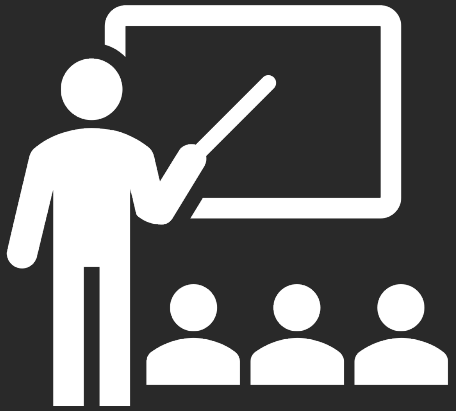
Templates

Game Plan



- erase in containers
- template functions
- concept lifting
- implicit interfaces
- overload resolution

erasing in STL containers

part 1: invalidated iterators

How do you erase from an STL collection?

```
vector<int> v{3, 1, 4, 1, 5, 2, 6};

auto iter = v.begin();

std::advance(iter, 4);                  // could also do iter += 4
v.erase(iter);                          // {3, 1, 4, 1, 2, 6}

// alas, can’t erase by index
```

more generic for advance(verb: void) or distance

forward: std::advance(iter, step)/std::ditance(iter, step) is a better way to move iterator since it applies to all kinds of iterator instead of random access iterator only.

How do you erase from an STL collection?

```
deque<int> d{3, 1, 4, 1, 5, 2, 6};

auto iter = d.begin();

std::advance(iter, 4);              // could also do iter += 4
d.erase(iter);                      // {3, 1, 4, 1, 2, 6}

// alas, can't erase by index
```

How do you erase from an STL collection?

```
list<int> l{3, 1, 4, 1, 5, 2, 6};

auto iter = l.begin();

std::advance(iter, 4);                  // can’t do iter += 4!
l.erase(iter);                          // {3, 1, 4, 1, 2, 6}

// alas, can’t erase by index
```

How do you erase from an STL collection?

```
set<int> s{1, 3, 5, 7, 9, 11};

s.erase(3);                                  // {1, 5, 7, 9, 11}
s.erase(s.begin());                          // {5, 7, 9, 11}
```

Problem of invalidated iterators!

```
list<int> l{3, 1, 4, 1, 5, 2, 6};

auto iter = l.begin();
auto temp = --l.end();                    // points to 6
std::advance(iter, 4);
l.erase(iter);                            // {3, 1, 4, 1, 2, 6}

auto val = *iter;                         // prints 6
```

Problem of invalidated iterators!

```
deque<int> d{3, 1, 4, 1, 5, 2, 6};

auto iter = d.begin();
auto temp = --d.end();                  // points to 6
std::advance(iter, 4);
d.erase(iter);                          // {3, 1, 4, 1, 2, 6}

auto val = *iter;                       // Undefined behavior!
```

Problem of invalidated iterators!

```
vector<int> v{3, 1, 4, 1, 5, 2, 6};

auto iter = v.begin();
auto temp = --v.end();                    // points to 6
std::advance(iter, 4);
v.erase(iter);                            // {3, 1, 4, 1, 2, 6}

auto val = *iter;                         // Undefined behavior!
```

Different containers have different rules for invalidated containers!

iterator to erasure point always invalidated

vector: all iterators after erasure point invalidated.

deque: all iterators invalidated
(unless erasure point was front or back)

list/set/map: all other iterators are still valid!

When might this problem appear?

```
// This code is buggy!
void erase_all(vector<int>& vec, int val) {
  for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
    if (*iter == val) {
      vec.erase(iter);
    }


  }
}
```

iter invalidated here

incrementing
invalidated iterator

When might this problem appear?

```
// This code is buggy!
void erase_all(vector<int>& vec, int val) {
  for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
    if (*iter == val) {
      vec.erase(iter);
    }


  }
}
```

When might this problem appear?

```
// This code is good!
void erase_all(vector<int>& vec, int val) {
  for (auto iter = vec.begin(); iter != vec.end();) {
    if (*iter == val) {
      iter = vec.erase(iter);
    } else {
      ++iter;
    }
  }
}
```

When might this problem appear?

don't increment if something was erased

```
// This code is good!
void erase_all(vector<int>& vec, int val) {
  for (auto iter = vec.begin(); iter != vec.end();) {
    if (*iter == val) {
      iter = vec.erase(iter);
    } else {
      ++iter;
    }
  }
}
```

erase returns valid iterator of element after the erased one

only increment if nothing was erased

erase update!

When might this problem appear?

```
// This code is buggy!
void erase_all_even_keys(map<int, int>& map, int val) {
  for (auto iter = map.begin(); iter != map.end(); ++iter) {
    if (iter->first == val) {
      iter = map.erase(iter);
    }
  }
}
```

When might this problem appear?

```
// Equivalently, bad code in the Stanford library
void erase_all_even_keys(map<int, int>& map, int val) {
  for (int key : map) {
    if (map[key] == val) {
      map.remove(key); // messes up the iterators!
    }
  }
}

// recall range-based for loop is implemented using iterators
```

When might this problem appear?

```
// This code is buggy!
void erase_all_even_keys(map<int, int>& map, int val) {
  for (auto iter = map.begin(); iter != map.end(); ++iter) {
    if (iter->first == val) {
      iter = map.erase(iter);
    }


  }
}
```

incrementing
invalidated iterator

iter invalidated here

When might this problem appear?

don’t increment if something was erased

```
// This code is good!
void erase_all_even_keys(map<int, int>& map, int val) {
  for (auto iter = map.begin(); iter != map.end();) {
    if (iter->first == val) {
      iter = map.erase(iter);
    } else {
      ++iter;
    }
  }
}
```

erase returns valid iterator of element after the erased one

only increment if nothing was erased

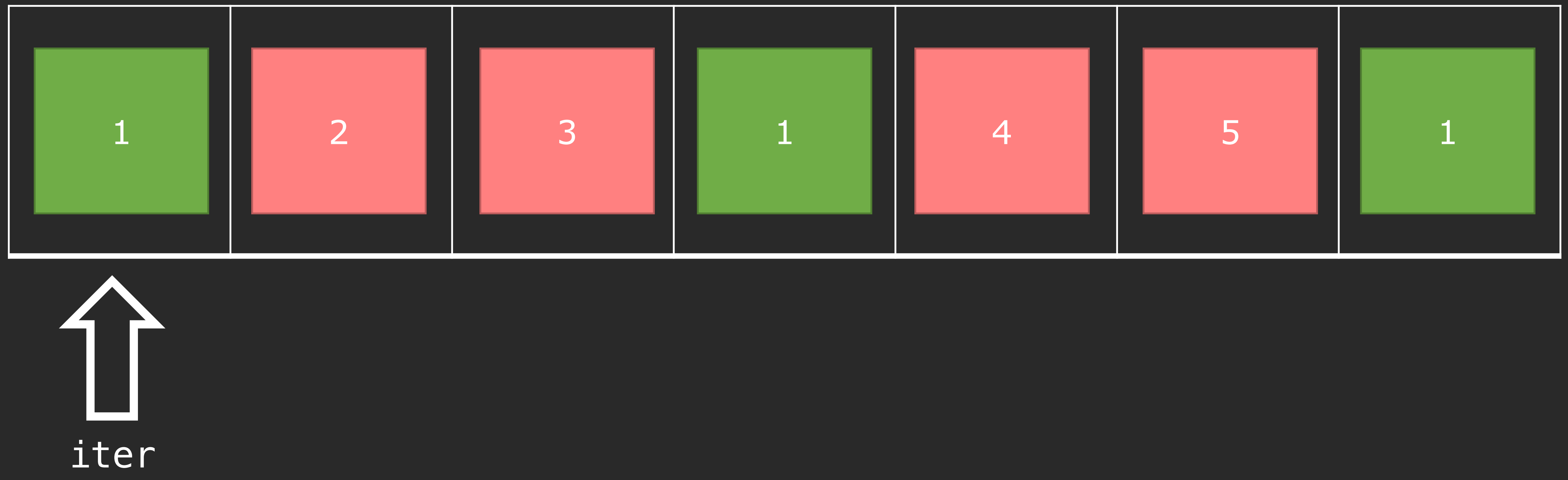
erasing in STL containers

part 2: erase-remove idiom

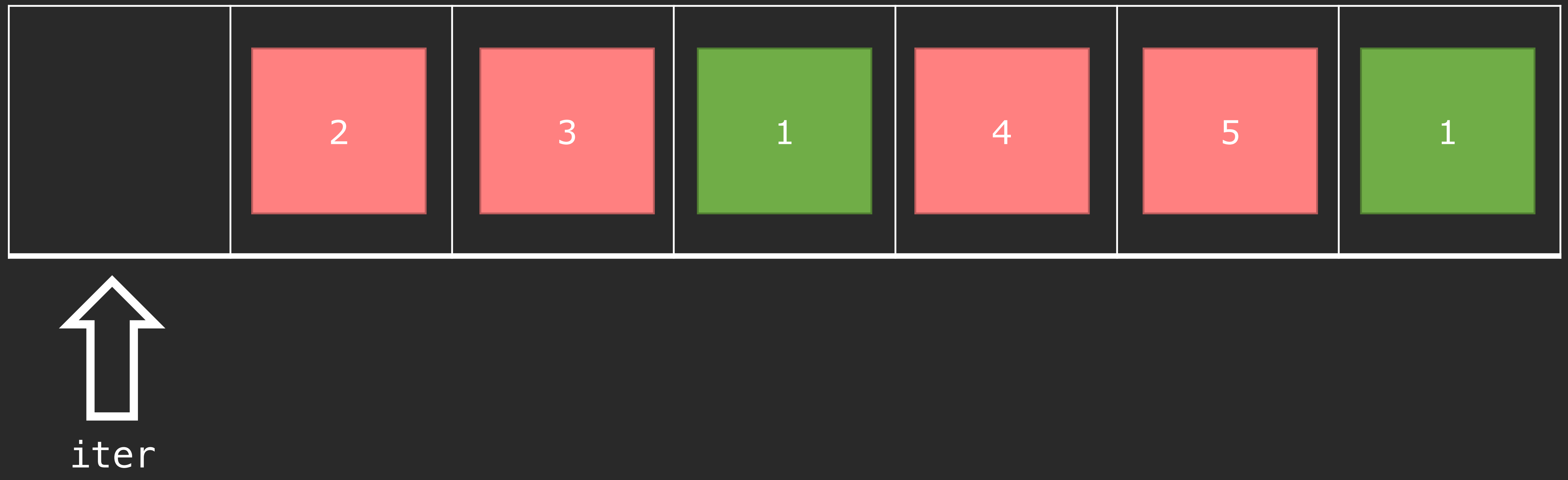
How efficient is this code?

```
void erase_all(vector<int>& vec, int val) {
  for (auto iter = vec.begin(); iter != vec.end();) {
    if (*iter == val) {
      iter = vec.erase(iter); // this is kinda slow
    } else {
      ++iter;
    }
  }
}
```

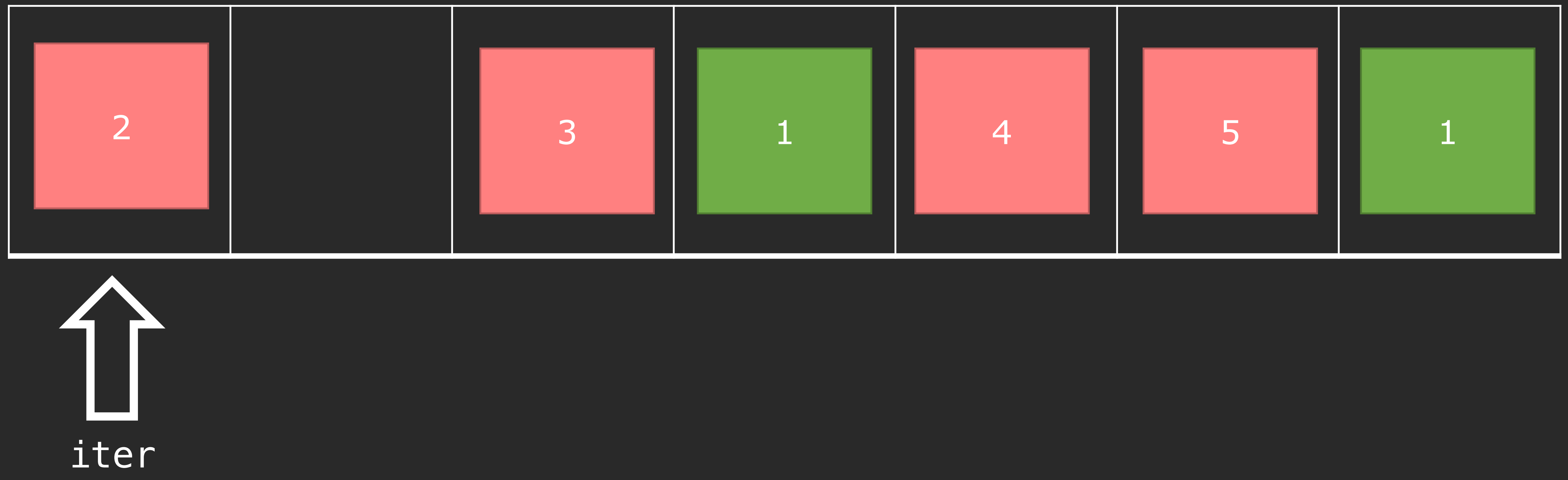
Let’s run through this code!



