Functions and Algorithms

Game Plan

- concept lifting part 2
- lambda functions
- algorithms
- iterator adaptors

concept lifting part 2

How many times does the [type] [val] appear in [a range of elements]?

```
template <typename InputIt, typename DataType>
int countOccurences(InputIt begin, InputIt end,
                                   DataType val) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) ++count;
  }
  return count;
}
```

How many times does the [type] [val] appear in [a range of elements]?

```
template <typename InputIt, typename DataType>
int countOccurences(InputIt begin, InputIt end,
                                   DataType val) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) ++count;
  }
  return count;
}
```

Let’s look at this part.

How many times does the element satisfy "equal [val]" in [a range of elements]?

```
template <typename InputIt, typename DataType>
int countOccurences(InputIt begin, InputIt end,
                                   DataType val) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) ++count;
  }
  return count;
}
```

This is another way to phrase what we are counting.

A predicate is a function which takes in some number of arguments and returns a boolean.

```
// Unary Predicate (one argument)
bool isEqualTo3(int val) {
    return val == 3;
}

// Binary Predicate (two arguments)
bool isDivisibleBy(int dividend, int divisor) {
    return dividend % divisor == 0;
}
```

How many times does the element satisfy [predicate] in [a range of elements]?

```
template <typename InputIt, typename DataType,
          typename UniPred>
int countOccurences(InputIt begin, InputIt end,
                                UniPred predicate) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (predicate(*iter)) ++count;
  }
  return count;
}
```

"equals [val]" is essentially a predicate function. Let's further generalize the function.

We can then call this function with a predicate.

```
bool isLessThan5(int val) {
    return val < 5;
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};

  countOccurences(vec.begin(), vec.end(), isLessThan5);
  // prints 2
  return 0;
}
```

Problem 1: what if we wanted to use some upper limit other than 5?

```
bool isLessThan5(int val) {
    return val < 5;
}

bool isLessThan6(int val) {
    return val < 6;
}

bool isLessThan7(int val) {
    return val < 7;
}
```

Problem 2: scope issue with having a variable limit in the calling function.

```
bool isLessThanLimit(int val) {
    return val < limit; // out of scope!
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;
  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
  return 0;
}
```

Problem 3: we can’t add an extra parameter to the predicate function.

```
bool isLessThanLimit(int val, int limit) {
    return val < limit; // not out of scope, but...
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;
  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
  return 0;                          // template error!
}
```

Predicate must be a unary predicate because of how we use it in countOccurences.

```
template <typename InputIt, typename DataType,
          typename UniPred>
int countOccurences(InputIt begin, InputIt end,
                                UniPred predicate) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (predicate(*iter)) ++count;
  }
  return count;
}
```

The core fundamental issue is about scope!

```
bool isLessThanLimit(int val) {
    return val < limit; // out of scope!
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;
  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
  return 0;
}
```

lambda functions

Old approach: function pointers

```
bool isLessThanLimit(int val) {
    return val < limit; // compiler error!
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 5;



  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
  return 0;
}
```

New approach: lambda functions

```
bool isLessThanLimit(int val) {
    return val < 5;
}

int main() {
   vector<int> vec{1, 3, 5, 7, 9};
   int limit = 5;
   auto isLessThanLimit = [limit](auto val) -> bool {
      return val < limit;
   }
   countOccurences(vec.begin(), vec.end(), isLessThanLimit);
   return 0;
}
```

New approach: lambda functions

```
bool isLessThanLimit(int val) {
    return val < 5;
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 5;
  auto isLessThanLimit = [limit](auto val) -> bool {
    return val < limit;
  }
  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
  return 0;
}
```

New approach: lambda functions

We don’t know the type, ask compiler.

capture clause, gives access to outside variables

parameter list, can use auto!

return type, optional

```
auto isLessThanLimit = [limit](auto val) -> bool {
  return val < limit;
}
```

Scope of lambda limited to capture clause and parameter list.

You can also capture by reference.

```
set<string> teas{“black”, “green”, “oolong”};
string banned = “boba”; // pls … this is not a tea
auto likedByAvery = [&teas, banned](auto type) {
    return teas.count(type) && type != banned;
};
```

You can also capture everything by value or reference.

```
// capture all by value, except teas is by reference
auto func1 = [=, &teas](parameters) -> return-value {
    // body
};

// capture all by reference, except banned is by value
auto func2 = [&, banned](parameters) -> return-value {
    // body
};
```

FYI, std::function<R(Args…)> is a generic wrapper for all things callable.

generally prefer auto or template deduction for functions, since std::function has a performance problem

```
int main() {
  std::function<void()> func1 = []() { return 137; };
  std::function<bool(int)> func2 = isLessThanLimit;

  std::function< int(iterator, iterator,
                     std::function<bool(int)>) >
     = countOccurences< v.begin(), v.end(), func2 >;
}
```

Lambdas are a type of function object (“functor”)

```
{
  auto mult = [](int param, int factor) {
     return param * factor;
  };

  // call mult’s () operator, like a function
  auto val = mult(3, 2);  // val is 6

  // bind takes a functor and returns a functor
  auto multBound = std::bind(mult, _1, 2);
} // destructor for mult called
```

Is there a way we can adapt this function we have to be usable in our generic function?

```
bool isLessThanLimit(int val, int limit) {
    return val < limit; // not out of scope, but...
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;



  countOccurences(vec.begin(), vec.end(), isLessThanLimit);
}
```

Solution 1: Write a lambda which wraps the call to isLessThanLimit.

```
bool isLessThanLimit(int val, int limit) {
    return val < limit; // not out of scope, but...
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;
  auto isLessThan = [limit] (int val) {
      return isLessThanLimit(val, limit);
  };
  countOccurences(vec.begin(), vec.end(), isLessThan);
}
```

Solution 2: std::bind, basically does the same thing.

```
bool isLessThanLimit(int val, int limit) {
    return val < limit; // not out of scope, but...
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 8;
  auto isLessThan = std::bind(isLessThanLimit, _1, limit);


  countOccurences(vec.begin(), vec.end(), isLessThan);
}
```

algorithms

The STL algorithms library has a highly optimized version of what we wrote!

C++ | Algorithm library

std::count, std::count_if

Defined in header <algorithm>

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T &value );                    (1)  (until C++20)

template< class InputIt, class T >
constexpr typename iterator_traits<InputIt>::difference_type
              count( InputIt first, InputIt last, const T &value );                    (since C++20)

template< class ExecutionPolicy, class ForwardIt, class T >
typename iterator_traits<ForwardIt>::difference_type
    count( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, const T &value );   (2)  (since C++17)

template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
    count_if( InputIt first, InputIt last, UnaryPredicate p );               (3)  (until C++20)

template< class InputIt, class UnaryPredicate >
constexpr typename iterator_traits<InputIt>::difference_type
              count_if( InputIt first, InputIt last, UnaryPredicate p );               (since C++20)

template< class ExecutionPolicy, class ForwardIt, class UnaryPredicate >
typename iterator_traits<ForwardIt>::difference_type
    count_if( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryPredicate p );   (4)  (since C++17)
```

(this one's replaced in C++20)

(this one runs in parallel)

(this one's replaced in C++20)

(this one runs in parallel)

Province of Value Queries
Province of Property Queries
Province of 2-Ranges Properties
Glorious County of Algos on Sets
Lonely Islands
Land of Queries
Value Searchers
Peninsula of Raw Memory
Territory of Movers
Province of Research
Relative Value Searchers
Island of Structure Changers
Land of Value Modifiers
Land of Permutations
Secret Runes
The World of C++ STL Algorithms
Fluent C++
Fluent C++
fluentcpp.com

Let’s try some basic operations on information from Carta!

```
struct Course {
  string code;
  double rating;
};
```

Algorithms we will explore!

```
std::sort
std::nth_element
std::stable_partition
std::copy_if
std::remove_if
```

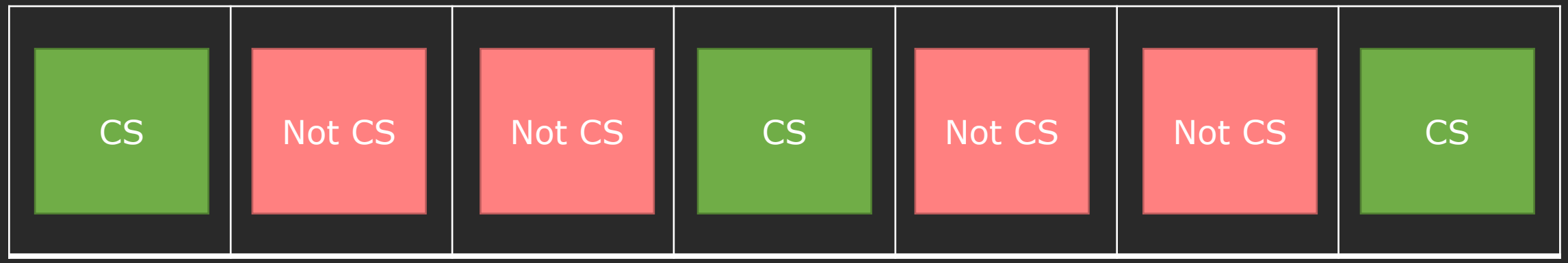
Calculate the median course rating. O(Nlog N)

```
auto compRating = [](const auto& s1,
                     const auto& s2) {
    return s1.rating < s2.rating;
};

size_t size = classes.size();

// O(N log N) sort
std::sort(classes.begin(), classes.end(), compAvg);

Course median = classes[size/2];
```

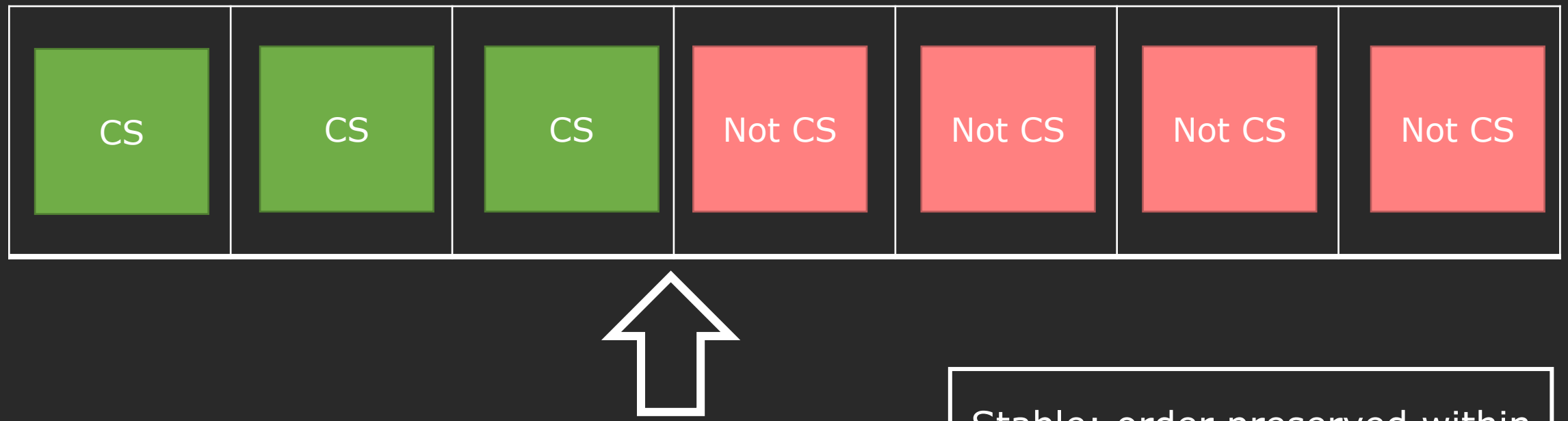
Calculate the median course rating. O(N)

```
auto compRating = [](const auto& s1,
                     const auto& s2) {
    return s1.rating < s2.rating;
};

size_t size = classes.size();

// O(N), sorts so nth_element is in correct position
// all elements smaller to left, larger to right
Course median = *std::nth_element(classes.begin(),
                 classes.end(),  size/2, compAvg);
```

What does stable partition do?



After a call to stable_partition:



Stable: order preserved within "CS" and "Not CS" group.

Using stable_partition.

