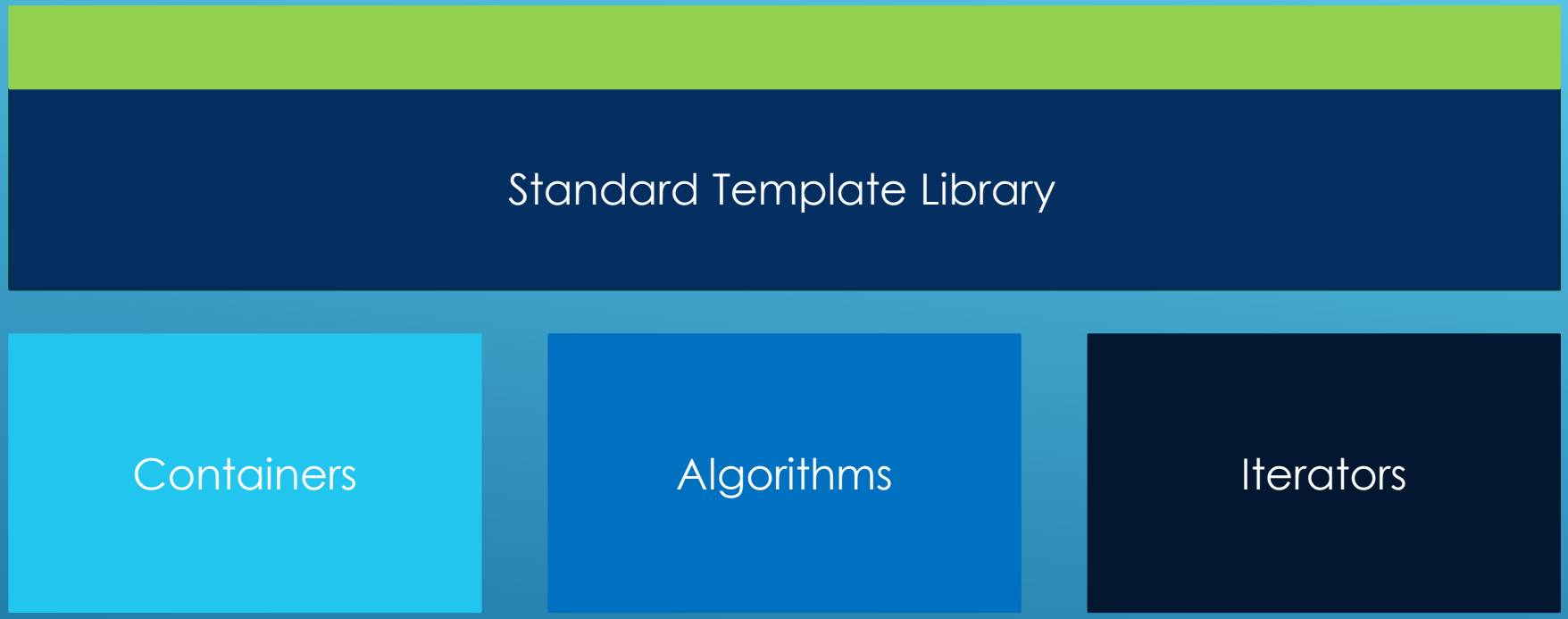
Slides

Section : STL Containers and Iterators

Slide intentionally left empty

STL Containers and Iterators : Introduction

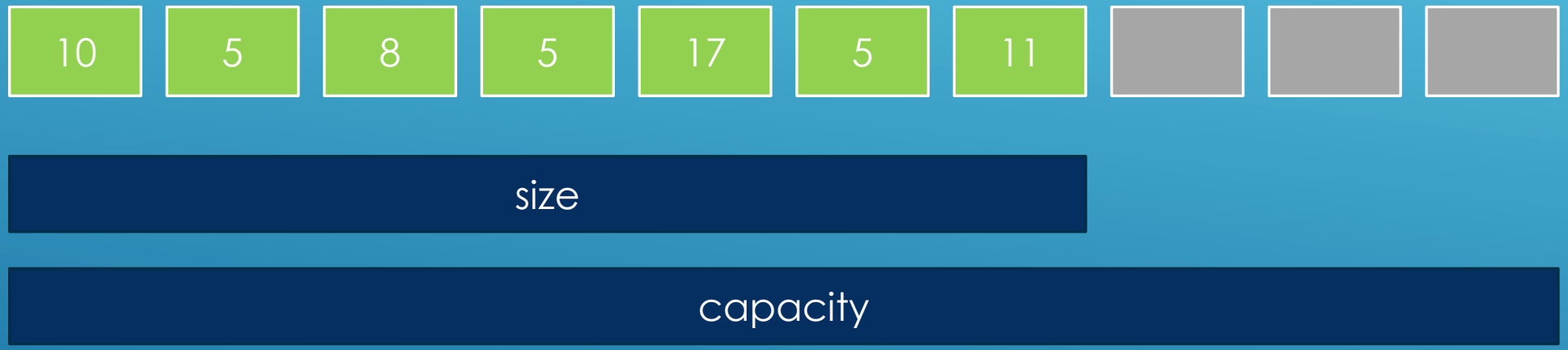

Standard Template Library
Containers
Algorithms
Iterators

Collections (Containers)

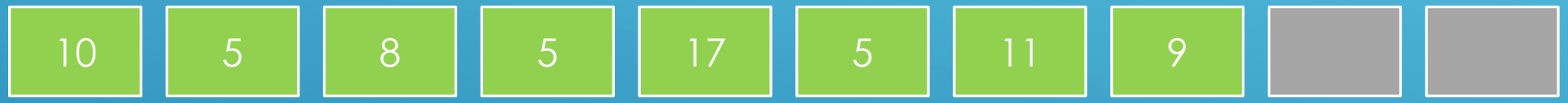
. BoxContainer is a container class that provides the features
    . add_item
    . remove_item
    . remove_all

. We can add BoxContainer's up with :
    . +=
    . +

. Additionaly we can :
    . Stream insert BoxContainers with opreator<<
    . Copy construct BoxContainer's
    . Copy assign BoxContainer's