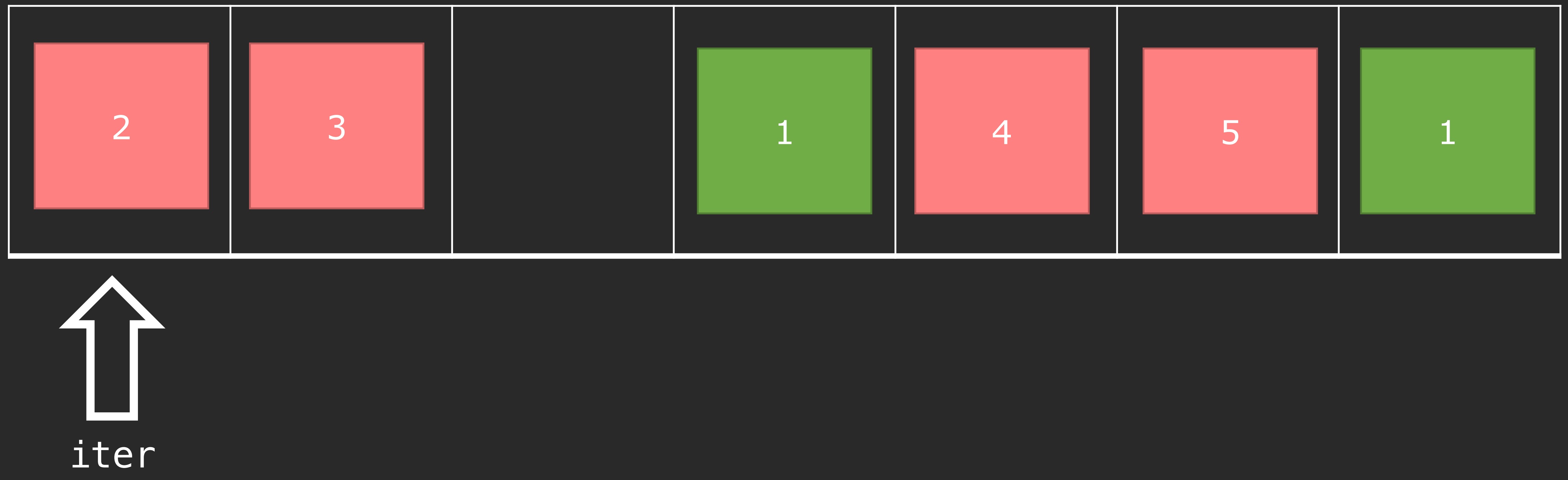
Let’s run through this code!



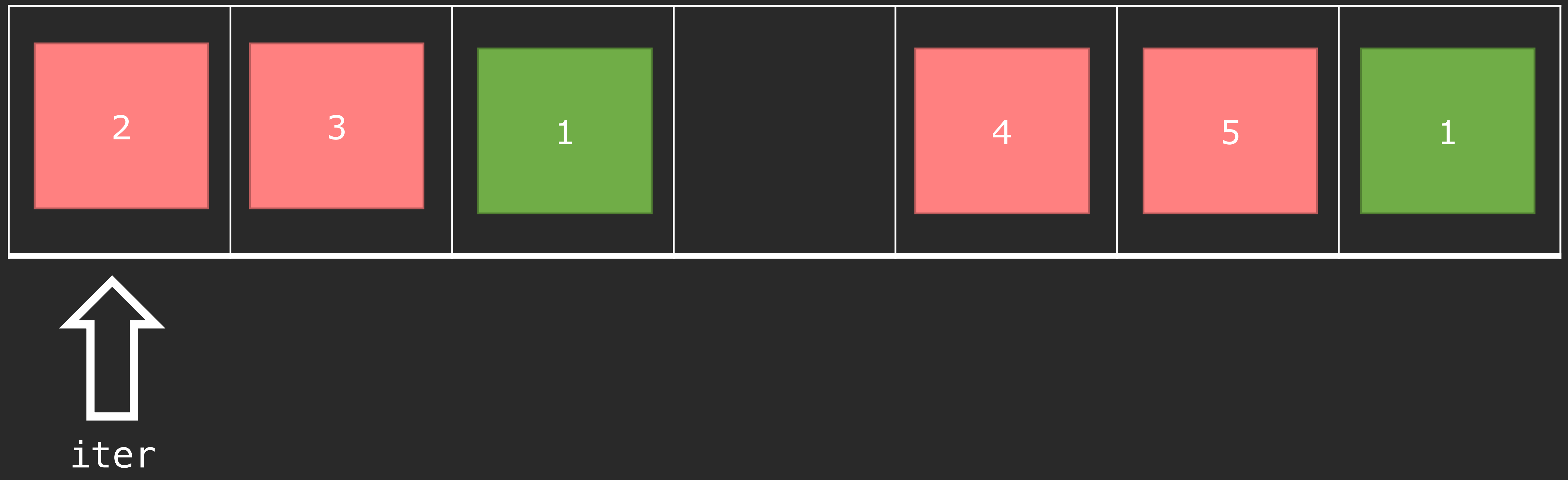
Let’s run through this code!



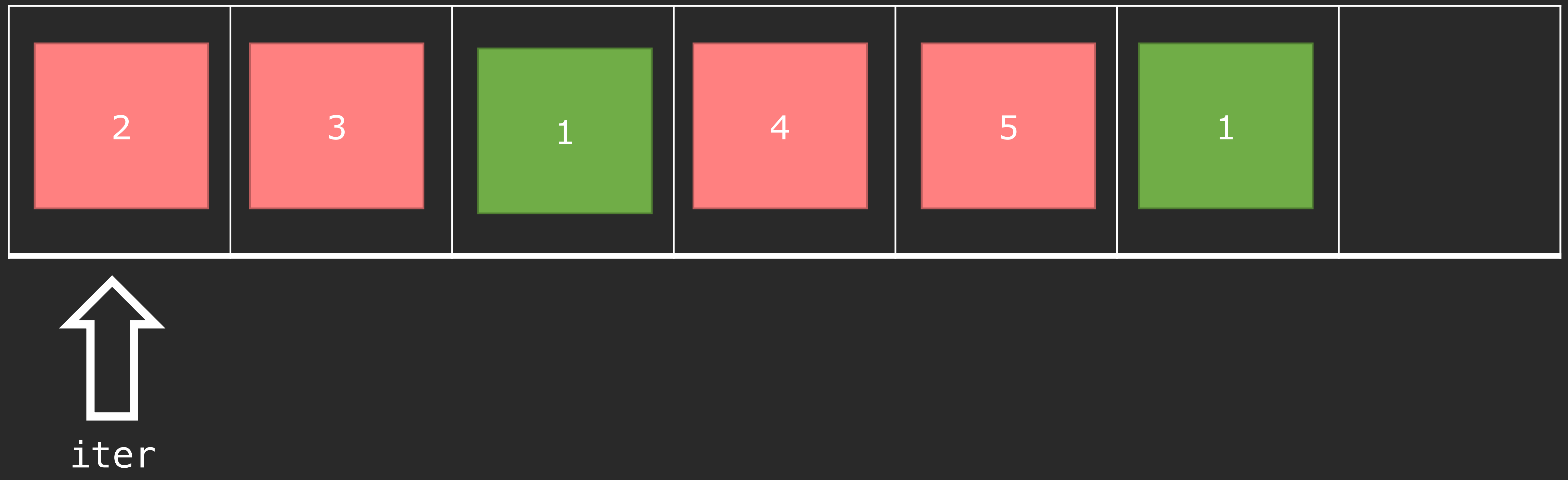
Let's run through this code!



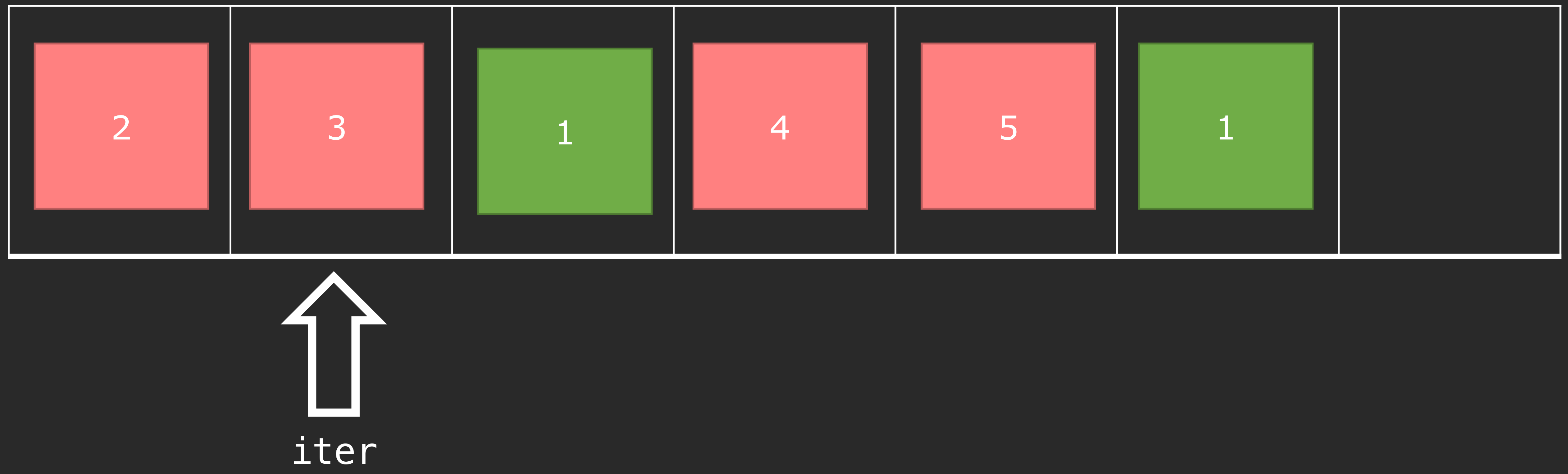
Let’s run through this code!



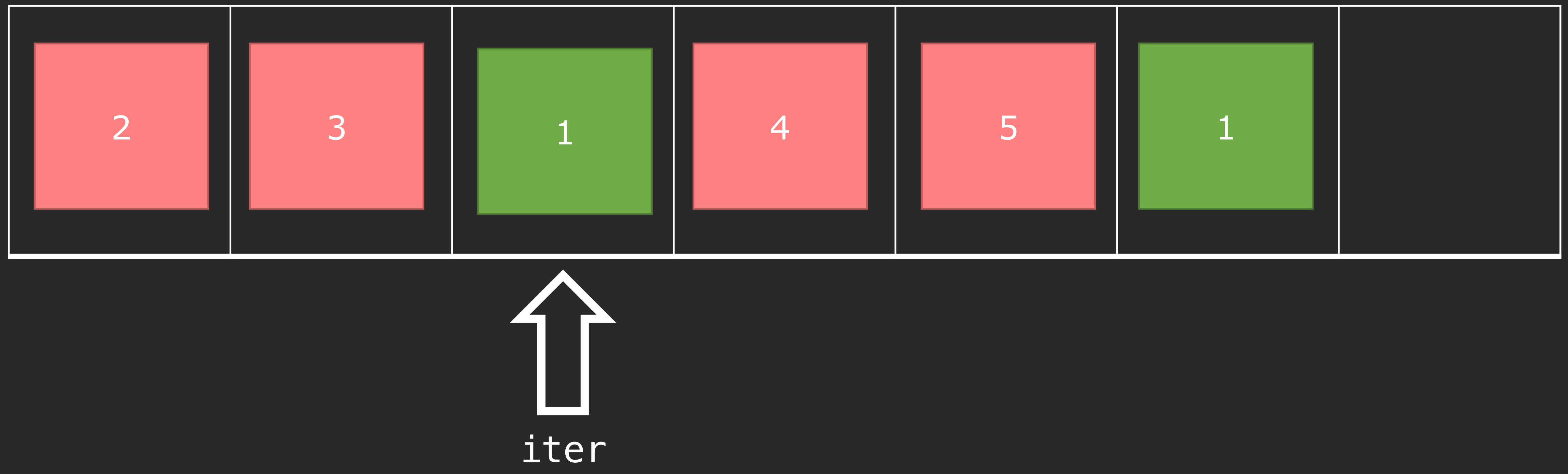
Let’s run through this code!



