic_regex`: Common Node

```
[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})
```



`constexpr` `std::basic_regex`: Requirements

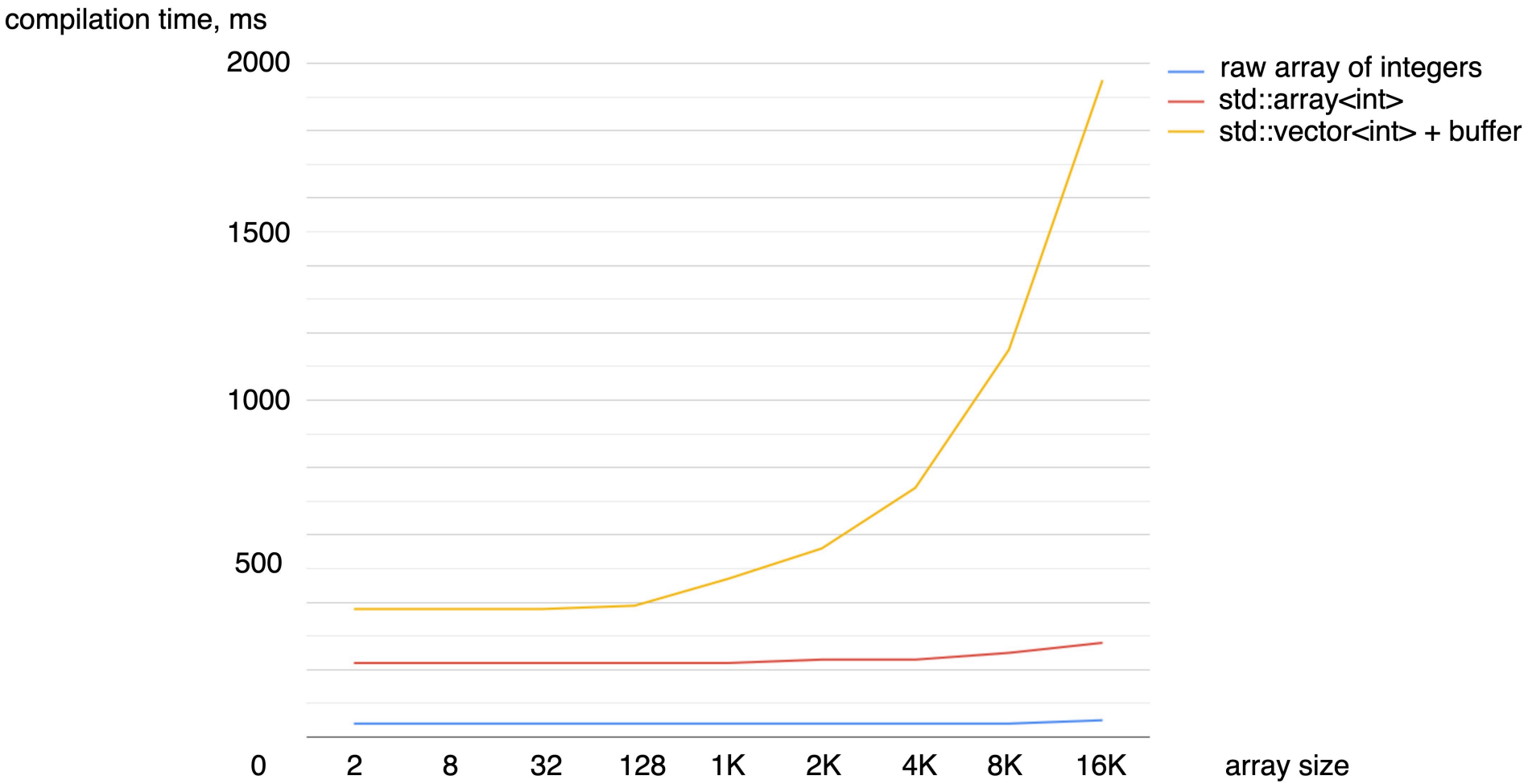
- **Avoid non-`constexpr`-compatible constructs**:
  - Mark **all** methods as `constexpr`.
  - Do **not** use `reinterpret_cast` or other incompatible operations.
  - Wait for `constexpr` support in `std::shared_ptr`.
- **Make locale-dependent functions `constexpr` (ASCII only):**
  - Implement `tolower`, `toupper` as `constexpr`
  - Provide a **table** of **caracter flags** (e.g., digit, alpha, etc.) usable at compile-time.
- **Node creation via allocator:**
  - Allow traits to **optionally** provide a node allocator (for backward compatibility).
  - Add a `std::basic_regex` constructor that accepts an allocator.

Results

Results

- Works on Clang and MSVC compilers.
- Does **not** work on GCC
  (due to bug https://gcc.gnu.org/bugzilla/show_bug.cgi?id=115233).

Results



Duration of the compiling a file with different data structures (Apple M2 Pro, clang 15.0.0)

Results

- Works on Clang and MSVC compilers.
- Does **not** work on GCC
  (due to bug https://gcc.gnu.org/bugzilla/show_bug.cgi?id=115233).
- constexpr `std::unordered_map` works correctly with this allocator.
- Memory overhead is approximately 200 bytes per container instance.

Thank you for your attention!

Feel free to reach out:

Telegram: @sssersh