10
5
8
5
17
5
11
size
capacity

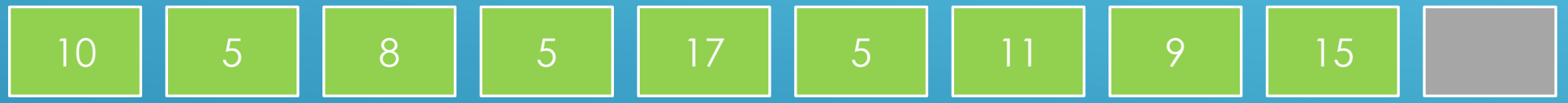
Adding elements



size

capacity

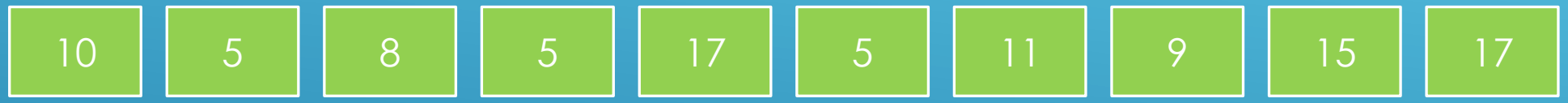
Adding elements



size

capacity

Adding elements



size

capacity

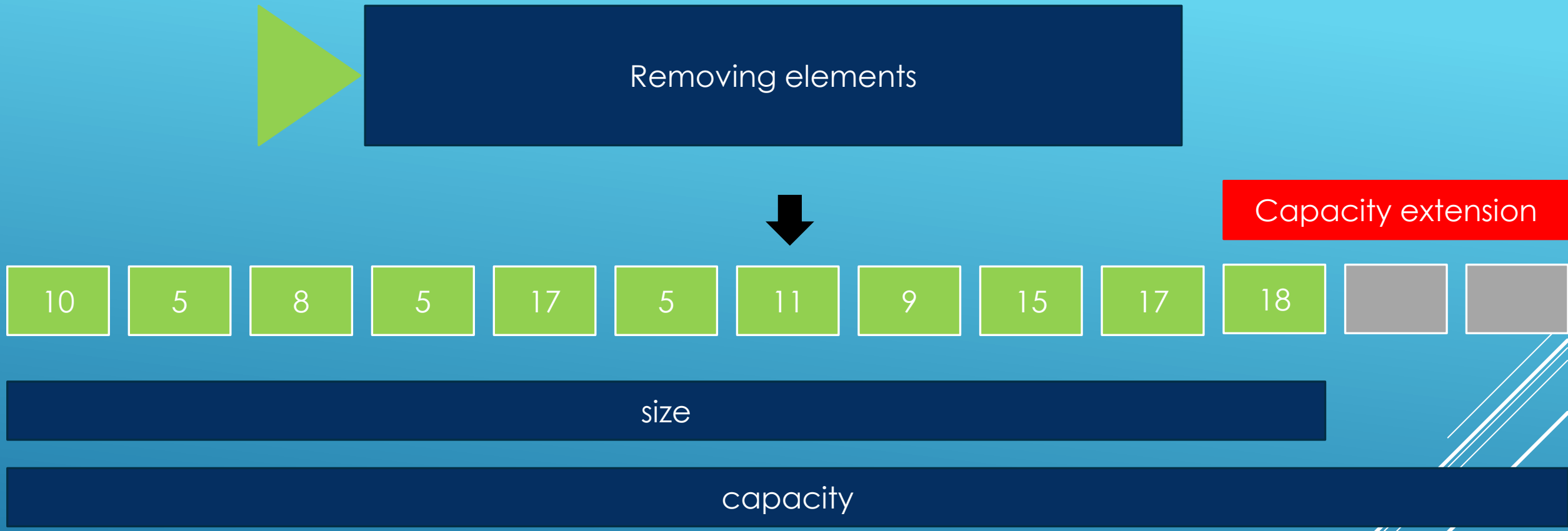

Adding elements

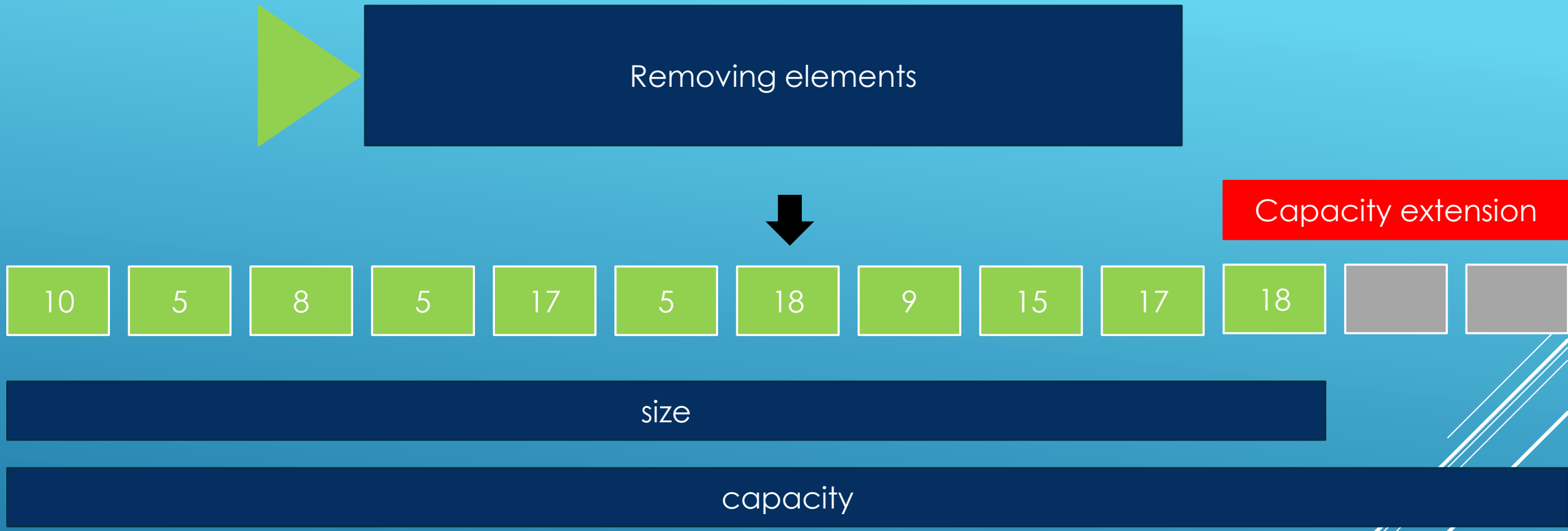

Capacity extension


10
5
8
5
17
5
11
9
15
17
18


size


capacity


Removing elements
Capacity extension
10
5
8
5
17
5
11
9
15
17
18
size
capacity

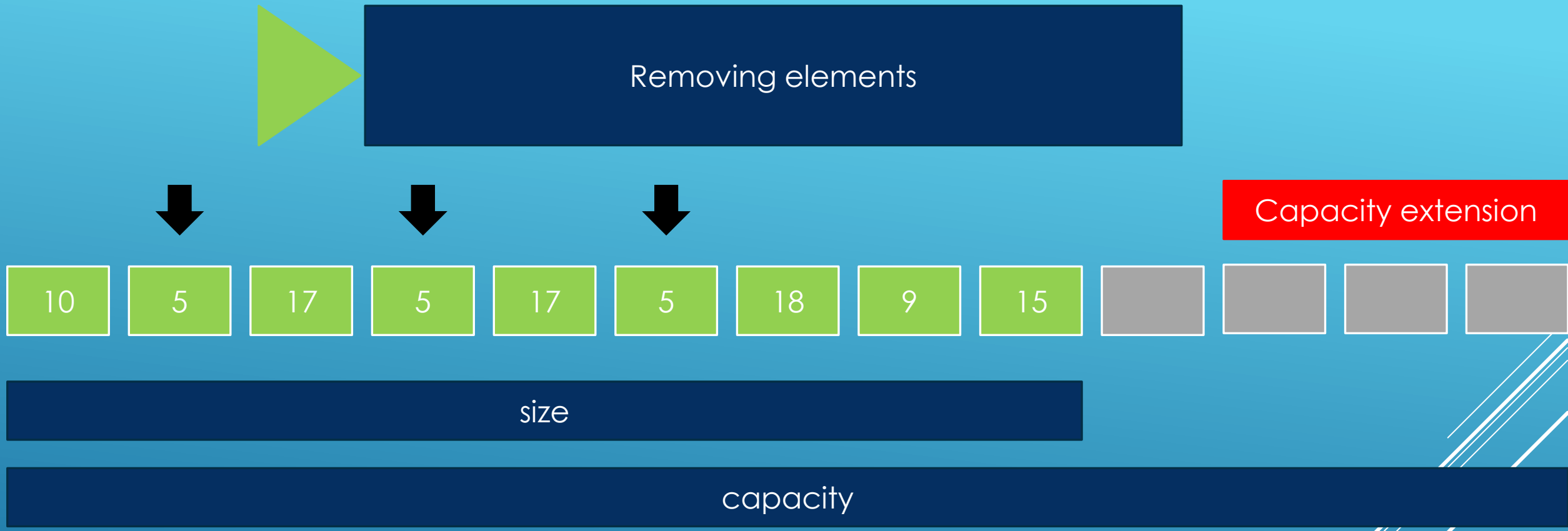

Removing elements
Capacity extension
10
5
8
5
17
5
18
9
15
17
18
size
capacity

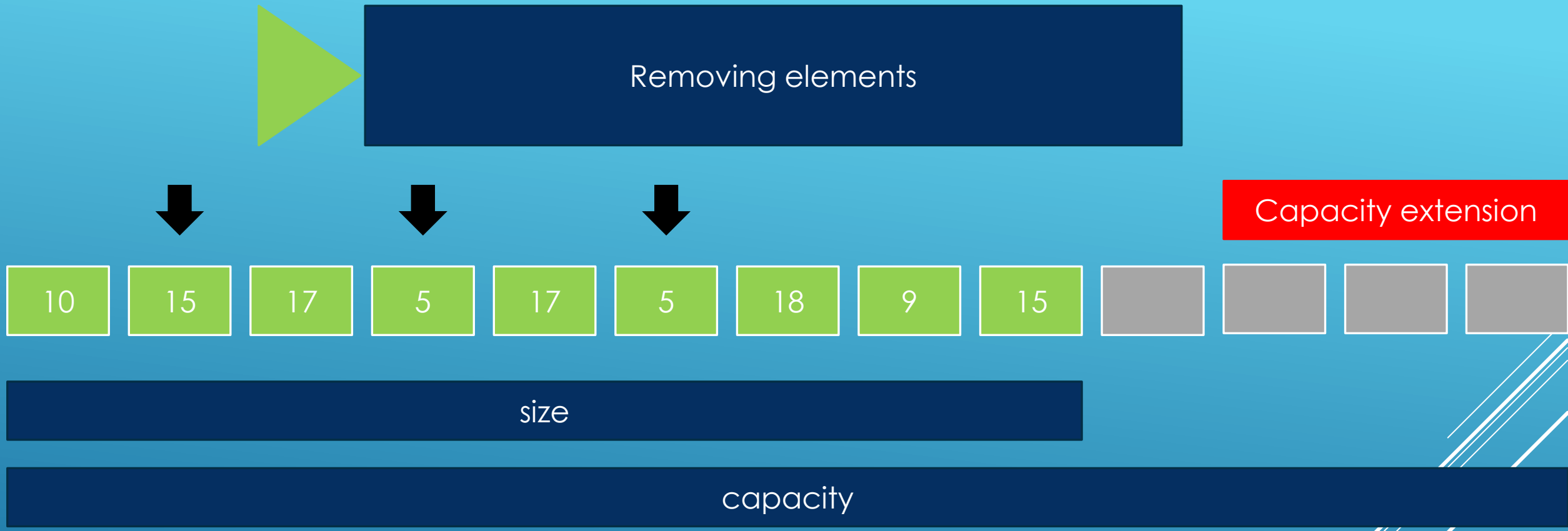
Removing elements

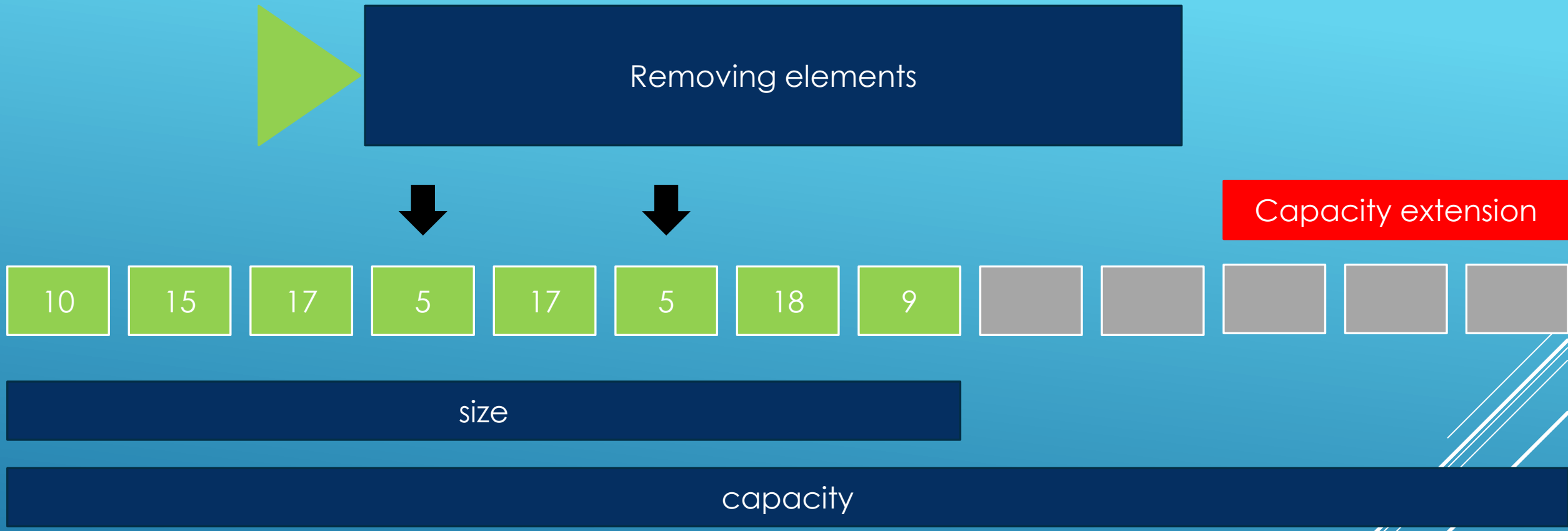

Removing elements
Capacity extension
10
5
8
5
17
5
18
9
15
17
size
capacity

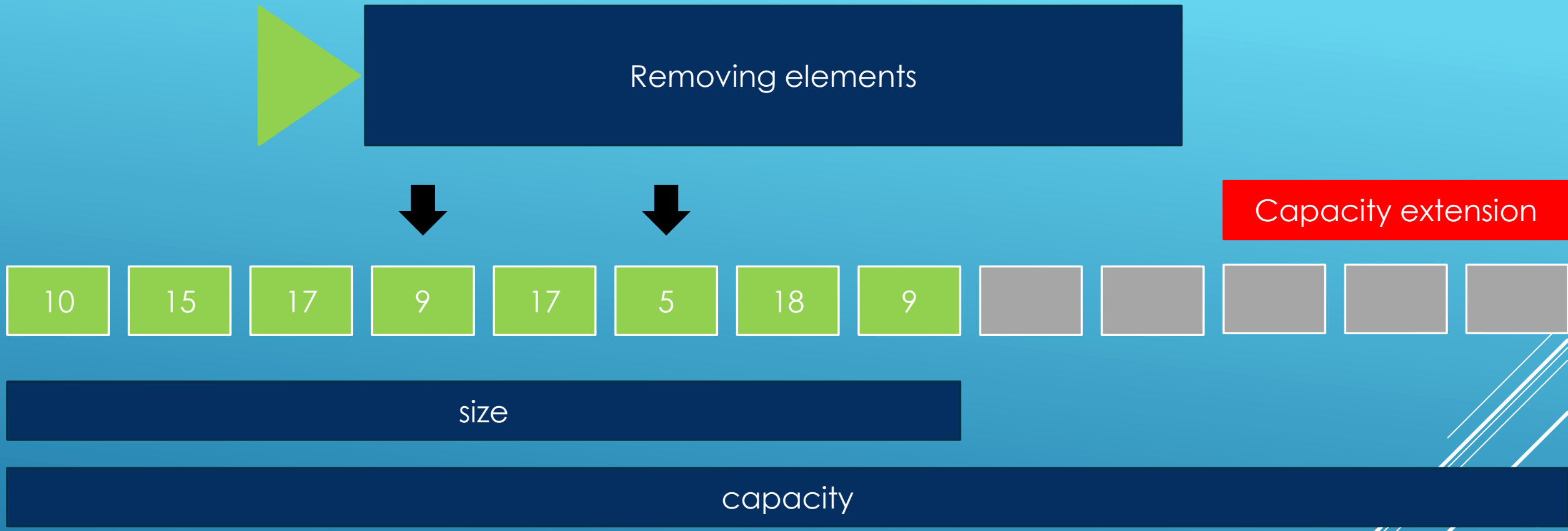

Removing elements
Capacity extension
10
5
17
5
17
5
18
9
15
17
size
capacity

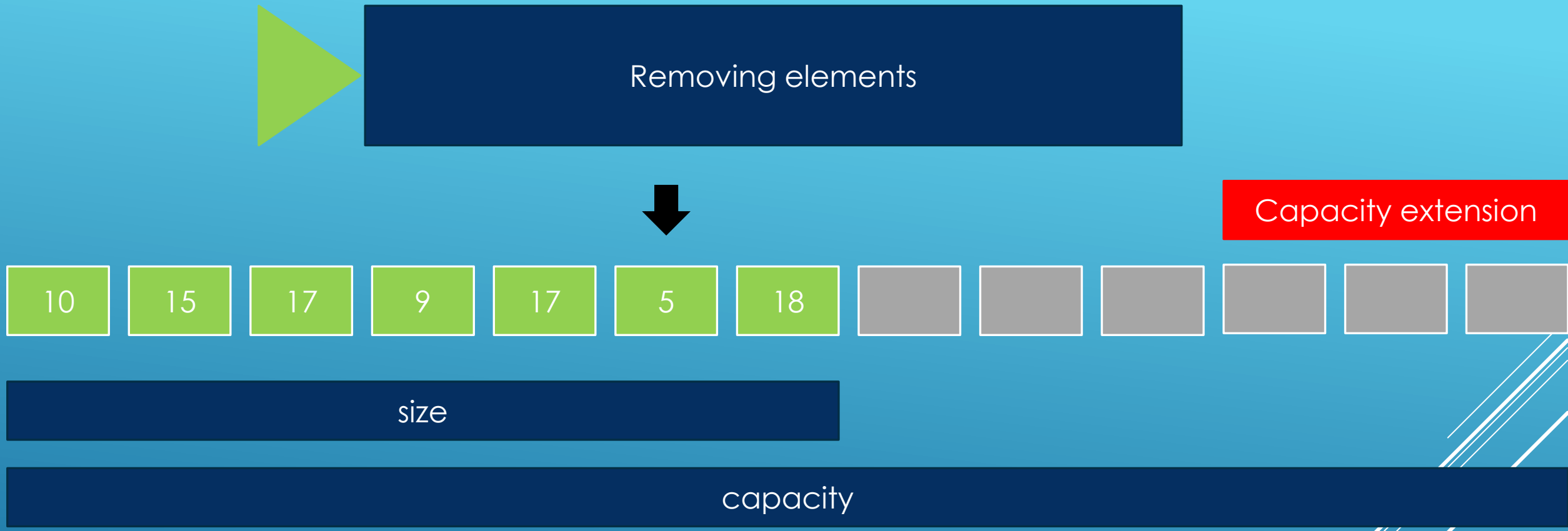

Removing elements
Capacity extension
10
5
17
5
17
5
18
9
15
size
capacity

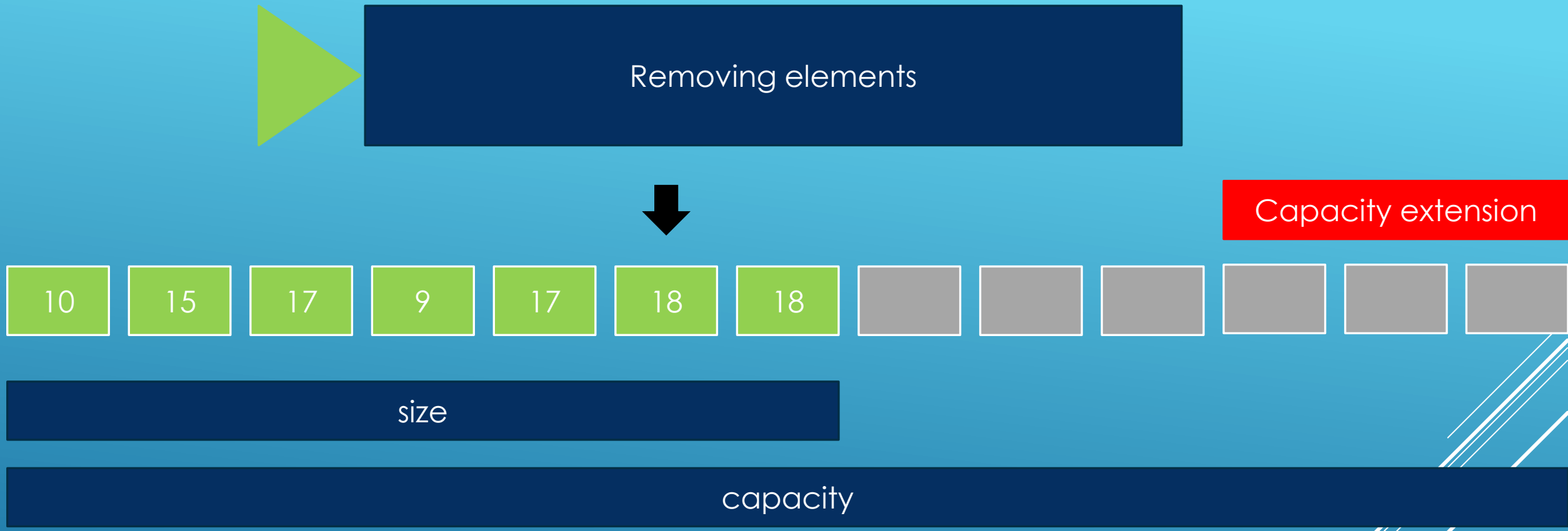

Removing elements
Capacity extension
10
5
17
5
17
5
18
9
15
size
capacity


Removing elements
Capacity extension
10
15
17
5
17
5
18
9
15
size
capacity

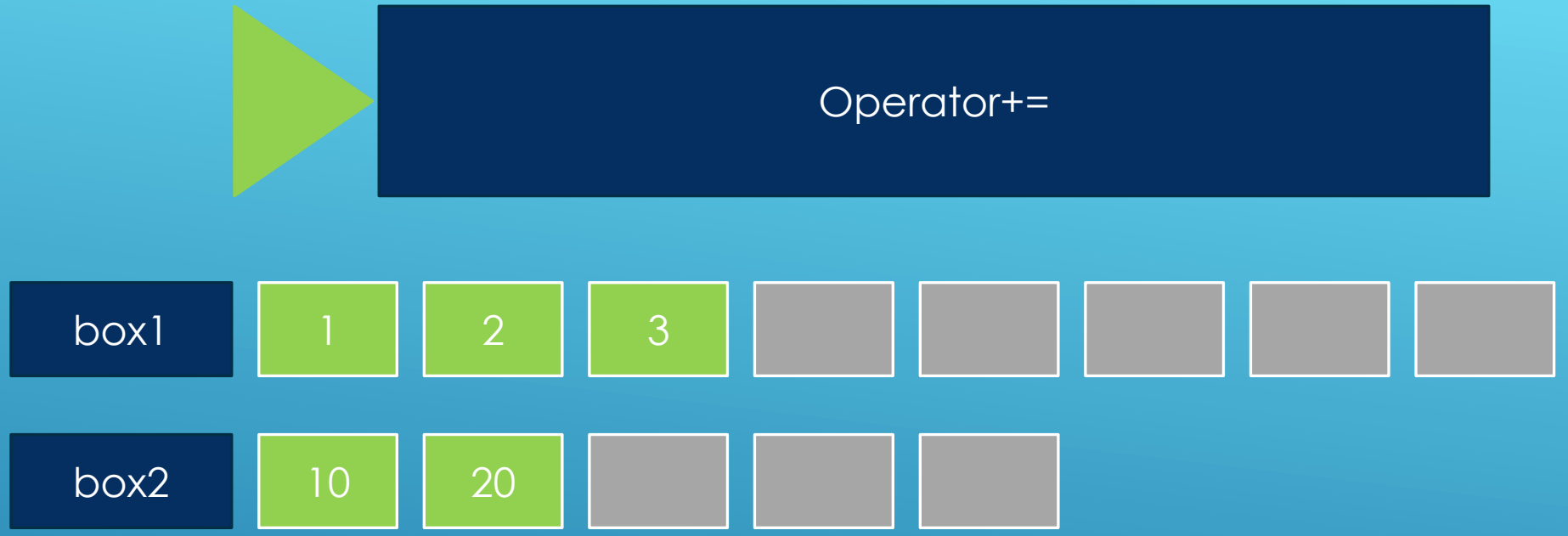

Removing elements
Capacity extension
10
15
17
5
17
5
18
9
size
capacity


Removing elements
Capacity extension
10
15
17
9
17
5
18
9
size
capacity

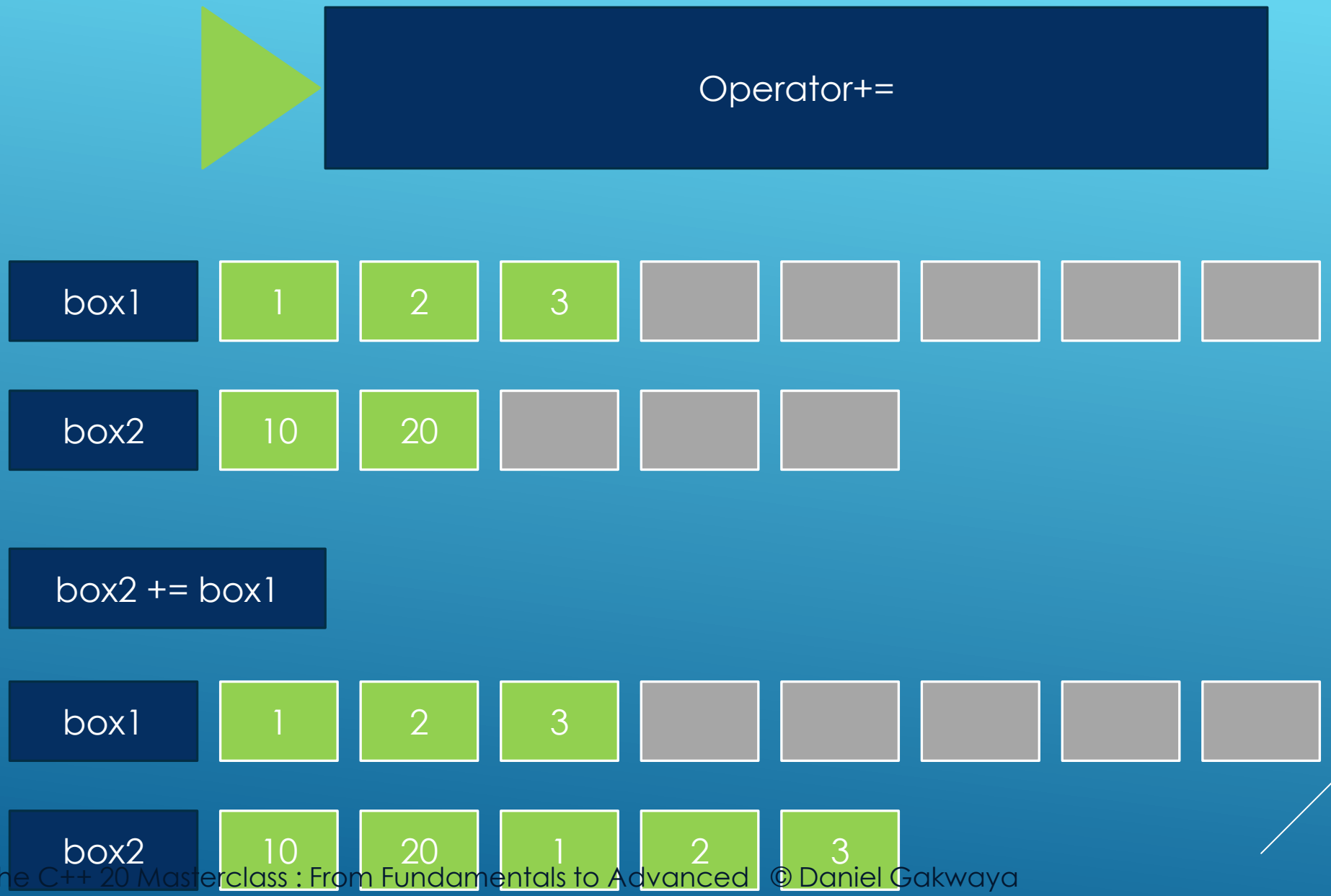

Removing elements
Capacity extension
10
15
17
9
17
5
18
size
capacity


Removing elements
Capacity extension
10
15
17
9
17
18
18
size
capacity


Removing elements
Capacity extension
10
15
17
9
17
18
size
capacity


Operator+=
box1
1
2
3
box2
10
20

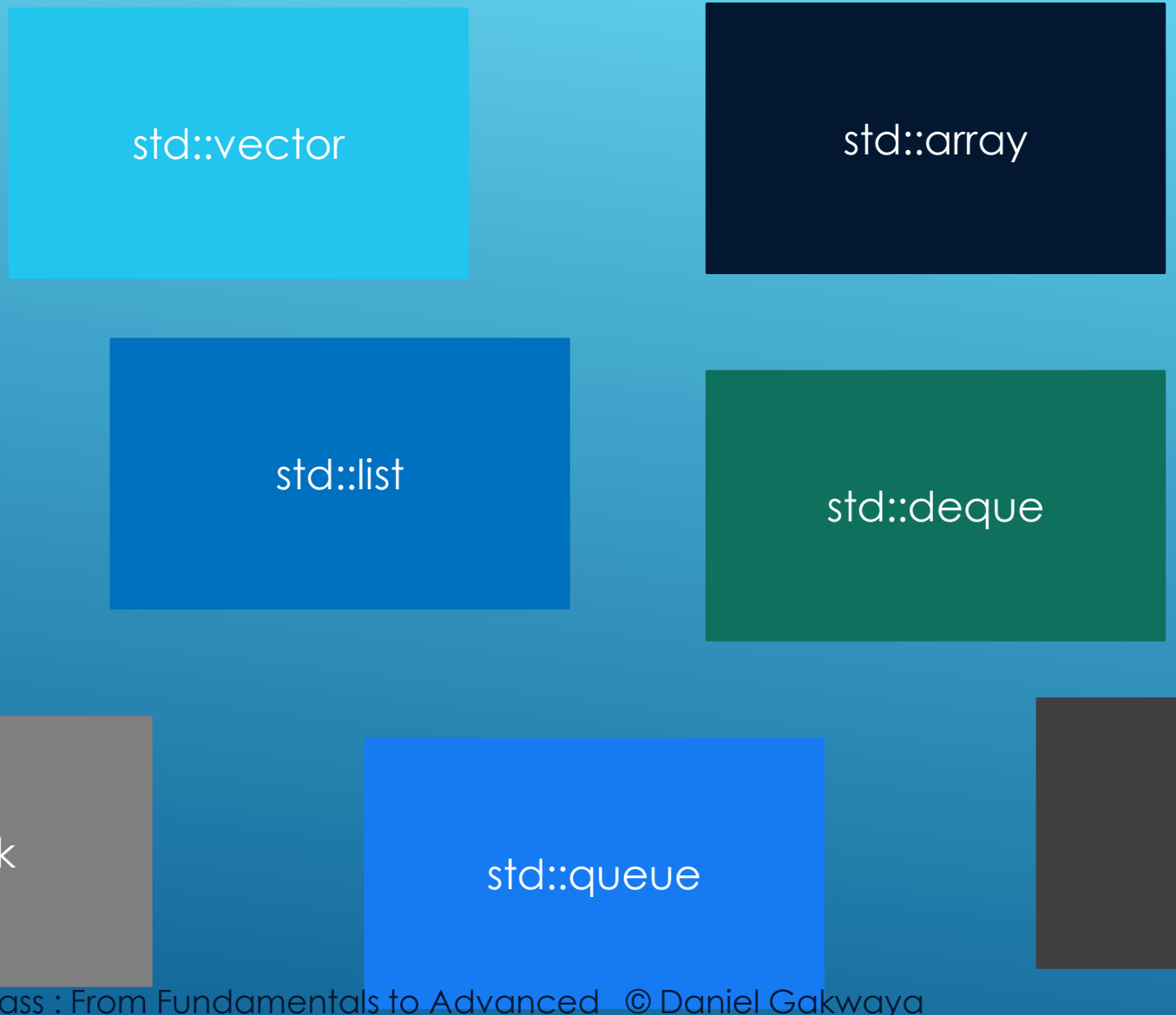

Operator+=
box1
1
2
3
box2
10
20
box2 += box1


Operator+=
box1
1
2
3
box2
10
20
box2 += box1
box1
1
2
3
box2
10
20
1
2
3


Operator+
box1
1
2
3
box2
10
20
box2 + box1
New box
10
20
1
2
3

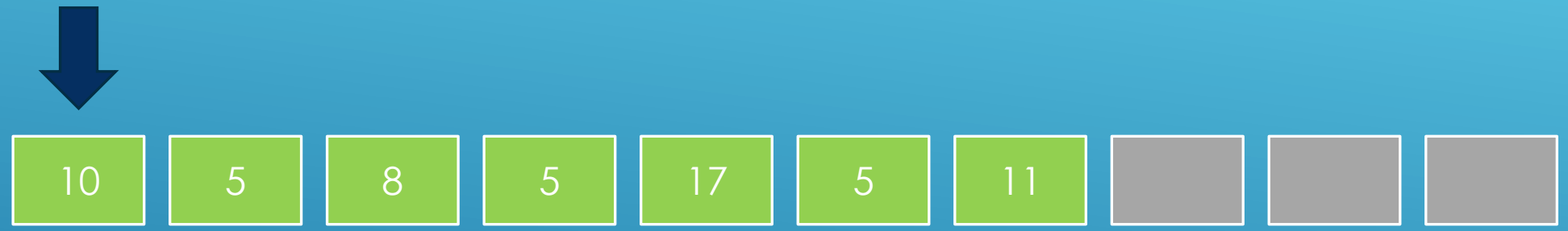
Class wrapping on top of raw array

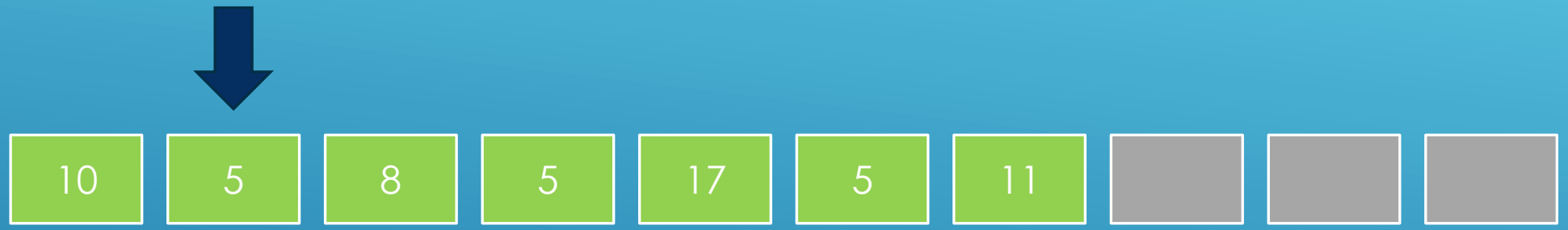
```
class BoxContainer : public StreamInsertable
{
        typedef int value_type; // Allows us to change what's stored in the vector on the fly
                                // Can make it store int, double,...
        static const size_t DEFAULT_CAPACITY = 30;
public:
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer& source);
    ~BoxContainer();

    //StreamInsertable Interface
    virtual void stream_insert(std::ostream& out)const;

    // Helper getter methods
    size_t size( ) const { return m_size; }
    size_t capacity() const{return m_capacity;};

    /* ...

private :
    value_type * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

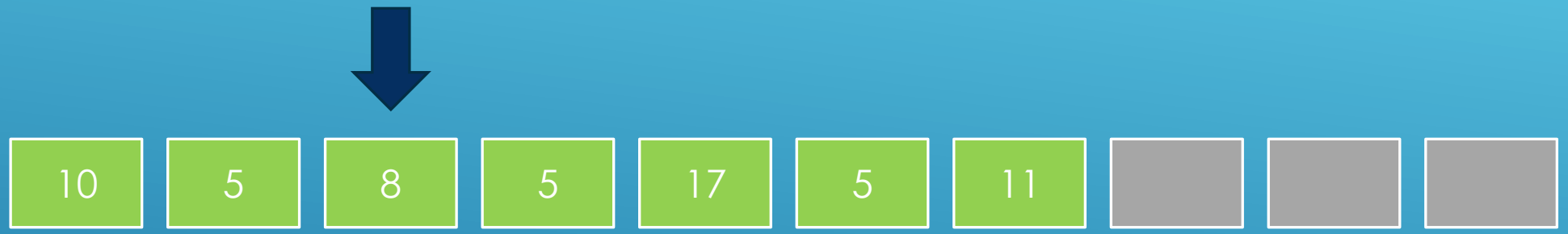

