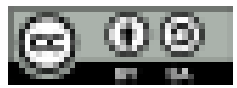
Scientific programming (in C++)

F. Giacomini

INFN-CNAF

School on Open Science Cloud
Perugia, June 2017

https://baltig.infn.it/giaco/201706_perugia_cpp



What is C++

C++ is a programming language that is:

- strongly and statically typed

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm

What is C++

C++ is a programming language that is:

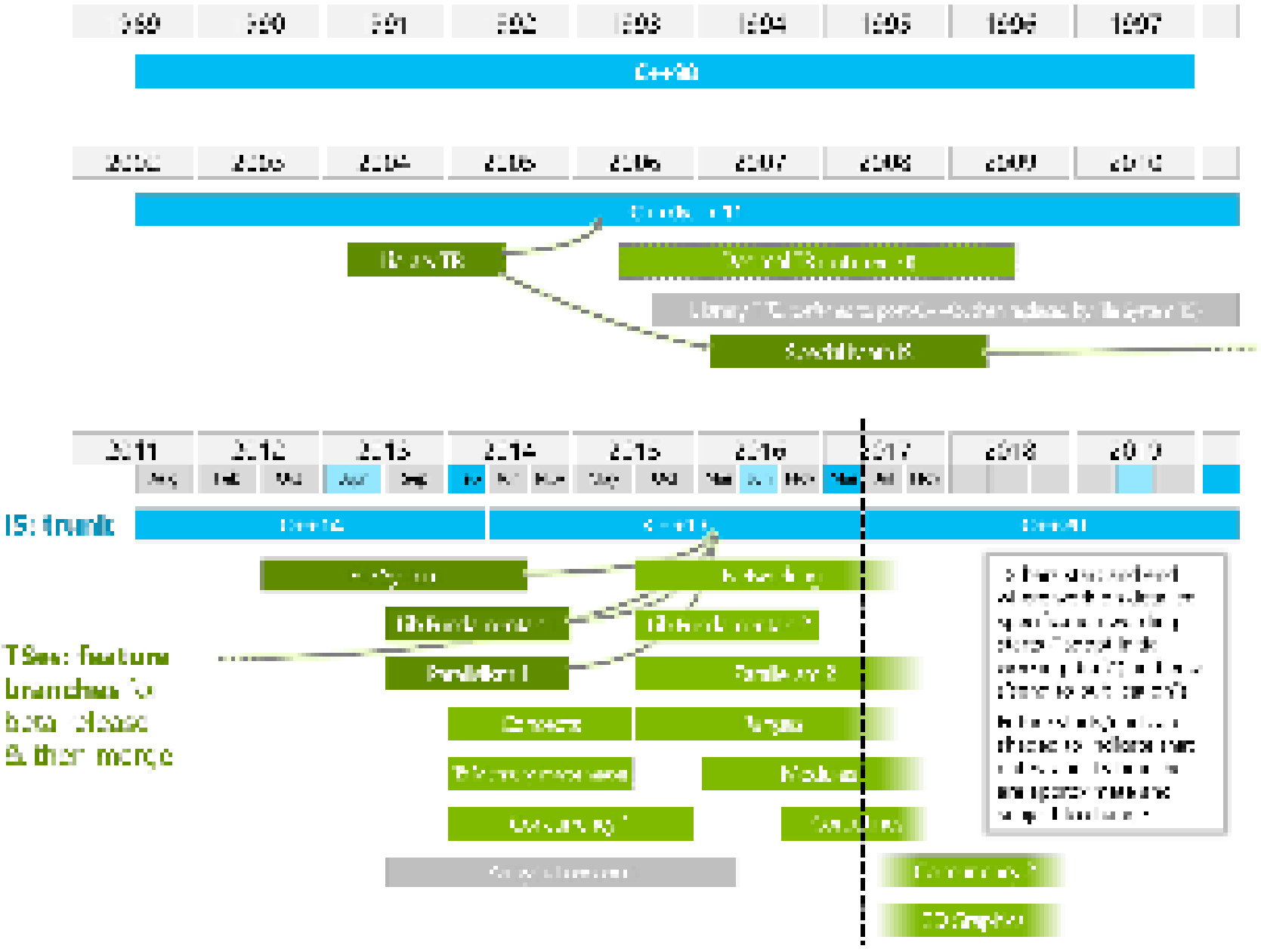
- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)

What is C++

C++ is a programming language that is:

- strongly and statically typed
- general-purpose
- multi-paradigm
- efficient (*"you don't pay for what you don't use"*)
- standard

C++ timeline



Online resources

- http://www.isocpp.org/
- http://en.cppreference.com/
- http://gcc.godbolt.org/
- http://coliru.stacked-crooked.com/

Standards

Working drafts, almost the same as the final published document

- C++03 `http://wg21.link/n1905`
- C++11 `http://wg21.link/n3242`
- C++14 `http://wg21.link/n4296`
- C++17 `http://wg21.link/n4659`

For the LaTeX sources see https://github.com/cplusplus/draft

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard Template Library)

The C++ standard library

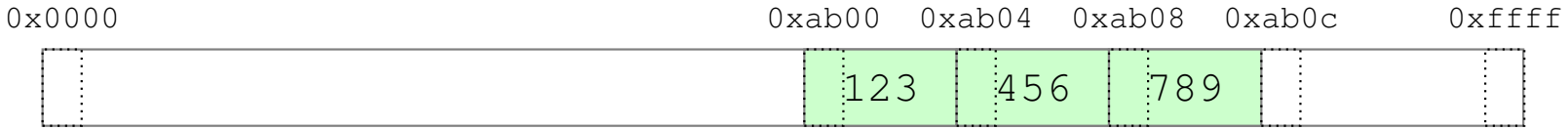
- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard Template Library)
- But templates are everywhere

“C” Arrays

Contiguous sequence of homogeneous objects in memory

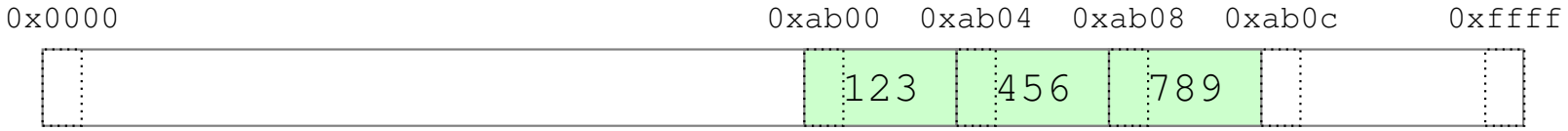
“C” Arrays

Contiguous sequence of homogeneous objects in memory



“C” Arrays

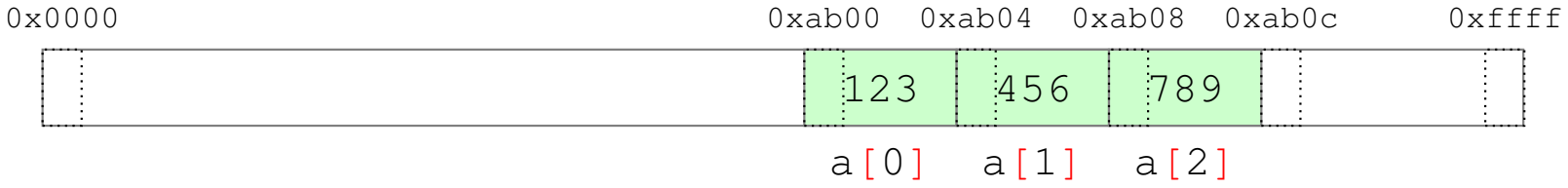
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
```

“C” Arrays

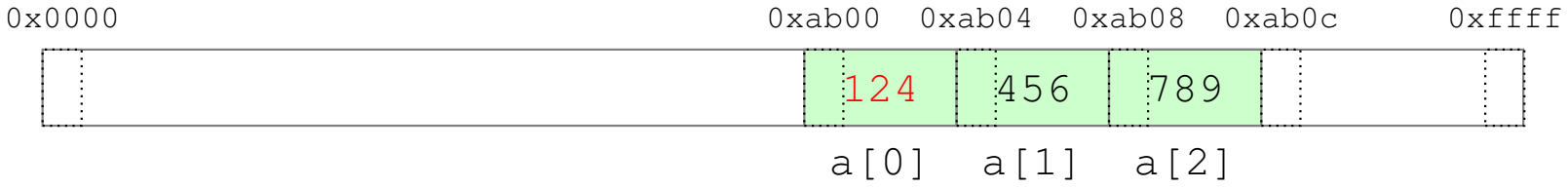
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
```