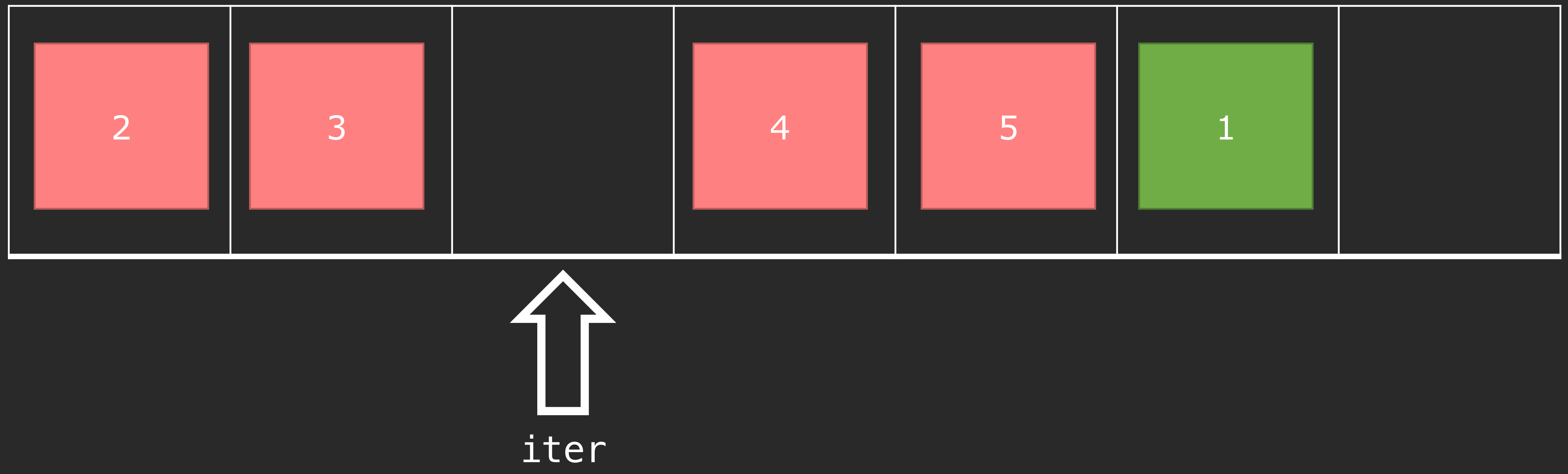
Let’s run through this code!



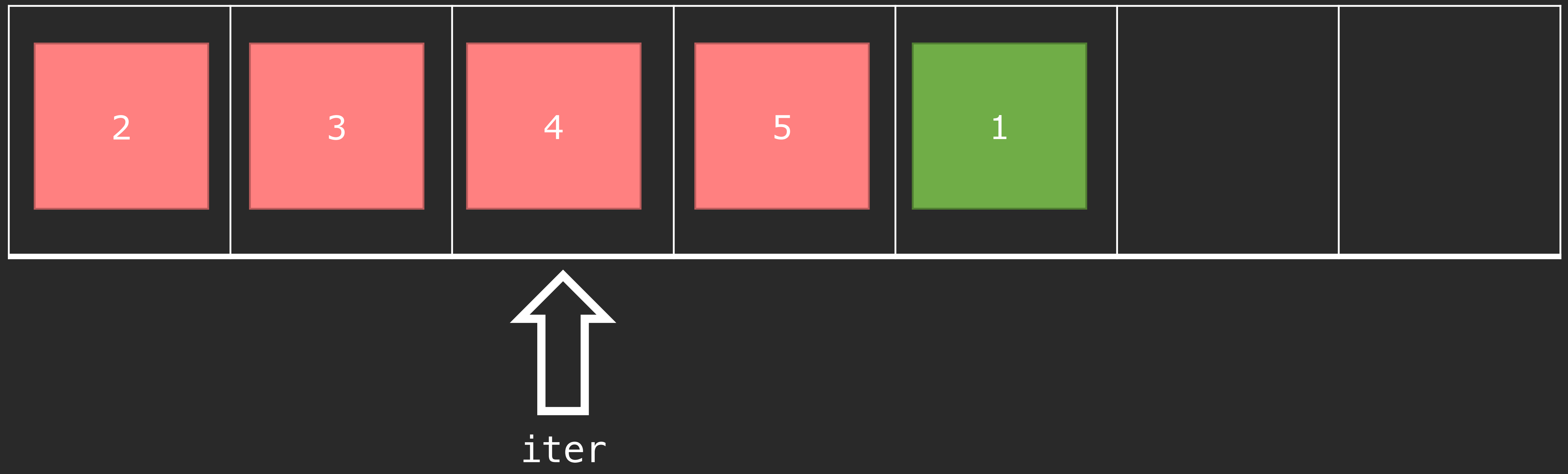
Let’s run through this code!



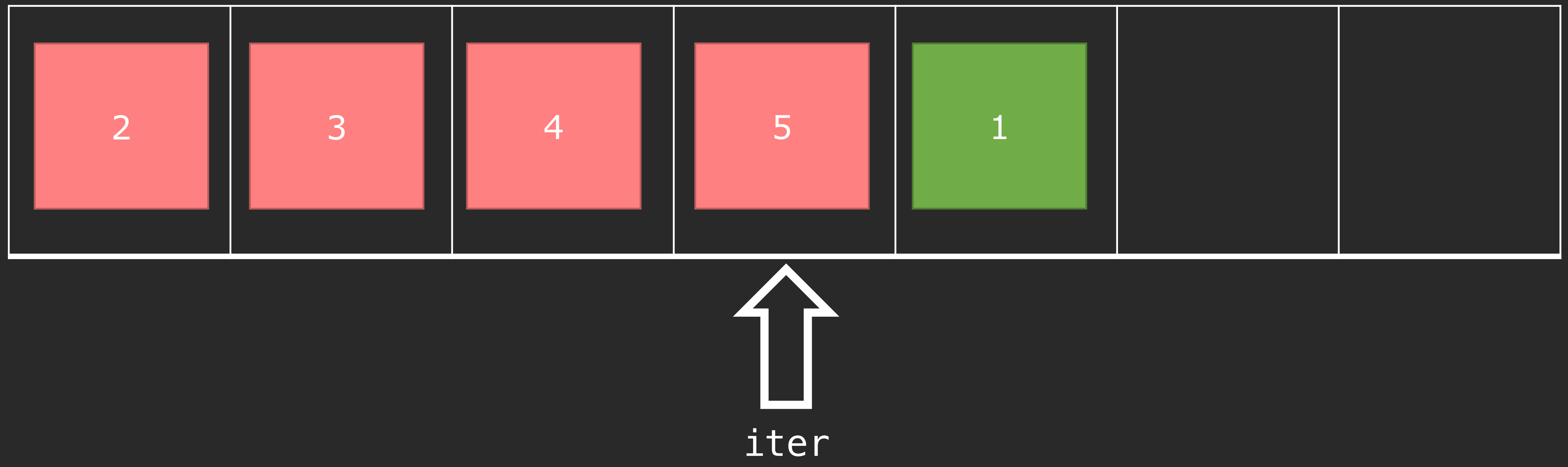
Let’s run through this code!



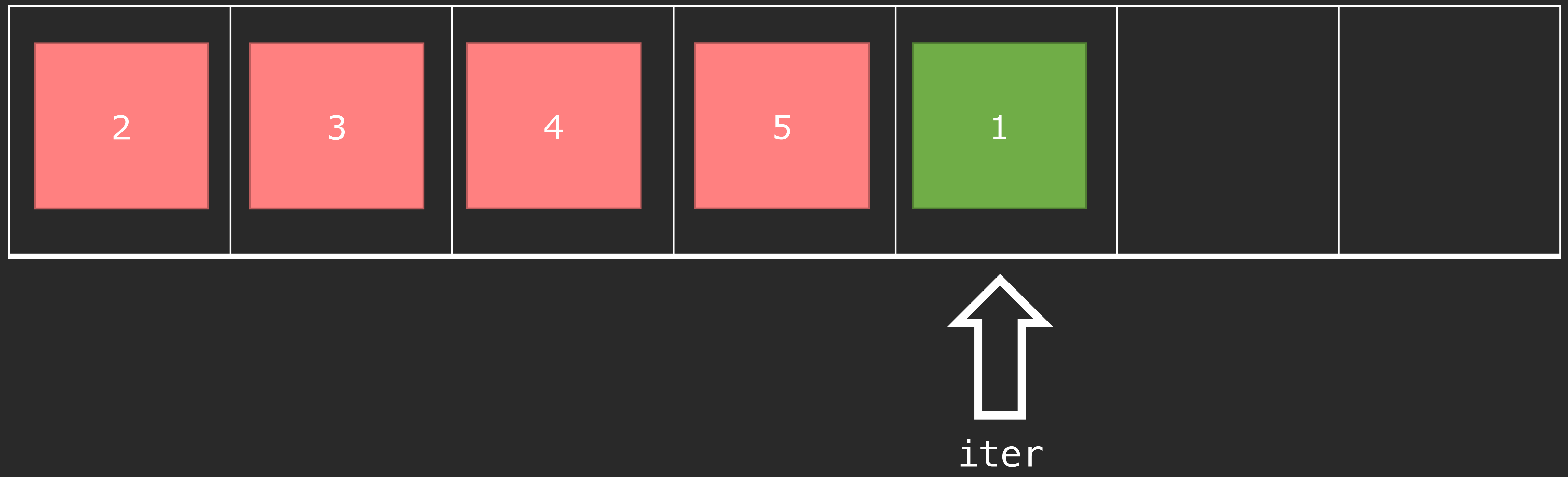
Let’s run through this code!



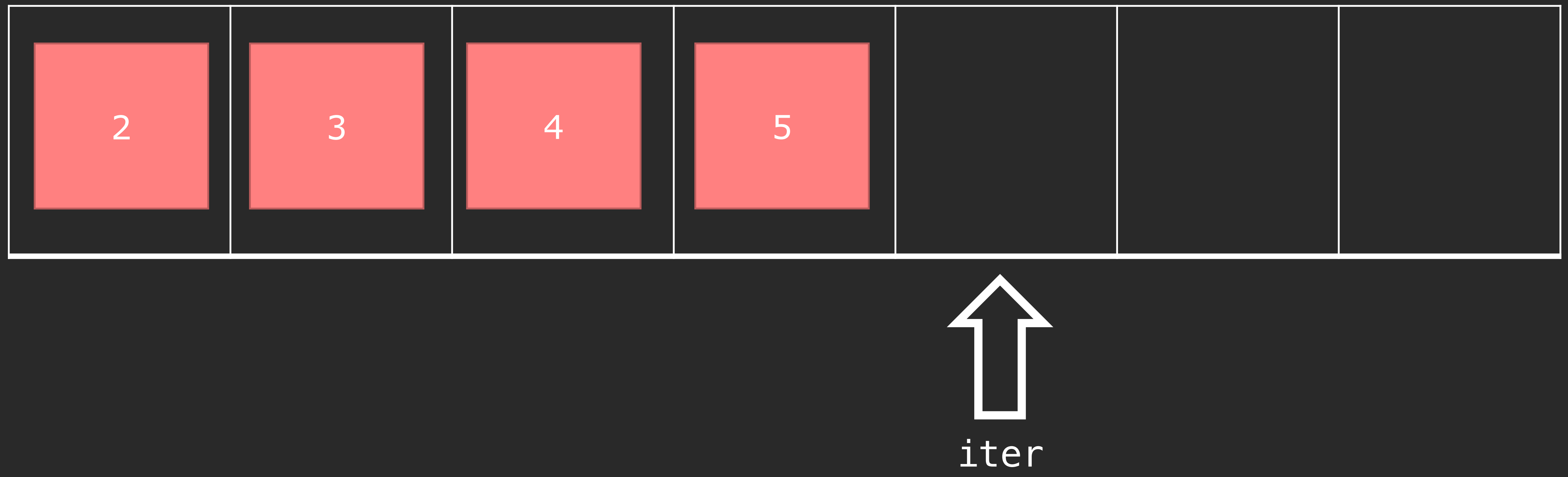
Let’s run through this code!