std::vector
std::array
std::list
std::deque
std::stack
std::queue
...

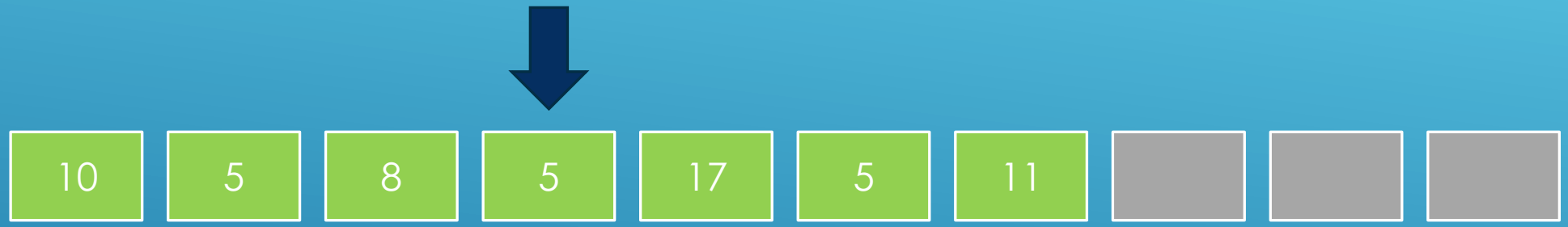
Slide intentionally left empty

Iterators


10
5
8
5
17
5
11

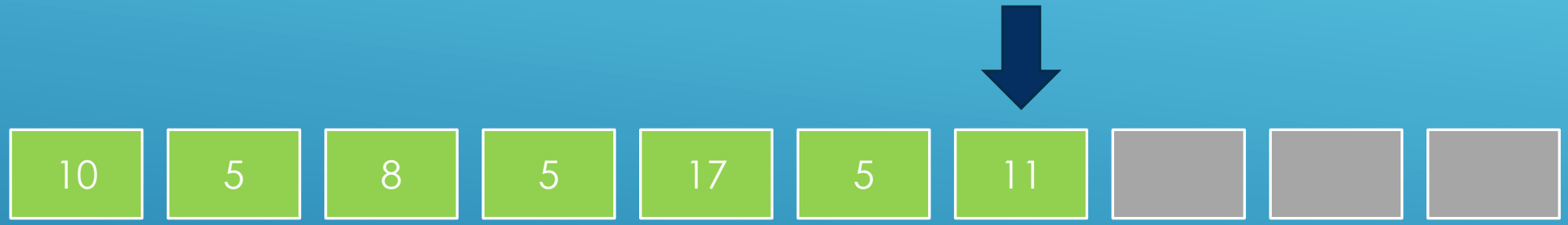

10
5
8
5
17
5
11

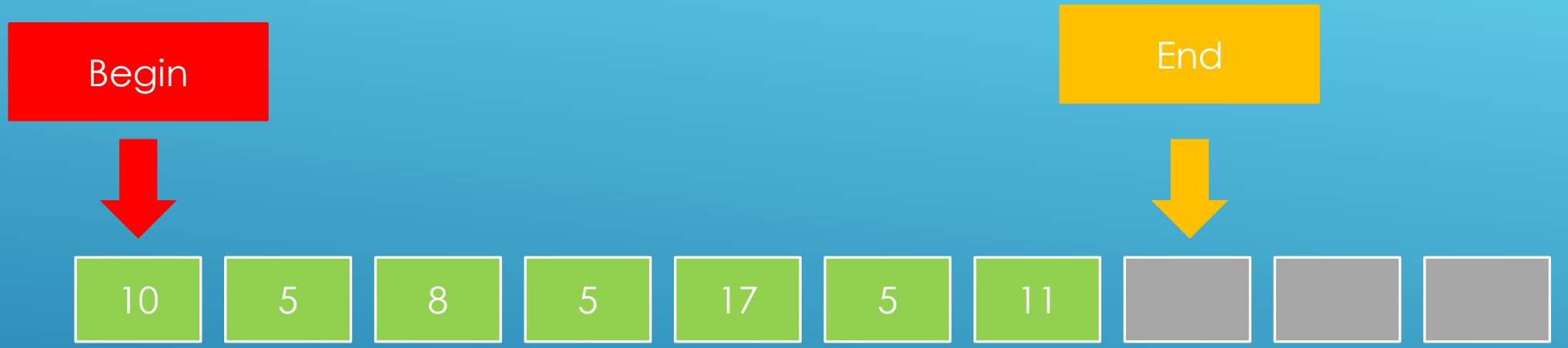

10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


Begin
End
10
5
8
5
17
5
11

Algorithms

sorting

finding

copying

filling

generating

transforming

...

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms

C++20 provides constrained versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an iterator-sentinel pair or as a single range argument, and projections and pointer-to-member callables are supported. Additionally, the return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm. (since C++20)

```
std::vector<int> v = {7, 1, 4, 0, -1};
std::ranges::sort(v); // constrained algorithm
```

Slide intentionally left empty

std::vector


10
5
8
5
17
5
11

Std::vector

Storing stuff contiguously in memory and providing helper methods to manipulate the data

Std::vector

#include <vector>

Constructing std::vector's

```
//Constructing vectors
std::cout << "Constructing vectors " << std::endl;
std::vector<std::string> vec_str{"The","sky","is","blue","my","friend"};
//std::cout << vec_str << std::endl;
print_vec(vec_str);

std::vector<int> ints1;
print_vec(ints1); // Won't print anything, the vector has no content

std::vector<int> ints2 = { 1,2,3,4 };
std::vector<int> ints3{ 11,22,33,44 };

print_vec(ints2);
print_vec(ints3);

std::vector<int> ints4(20, 55); // A vector with 20 items, all initialized to 55
print_vec(ints4);

//Be careful about uniform initialization
std::vector<int> ints5{20, 55}; // A vector with 2 items : 20 and 55
print_vec(ints5);
```

Looping around and printing

```
template <typename T>
void print_vec( const std::vector<T>& vec){
    for(size_t i{}; i < vec.size();++i){
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;
}
```

Accessing elements

```
//Accessing elements
std::cout << std::endl;
std::cout << "Accessing elements in a vector: " << std::endl;
std::cout << "vec_str[2] : " << vec_str[2] << std::endl;
std::cout << "vec_str.at(3) : " << vec_str.at(3) << std::endl;
std::cout << "vec_str.front() : " << vec_str.front() << std::endl;
std::cout << "vec_str.back() : " << vec_str.back() << std::endl;

//The data method : getting direct access to the underlying array
print_raw_array(vec_str.data(),vec_str.size());
```

Using the underlying raw array

```
template <typename T>
void print_raw_array(const T* p, std::size_t size)
{
    std::cout << "data = ";
    for (std::size_t i = 0; i < size; ++i)
        std::cout << p[i] << ' ';
    std::cout << std::endl;
}
```