“C” Arrays

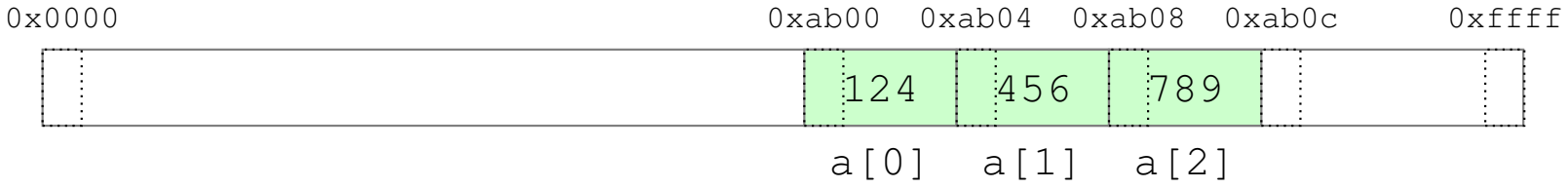
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
```

“C” Arrays

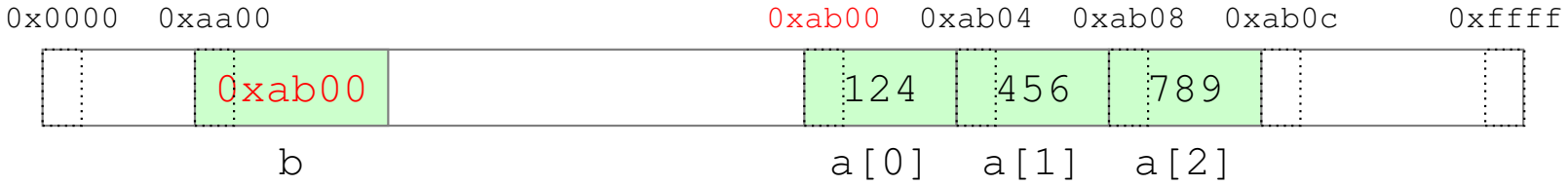
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
```

“C” Arrays

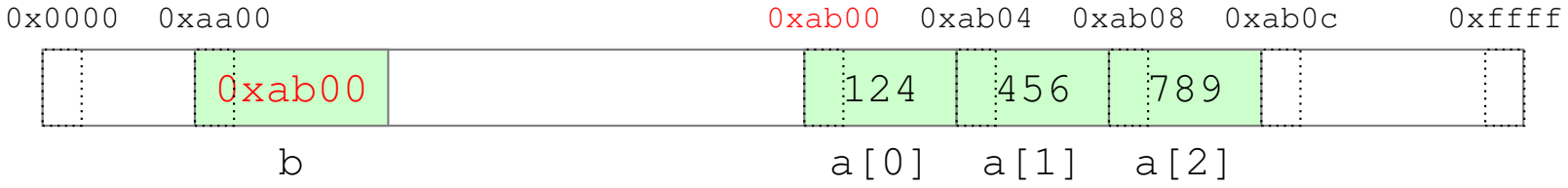
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
```

“C” Arrays

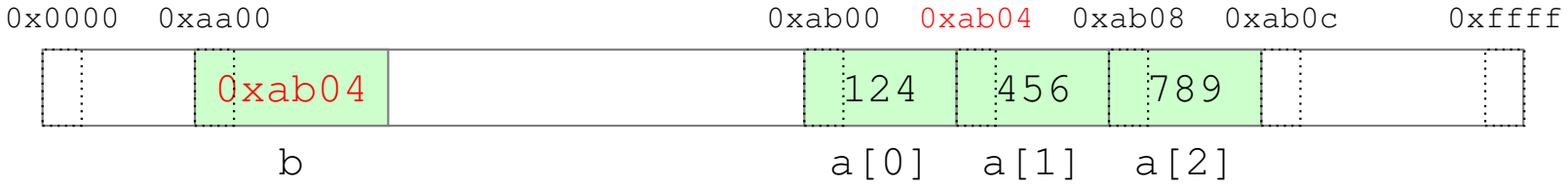
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
```

“C” Arrays

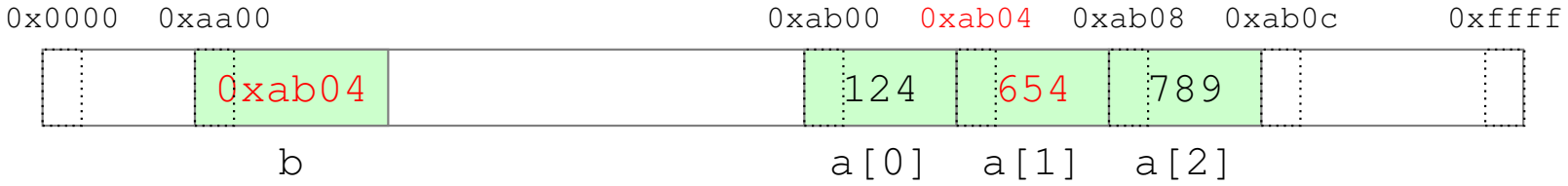
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
```

“C” Arrays

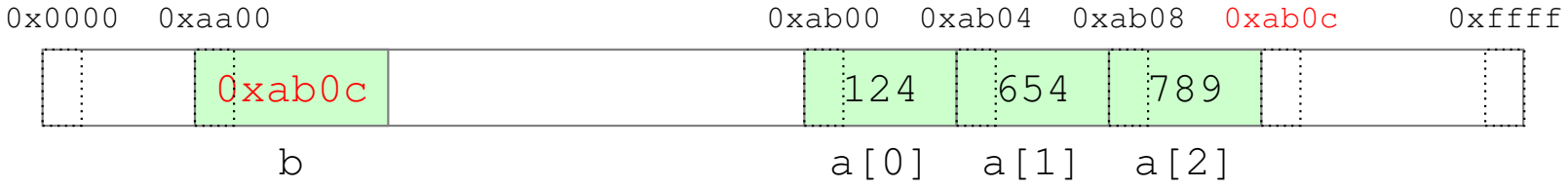
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
```

“C” Arrays

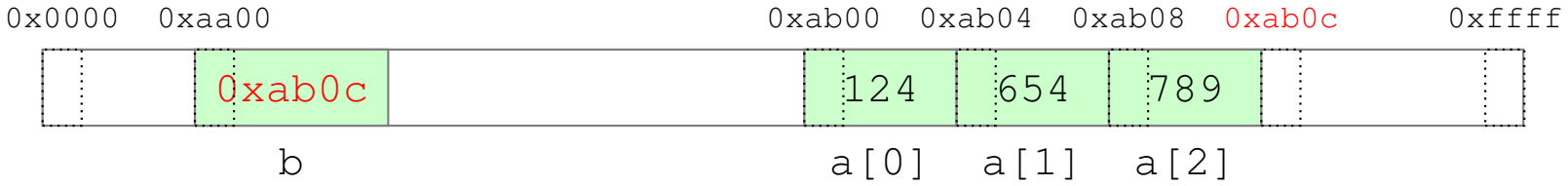
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
```

“C” Arrays

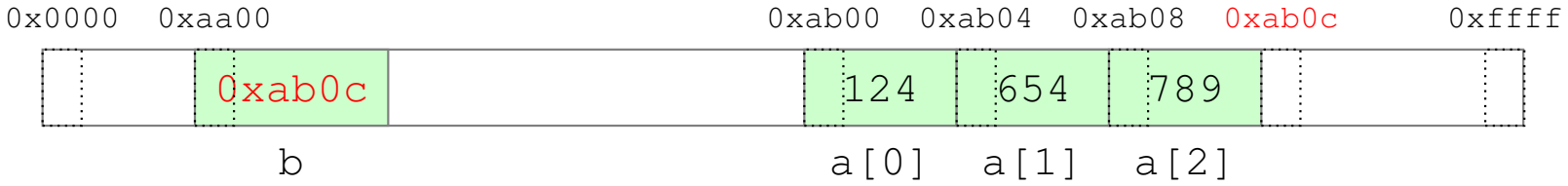
Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
```

“C” Arrays

Contiguous sequence of homogeneous objects in memory




```
int a[3] = {123, 456, 789}; // int[3]
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

Containers

- Objects that contain and own other objects
- Different characteristics and operations, some common traits
- Implemented as class templates

Sequence The client decides where an element gets inserted

- `array`, `vector`, `list`, `forward_list`, `deque`

Associative The container decides where an element gets inserted

Ordered The elements are sorted

- `map`, `multimap`, `set`, `multiset`

Unordered The elements are hashed

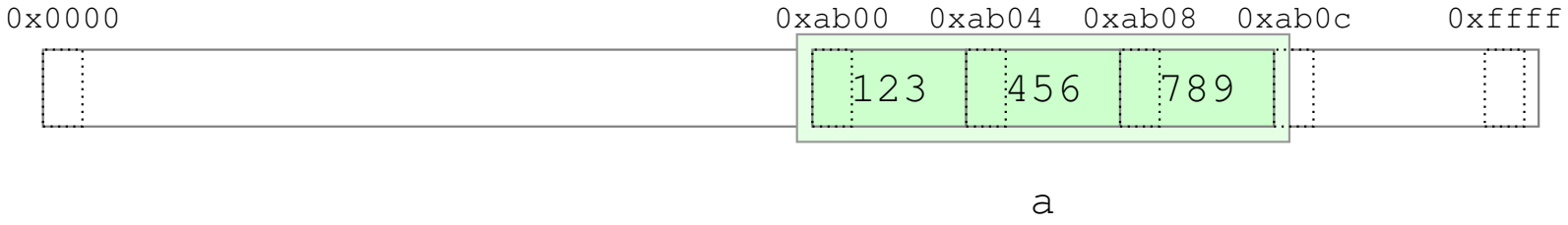
- `unordered_*`

std::array

Type-safe, no-runtime overhead alternative to a native array

std::array

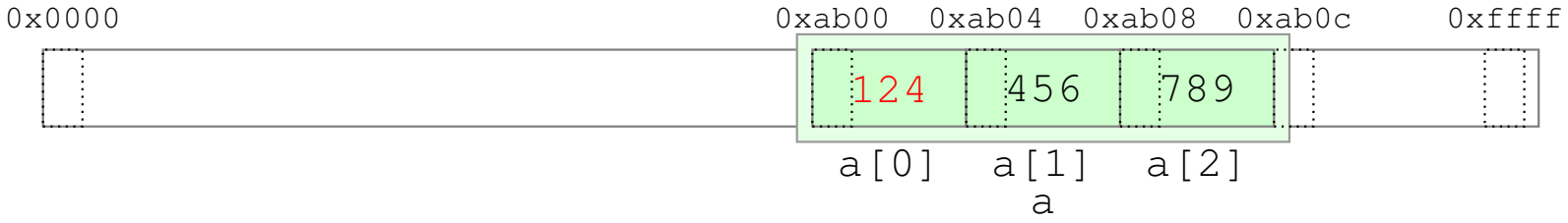
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
```