```
string dep = "CS";
auto isDep = [dep](const auto& course) {
    return course.name.size() >= dep.size &&
           course.substr(0, dep.size()) == dep;
};

auto iter = std::stable_partition(courses.begin(),
                                  courses.end(), isDep);
courses.erase(iter, courses.end());
```

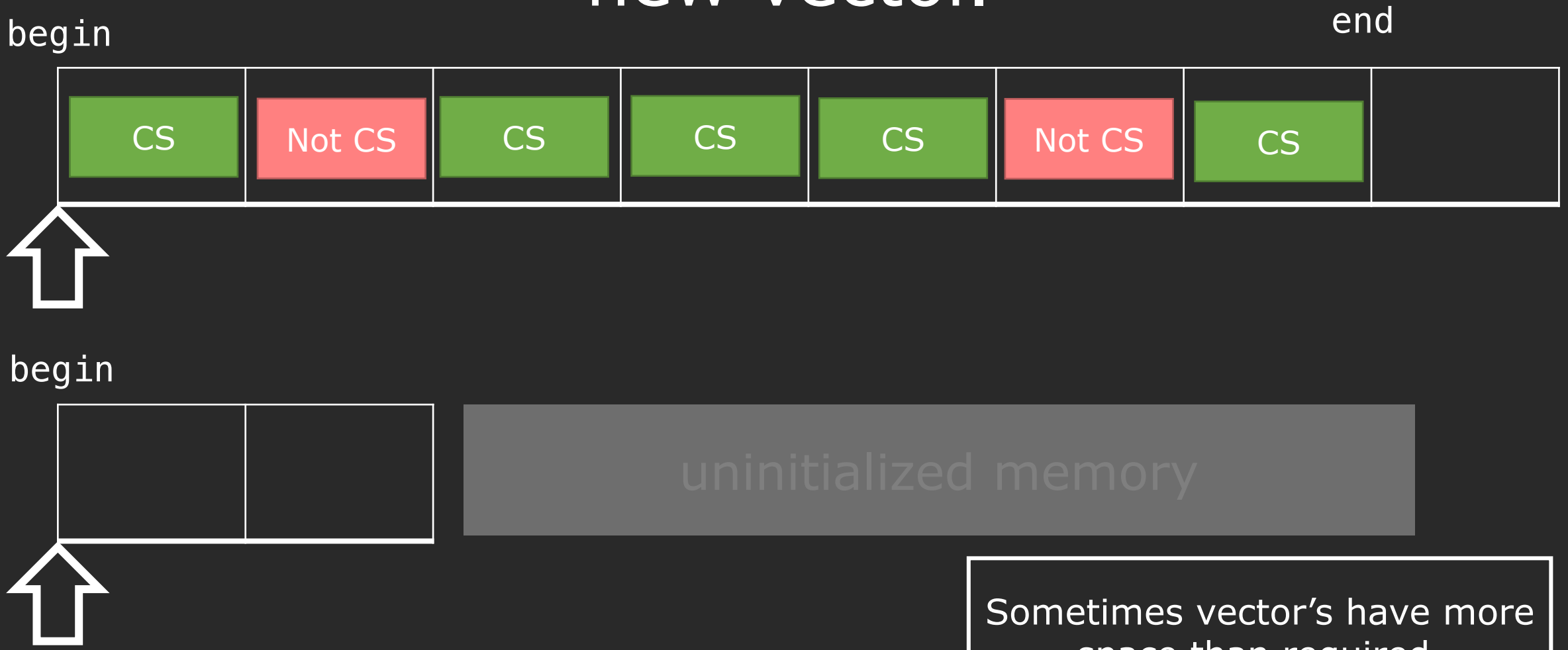
Why use algorithms?

- Abstraction: perform algorithms without looking at elements.
- Generic: operations are based on ranges, not containers.
- Correct: heavily tested, most definitely correct.
- Heavily optimized: performs optimizations using features we haven’t/won’t even learn about.

This code unfortunately doesn’t work!

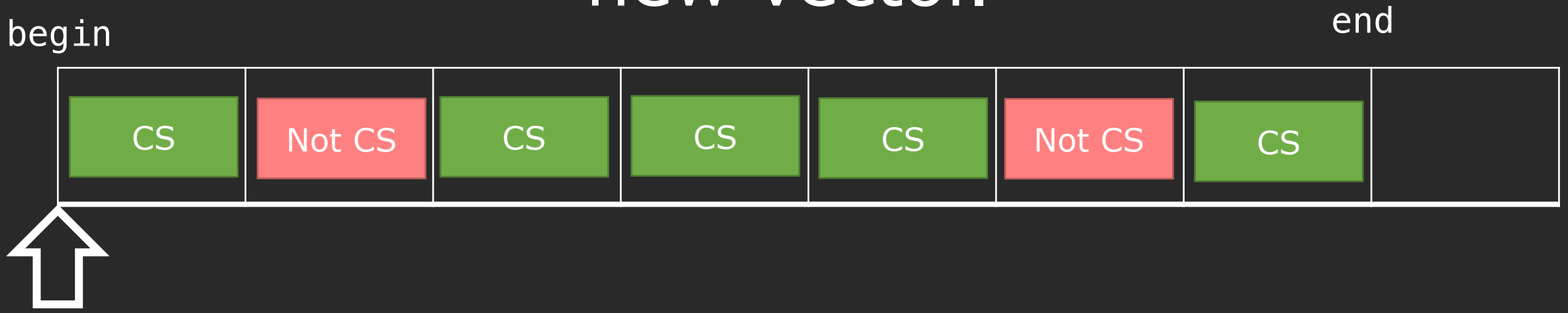
```
string dep = ”CS”;
auto isDep = [dep](const auto& course) {
    return course.name.size() >= dep.size &&
           course.substr(0, dep.size()) == dep;
};