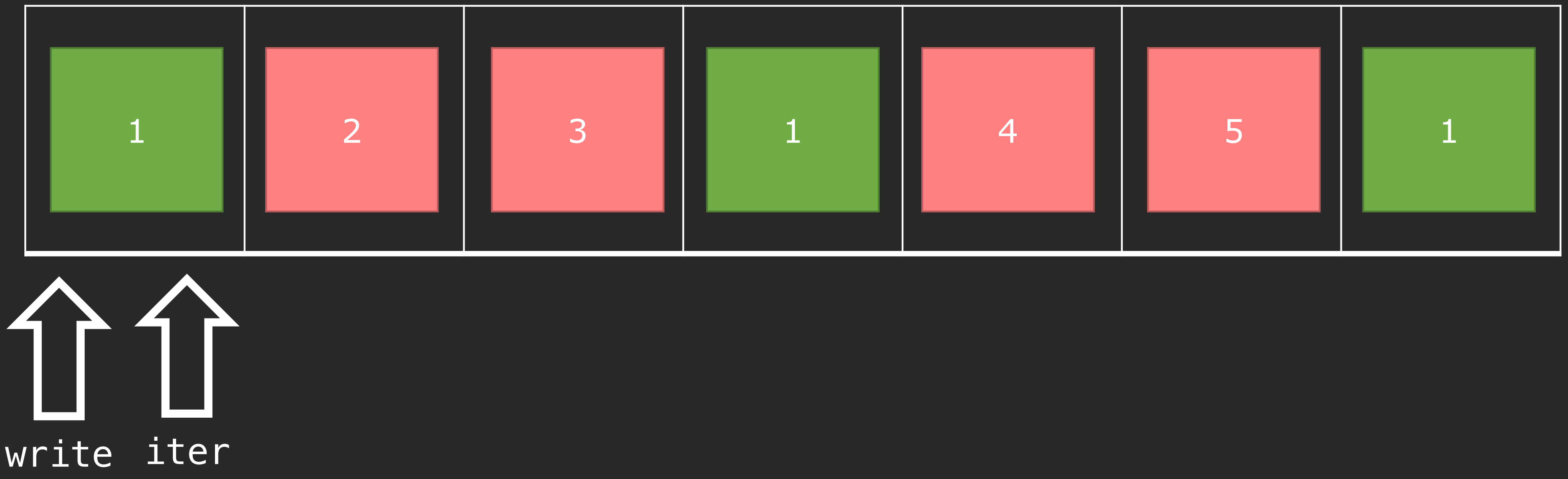
Let’s run through this code!



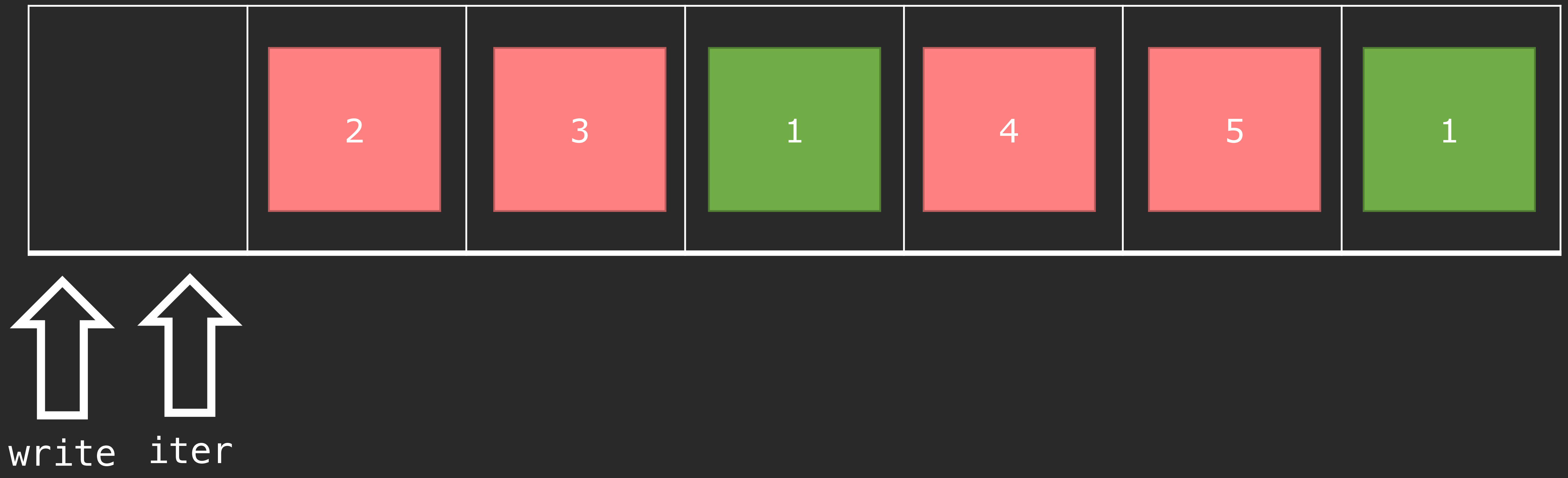
Let’s run through this code!



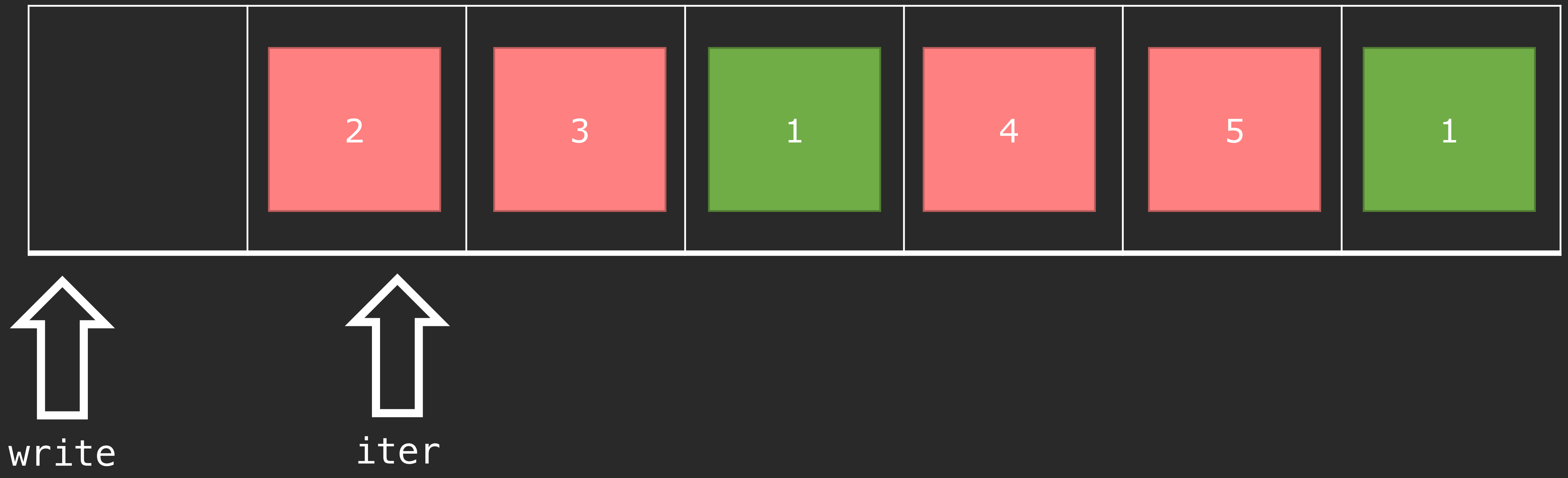
Let’s run through this code!



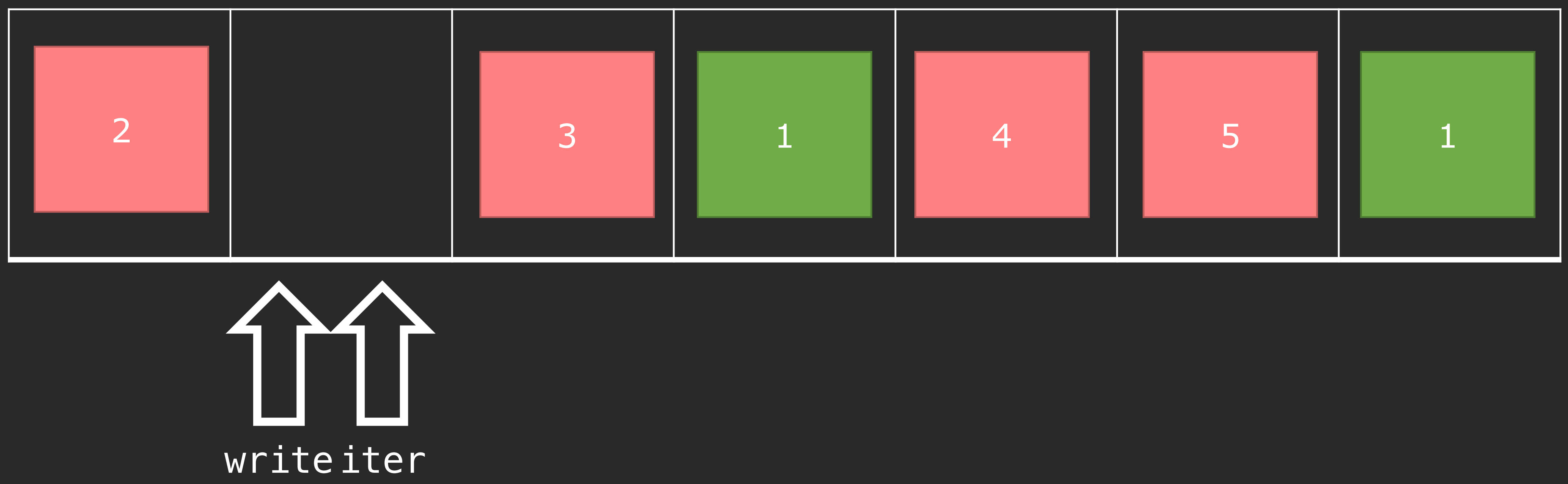
Let’s run through this code!



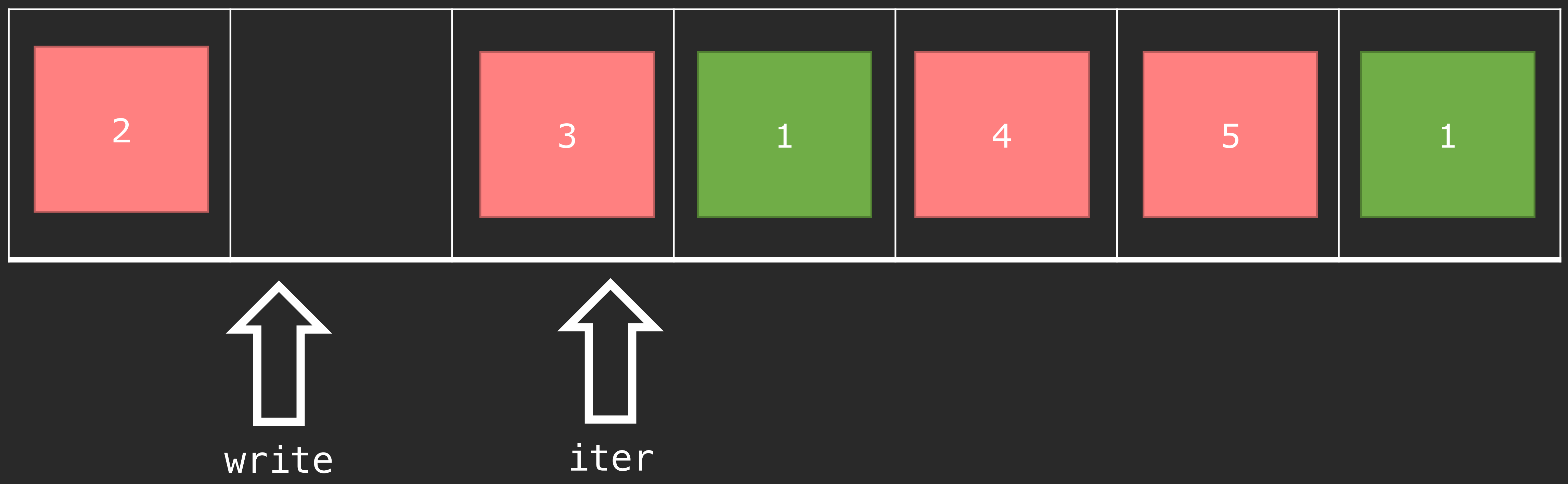
Let’s run through this code!