std::array

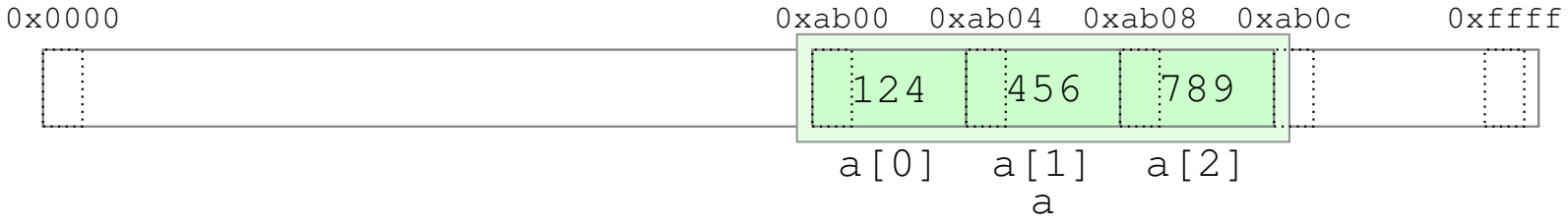
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
```

std::array

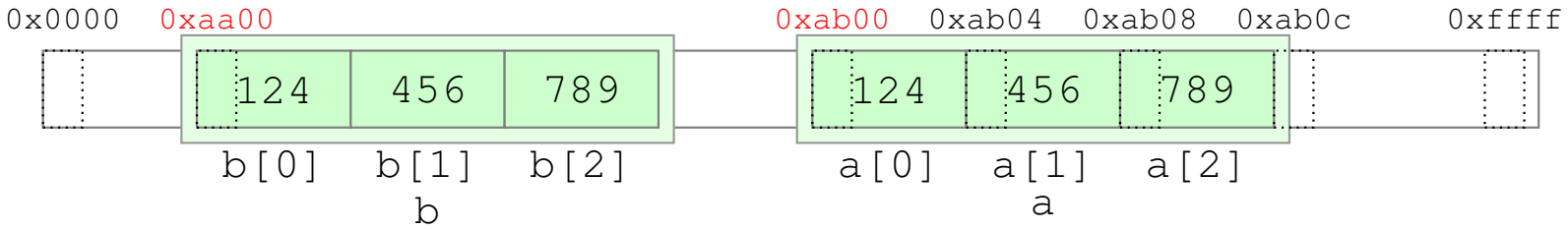
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                          // undefined behavior
```

std::array

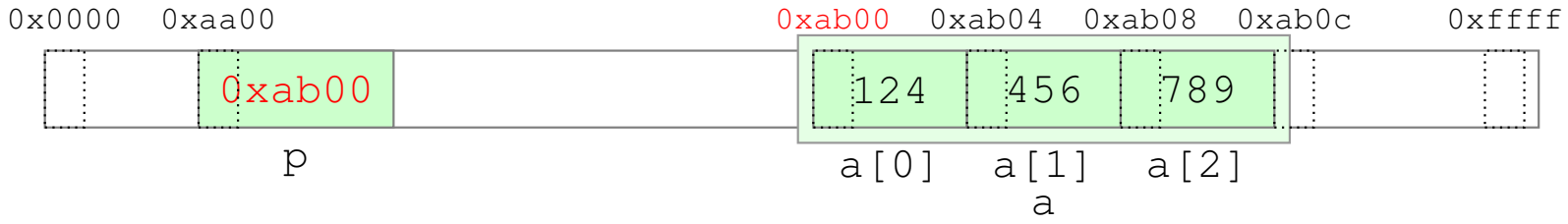
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                          // undefined behavior
auto b = a;                    // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
```

std::array

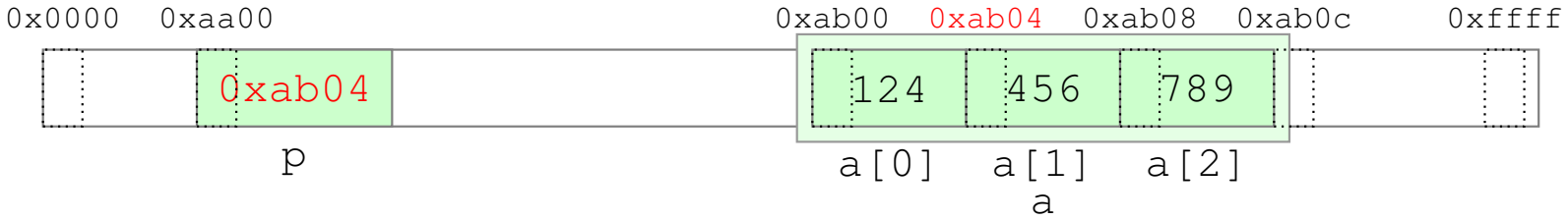
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
```

std::array

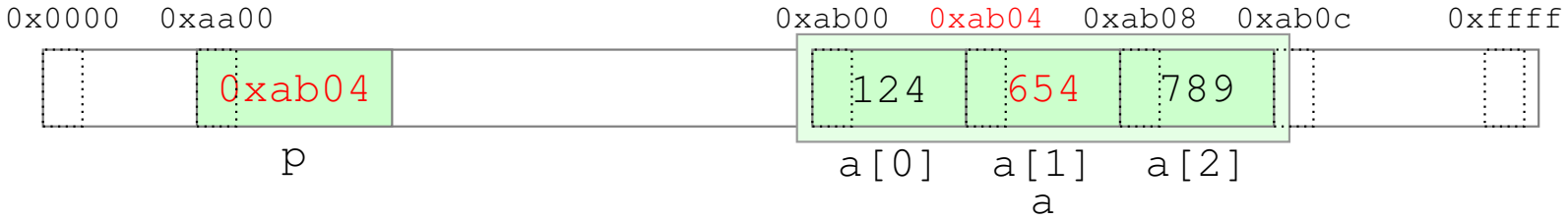
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
```

std::array

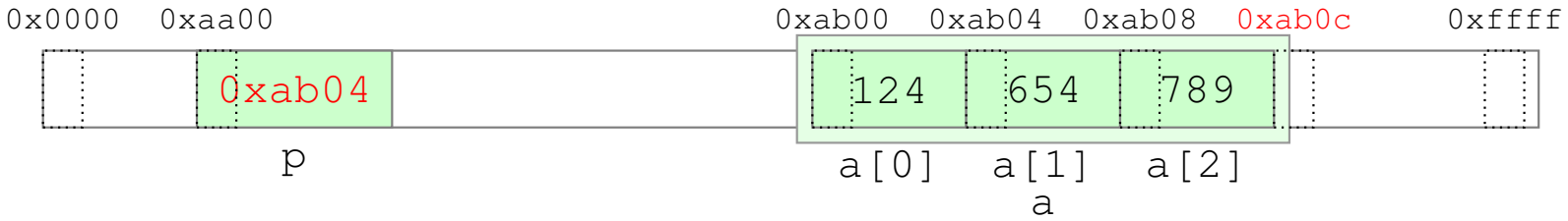
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
```