std::copy_if(csCourses.begin(), csCourses.end(),
           csCourses, isDep);
```

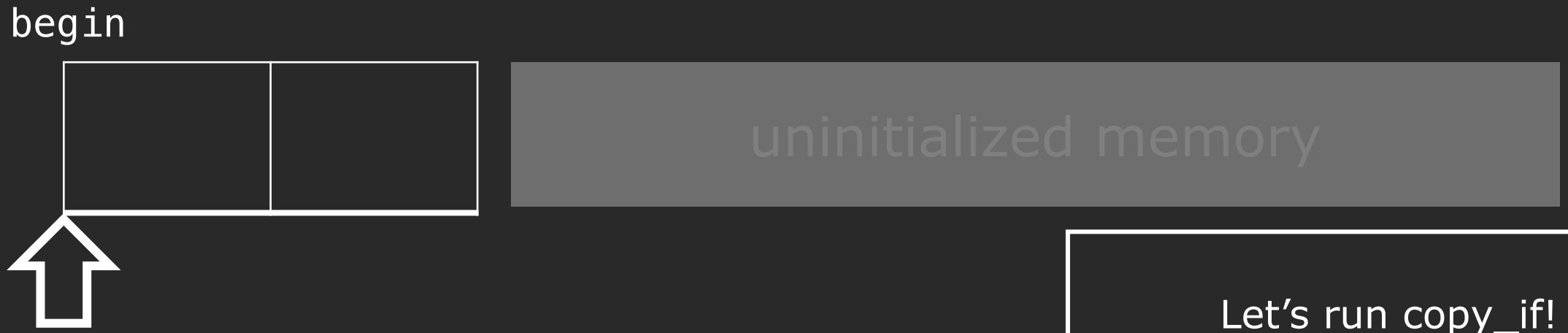
Copy all the CS courses into a new vector.



Sometimes vector's have more space than required.

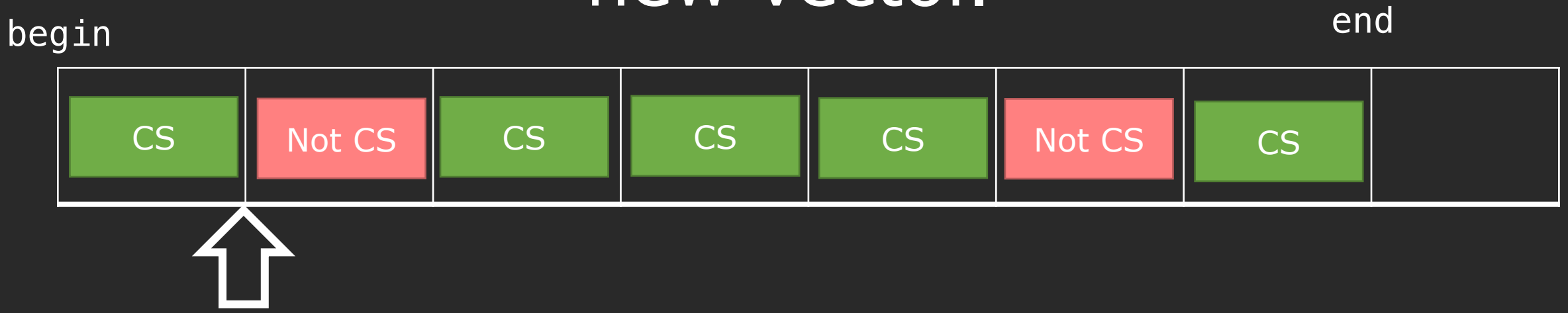
Copy all the CS courses into a new vector.

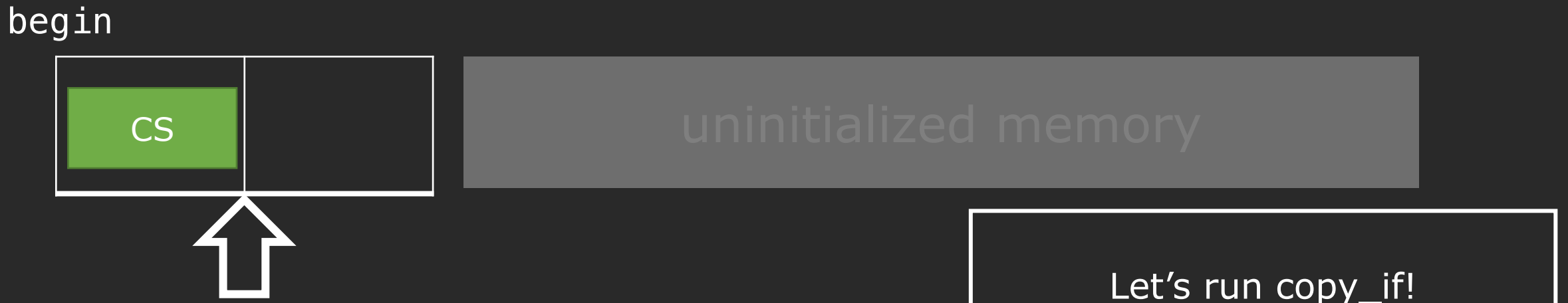




Let’s run copy_if!

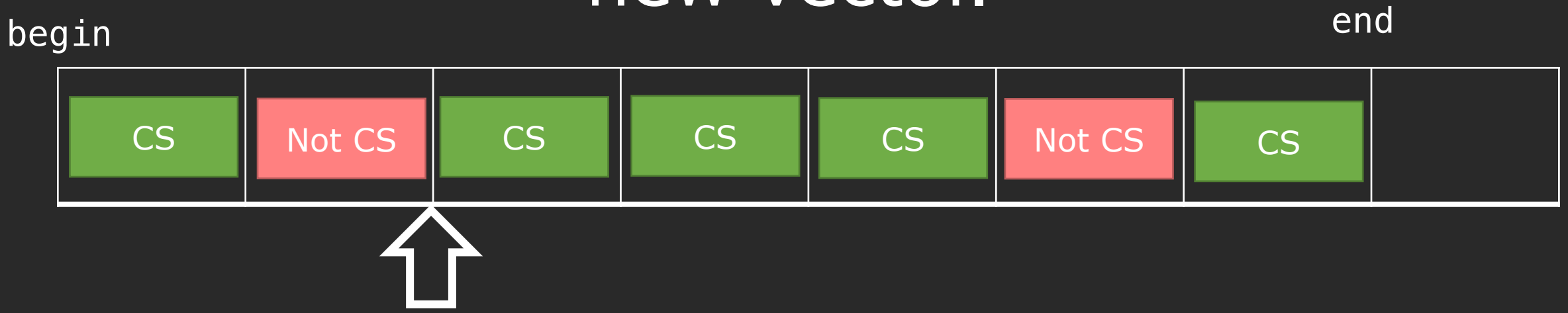
Copy all the CS courses into a new vector.

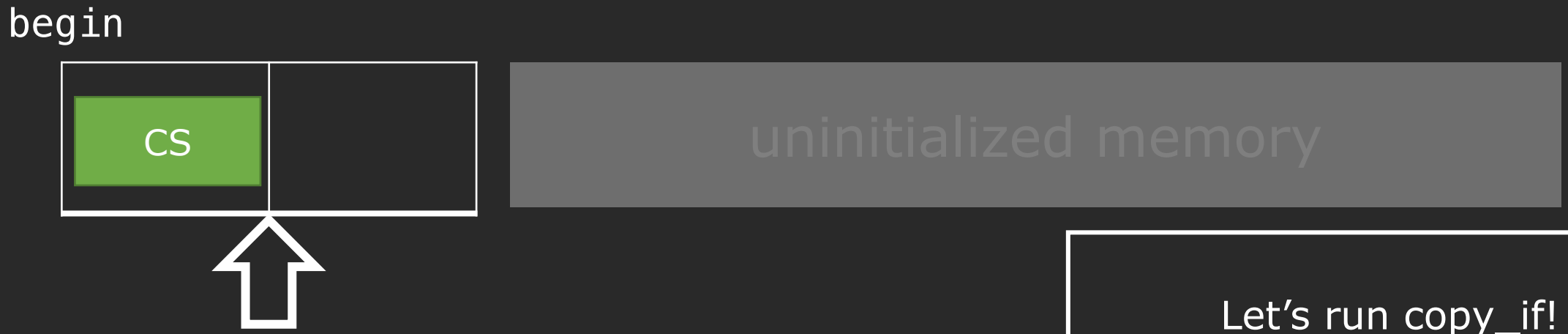




Let’s run copy_if!

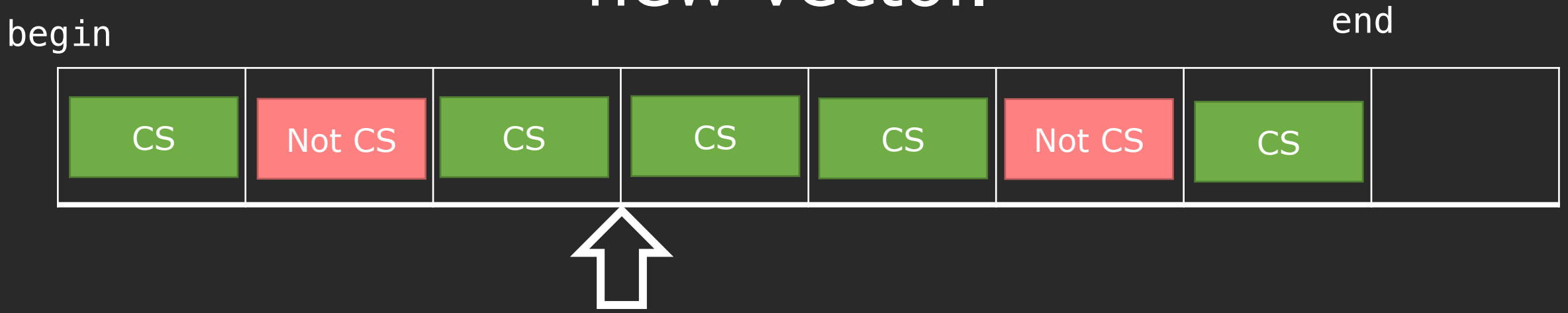
Copy all the CS courses into a new vector.

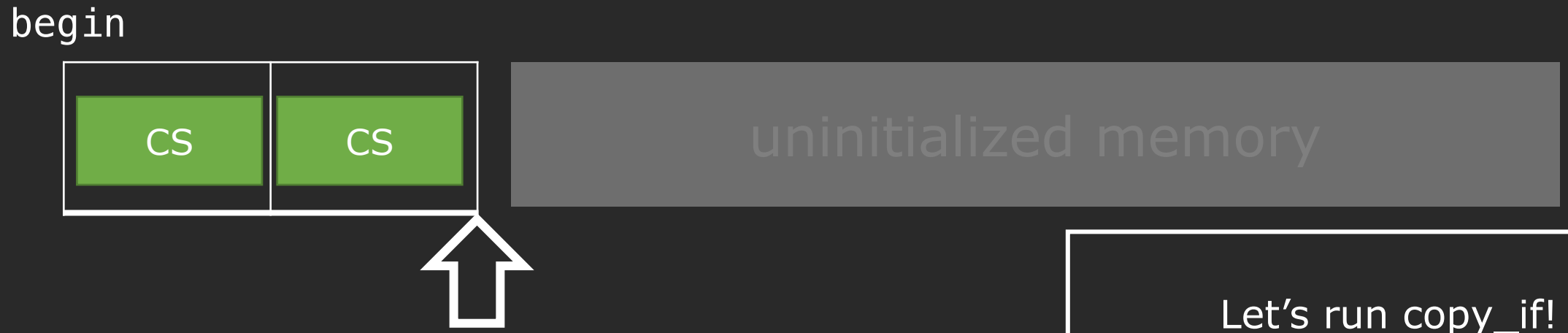




Let’s run copy_if!

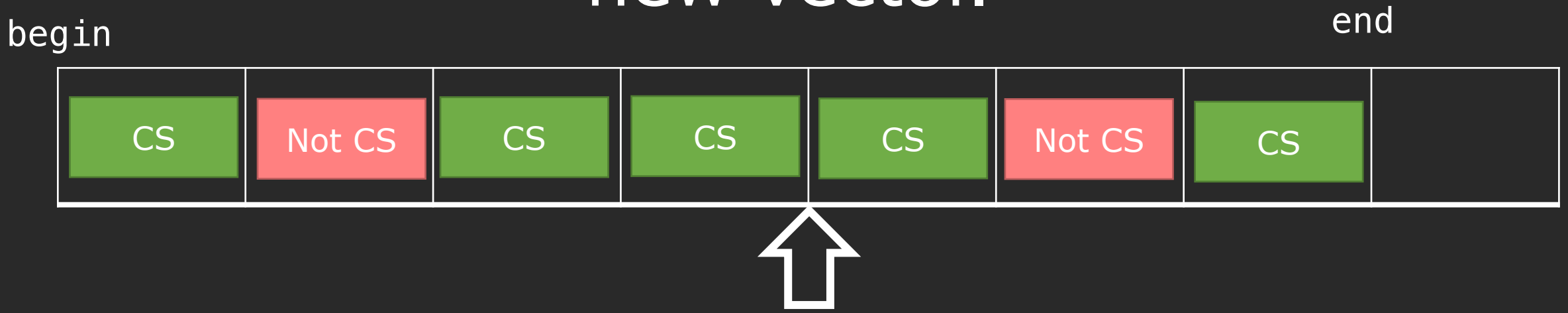