Adding and removing stuff

```
std::cout << "ints1 : " ;
print_vec(ints1);

//Pushing back
ints1.push_back(100);
ints1.push_back(200);
ints1.push_back(300);
ints1.push_back(500);
std::cout << "ints1 : " ;
print_vec(ints1);

//Poping back
ints1.pop_back();
std::cout << "ints1 : " ;
print_vec(ints1);
```

Slide intentionally left empty

std::array


10
5
8
5
17
5
11

Std::array

Storing stuff in a fixed size container

std::array

#include <array>

Constructing std::array's

```
std::array<int, 3> int_array1; // Will contain junk by default
std::array<int, 3> int_array2{ 1,2 }; // Will contain 1,2,0
std::array<int, 3> int_array3{}; // Will contain 0 0 0
std::array int_array4{ 1,2 }; //Compiler will deduce std::array<int,2>
//std::array<int, 3> int_array5{1,2,3,4,5}; // Compiler error : More than enough elements
//Can deduce the type with auto.
 auto int_array6 = std::experimental::make_array(1, 2, 3, 4, 5);

std::cout << "int_array1 : " ;
print_array(int_array1);

std::cout << "int_array2 : " ;
print_array(int_array2);

std::cout << "int_array3 : " ;
print_array(int_array3);

std::cout << "int_array4 : " ;
print_array(int_array4);

std::cout << "int_array6 : " ;
print_array(int_array6);
```

Adding stuff

```
//Adding and removing stuff
//Can't really add stuff. Can specify content at initialization
//Can also fill the entire array with an element
std::cout << std::endl;
std::cout << "Filling the array : " << std::endl;
int_array1.fill(321);
int_array4.fill(500);
std::cout << "int_array1 : " ;
print_array(int_array1);
std::cout << "int_array4 : " ;
print_array(int_array4);
```

Looping around and printing

```
template <typename T,size_t Size> // The second template argument has to be a size.
void print_array( const std::array<T,Size>& arr){
    for(size_t i{}; i < arr.size();++i){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

Adding and removing stuff

```
//Adding and removing stuff
//Can't really add stuff. Can specify content at initialization
//Can also fill the entire array with an element
std::cout << std::endl;
std::cout << "Filling the array : " << std::endl;
int_array1.fill(321);
int_array4.fill(500);
std::cout << "int_array1 : " ;
print_array(int_array1);
std::cout << "int_array4 : " ;
print_array(int_array4);
```

Accessing elements

```
//Accessing elements
std::cout << std::endl;
std::cout << "Accessing elements in an array: " << std::endl;
std::cout << "int_array2[0] : " << int_array2[0] << std::endl;
std::cout << "int_array2.at(1) : " << int_array2.at(1) << std::endl;
std::cout << "int_array2.front() : " << int_array2.front() << std::endl;
std::cout << "int_array2.back() : " << int_array2.back() << std::endl;
//data method
print_raw_array(int_array2.data(),int_array2.size());
```