std::array

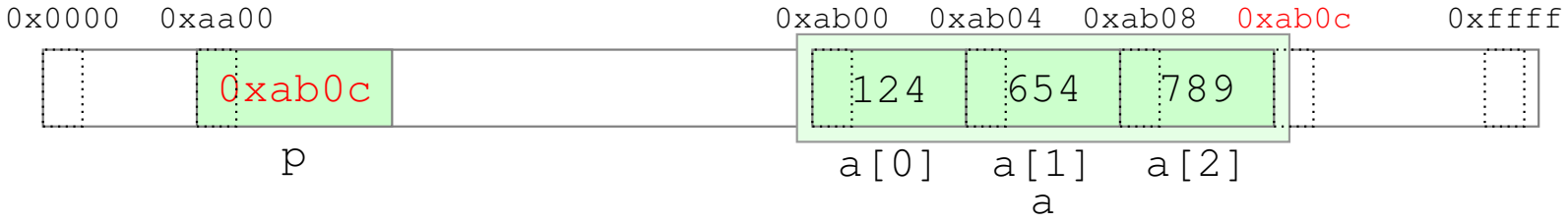
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
```

std::array

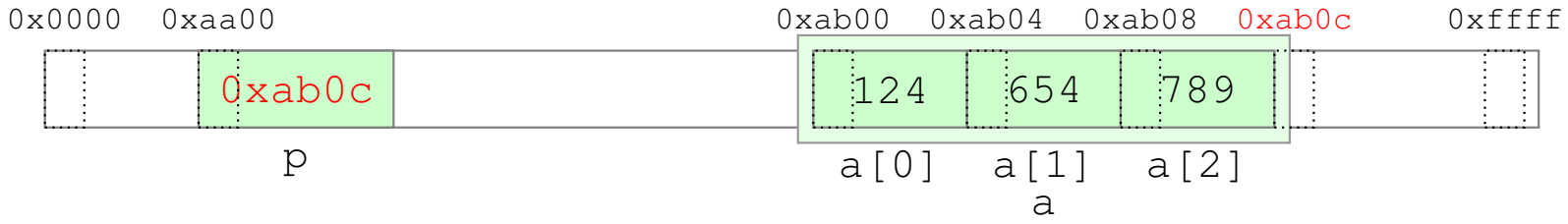
Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
```

std::array

Type-safe, no-runtime overhead alternative to a native array




```
std::array<int,3> a={123, 456, 789}; // size is part of the type
++a[0];
a[3];                       // undefined behavior
auto b = a;                 // std::array<int, 3>, make a copy!
assert(&b[0] != &a[0]);
auto p = &a[0];
++p;                        // increase by sizeof(int)
assert(p == &a[1]);
*p = 654;
p += 2;                     // increase by 2 * sizeof(int)
*p;                         // undefined behavior
if (p == a.data() + a.size()) { ... } // ok
```

std::array

```
template<class T, unsigned N> class array {
  T elems[N]; // no space overhead wrt to native array
public:
  size_t size() const { return N; }
  T* data() { return elems; }
  T const* data() const { return elems; }
  T& operator[](unsigned n) { return elems[n]; } // no checks!
  T const& operator[](unsigned n) const { return elems[n]; }
  ...
};

template<class T, unsigned N>
bool operator==(array<T, N> const& l, array<T, N> const& r) {
  return std::equal(l.data(), l.data() + N, r.data());
}

template<class T, unsigned N>
bool operator<(array<T, N> const& l, array<T, N> const& r) {
  return std::lexicographical_compare(
      l.data(), l.data() + N
    , r.data(), r.data() + N
  );
}
...
```

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying `all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying `all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`

Modifying `copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`

Algorithms

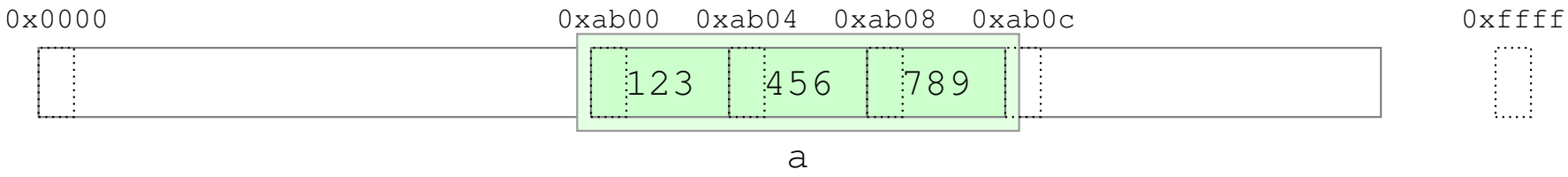
- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Range

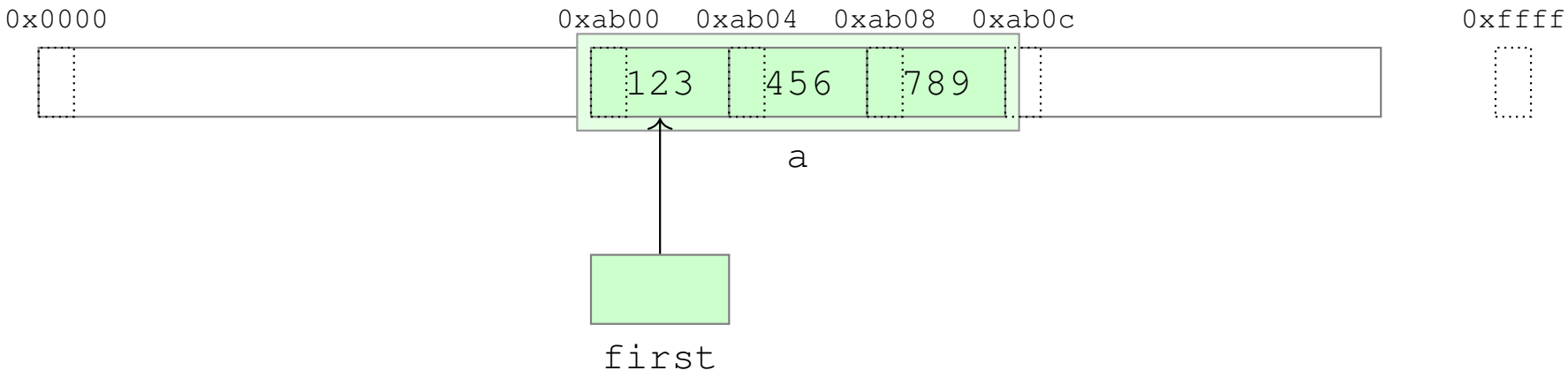
- A range is defined by a pair of iterators [*first*, *last*), with *last* referring to one past the last element in the range
  - the range is *half-open*
  - *first* == *last* means the range is empty
  - *last* can be used to return failure
- An iterator is a generalization of a pointer
  - it supports the same operations, possibly through overloaded operators
  - certainly `*` `++` `->` `==` `!=`, maybe `--` `+` `-` `+=` `-=` `<`
- Ranges are typically obtained from containers calling specific methods

Range (cont.)



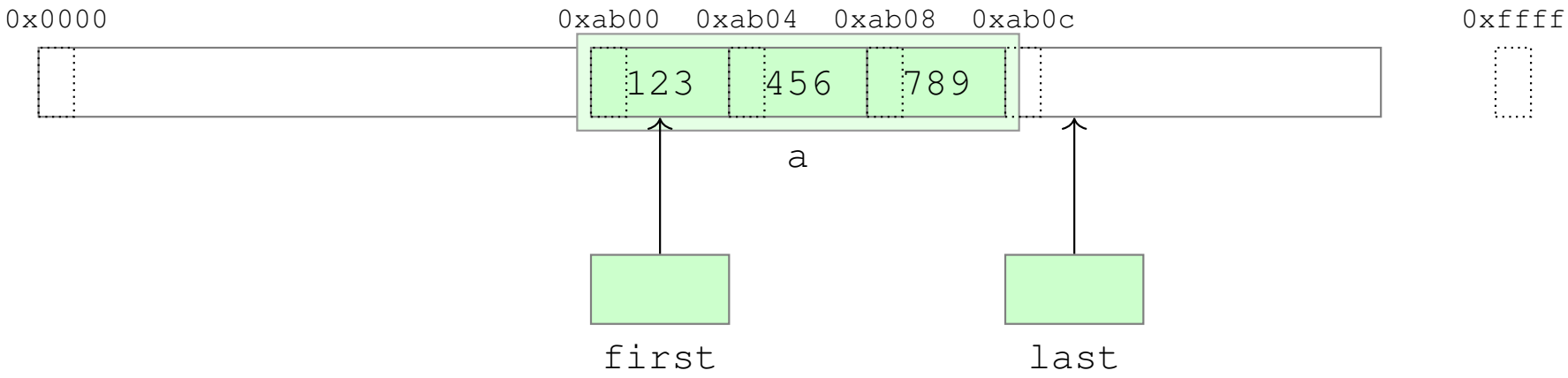

```
std::array<int,3> a = {123, 456, 789};
```

Range (cont.)