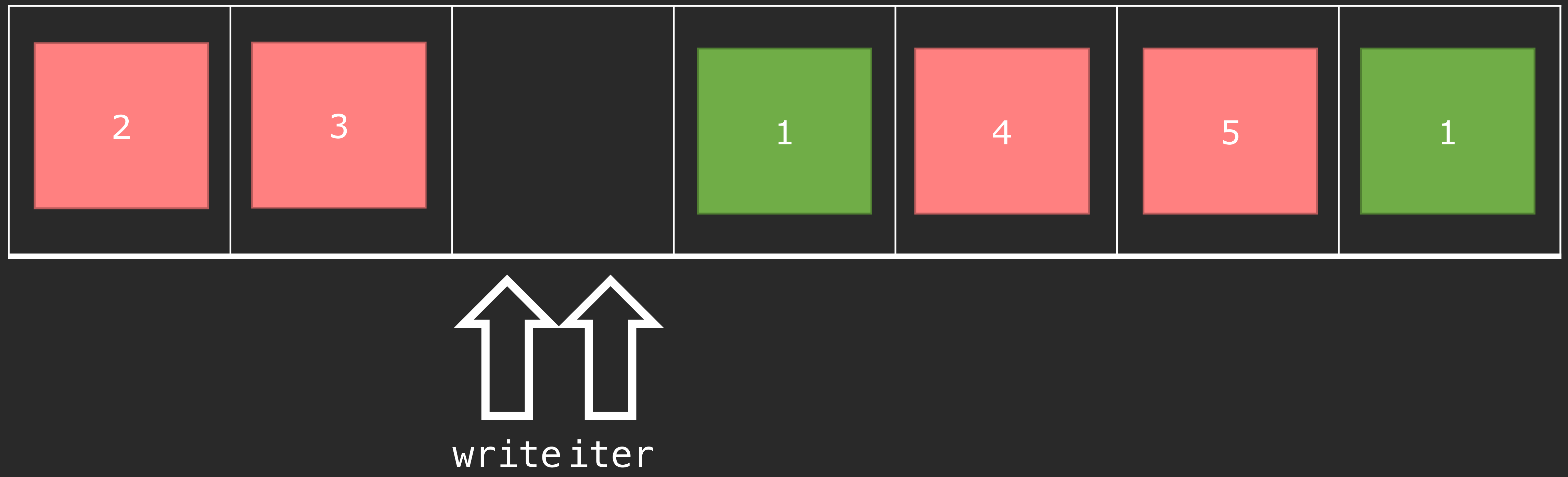
Let’s run through this code!



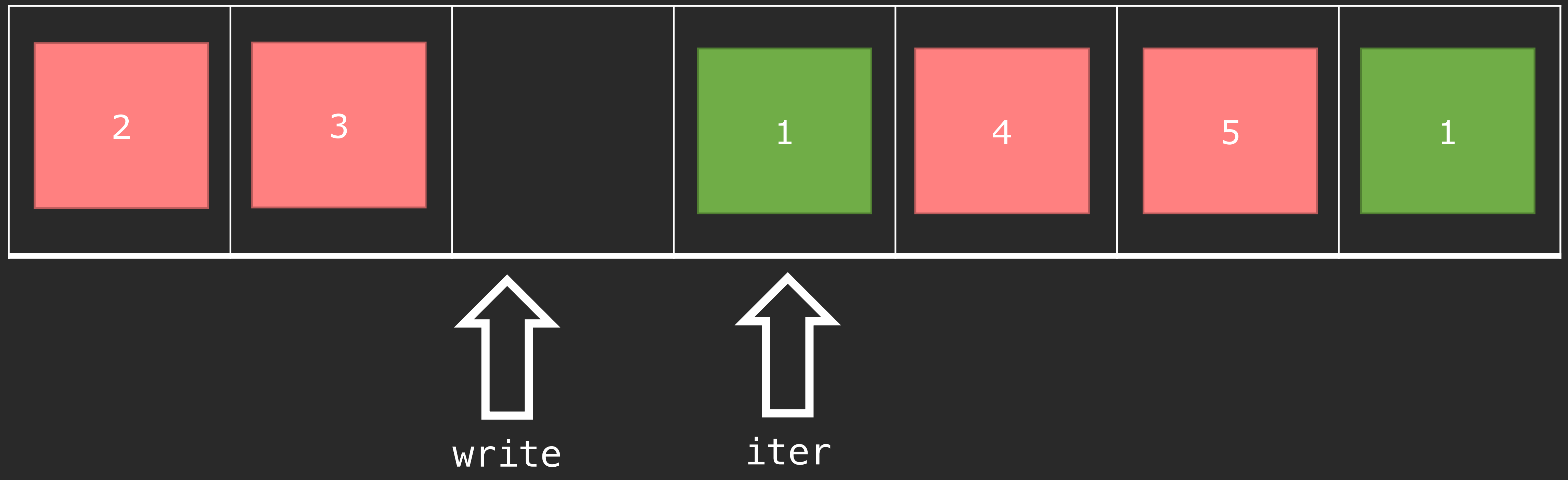
Let's run through this code!



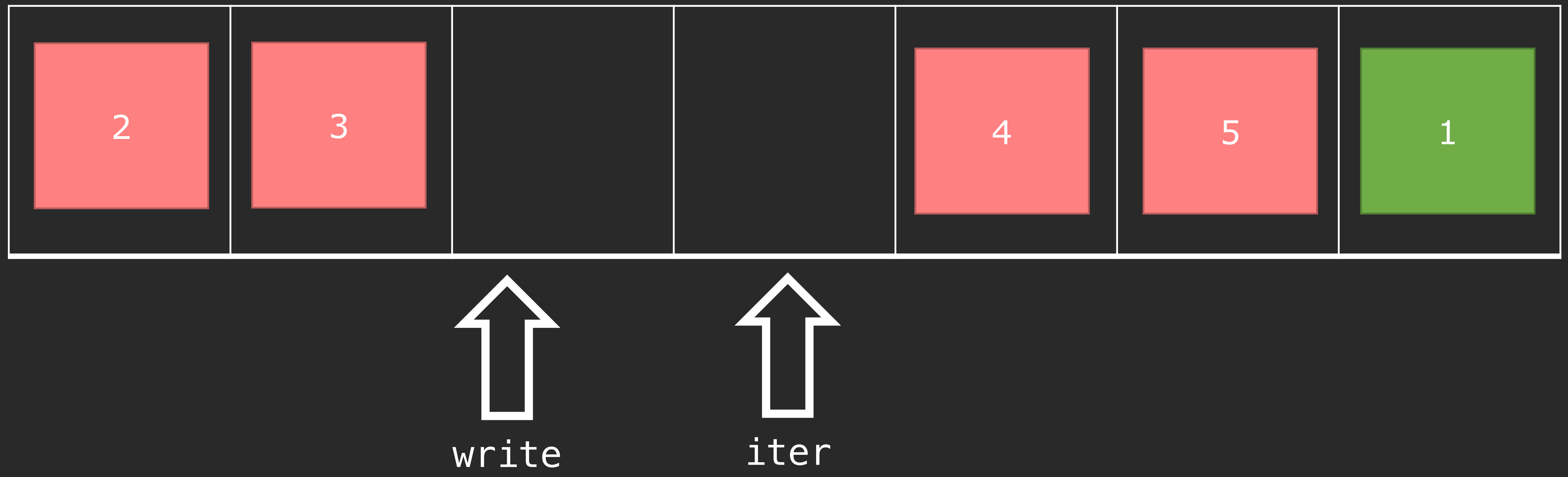
Let’s run through this code!



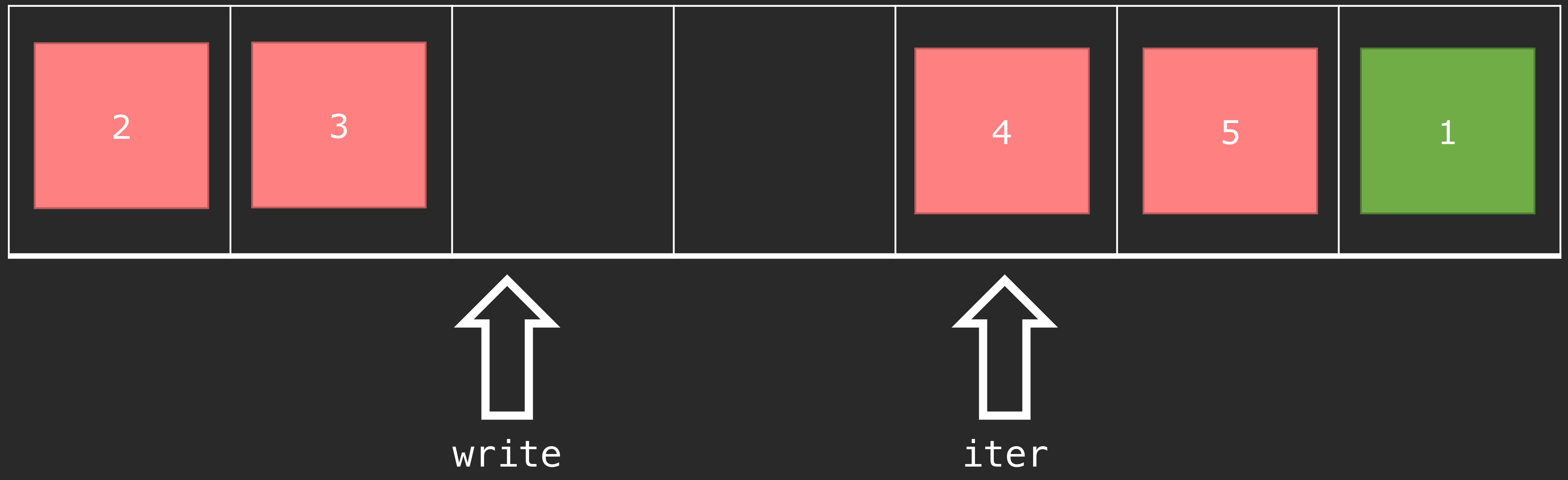
Let’s run through this code!



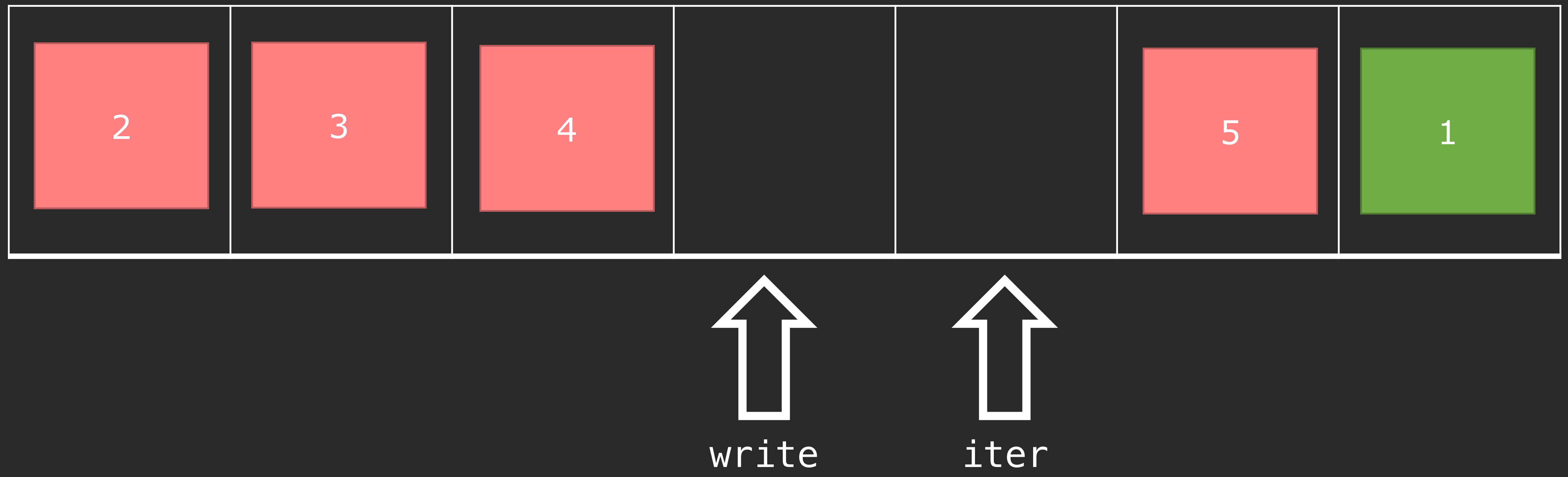
Let’s run through this code!