Using the underlying raw array

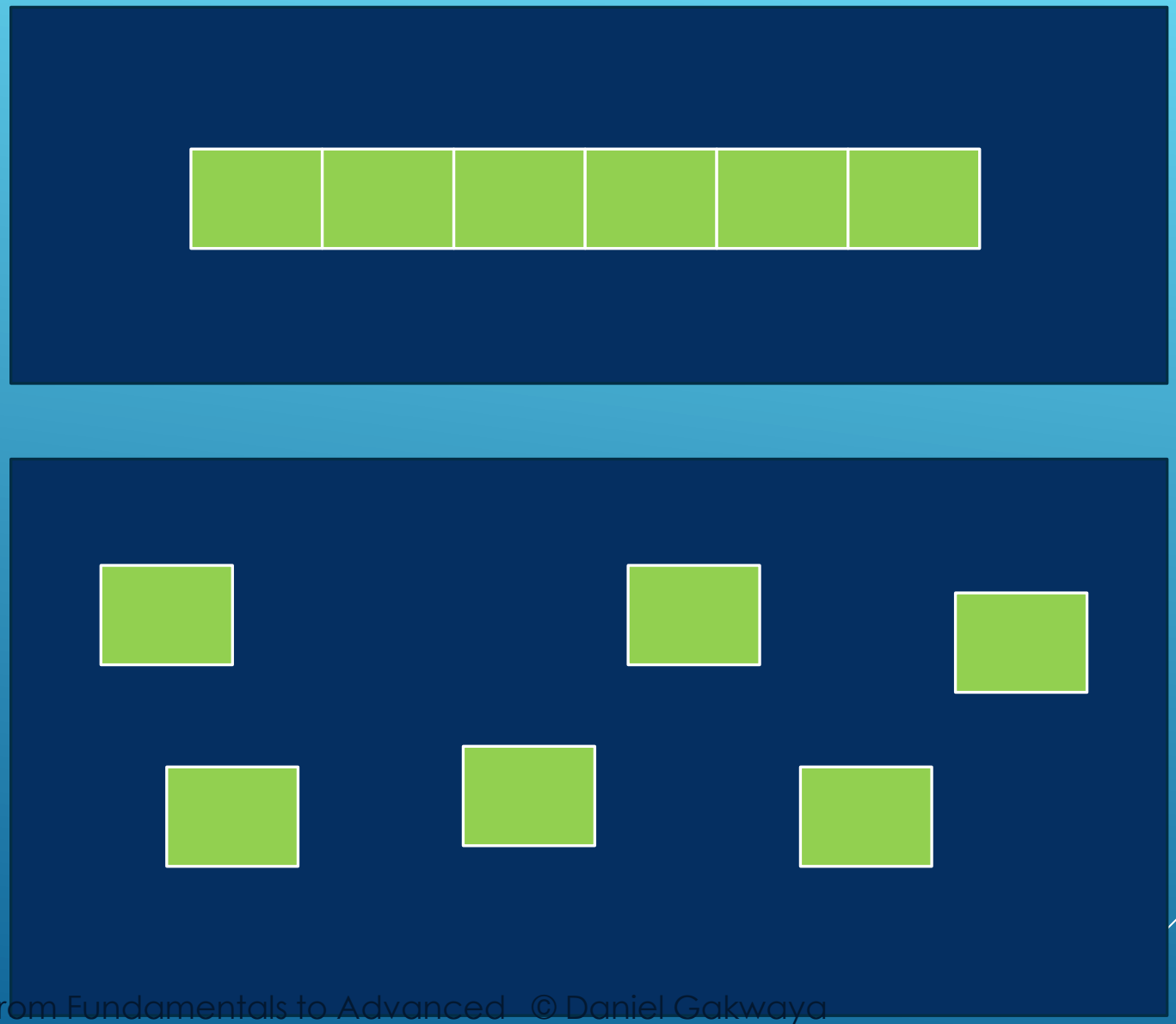
```
template <typename T>
void print_raw_array(const T* p, std::size_t size)
{
    std::cout << "data = ";
    for (std::size_t i = 0; i < size; ++i)
        std::cout << p[i] << ' ';
    std::cout << std::endl;
}
```

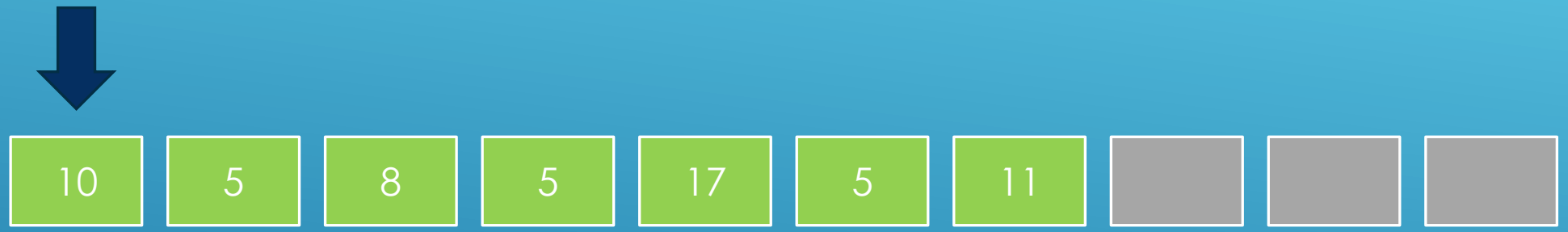
Slide intentionally left empty

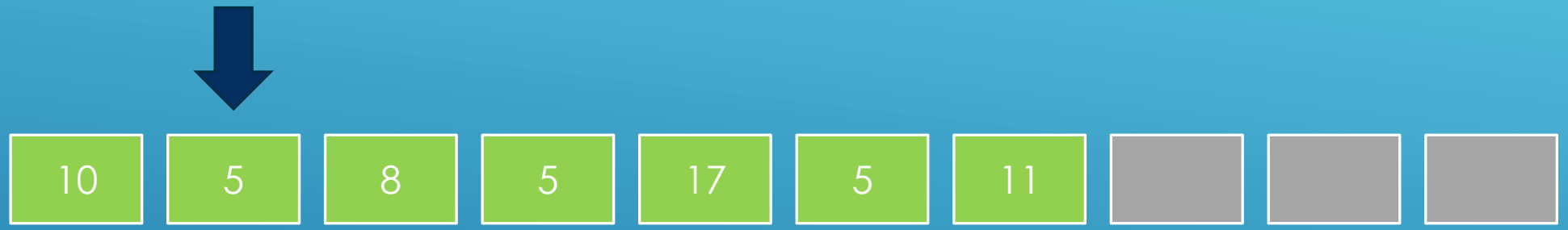
Iterators

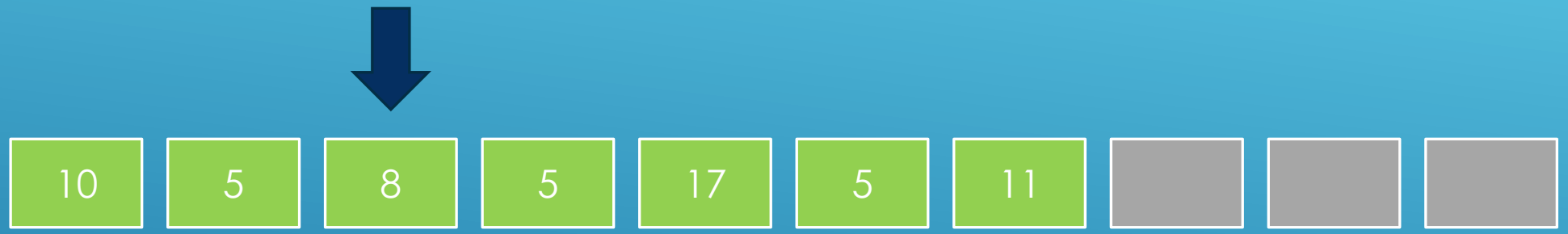
Iterators

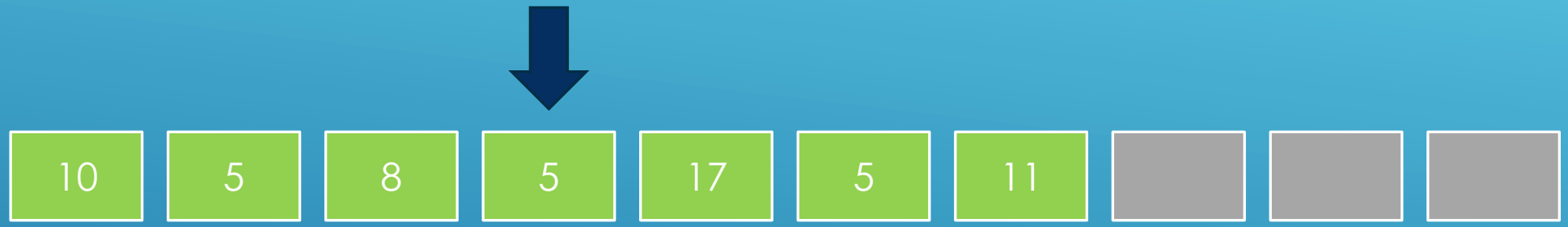
Traversing containers in a unified way, regardless of the internal structure if the container. Each C++ container usually also defines iterators that traverse it.




10
5
8
5
17
5
11

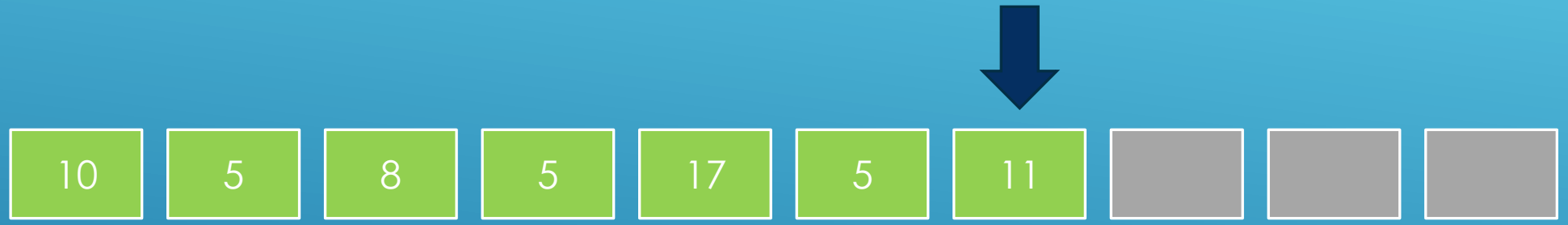

10
5
8
5
17
5
11

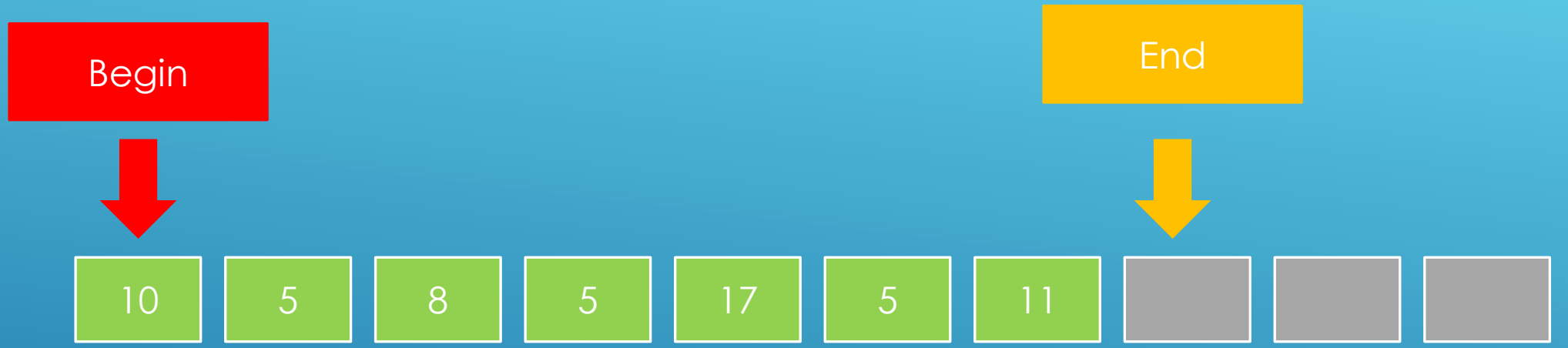

10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


Begin
End
10
5
8
5
17
5
11

begin() and end() iterators

```
std::vector<int> ints1{ 11,22,33,44 };
std::vector<int>::iterator it = ints1.begin();
std::vector<int>::iterator end_it = ints1.end();

std::cout << std::boolalpha;
std::cout << "first elt : " << *it << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;

++it;
std::cout << std::endl;
std::cout << "second elt : " << *it << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;

/* ...

++it;
std::cout << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;
```

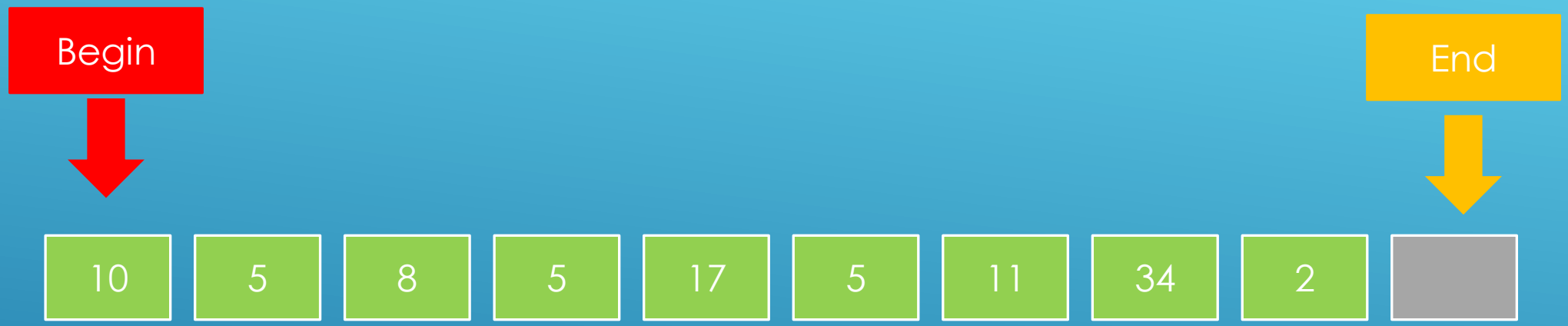
Uniform traversal of containers

```
template <typename T>
void print_collection(const T& collection){

    auto it = collection.begin();

    std::cout << " [";
    while(it != collection.end()){
        std::cout << " " << *it ;
        ++it;
    }
    std::cout << "]" << std::endl;
}
```

```
std::vector<int> ints1{ 11,22,33,44 };
std::array<int,4> ints2 {100,200,300,400};
print_collection(ints1);
print_collection(ints2);
```