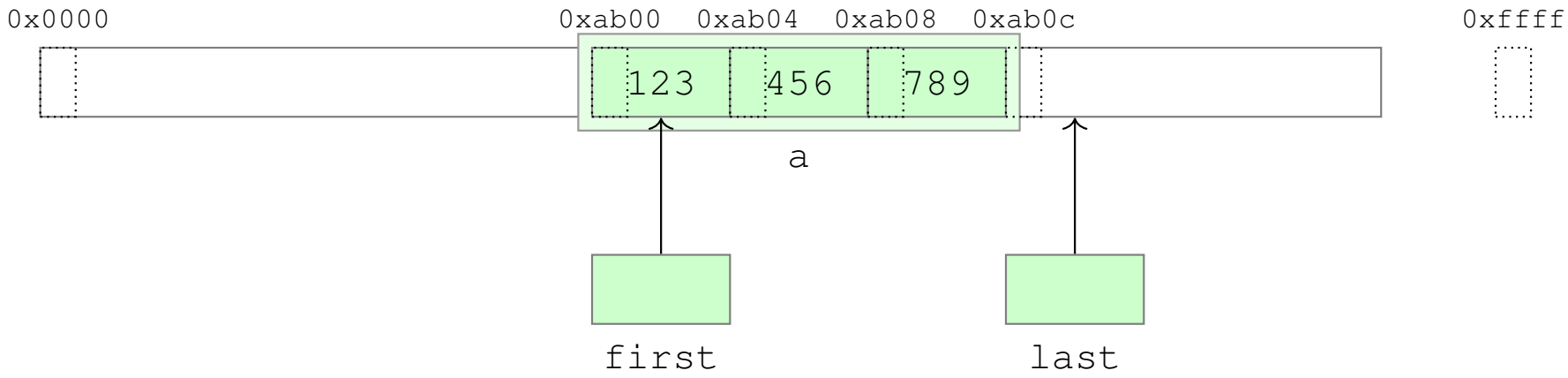

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();          // or std::begin(a)
```

Range (cont.)



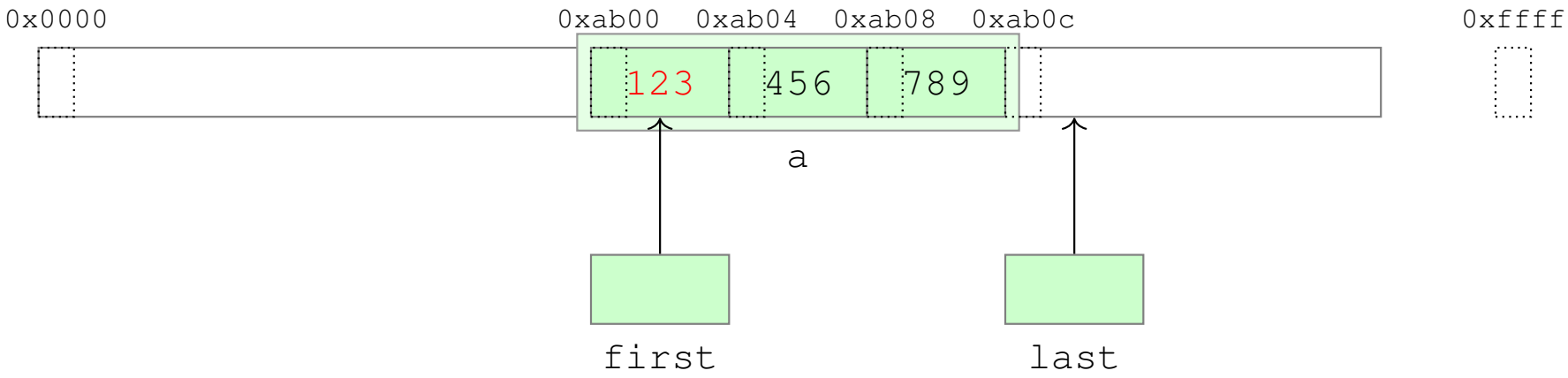

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();          // or std::begin(a)
auto last = a.end();             // or std::end(a)
```

Range (cont.)