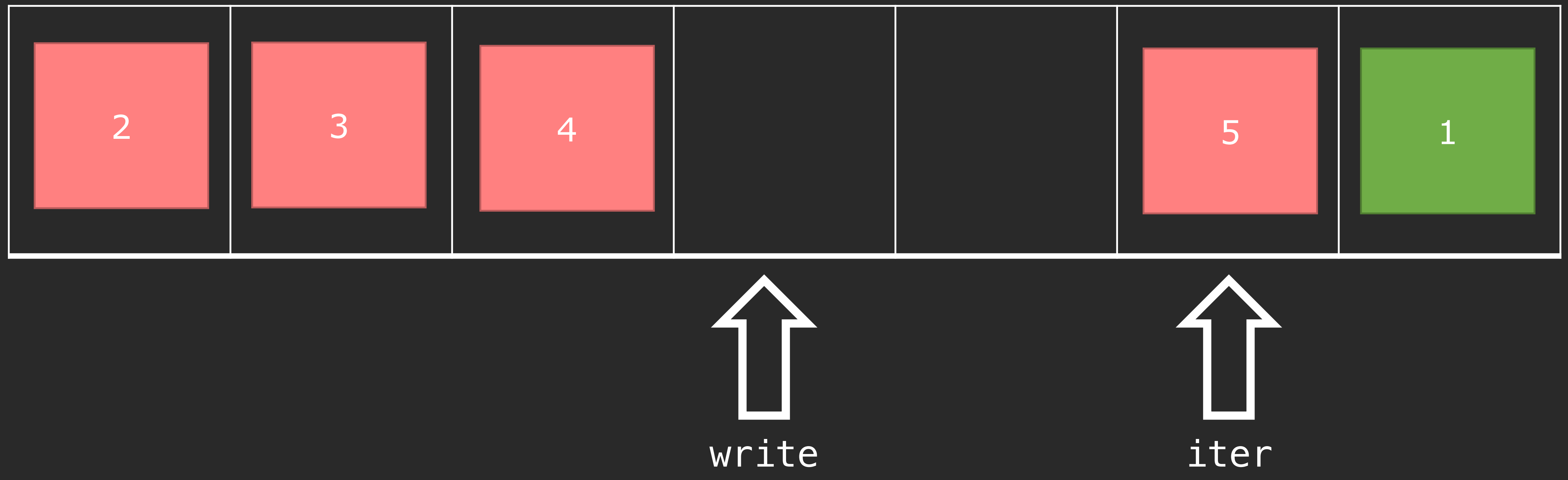
Let’s run through this code!



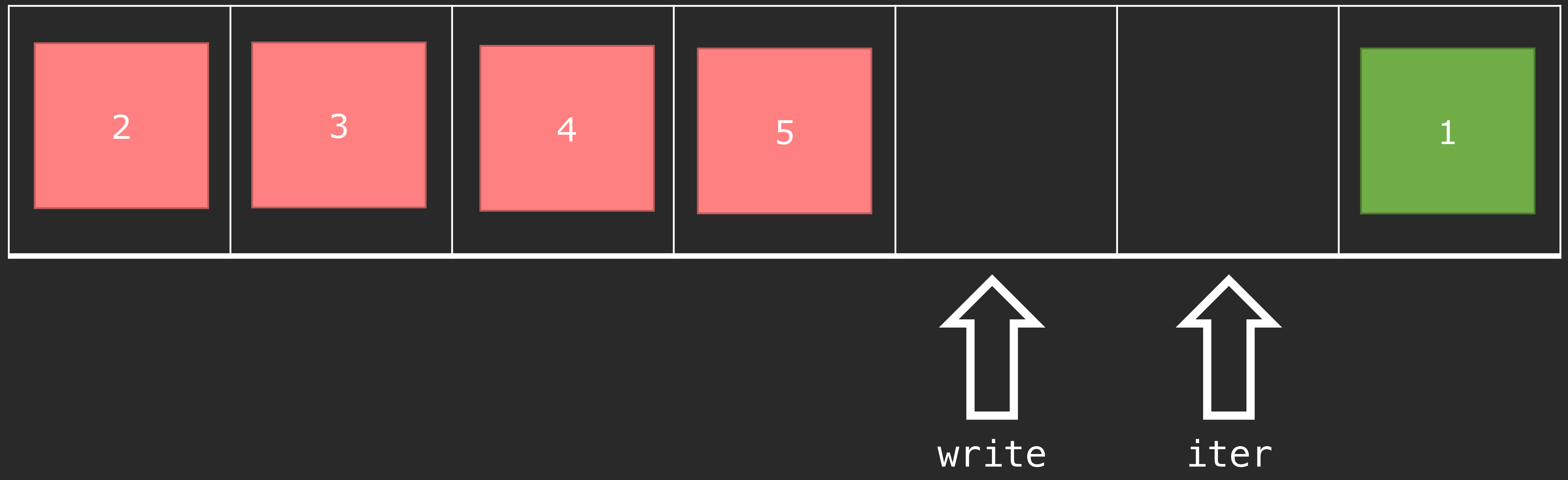
Let’s run through this code!



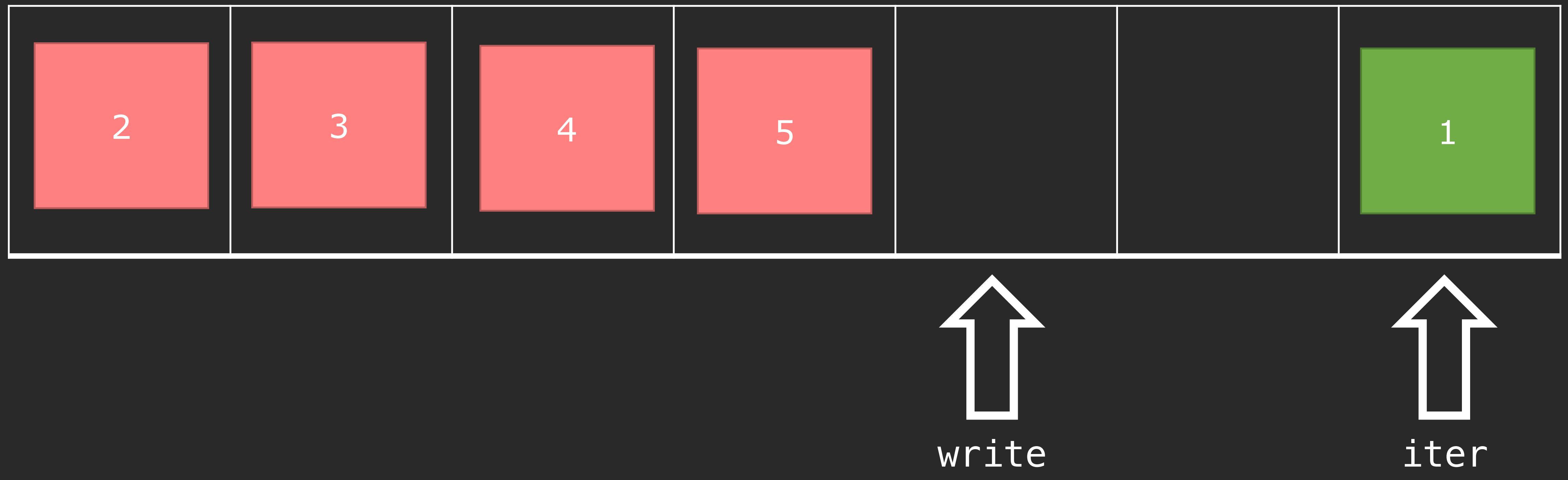
Let’s run through this code!



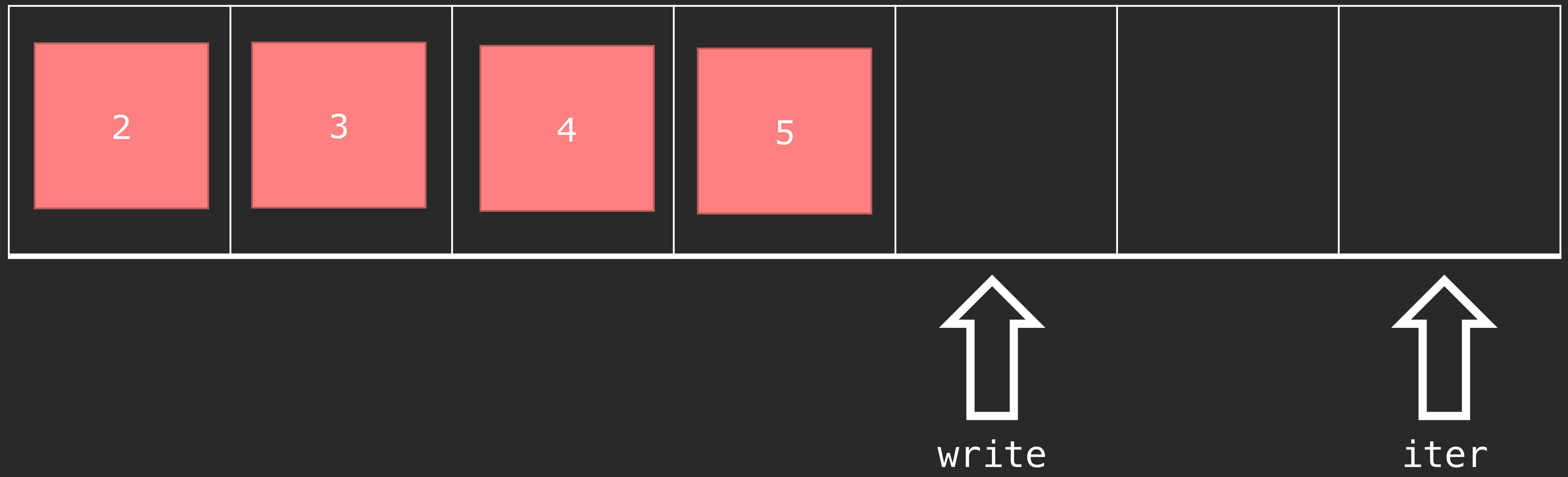
Let’s run through this code!



Let’s run through this code!



Let’s run through this code!



we’ll discuss more next lecture!

template functions

Can we handle different types?

```
int main() {
  auto [min, max] = my_minmax(3, 6);
  cout << min << endl; // 3
  cout << max << endl; // 6
}

pair<int, int> my_minmax(int a, int b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Can we handle different types?

```
int main() {
  auto [min, max] = my_minmax(“Anna”, “Avery”);
  cout << min << endl; // Anna – first alphabetical
  cout << max << endl; // Avery
}
```

One way: overloaded functions.

```
pair<int, int> my_minmax(int a, int b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

```
pair<double, double> my_minmax(double a, double b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

```
pair<string, string> my_minmax(string a, string b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Bigger problem: how do you handle user defined types?

An observation: the highlighted parts are identical.

```
pair<int, int> my_minmax(int a, int b) {
  if (a < b) return {a, b};
  else return {b, a};
}
--------------------------------------------------------------
pair<double, double> my_minmax(double a, double b) {
  if (a < b) return {a, b};
  else return {b, a};
}
--------------------------------------------------------------
pair<string, string> my_minmax(string a, string b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Only the types are different.

```
pair<int, int> my_minmax(int a, int b) {
  if (a < b) return {a, b};
  else return {b, a};
}
-----------------------------------------------------------------
pair<double, double> my_minmax(double a, double b) {
  if (a < b) return {a, b};
  else return {b, a};
}
-----------------------------------------------------------------
pair<string, string> my_minmax(string a, string b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Let’s write a general form in terms of a type T.

```
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

```
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

```
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Let’s write a general form in terms of a type T.

```
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

We now have a generic function!

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

We now have a generic function!

Declares the next function is a template.

Specifies T is some arbitrary type.

List of template arguments.

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Scope of template argument T limited to function.

是一个template，并不是一个function，把类型明确了之后就是个function

How do you call such a function?

```
my_minmax<string>(“Anna”, “Avery”);
```

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Explicit Instantiation of Templates

```
my_minmax<string>("Anna", "Avery");
```

Explicitly states
T = string

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Compiler replaces
every T with string

How do you call such a function?

```
my_minmax<string>(“Anna”, “Avery”);
```

```
template <typename string>
pair<string, string> my_minmax(string a, string b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

How do you call such a function?

```
my_minmax(3, 4);
```

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

How do you call such a function?

uniform generalization fails

```
my_minmax(3, 4);
```

Compiler deduces
T = int

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Compiler replaces
every T with int

Be careful: type deduction can’t read your mind!

```
my_minmax(“Anna”, “Avery”);
```

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Be careful: type deduction can’t read your mind!