Slide intentionally left empty

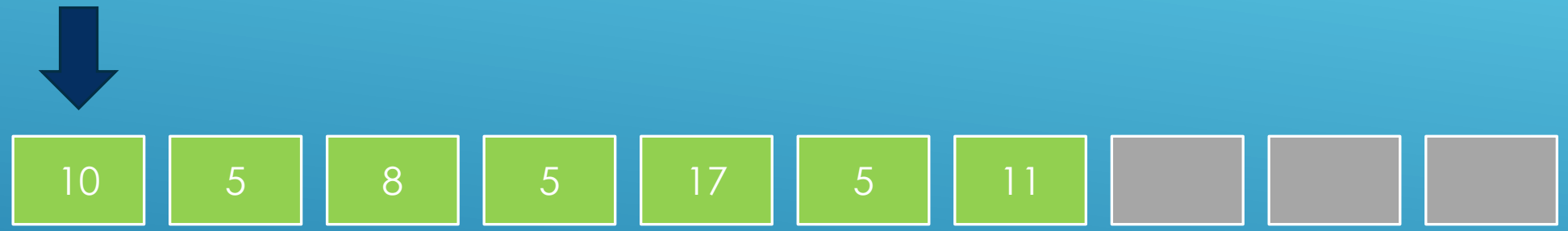
Traversing container subsets with iterators

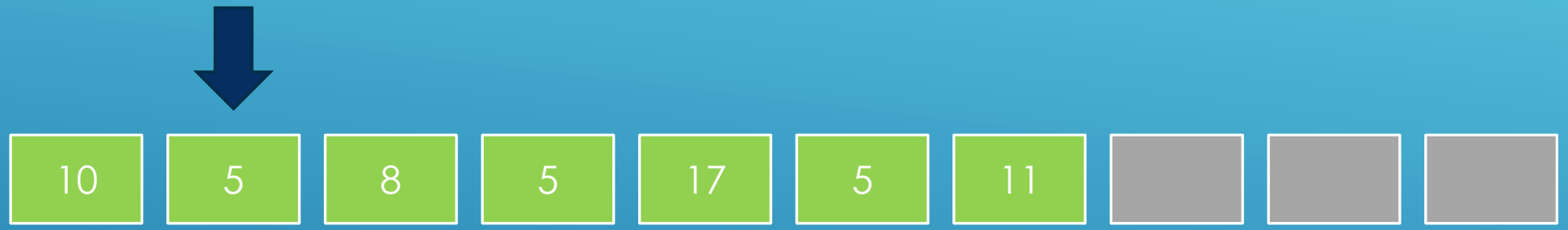

Begin
End
10
5
8
5
17
5
11
34
2

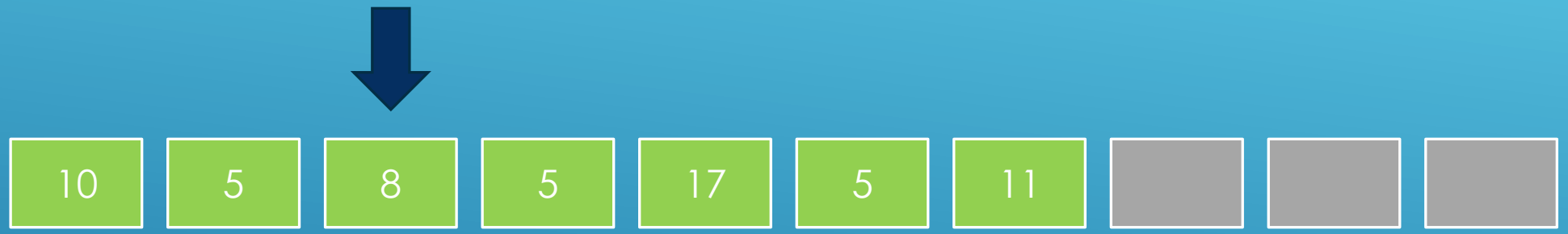
```
template <typename T>
void print_collection(const T& collection , size_t begin_adjustment,
                                            size_t end_adjustment){
    //Adjudt begining and end
    auto start_point = collection.begin() + begin_adjustment;
    auto  end_point = collection.end() - end_adjustment;

    std::cout << " [";
    while(start_point != end_point){
        std::cout << " " << *start_point ;
        ++start_point;
    }
    std::cout << "]" << std::endl;
}

int main(int argc, char **argv)
{
    std::vector<int> ints1{ 11,22,33,44,55,66,77 };
    std::array<int,6> ints2 {100,200,300,400,500,600};

    print_collection(ints1,2,2);
    print_collection(ints2,1,1);
    return 0;
}
```

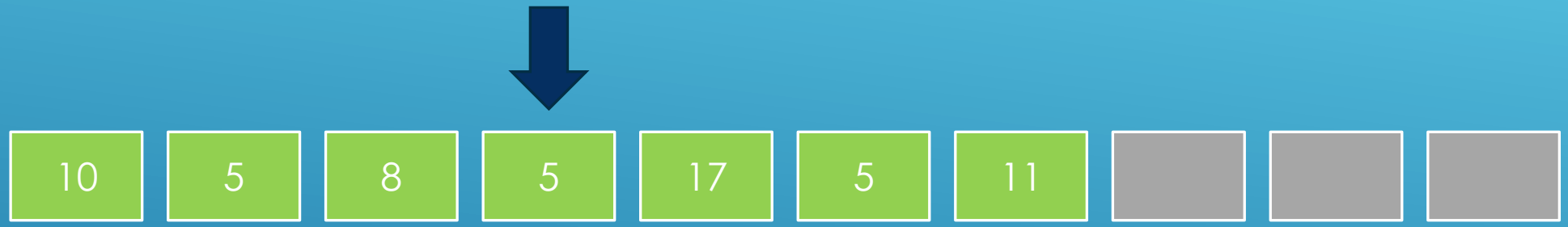
Slide intentionally left empty

Reverse iterators


10
5
8
5
17
5
11

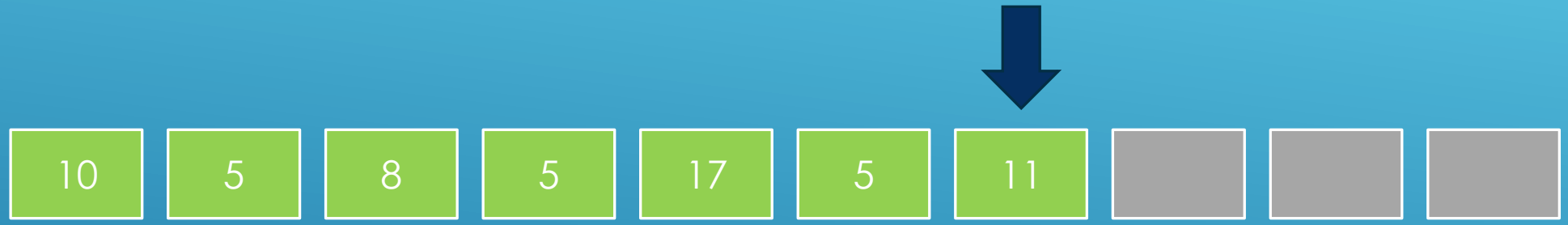

10
5
8
5
17
5
11

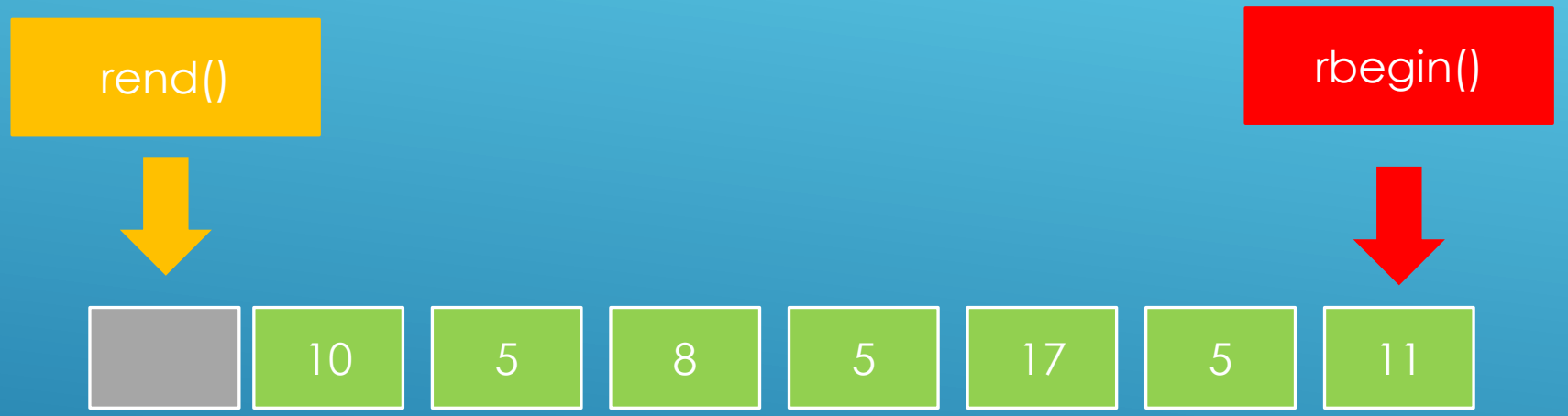

10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


10
5
8
5
17
5
11


rend()
rbegin()
10
5
8
5
17
5
11

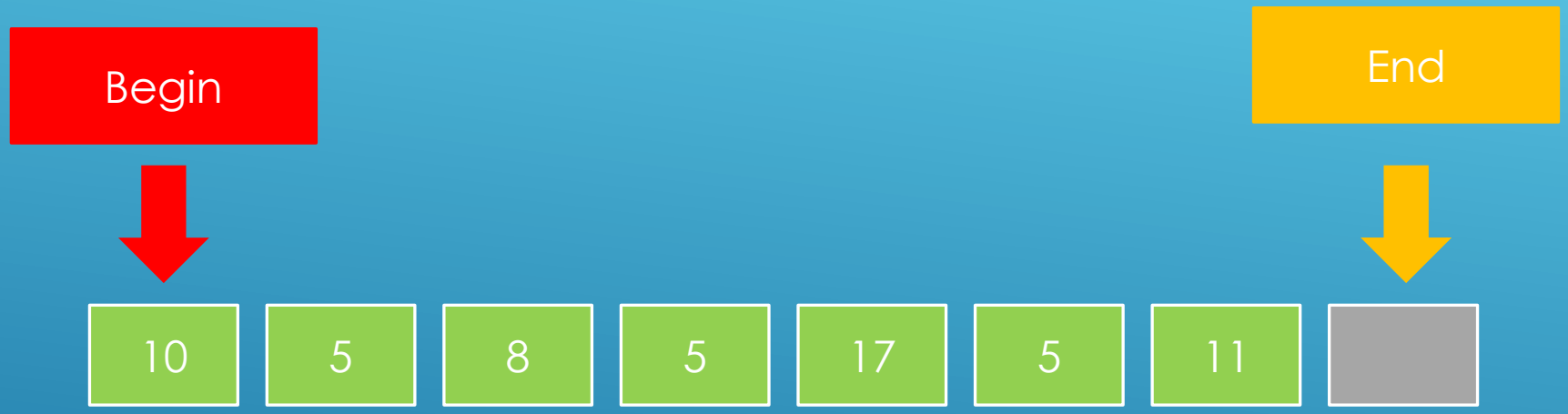
```cpp
std::vector<int> numbers {1,2,3,4,5,6,7,8,9,10};

//auto it = numbers.rbegin(); // A reverse iterator increments backwards from the end.
std::vector<int>::reverse_iterator it= numbers.rbegin();
std::cout << "Numbers : [";
while(it != numbers.rend()){
    std::cout << " " << *it ;
    ++it;
}
std::cout << "] " << std::endl;
```

Comparing iterators of different types

```
//Can't compare iterators of diffeent types
auto it_rev = numbers.rbegin();

if( it_rev != numbers.end()){ // Compiler error.
    std::cout << "Do something..." << std::endl;
}
```

Slide intentionally left empty

Constant iterators


Begin
End
10
5
8
5
17
5
11

Regular non const iterators

```
std::vector<int> numbers{ 11,22,33,44,55,66,77};
auto it = numbers.begin(); // Non const iterator, can modify underlying data through it.

while( it != numbers.end()){
    *it = 100;
    ++it;
}
print_collection(numbers);
```

Const iterator

```
std::vector<int> numbers{ 11,22,33,44,55,66,77};
//std::vector<int>::const_iterator it = numbers.begin();
//std::vector<int>::const_iterator it = numbers.cbegin();
auto it = numbers.cbegin(); // const iterator

while( it != numbers.end()){
   // std::cout << " " << *it ;
    *it = 100; // Can't change underlying data through a const iterator
    ++it;
}
print_collection(numbers);
```