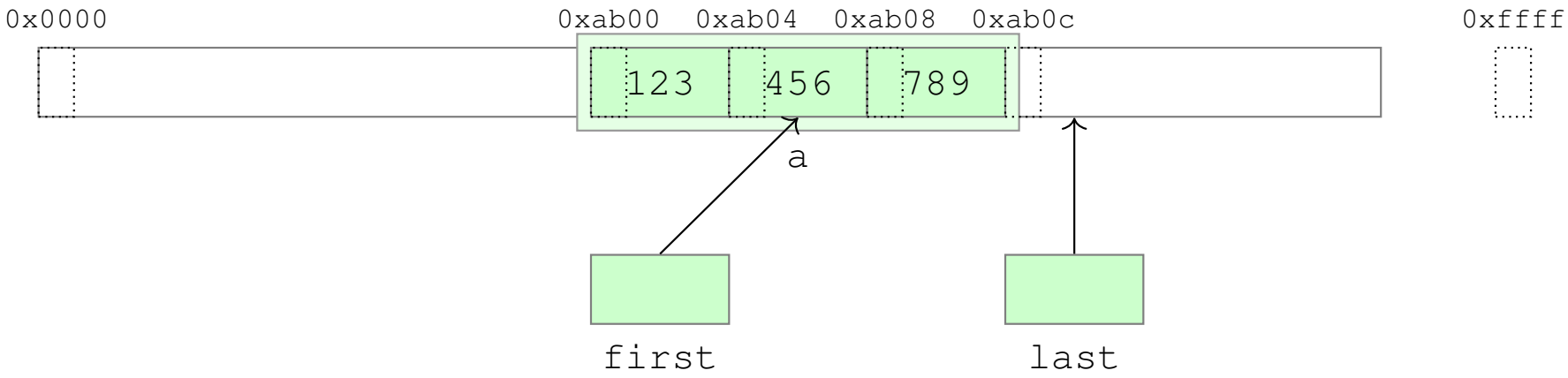

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



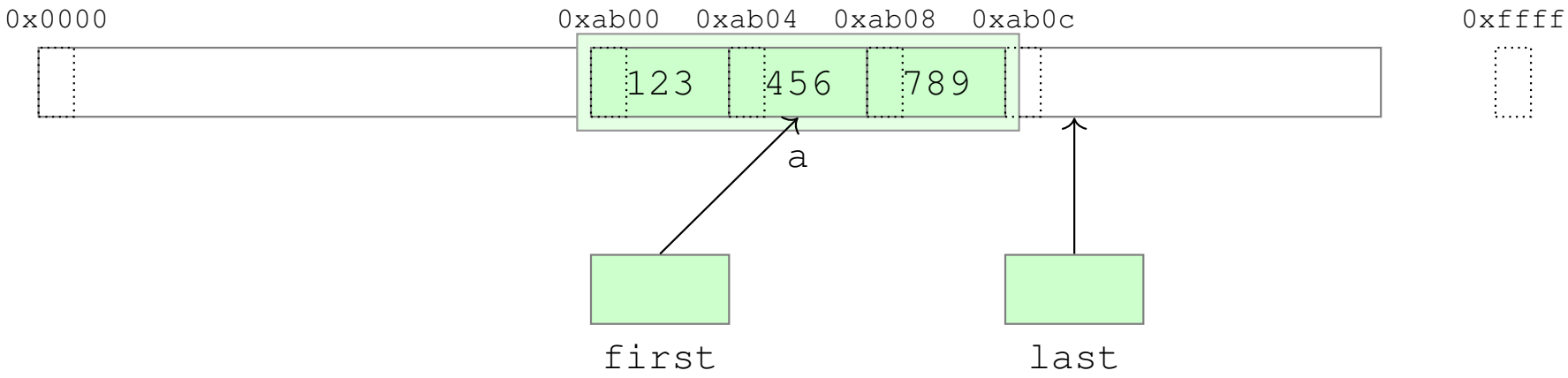

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



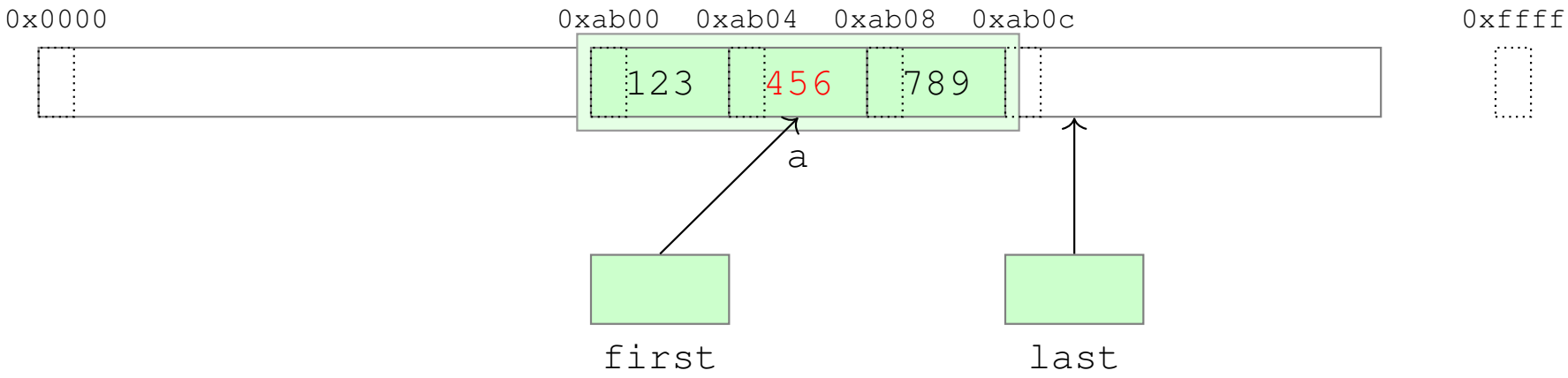

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



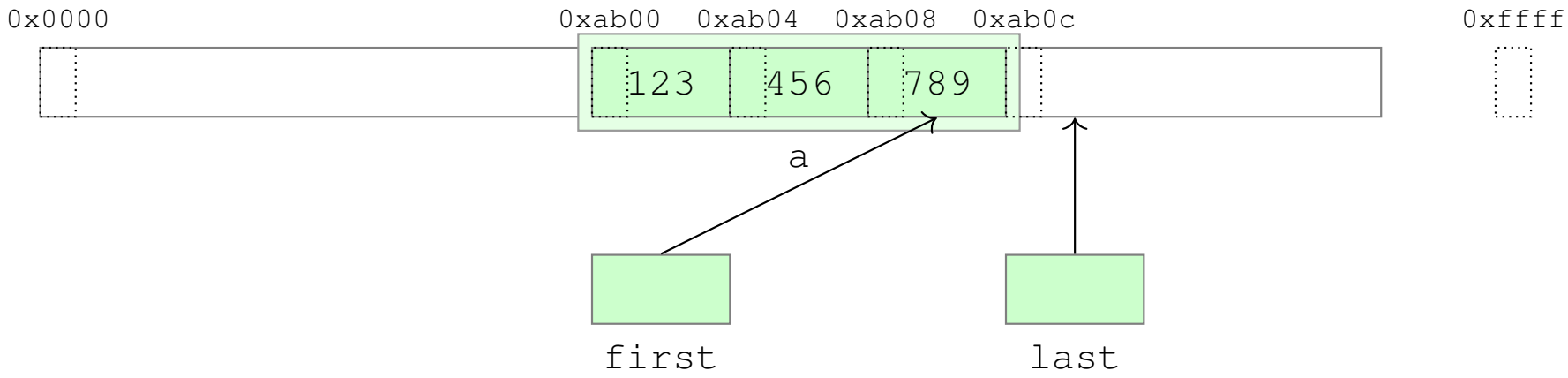

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



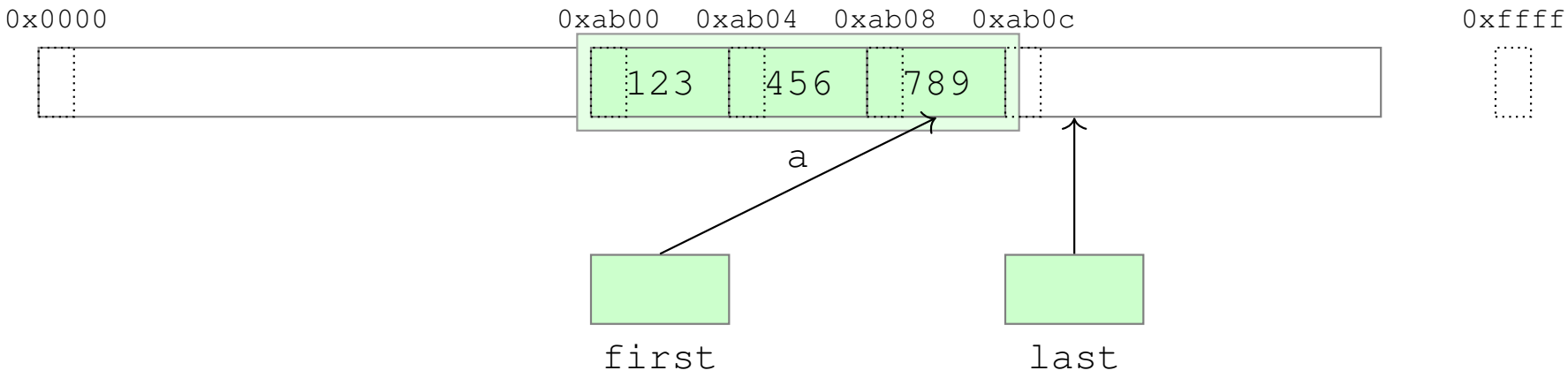

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



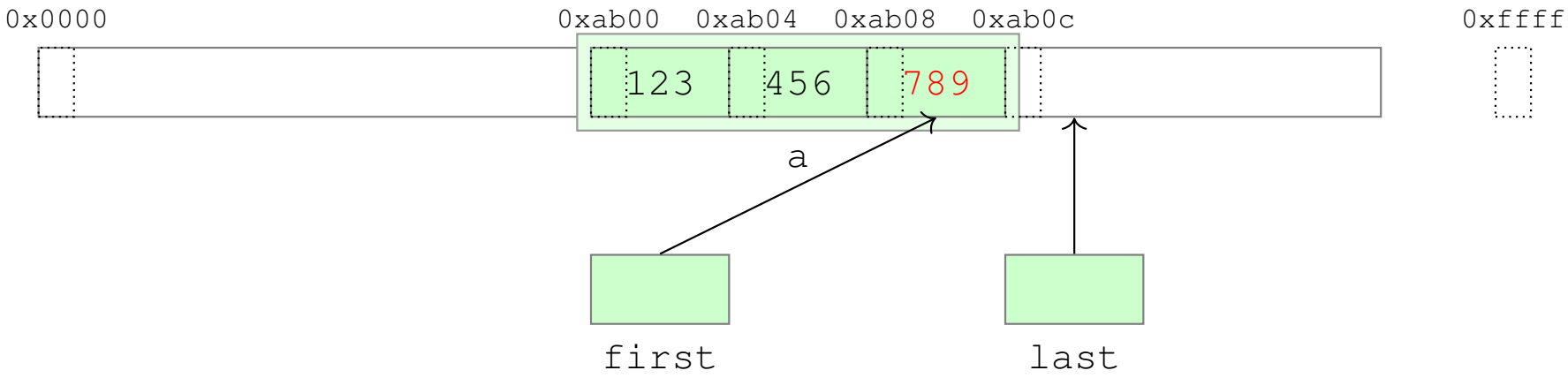

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



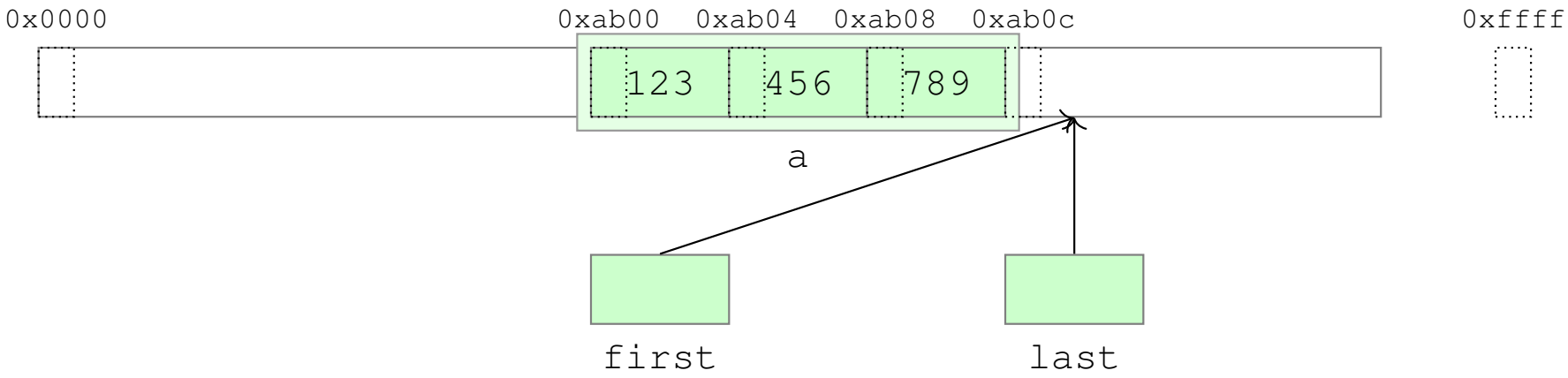

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



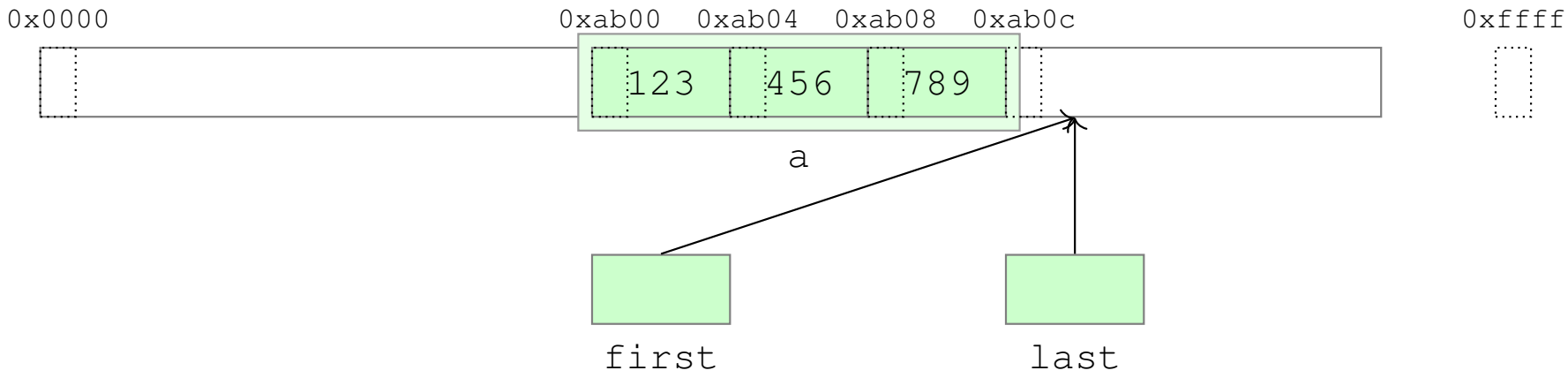

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

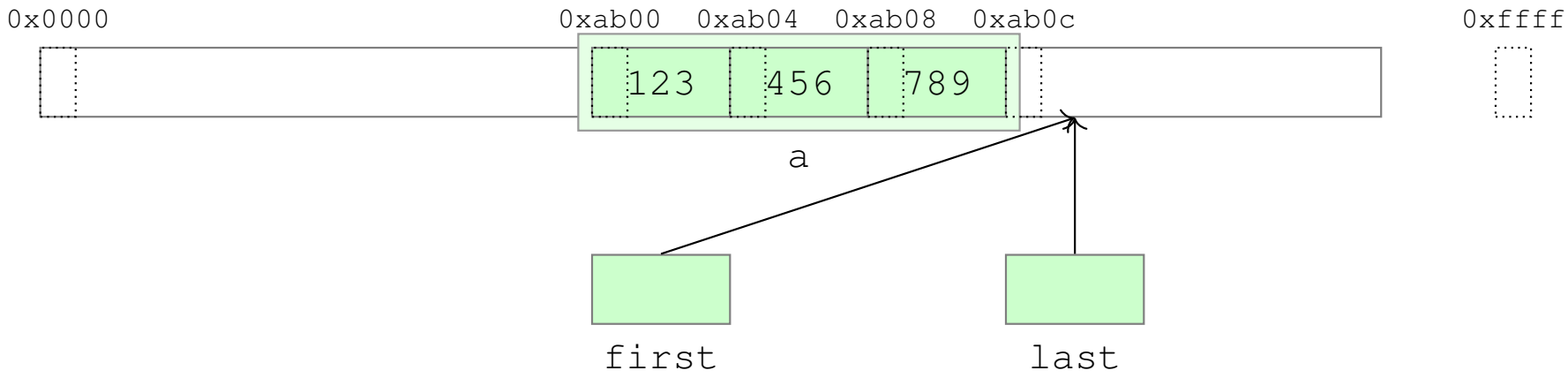
Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::array<T>::iterator` models the *RandomAccessIterator* concept