Copy all the CS courses into a new vector.

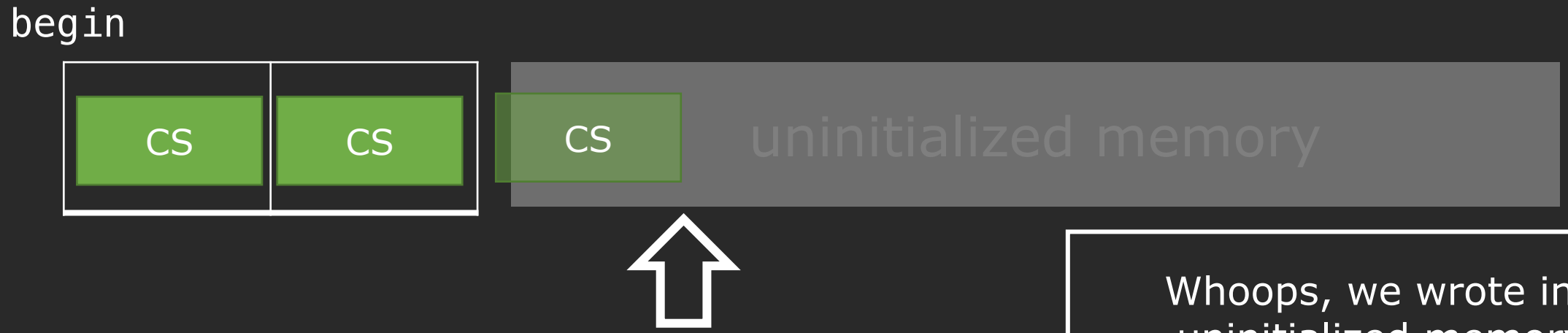




Let’s run copy_if!

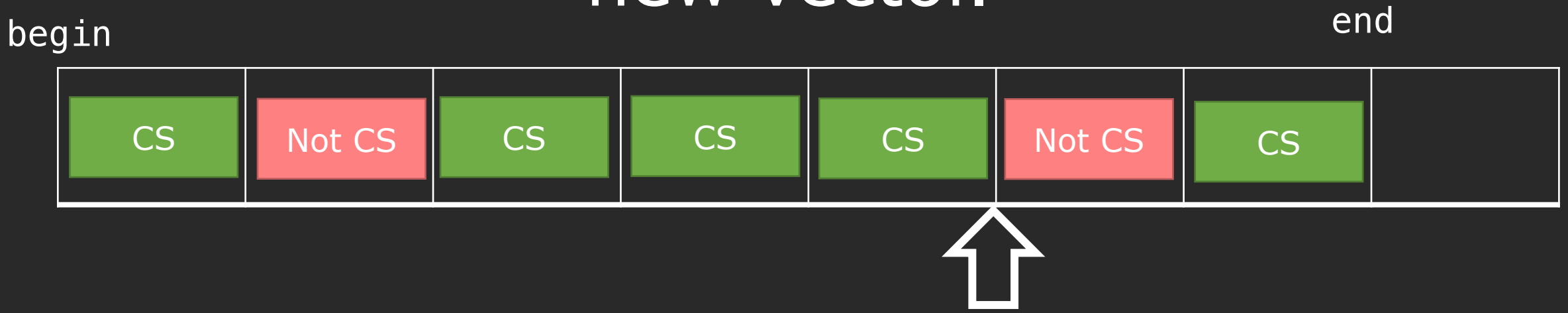
Copy all the CS courses into a new vector.

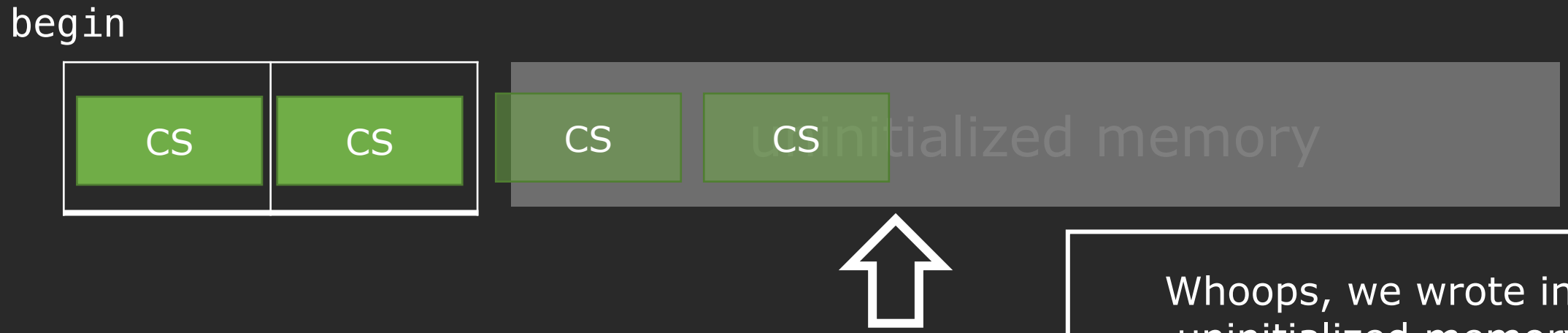




Whoops, we wrote into uninitialized memory!

Copy all the CS courses into a new vector.



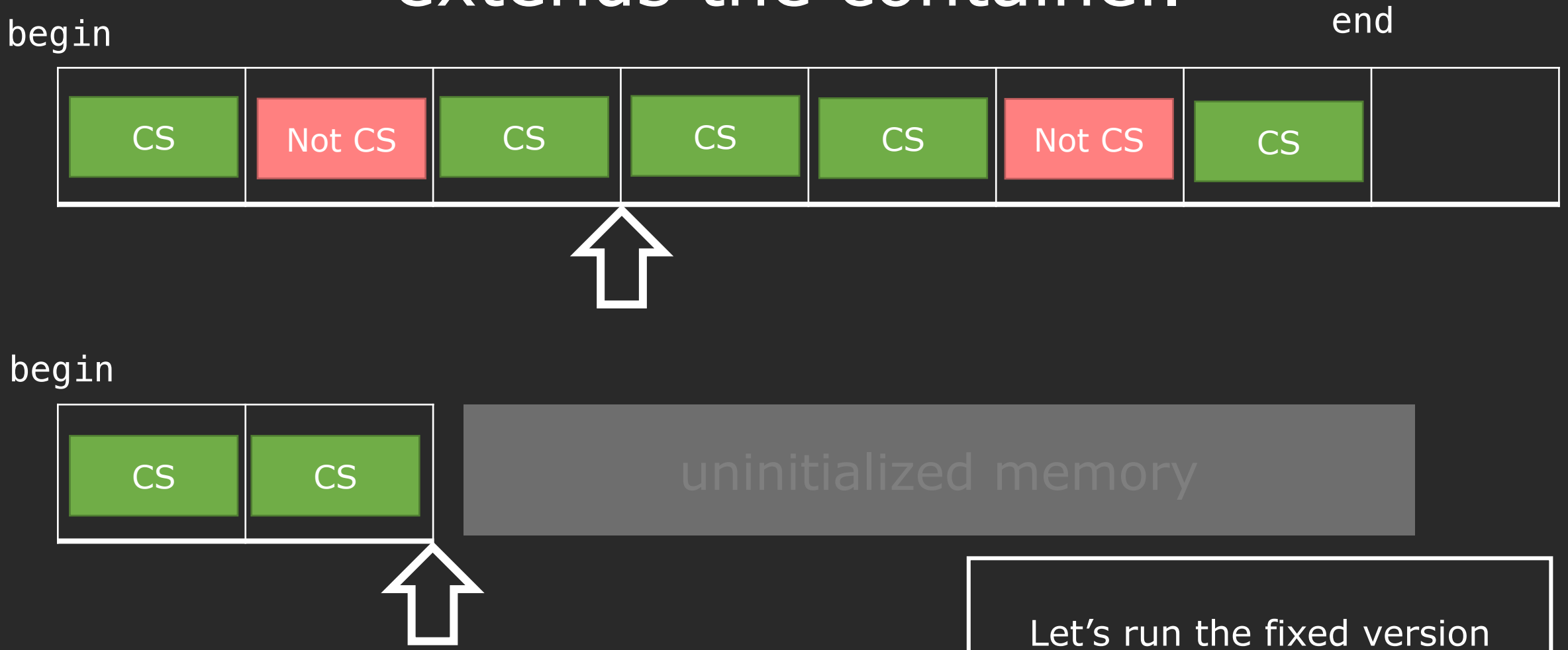


Whoops, we wrote into uninitialized memory!

We need a special iterator which extends the container.

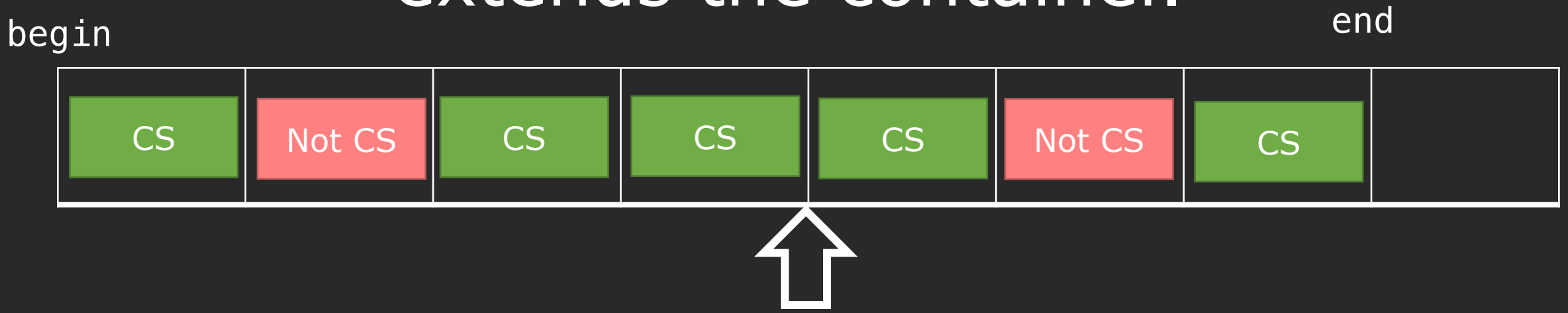
```
string dep = "CS";
auto isDep = [dep](const auto& course) {
    return course.name.size() >= dep.size &&
           course.substr(0, dep.size()) == dep;
};