Constant reverse iterators

```
std::vector<int> numbers{ 11,22,33,44,55,66,77};
auto it1 = numbers.crbegin();
//std::vector<int>::const_reverse_iterator it1= numbers.crbegin();

while(it1 != numbers.crend()){
  *it1 = 600; // Compiler error, it1 is a const iterator, we can't modify
              // container data through it.
  ++it1;

}
```

Const iterators from const containers

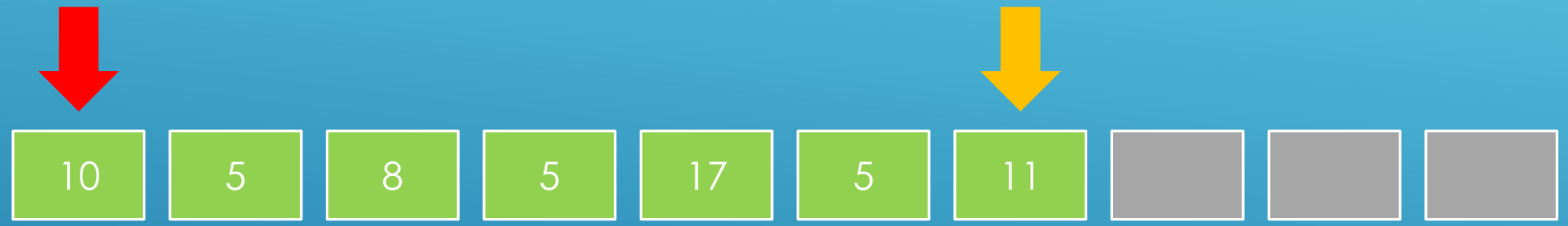
```
//Const container
const std::vector<int> numbers1 {1,2,3,4,5,6,7,8,9,10};

auto it_modify = numbers1.begin();
std::cout << *it_modify << std::endl;

//Because the container is const, begin() here returns a const iterator.
*it_modify =4; // Compiler error
```

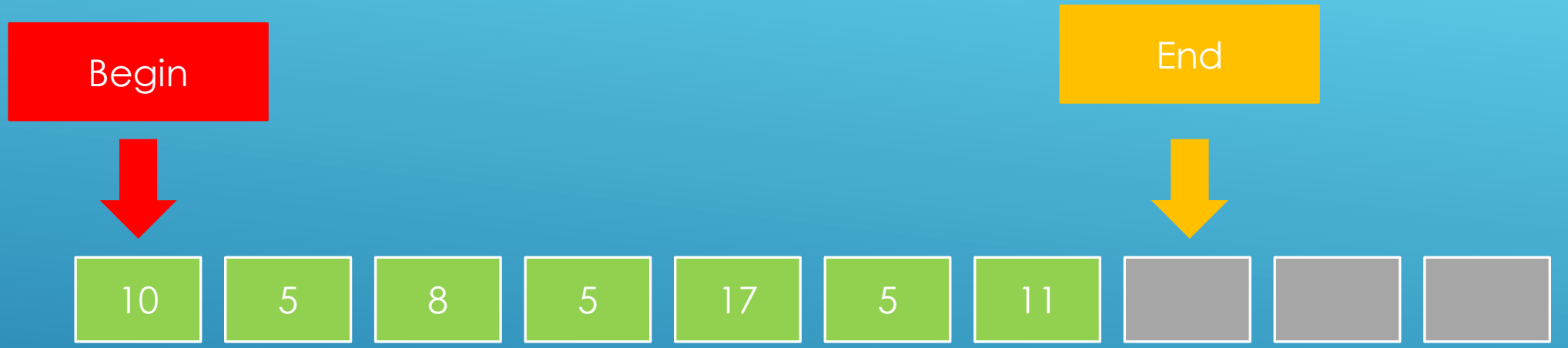
Slide intentionally left empty

Iterator types

Notes to self

Use these references :
https://www.cplusplus.com/reference/iterator/
https://en.cppreference.com/w/cpp/iterator
https://en.cppreference.com/w/cpp/iterator/contiguous_iterator


10
5
8
5
17
5
11


Begin
End
10
5
8
5
17
5
11

Input iterators

```
. Input iterators :
    - Used to read stuff from containers
    - single pass from begining to end.
    - Some perators :
            * operator ++
            * operator * (read)
            * operator -> (read)
            * operator==
            * operator!=
```

Output iterators

```
. Output iterators
    - Used to write stuff into containers
    - single pass from beginning to end
    - Some operators :
            * operator ++
            * operator * (write)
            * operator -> (write)
            * operator==
            * operator!=
```

Bidirectional iterators

```
. Bidirectional iterators
    - Like input iterators, used to read stuff. But can read
      forward and backwards
    - Single pass
    - Some operators :
            * operator ++
            * operator --
            * operator * (read)
            * operator -> (read)
            * operator==
            * operator!=
```

Forward iterators

```
. Forward iterators
    - Combination of input and output iterators
    - Can't read backwards though, only forward
    - multipass
    - Some perators :
            * operator ++
            * operator * (read,write)
            * operator -> (read,write)
            * operator==
            * operator!=
```

Random access iterators

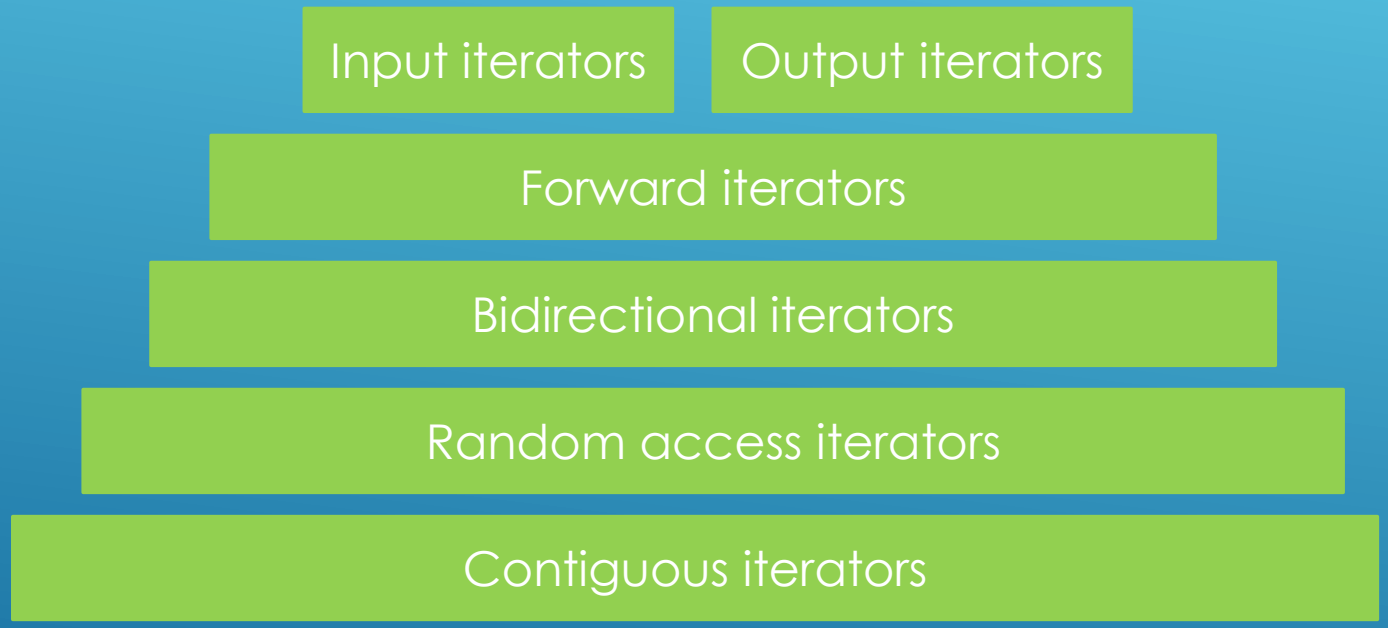
```
. Random access iterators
    - Can read/write randomly from/to any index in the container
    - multipass
    - Some perators :
            * operator[]
            * operator ++
            * operator * (read,write)
            * operator -> (read,write)
            * operator==
            * operator!=
```

Contiguous iterators

What are contiguous iterators useful for ?

StackOverflow

https://stackoverflow.com/q/60587869


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Slide intentionally left empty

std::begin(T) & std::end(T)

- std::begin and std::end() template functions return the begin and end iterator respectively for the underlying container passed as parameter

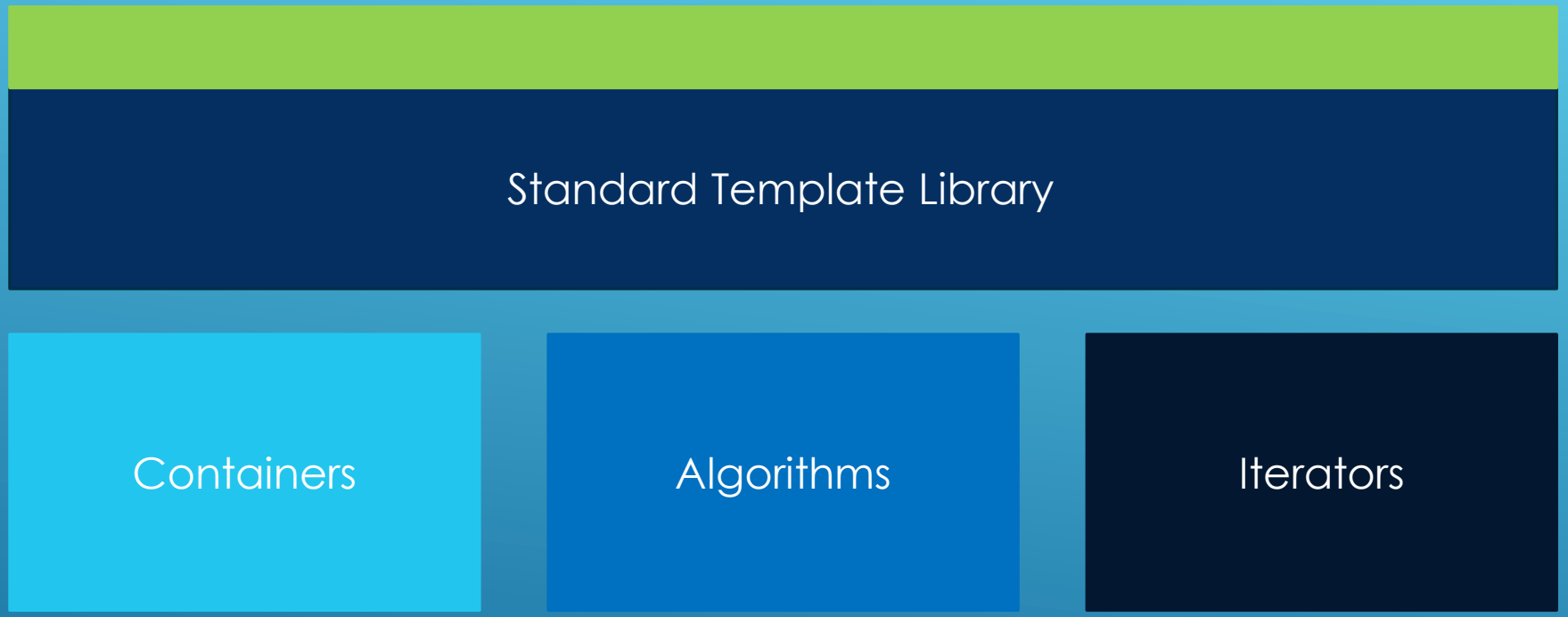
- These functions are usually helpful when you want your iterator based code to work even for regular raw c arrays. C arrays support pointers and pointers meet all the requirements for random access iterators.

- The requirement for the template argument is that the collection passed in should support these begin and end iterators.