```
my_minmax(“Anna”, “Avery”);
```

Compiler deduces
T = char* (C-string)

```
template <typename T>
pair<T, T> my_minmax(T a, T b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

Compiler replaces
every T with char*

Comparing pointers,
not what you want!

And just in case the type is a large collection.

```
template <typename T>
pair<T, T> my_minmax(const T& a, const T& b) {
  if (a < b) return {a, b};
  else return {b, a};
}
```

T = vector<int> would be okay here.

Preview of template errors

```
my_minmax(cout, cout);
```

- Semantic error: you can’t call operator < on two streams.
- Conceptual error: you can’t find the min or max of two streams.

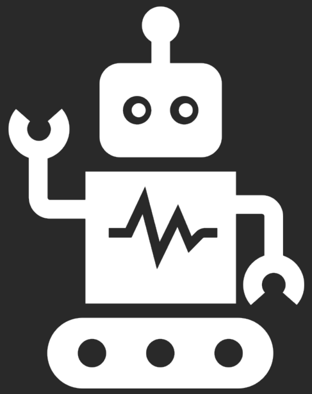
- The compiler deduces the types and literally replaces the types. Compiler will produce semantic errors, not conceptual error.
- This turns out to be a headache!

Preview of template errors

```
std::find(X, Y, Z, W);
```

- Semantic error: [some horrifying code you didn't write failed]
- Conceptual error: [you called the function incorrectly]

- Every quarter on CS 106B Piazza: "Compiler points to an error in the Stanford library. Stanford library is broken!"



Example

Templates: declaration and instantiation

Your turn: make this function generic!

```
int getInteger(const string& prompt, const string& reprompt) {
  while (true) {
    cout << prompt;
    string line; int result; char extra;
    if (!getline(cin, line))
      throw domain_error("[shortened]");
    istringstream iss(line);
    if (iss >> result && !(iss >> extra)) return result;
    cout << reprompt << endl;
  }
}
```

Your turn: make this function generic!

```
template <typename T>
T getType(const string& prompt, const string& reprompt) {
  while (true) {
    cout << prompt;
    string line; T result; char extra;
    if (!getline(cin, line))
      throw domain_error("[shortened]");
    istringstream iss(line);
    if (iss >> result && !(iss >> extra)) return result;
    cout << reprompt << endl;
  }
}
```

concept lifting

Concept Lifting

Looking at the assumptions you place on the parameters and questioning if they are really necessary.

Can you solve a more general problem by relaxing the constraints?

Why write generic functions?

Count how many times 3 appears in a vector<int>.
Count how many times 4.7 appears in a list<double>.
Count how many times ‘X’ appears in a string.
Count how many times ‘X’ appears in a deque<char>.
Count how many times 5 appears in the second half of a vector<int>.
Count how many elements in the second half of a vector<int> are at most 5.

How many times does the integer [val] appear in an entire vector of integers?

```
template <>
int countOccurences(const vector<int>& vec,
                                  int val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i] == val) ++count;
  }
  return count;
}
```

What unnecessary assumption does the function make?

How many times does the integer [val] appear in an entire vector of integers?

```
template <>
int countOccurences(const vector<int>& vec,
                                       int val) {
   int count = 0;
   for (size_t i = 0; i < vec.size(); ++i) {
      if (vec[i] == val) ++count;
   }
   return count;
}
```

What unnecessary assumption does the function make?

How many times does the [type] [val] appear in an entire vector of [type]?

```
template <typename DataType>
int countOccurences(const vector<DataType>& vec,
                                 DataType val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i] == val) ++count;
  }
  return count;
}
```

What unnecessary assumption does the function make?

How many times does the [type] [val] appear in an entire vector of [type]?

```
template <typename DataType>
int countOccurences(const vector<DataType>& vec,
                                DataType val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i] == val) ++count;
  }
  return count;
}
```

What unnecessary assumption does the function make?

How many times does the [type] [val] appear in an entire [collection] of [type]?

collection有assumption：sequence collection，但是还有associative collection

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                              DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

This code does not work. Why?

How many times does the [type] [val] appear in an entire [collection] of [type]?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                         DataType val) {
  int count = 0;
  }
  return count;
}
```

```
list<int> list = {1.1, 3.14, 3.14, 3.14, 1.1};
int count = countOccurences(list, 3.14);
```

Imagine we called countOccurences with a list.

How many times does the [type] [val] appear in an entire [collection] of [type]?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                          DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

We are indexing through a potentially unindexable collection.

How many times does the [type] [val] appear in an entire [collection] of [type]?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                             DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (*(iter + i) == val) ++count;
  }
  return count;
}
```

Bad! Why?

How many times does the [type] [val] appear in an entire [collection] of [type]?

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

Solved using iterators!

How many times does the [type] [val] appear in an entire [collection] of [type]?

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

This still makes one last assumption.

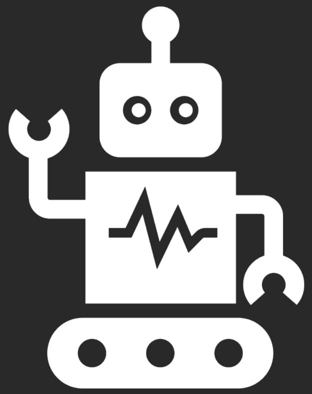
How many times does the [type] [val] appear in [a range of elements]?

用最通用的inputI ter

assumption lifting

```
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;
}
```

We even give control of where the start and end should be.



Example

Lifting countOccurences

Can we solve all of these now?

Count how many times 3 appears in a vector<int>.
Count how many times 4.7 appears in a list<double>.
Count how many times 'X' appears in a string.
Count how many times 'X' appears in a deque<char>.
Count how many times 5 appears in the second half of a vector<int>.
Count how many elements in the second half of a vector<int> are at most 5.

We are stuck on the last one. How do we customize the predicate?

```
countOccurences(v.begin(), v.end(), 3);
countOccurences(l.begin(), l.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);
countOccurences(v.begin() + v.size()/2, v.end(), 5);
```

Count how many elements in the second half of a vector<int> are at most 5.

We’ll tackle this next time!

implicit interfaces

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
vector<int> v1{1, 2, 3, 1, 2, 3};
vector<int> v2{1, 2, 3};
countOccurences(v1.begin(), v1.end(), v2);
```

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
vector<int> v1{1, 2, 3, 1, 2, 3};
vector<int> v2{1, 2, 3};
countOccurences(v1.begin(), v1.end(), v2);
```

`vector<int>`

`vector<int>::iterator`

`vector<int>::iterator`

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
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;
}
```

The compiler literally replaces each template parameter with whatever you instantiate it with.

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

依次替换
InputIter,vector<int>
iter datatype: vector

*iter is an int, can't
== to a vector

A template function defines an implicit interface that each template parameter must satisfy.

```
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;
}
```

What must be true of InputIt and DataType?

A template function defines an implicit interface that each template parameter must satisfy.

```
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;
}
```

begin must be copyable.

A template function defines an implicit interface that each template parameter must satisfy.

```
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;
}
```

iter must be equality comparable to end.

A template function defines an implicit interface that each template parameter must satisfy.

```
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;
}
```

You must be able to increment iter.

A template function defines an implicit interface that each template parameter must satisfy.

```
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;
}
```

You must be able to dereference iter and equality compare it to val.

Each template parameter must have the operations the function assumes it has.

`InputIt` must support
- copy assignment (`iter = begin`)
- prefix operator (`++iter`)
- comparable to end (`begin != end`)
- dereference operator (`*iter`)

`DataType` must support
- comparable to `*iter`

Nasty compile errors if instantiated type do not support these.

Each template parameter must have the operations the function assumes it has.

`InputIt` must support

- copy assignment (`iter = begin`) // bad: streams
- prefix operator (`++iter`) // bad: collections
- comparable to end (`begin != end`) // bad: anything not an iterator
- dereference operator (`*iter`) // bad: numeric types

`DataType` must support

- comparable to `*iter` // bad: iterators of wrong type

Nasty compile errors if instantiated type do not support these.

Template errors are not fun to debug.

There's a StackOverflow thread on maximizing lines of error messages with fewest lines of code.

Basically use a lot of template features incorrectly.

```
averyw09521@Averys-MacBook-Air lectures-new % g++ -std=c++2a templates-1.cpp -o templates && ./templates
templates-1.cpp:19:15: error: invalid operands to binary expression ('int' and 'std::__1::vector<int, std::__1::allocato
    if (*iter == val) ++count;
        ~~~~~ ^  ~~~
templates-1.cpp:32:3:       in instantiation of function template specialization 'countOccurences<std::__1::__wrap_iter<
      std::__1::vector<int, std::__1::allocator<int> > >' requested here
  countOccurences(v.begin(), v.end(), v2);
  ^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/system_error:391:1:       candidate function not viable: r
      from 'int' to 'const std::__1::error_code' for 1st argument
operator==(const error_code& __x, const error_code& __y) _NOEXCEPT
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/system_error:398:1:       candidate function not viable: r
      from 'int' to 'const std::__1::error_code' for 1st argument
operator==(const error_code& __x, const error_condition& __y) _NOEXCEPT
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/system_error:406:1:       candidate function not viable: r
      from 'int' to 'const std::__1::error_condition' for 1st argument
operator==(const error_condition& __x, const error_code& __y) _NOEXCEPT
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/system_error:413:1:       candidate function not viable: r
      from 'int' to 'const std::__1::error_condition' for 1st argument
operator==(const error_condition& __x, const error_condition& __y) _NOEXCEPT
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/utility:580:1:       candidate template ignored: could not
      'pair<type-parameter-0-0, type-parameter-0-1>' against 'int'
operator==(const pair<_T1,_T2>& __x, const pair<_T1,_T2>& __y)
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/iterator:712:1:       candidate template ignored: could no
      'reverse_iterator<type-parameter-0-0>' against 'int'
operator==(const reverse_iterator<_Iter1>& __x, const reverse_iterator<_Iter2>& __y)
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/iterator:941:1:       candidate template ignored: could no
      'istream_iterator<type-parameter-0-0, type-parameter-0-1, type-parameter-0-2, type-parameter-0-3>' against 'int'
operator==(const istream_iterator<_Tp, _CharT, _Traits, _Distance>& __x,
^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/iterator:1045:6:       candidate template ignored: could r
      'istreambuf_iterator<type-parameter-0-0, type-parameter-0-1>' against 'int'
bool operator==(const istreambuf_iterator<_CharT,_Traits>& __a,
     ^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/iterator:1153:1:       candidate template ignored: could r
      'move_iterator<type-parameter-0-0>' against 'int'
operator==(const move_iterator<_Iter1>& __x, const move_iterator<_Iter2>& __y)
```

Template interfaces: explicit vs. implicit

```
countOccurences(v1.begin(), v1.end(), v2);
```

- Semantic error: *iter == val compares int with vector<int>.
- Conceptual error: you can’t find the min or max of two streams.

- The compiler deduces the types and literally replaces the types. Compiler will produce semantic errors, not conceptual error.
- Really not fun to debug.

More practice: what is the implicit interface?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                                      DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

More practice: what is the implicit interface?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                          DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

Collection must have a method size() that returns an integer.

More practice: what is the implicit interface?

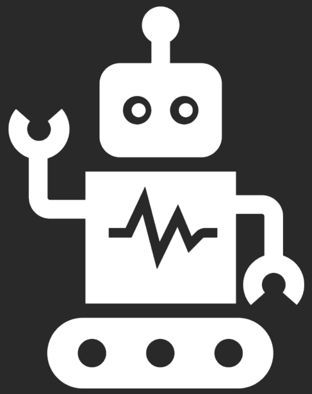
```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                         DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

Collection must support the subscript operator ([ ])

More practice: what is the implicit interface?

```
template <typename Collection, typename DataType>
int countOccurences(const Collection& list,
                         DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}
```

Furthermore, that return value must be equality comparable to DataType.