std::copy_if(csCourses.begin(), csCourses.end(),
        back_inserter(csCourses), isDep);
```

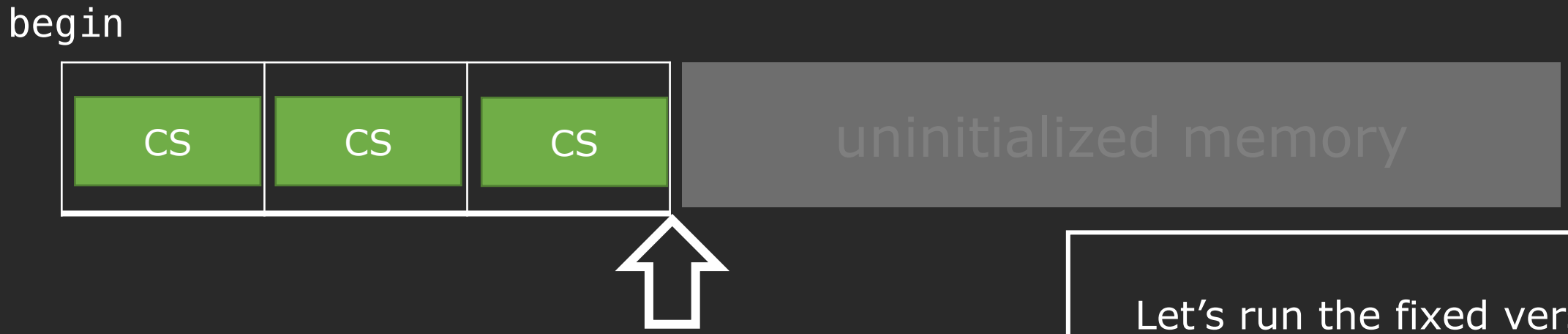
We need a special iterator which extends the container.



Let’s run the fixed version

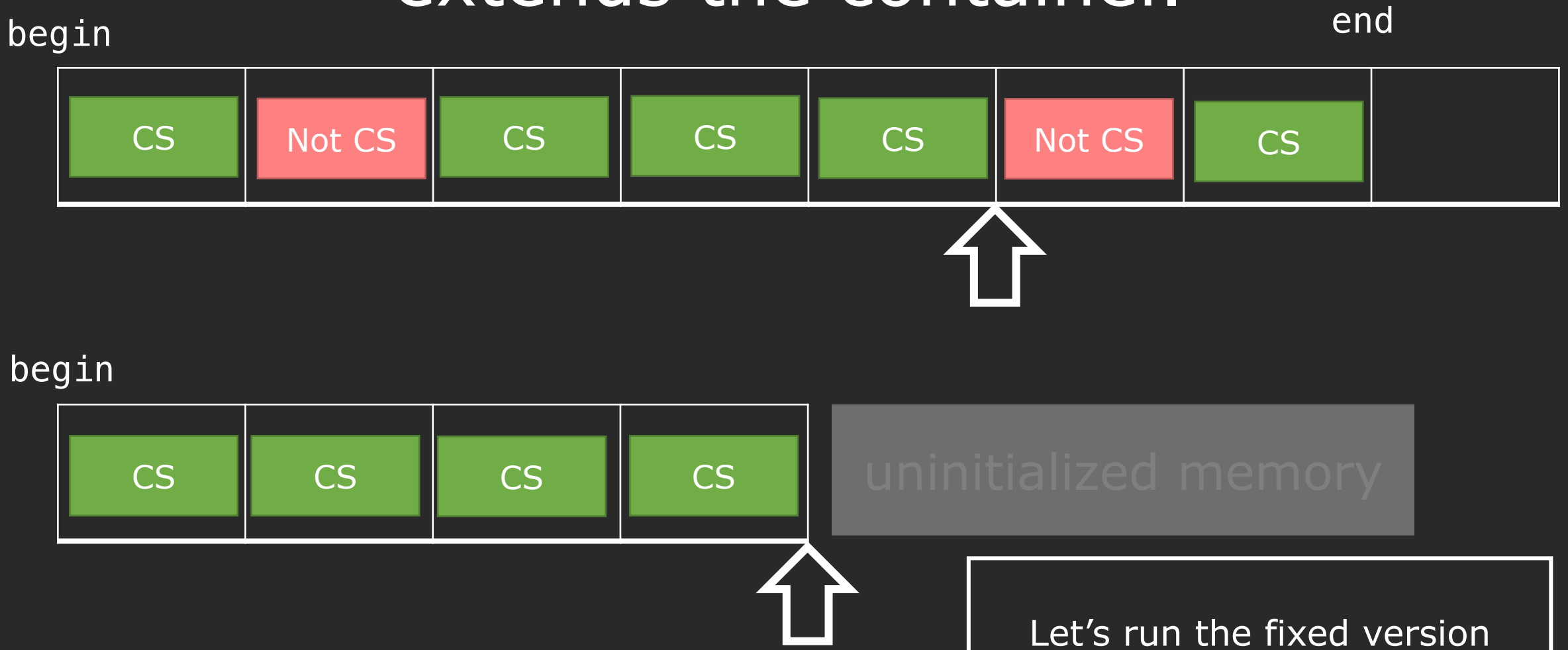
We need a special iterator which extends the container.





Let’s run the fixed version

We need a special iterator which extends the container.

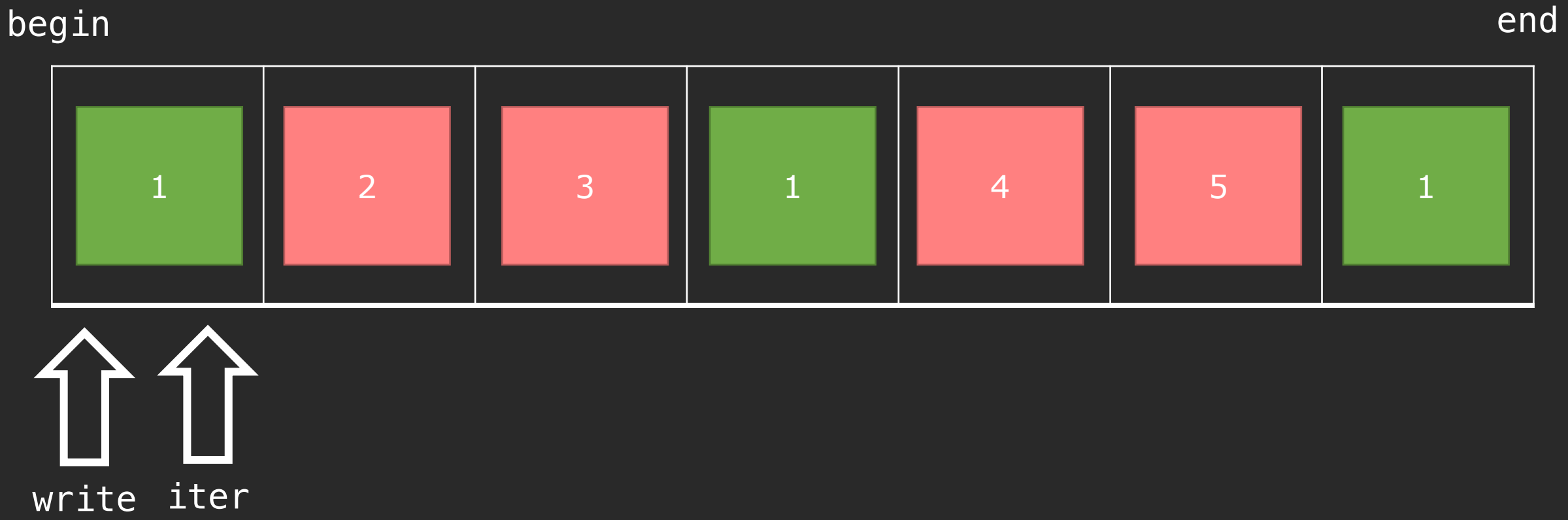


Let’s run the fixed version

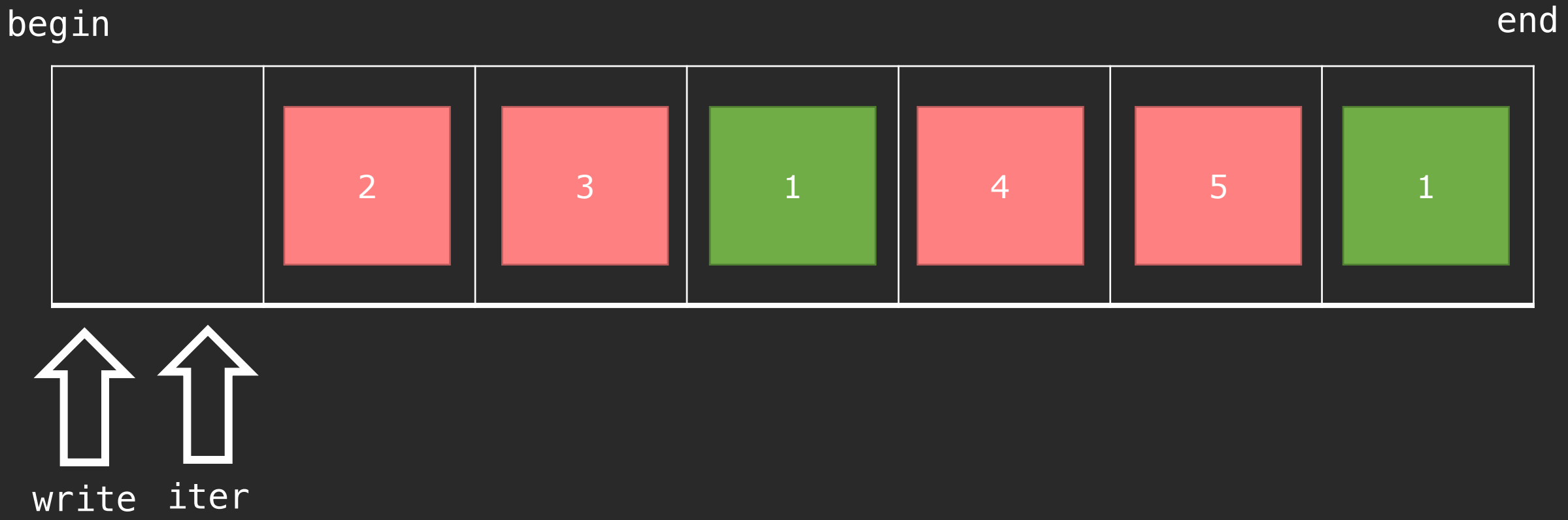
Stream iterators read from istreams or write to ostreams!