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto const last = a.end();      // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::array<T>::iterator` models the *RandomAccessIterator* concept

Generic programming

- A style of programming in which algorithms are written in terms of concepts

Generic programming

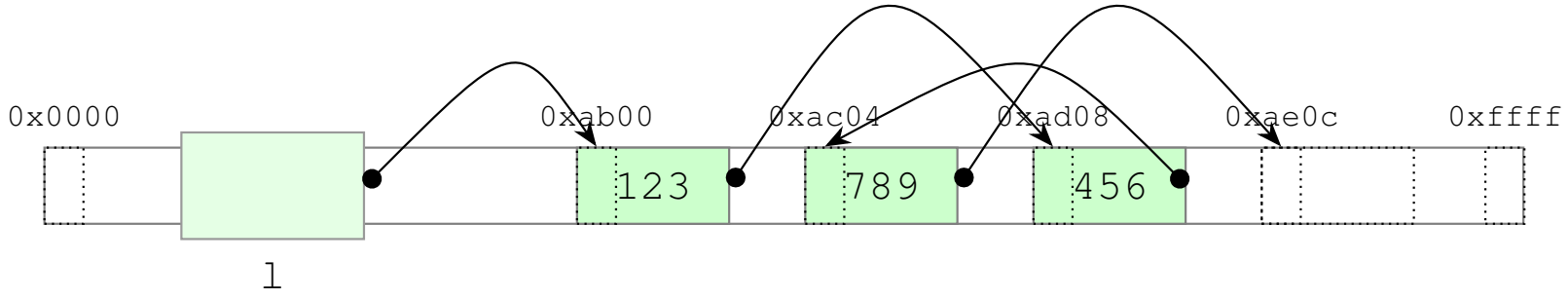
- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy
  - e.g. supported expressions, nested typedefs, memory layout, ...

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy
  - e.g. supported expressions, nested typedefs, memory layout, ...

```
template <class Iterator, class T>
Iterator
find(Iterator first, Iterator last, const T& value)
{
  for (; first != last; ++first)
    if (*first == value)
      break;
  return first;
}
```

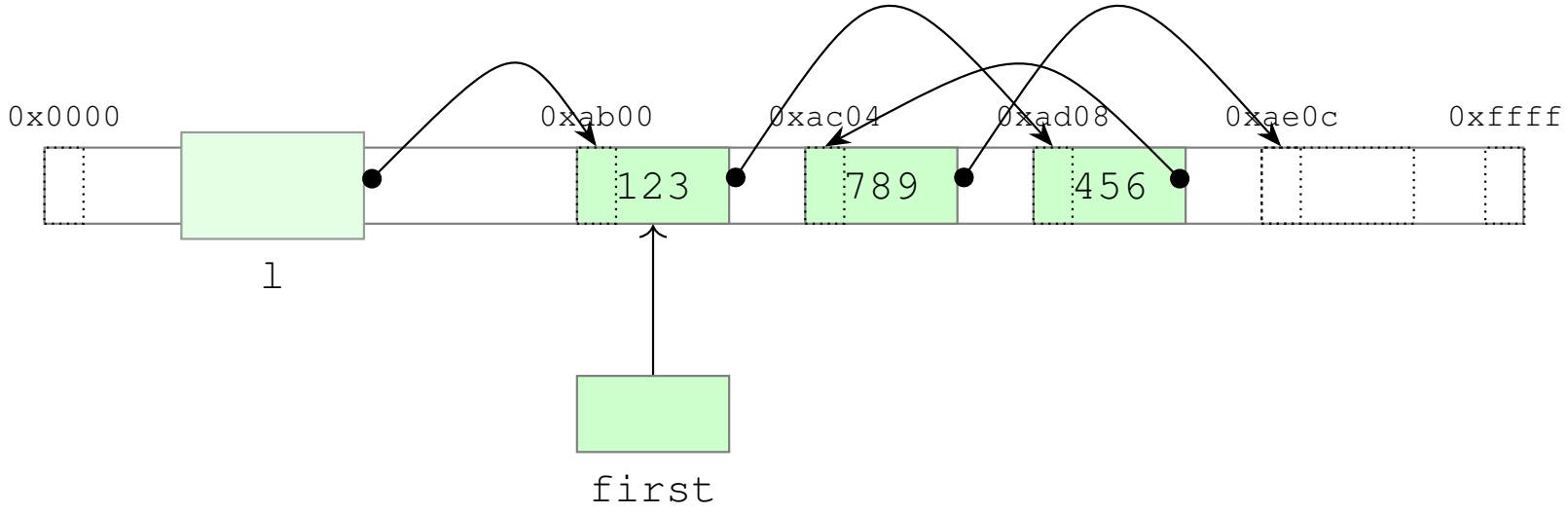
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

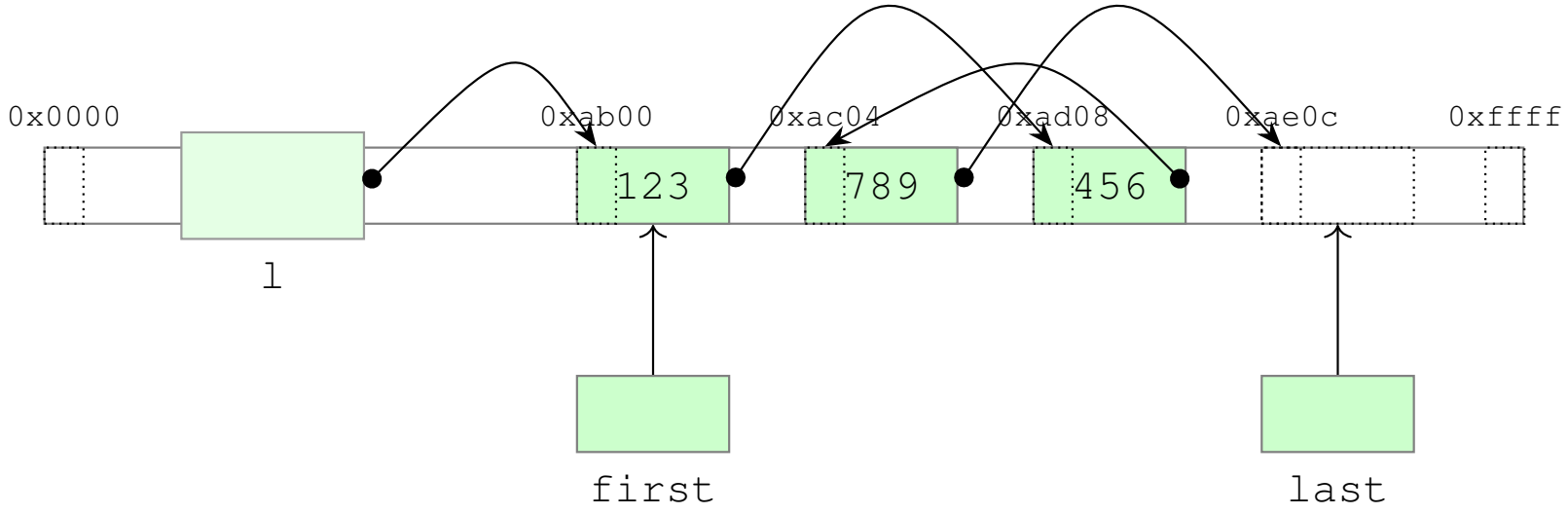
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