Example

When templates go wrong.

overload resolution

advanced topic

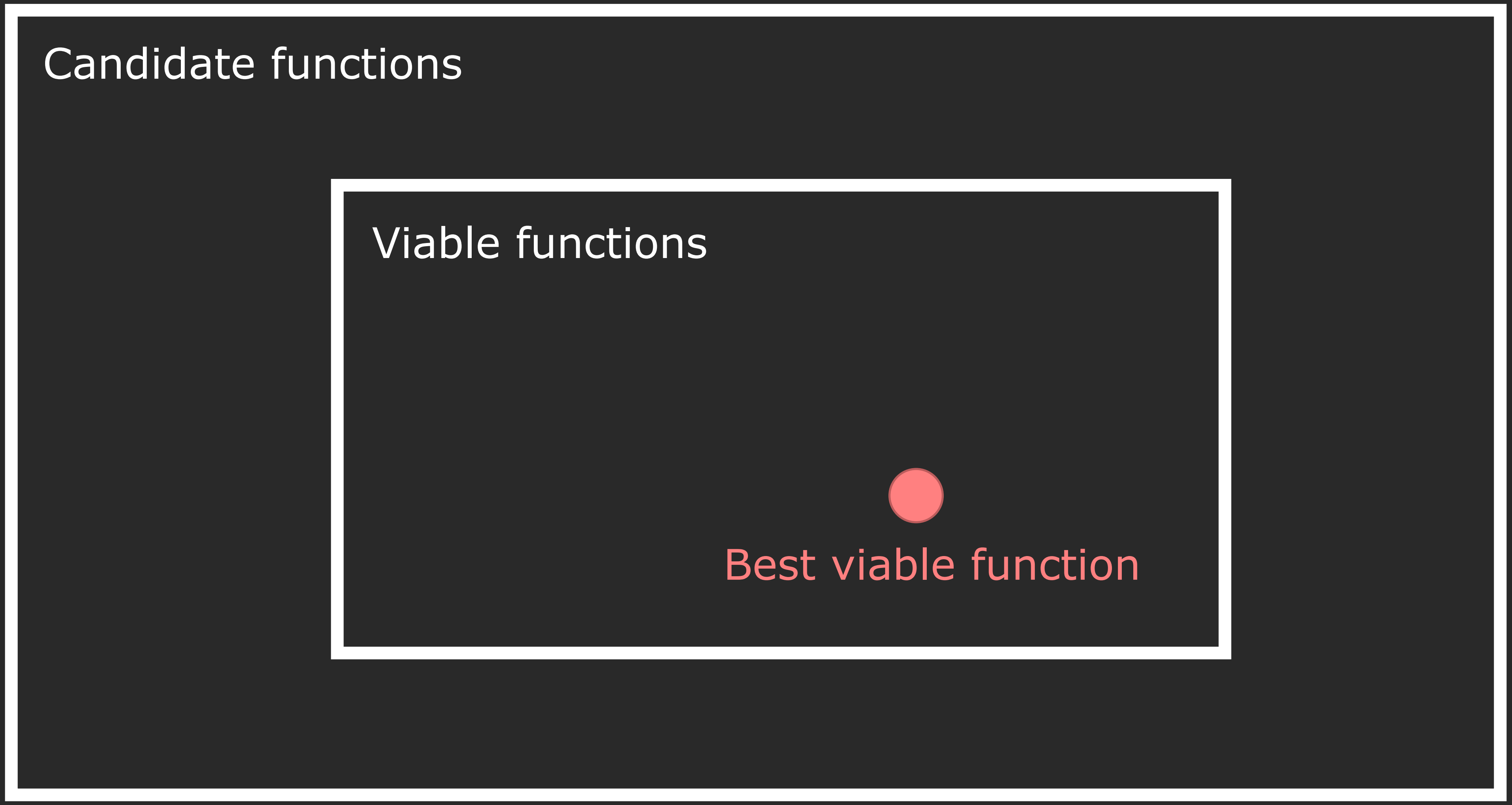
most C++ programmers don’t actively think about this

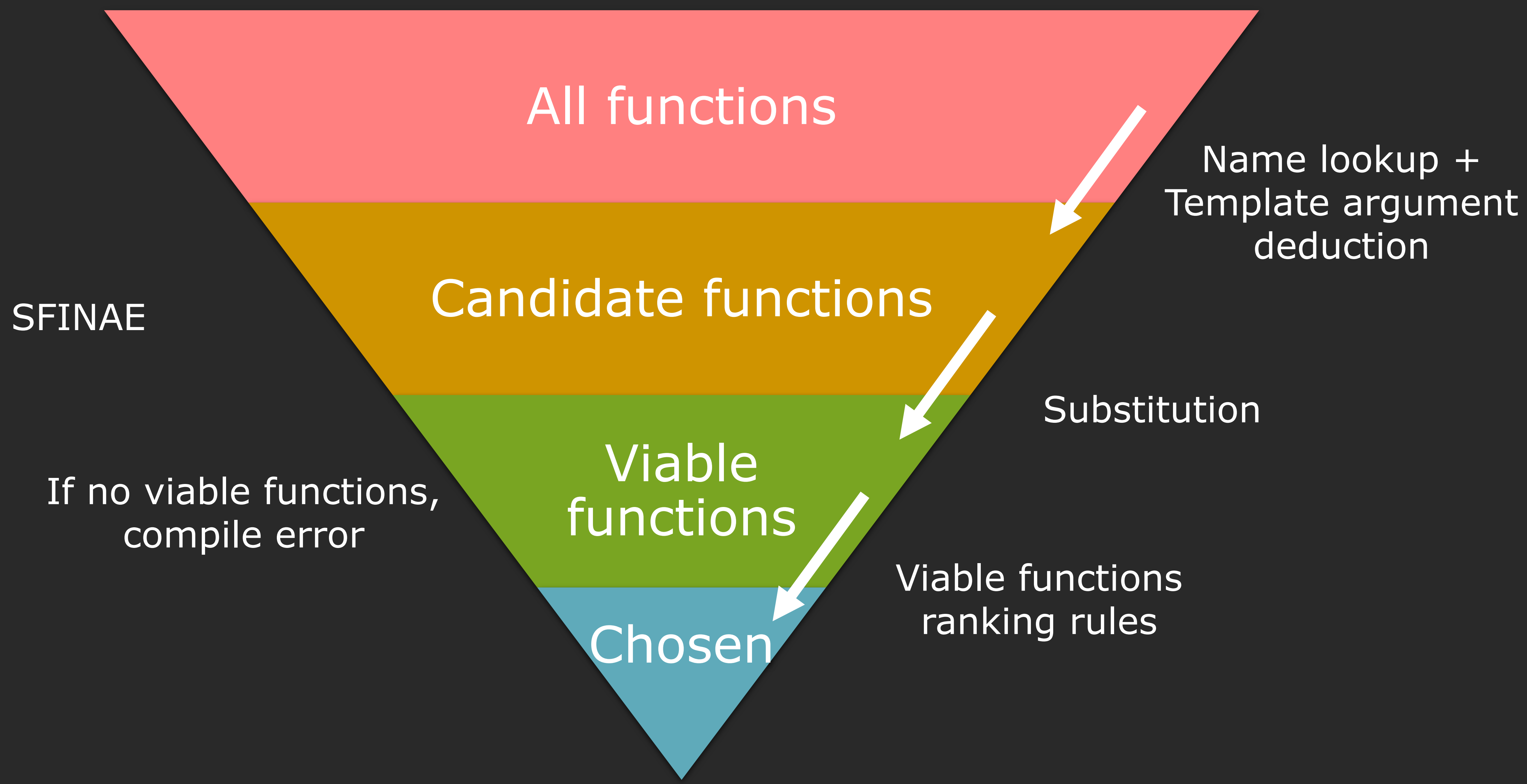
“what if there are multiple potential templates functions?”

My answer last quarter: don’t do it.

Better answer: sometimes people do that! Let’s see it.

All functions




All functions
Candidate functions
Viable functions
Chosen
Name lookup + Template argument deduction
Substitution
Viable functions ranking rules
SFINAE
If no viable functions, compile error

Overload resolution steps

- From all functions within scope, look up all functions that match the name of function call. If template is found, deduce the type.
- From all candidate functions, check the number and types of the parameters. For template instantiations, try substituting and see if implicit interface satisfied. If fails, remove these instantiations.
- From all viable functions, rank the viable functions based on the type conversions necessary and the priority of various template types. Choose the best viable function.

const –> conversion

SFINAE

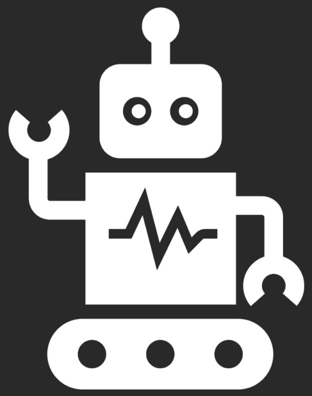
- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- When substituting the deduced types fails (in the immediate context) because the type doesn't satisfy implicit interfaces, this does not result in a compile error.
- Instead, this candidate function is not part of the viable function. The other candidates will still be processed.

Power of SFINAE

- We can implement this logic:

If substituted type does not satisfy some condition,
remove this overload from the candidate function set.

For Java users, this should remind you of Reflection.
For Python users, this might remind you of hasattr().



Example

SFNIAE example and enable_if

Substitution passes if T has a member size.

```
template <typename T>
auto printSize(const T& a) -> decltype(a.size()) {
  cout << "printing with size member function: ";
  cout << a.size() << endl;

  return a.size();
}

// T = int (fail)
// T = vector<int> (success)
// T = vector<int>* (fail)
```

Substitution passes if T can be negated.

```
template <typename T>
auto printSize(const T& a) -> decltype(-a) {
  cout << “printing with negative numeric function: ”;
  cout << -a << endl;

  return -a;
}

// T = int (success)
// T = vector<int> (fail)
// T = vector<int>* (fail)
```

Substitution passes if T can be dereferenced and called with size member function.

```
template <typename T>
auto printSize(const T& a) -> decltype(a->size()) {
   cout << "printing with pointer function: ";
   cout << a->size() << endl;

   return a->size();
}

// T = int (fail)
// T = vector<int> (fail)
// T = vector<int>* (success)
```

SFNIAE removes the overloads which do not compile, allowing you to call printSize on different types!

```
int main() {
  vector<int> vec{1, 2, 3};
  printSize(vec);          // calls first overload
  printSize(vec[1]);       // calls second overload
  printSize(&vec);         // calls third overload
  printSize(nullptr);      // compiler error
}
```

Power of SFINAE

std::enable_if<Predicate>

If Predicate is satisfied, proceed as normal.

If Predicate is not satisfied, <u>purposely</u> create a template error!

The signbit function can only be called if T is an arithmetic type.

```
template <typename T, typename
      std::enable_if<std::is_arithmetic<T>, bool>::type>
signbit(T x) {

      // implementation

}
```

The signbit function can only be called if T is an arithmetic type.

利用substitution来替换

```
template <typename T, typename
      std::enable_if<std::is_arithmetic<T>, bool>::type>
signbit(T x) {

      // implementation

}
```

This expression doesn’t compile if T is not arithmetic.

Next time

Functions and Algorithms

Your turn:
lift this function to its most generic form.

```
int main() {
    vector<int> v1{1, 2, 3, 4};
    vector<int> v2{1, 2, 4, 6};
    vector<int> v3{1, 2, 3, 4};
    vector<int> v4{1, 2, 3};

    auto [match, l1, l2] = mismatch(v1, v2); // {false, 3, 4}
    auto [match, r1, r3] = mismatch(r1, r3); // {true, 0, 0}
    auto [match, k1, k4] = mismatch(k1, k4); // undefined
}
```

Your turn:
lift this function to its most generic form.

```
tuple<bool, int, int> mismatch(const vector<int>& vec1,
                               const vector<int>& vec2)
  size_t i = 0;
  while (i < vec1.size() && vec1[i] == vec2[i]){
    ++i;
  }
  if (i == vec1.size()) return {false, 0, 0};
  else return {true, vec1[i], vec2[i]};
}
```

Your turn:
lift this function to its most generic form.

```
template <typename InputIt1, typename InputIt2>
pair<InputIt1, InputIt2> mismatch(InputIt1 first1,
                                  InputIt1 last1,
                                  InputIt2 first2)




}
```

What is the implicit interface of this template function?

the lenght of the second container >= first one

Your turn:
lift this function to its most generic form.

```
template <typename InputIt1, typename InputIt2>
pair<InputIt1, InputIt2> mismatch(InputIt1 first1,
                                  InputIt1 last1,
                                  InputIt2 first2)
  while (first1 != last1 && *first1 == *first2){
    ++first1; ++first2;
  }

  return {first1, first2};
}
```

What is the implicit interface of this template function?

Challenge Problem: Implement the logic of remove from before!

```
template <typename ForwardIt, typename T>
ForwardIt remove(ForwardIt first, ForwardIt last,
                    const T& value) {







}
```

Challenge Problem:
Implement the logic of remove from before!

```
template <typename ForwardIt, typename T>
ForwardIt remove(ForwardIt first, ForwardIt last,
                    const T& value) {
    first = std::find(first, last, value);
    if (first != last)
        for(ForwardIt i = first; ++i != last; )
            if (!(*i == value))
                *first++ = std::move(*i);
    return first;
}
```