```
string dep = ”CS”;
auto isDep = [dep](const auto& course) {
    return course.name.size() >= dep.size &&
           course.substr(0, dep.size()) == dep;
};


std::copy_if(csCourses.begin(), csCourses.end(),
std::ostream_iterator<Course>(cout, ”\n"), isDep);
```

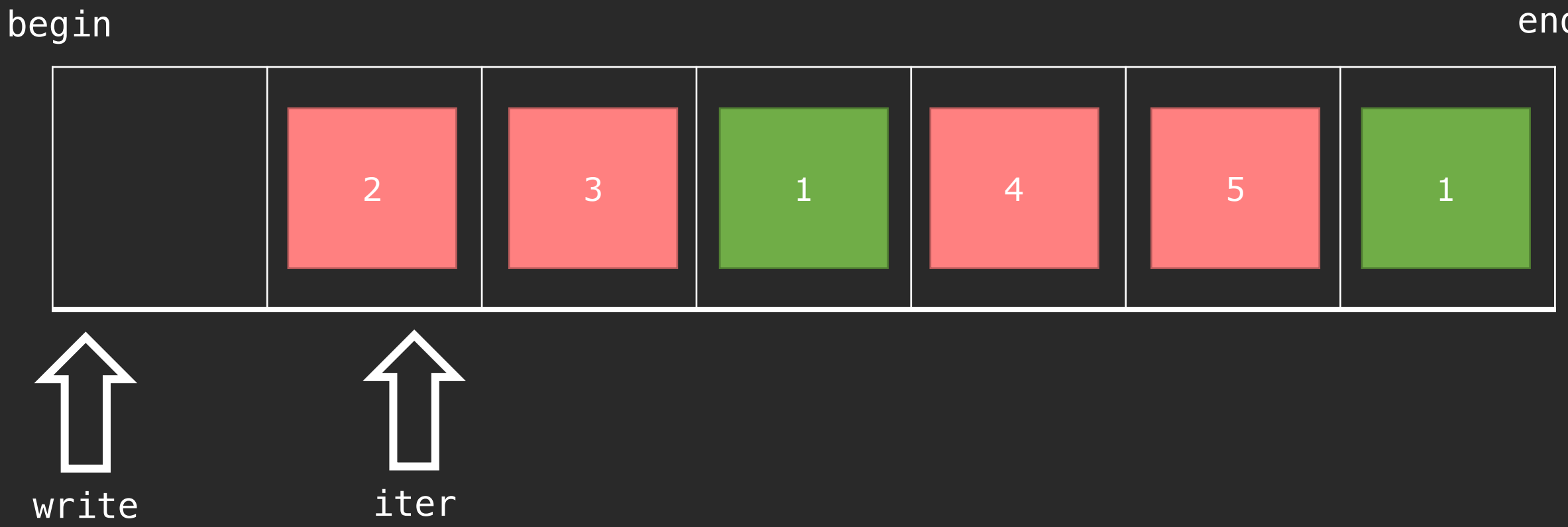
Let’s run through this code!



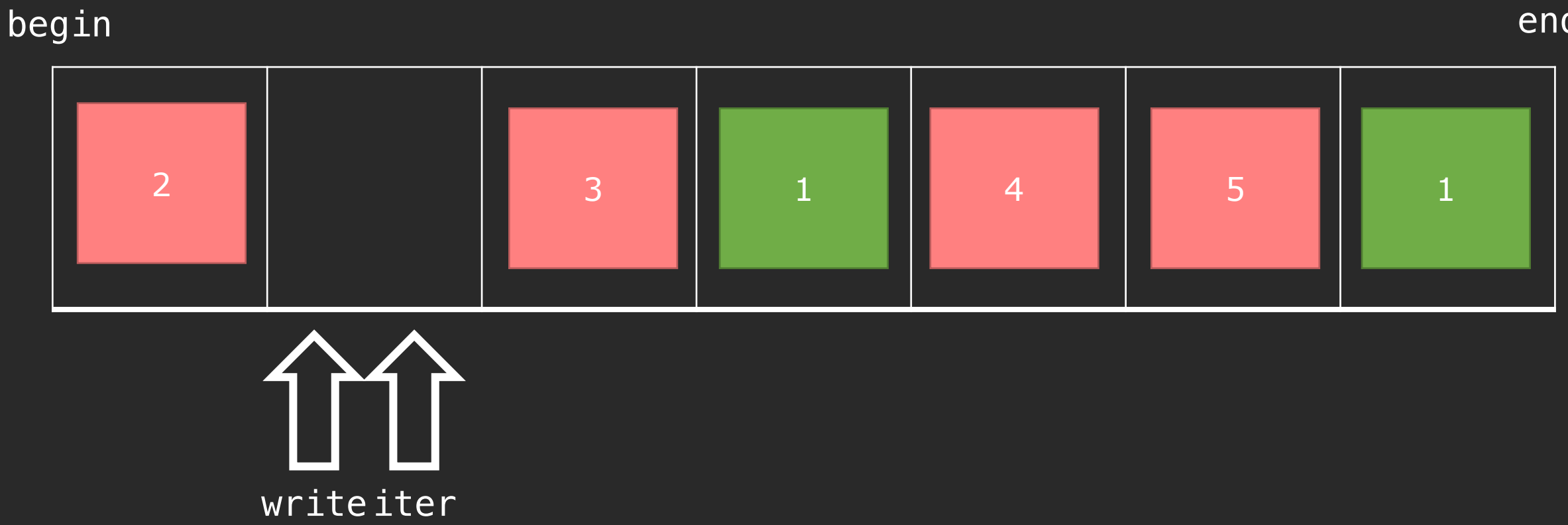
Let’s run through this code!



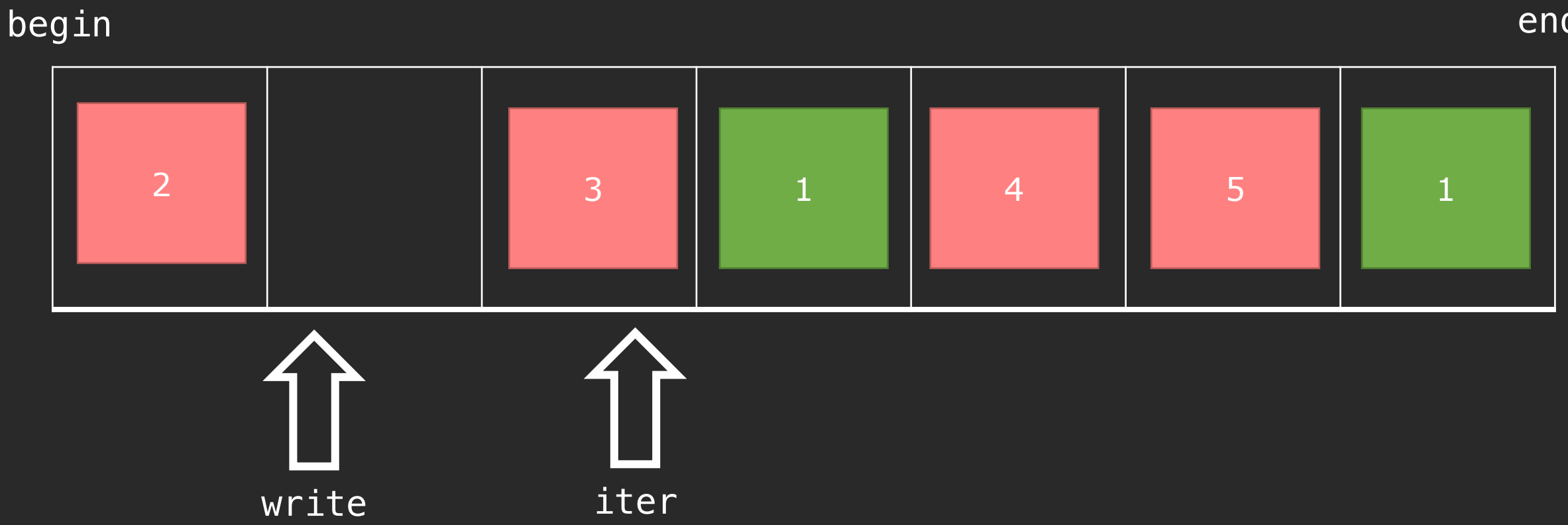
Let’s run through this code!



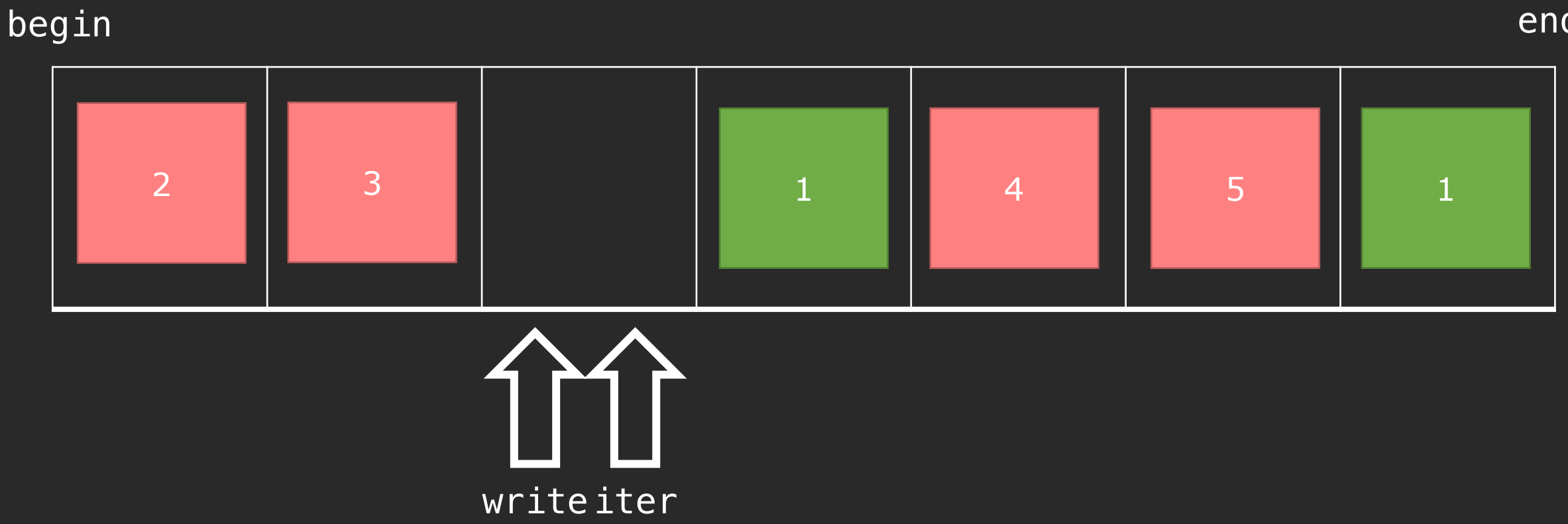
Let’s run through this code!



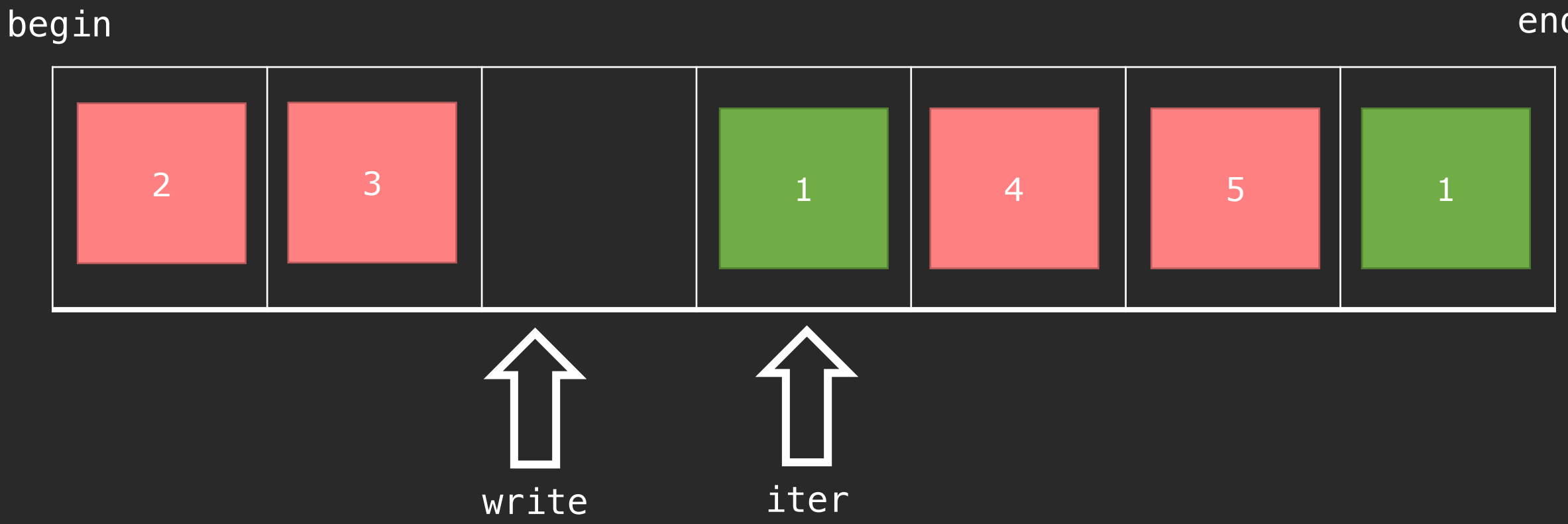
Let’s run through this code!



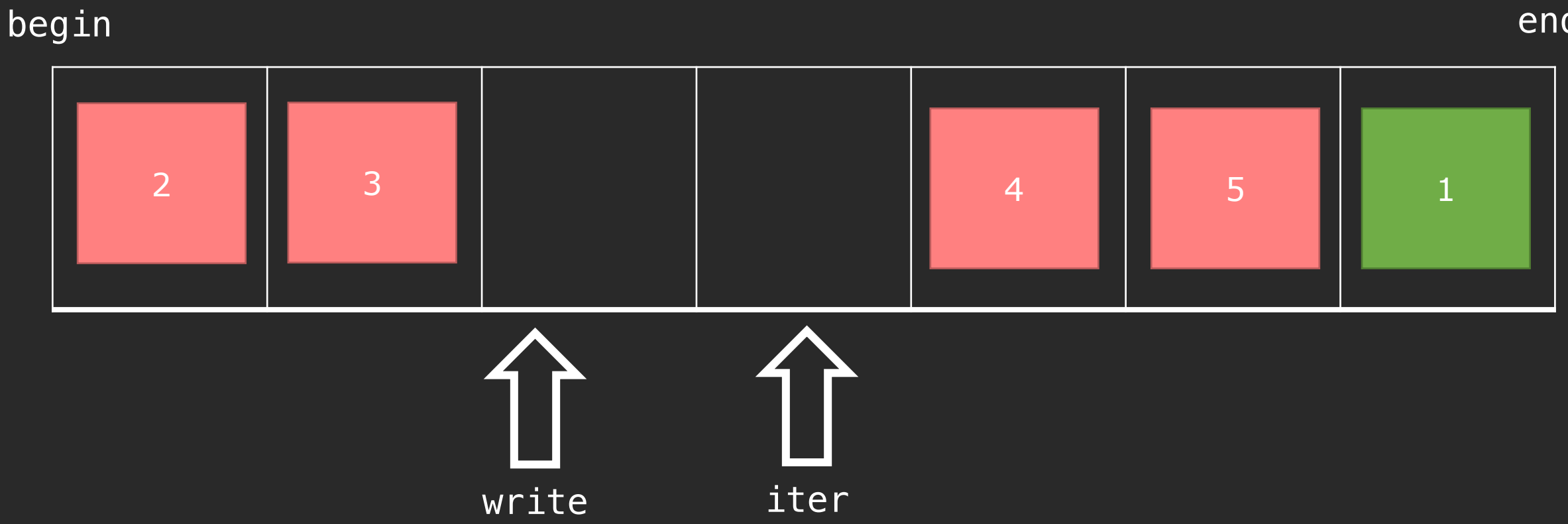