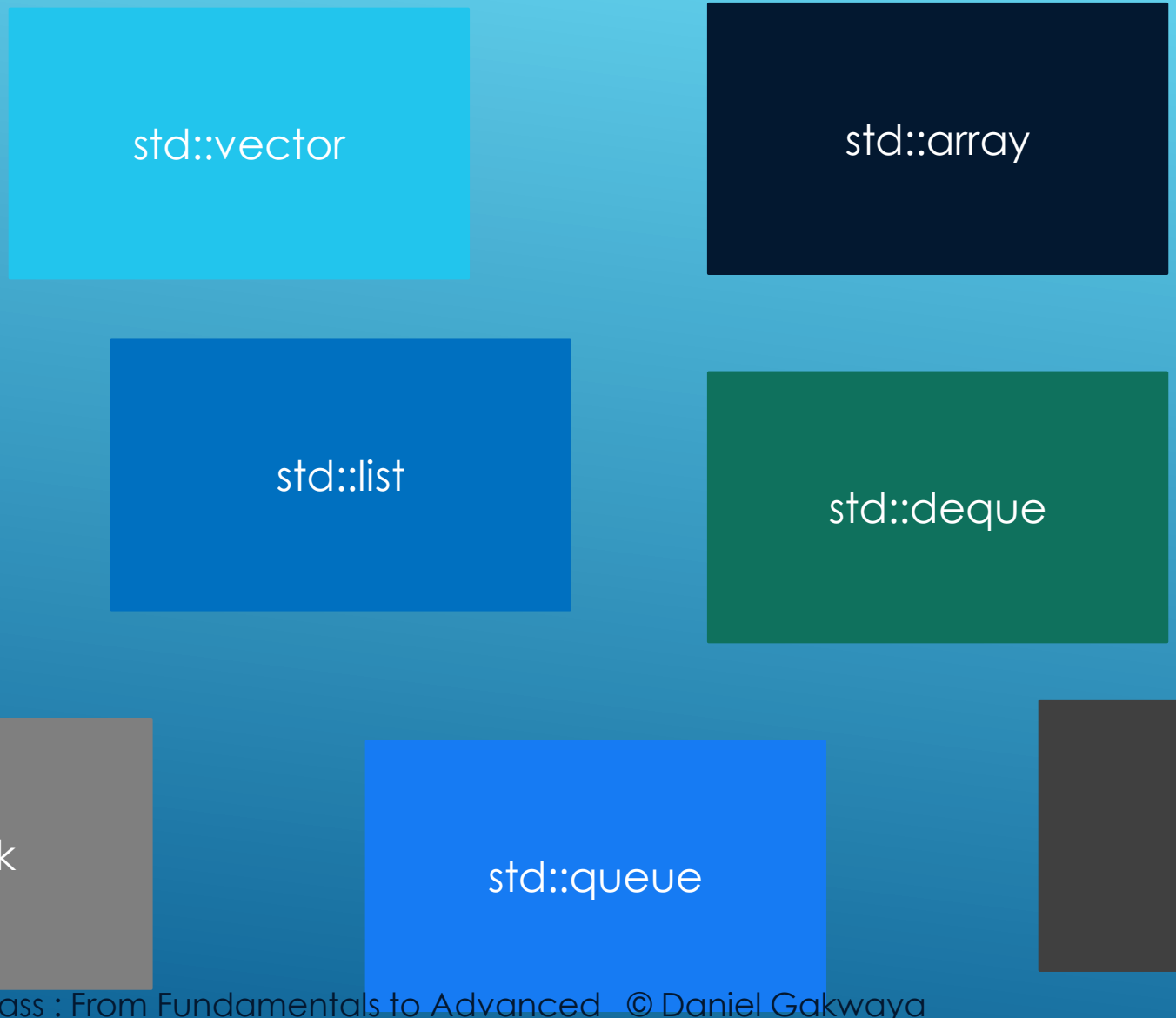
```cpp
std::vector<int> vi {1,2,3,4,5,6,7,8,9};
//int vi[] {1,2,3,4,5,6,7,8,9};

std::cout << " Collection : " ;
/*
for(auto it = vi.begin(); it!= vi.end(); ++it){
    std::cout << *it  << " ";
}
*/

for(auto it = std::begin(vi); it!= std::end(vi); ++it){
    std::cout << *it  << " ";
}
std::cout << std::endl;
```

Head to the IDE and show all this off.

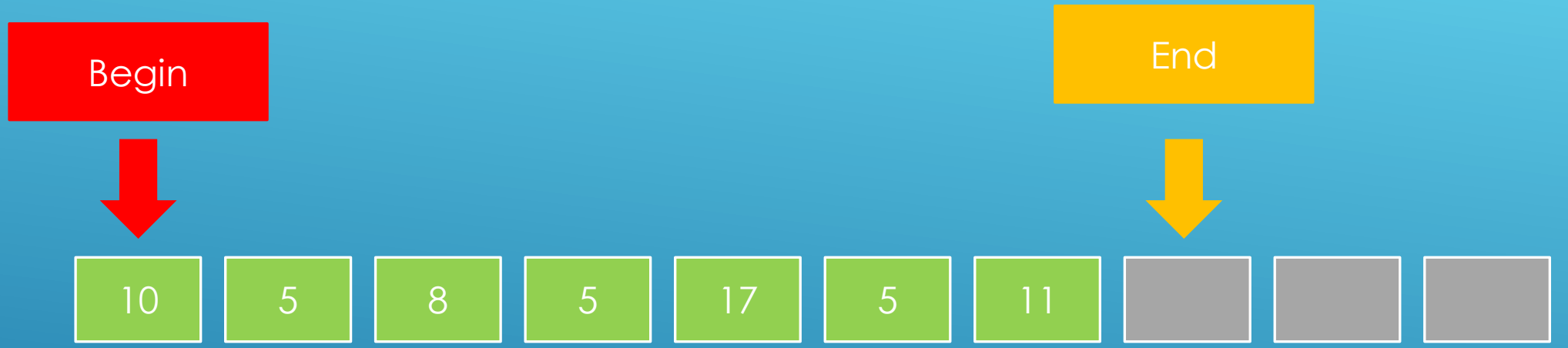
STL, Containers and Iterators : Summary

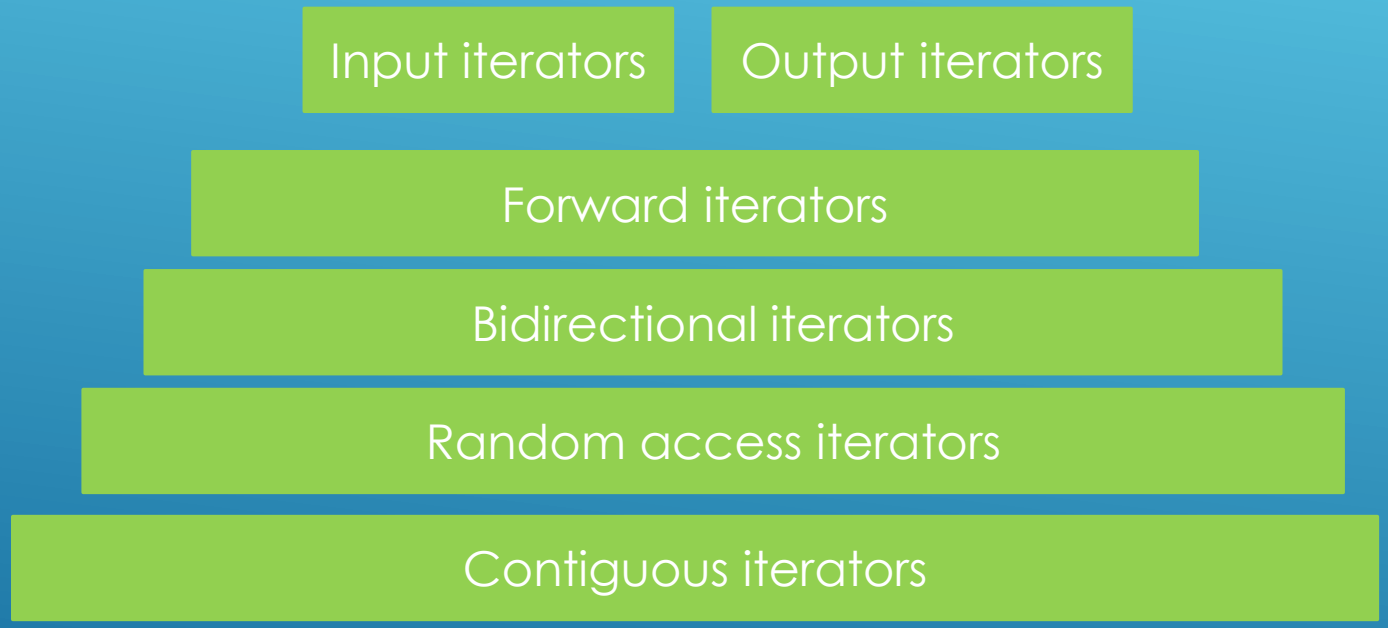

Standard Template Library
Containers
Algorithms
Iterators

Collections (Containers)


std::vector
std::array
std::list
std::deque
std::stack
std::queue
...

Iterators


Begin
End
10
5
8
5
17
5
11


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

Algorithms

sorting

finding

copying

filling

generating

transforming

...

Std::vector

```
//Constructing vectors
std::cout << "Constructing vectors " << std::endl;
std::vector<std::string> vec_str{"The","sky","is","blue","my","friend"};
//std::cout << vec_str << std::endl;
print_vec(vec_str);

std::vector<int> ints1;
print_vec(ints1); // Won't print anything, the vector has no content

std::vector<int> ints2 = { 1,2,3,4 };
std::vector<int> ints3{ 11,22,33,44 };

print_vec(ints2);
print_vec(ints3);

std::vector<int> ints4(20, 55); // A vector with 20 items, all initialized to 55
print_vec(ints4);

//Be careful about uniform initialization
std::vector<int> ints5{20, 55}; // A vector with 2 items : 20 and 55
print_vec(ints5);
```

Std::array

```
std::array<int, 3> int_array1; // Will contain junk by default
std::array<int, 3> int_array2{ 1,2 }; // Will contain 1,2,0
std::array<int, 3> int_array3{}; // Will contain 0 0 0
std::array int_array4{ 1,2 }; //Compiler will deduce std::array<int,2>
//std::array<int, 3> int_array5{1,2,3,4,5}; // Compiler error : More than enough elements
//Can deduce the type with auto.
 auto int_array6 = std::experimental::make_array(1, 2, 3, 4, 5);

std::cout << "int_array1 : " ;
print_array(int_array1);

std::cout << "int_array2 : " ;
print_array(int_array2);

std::cout << "int_array3 : " ;
print_array(int_array3);

std::cout << "int_array4 : " ;
print_array(int_array4);

std::cout << "int_array6 : " ;
print_array(int_array6);
```

Iterators

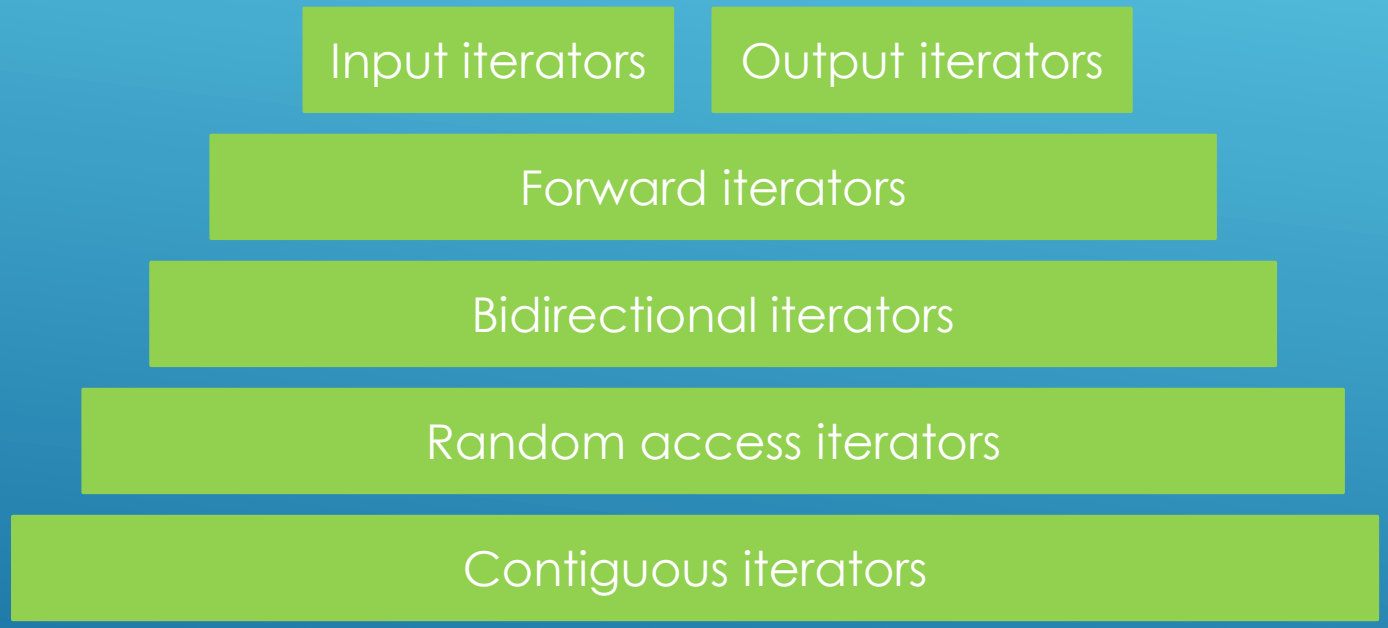
```
std::vector<int> ints1{ 11,22,33,44 };
std::vector<int>::iterator it = ints1.begin();
std::vector<int>::iterator end_it = ints1.end();

std::cout << std::boolalpha;
std::cout << "first elt : " << *it << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;

++it;
std::cout << std::endl;
std::cout << "second elt : " << *it << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;

/* ...

++it;
std::cout << std::endl;
std::cout << "it = end_it : " << (it == end_it) << std::endl;
```

Reverse iterators

```
std::vector<int> numbers {1,2,3,4,5,6,7,8,9,10};

//auto it = numbers.rbegin(); // A reverse iterator increments backwards from the end.
std::vector<int>::reverse_iterator it= numbers.rbegin();
std::cout << "Numbers : [";
while(it != numbers.rend()){
    std::cout << " " << *it ;
    ++it;
}
std::cout << "] " << std::endl;
```

Constant iterators

```
std::vector<int> numbers{ 11,22,33,44,55,66,77};
//std::vector<int>::const_iterator it = numbers.begin();
//std::vector<int>::const_iterator it = numbers.cbegin();
auto it = numbers.cbegin(); // const iterator

while( it != numbers.end()){
   // std::cout << " " << *it ;
    *it = 100; // Can't change underlying data through a const iterator
    ++it;
}
print_collection(numbers);
```


Input iterators
Output iterators
Forward iterators
Bidirectional iterators
Random access iterators
Contiguous iterators

std::begin() and std::end()

```
std::vector<int> vi {1,2,3,4,5,6,7,8,9};
//int vi[] {1,2,3,4,5,6,7,8,9};

std::cout << " Collection : " ;
/*
for(auto it = vi.begin(); it!= vi.end(); ++it){
    std::cout << *it  << " ";
}
*/

for(auto it = std::begin(vi); it!= std::end(vi); ++it){
    std::cout << *it  << " ";
}
std::cout << std::endl;
```

Slide intentionally left empty