Let’s run through this code!



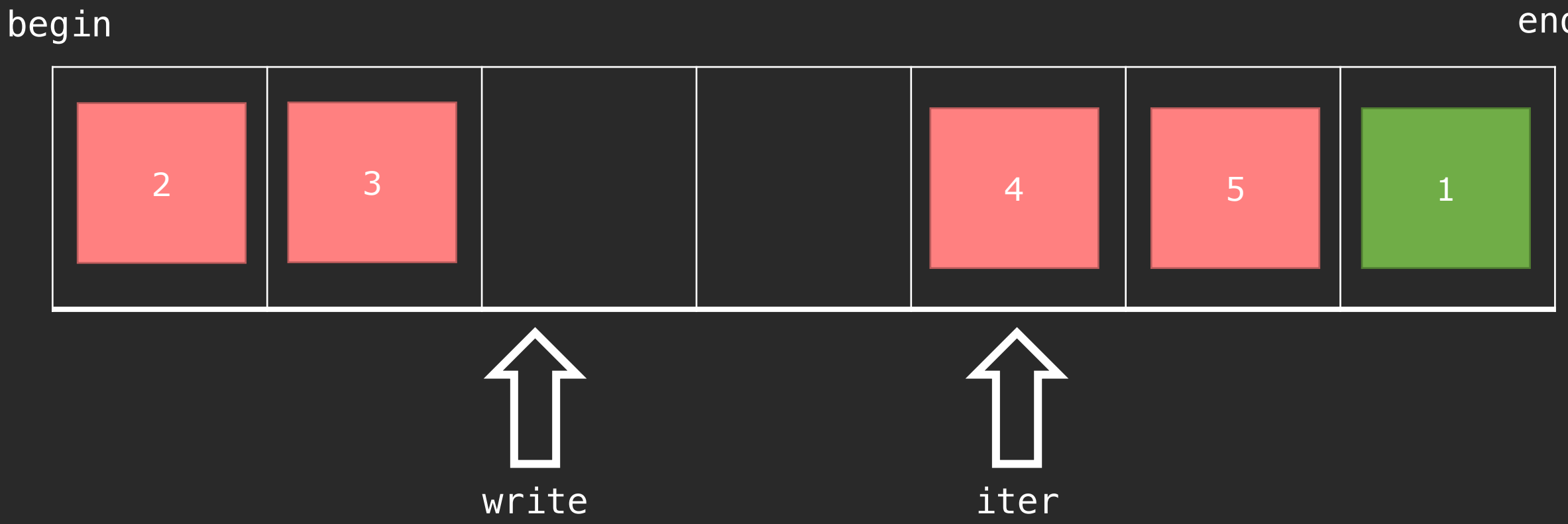
Let’s run through this code!



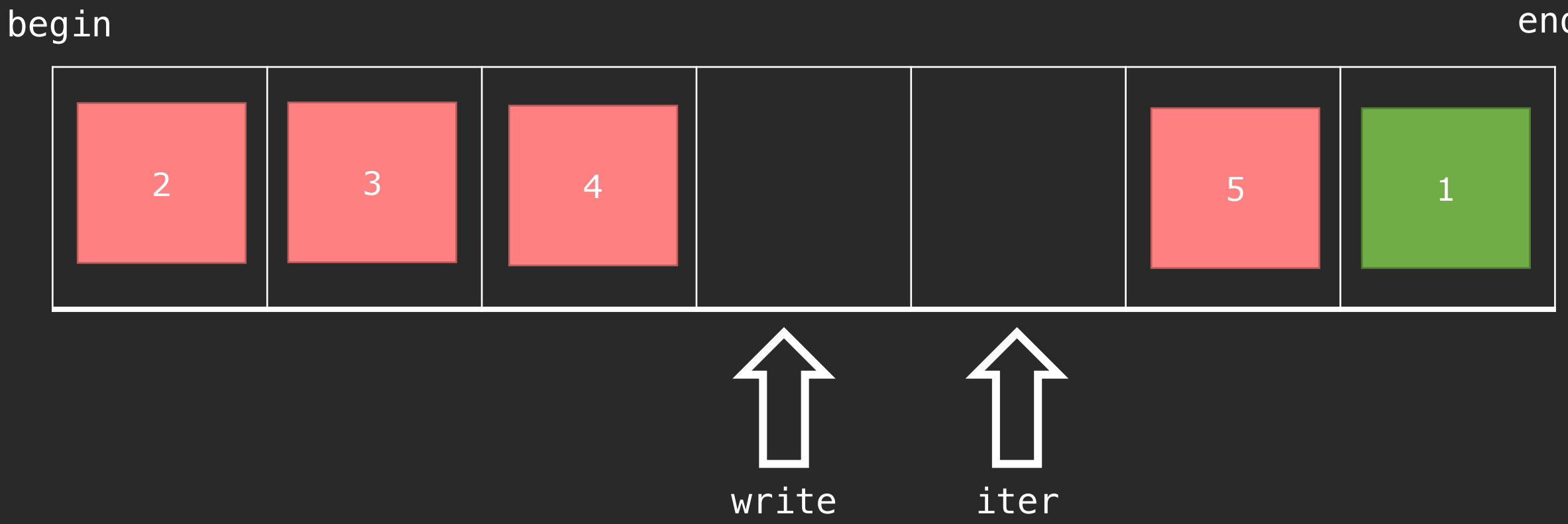
Let’s run through this code!



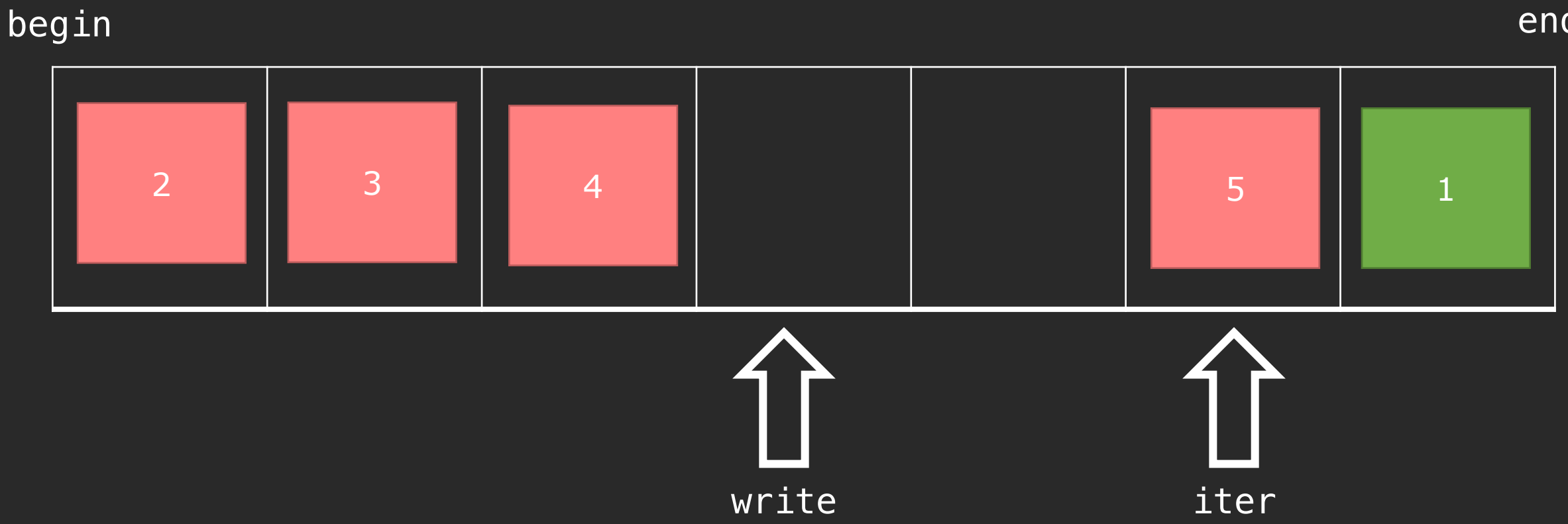
Let’s run through this code!



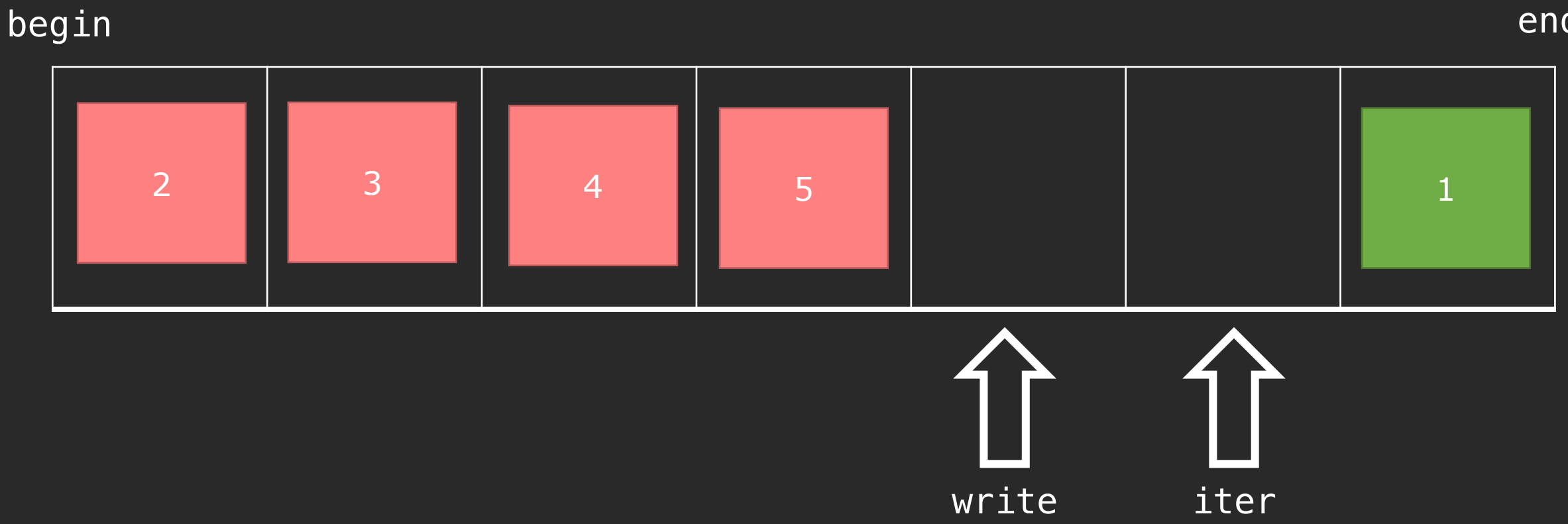
Let’s run through this code!



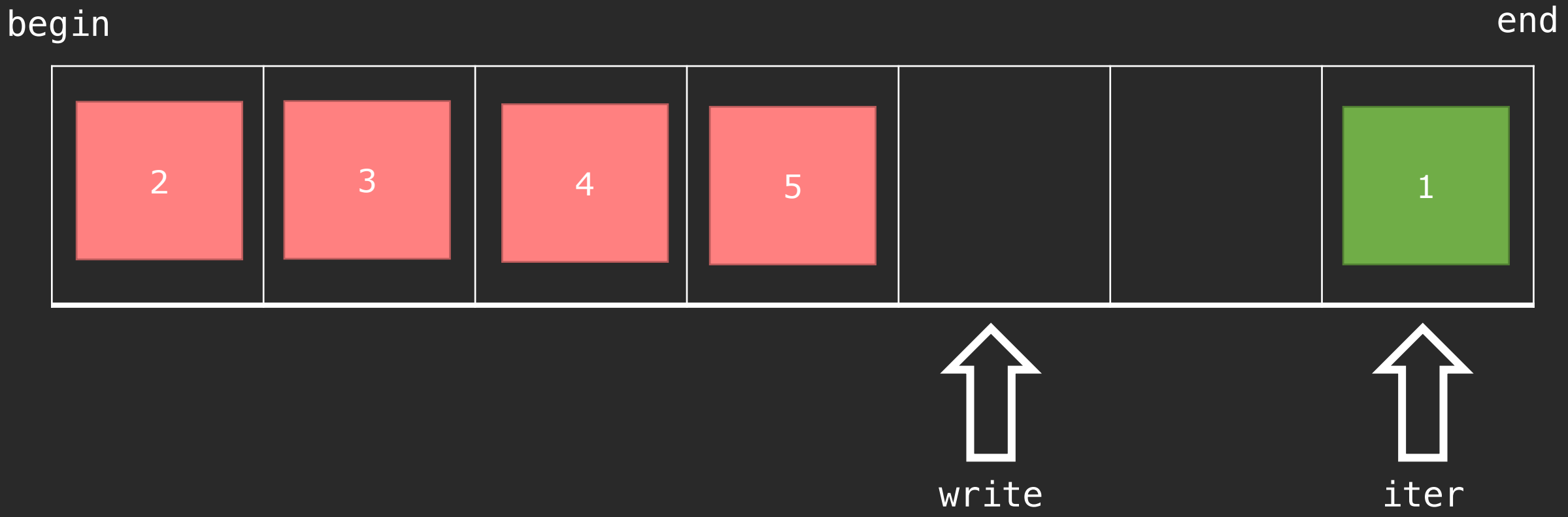
Let’s run through this code!



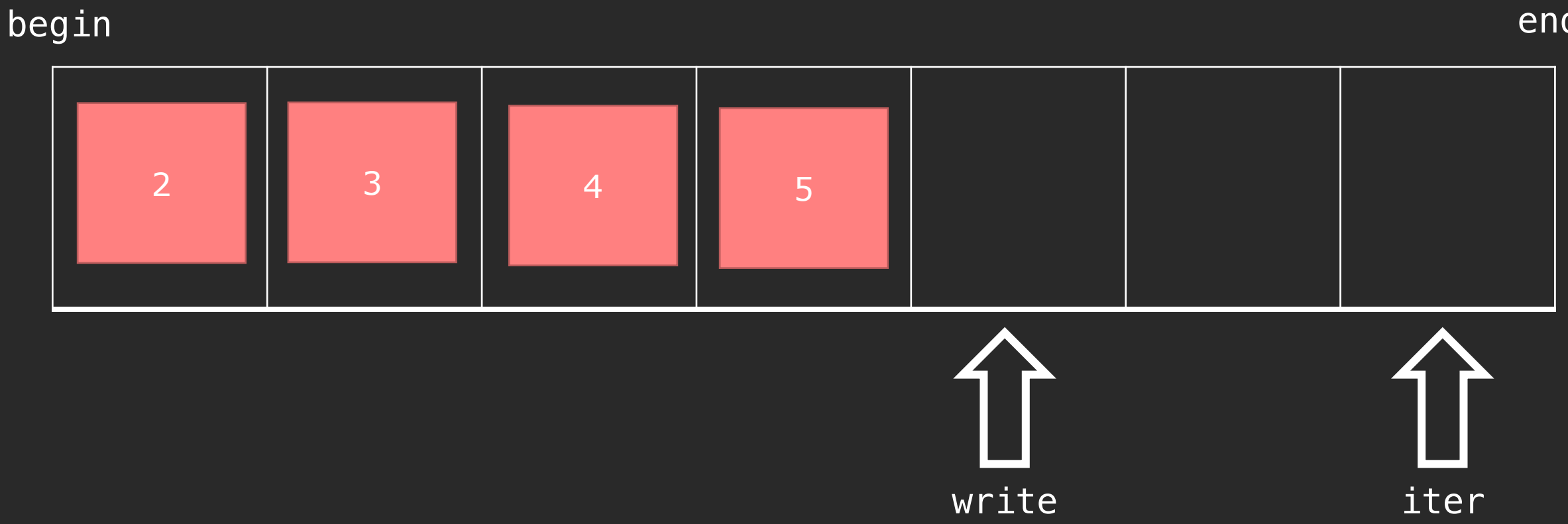
Let’s run through this code!



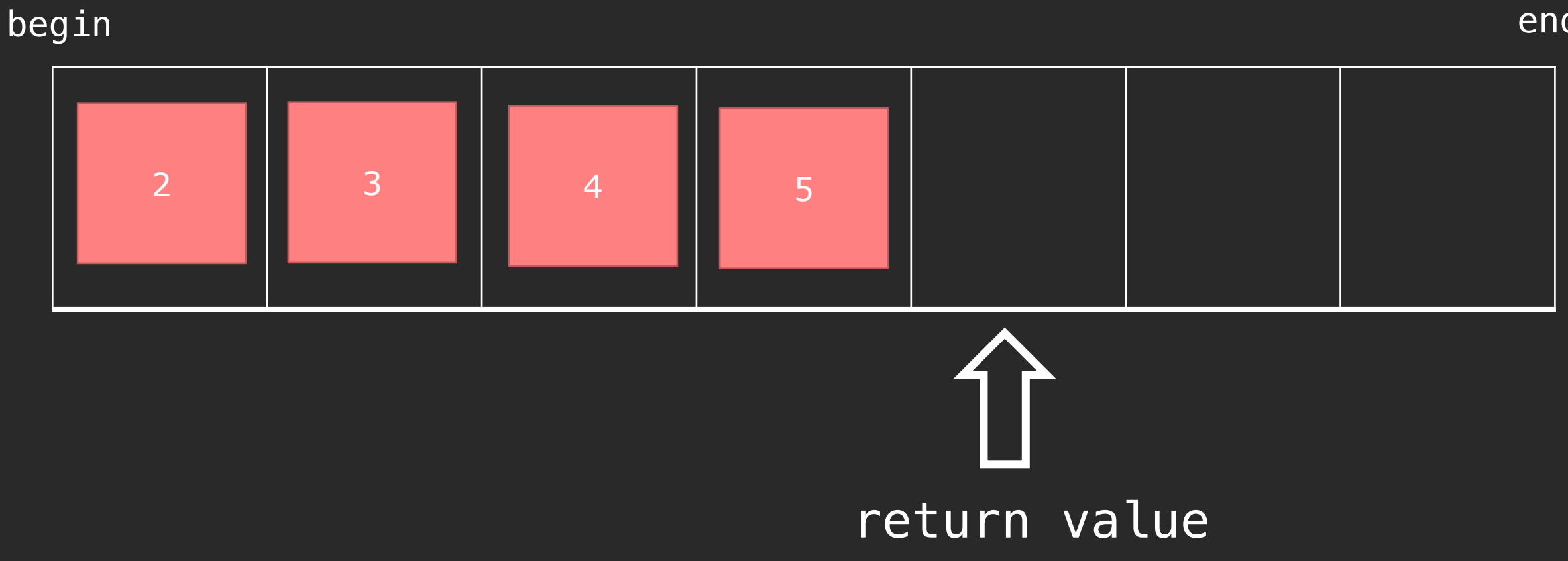
Let’s run through this code!



Let’s run through this code!



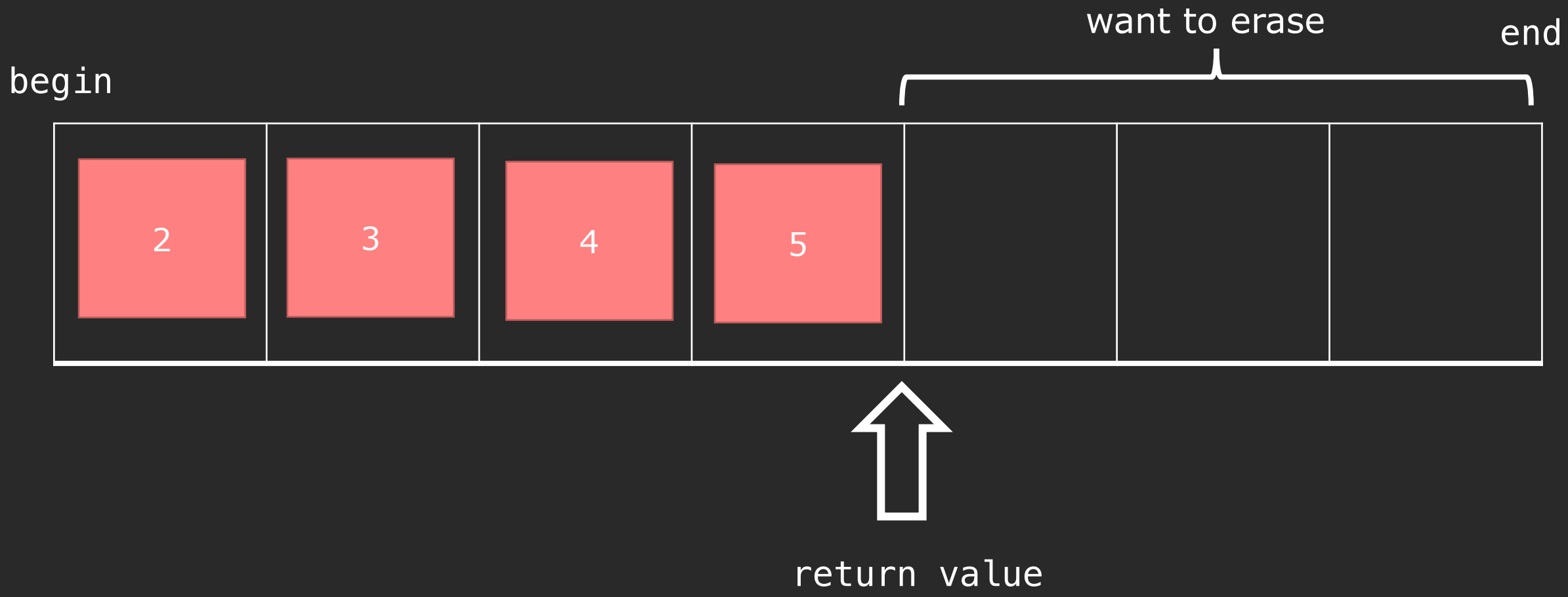
Let’s run through this code!



std::remove does not change the size of the container!

- It can’t!
- The algorithm is not a member of std::vector (or any other collection) so it can’t change its size member.

Let’s run through this code!



erase-remove idiom

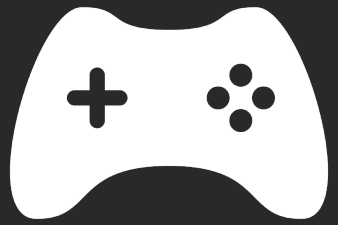
erases trash (everything between iterator and end)

returns iterator to beginning of trash.

```
v.erase(
    std::remove_if(v.begin(), v.end(), pred),
    v.end()
);
```

Homework Problem
Implement the logic of remove from before!

```
template <typename ForwardIt, typename UniPred>
ForwardIt remove_if(ForwardIt first, ForwardIt last,
                    UniPred pred) {




}
```



Next time

Applying the Algorithms + STL Review