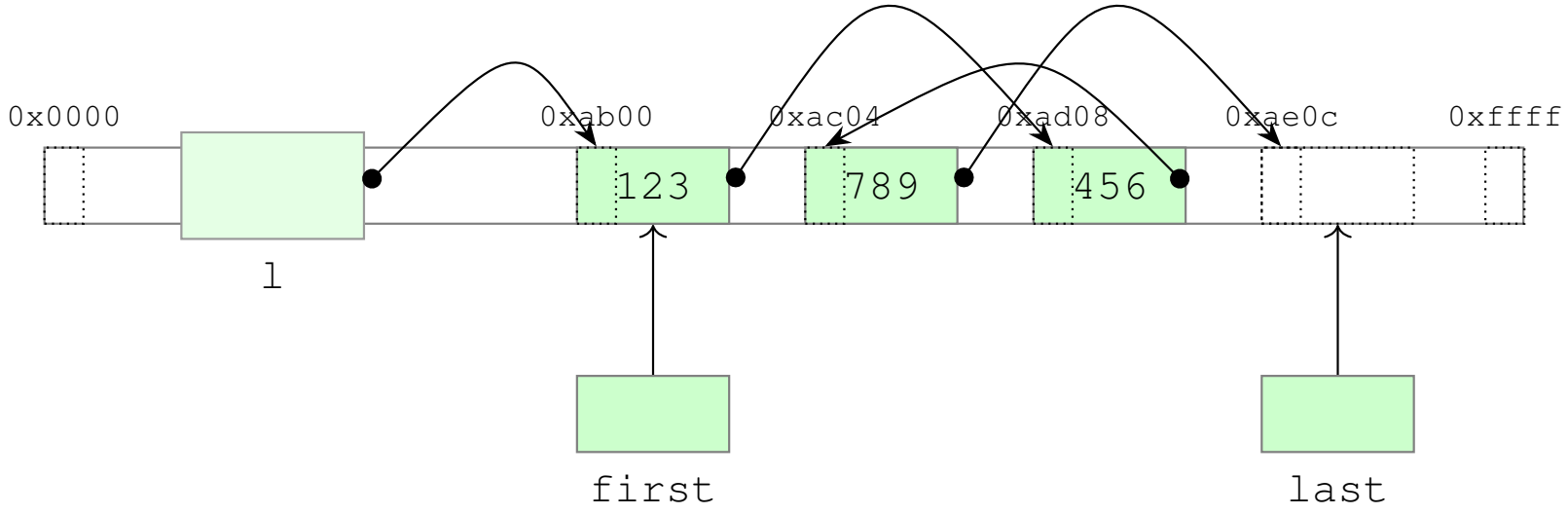
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

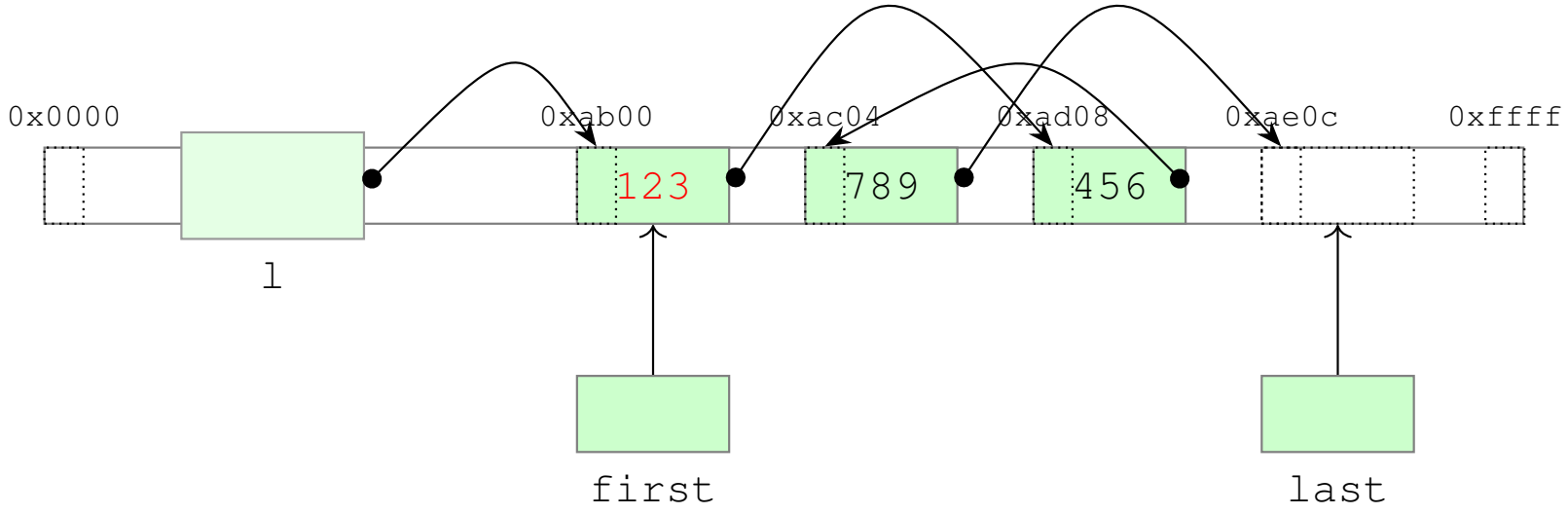
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

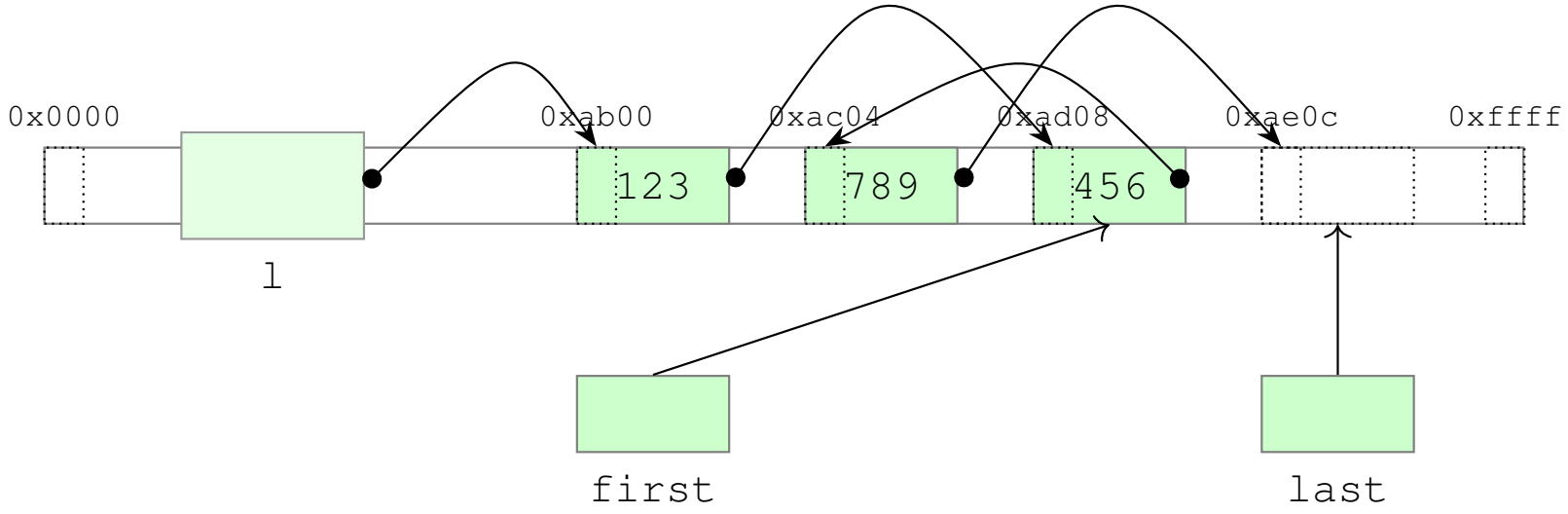
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

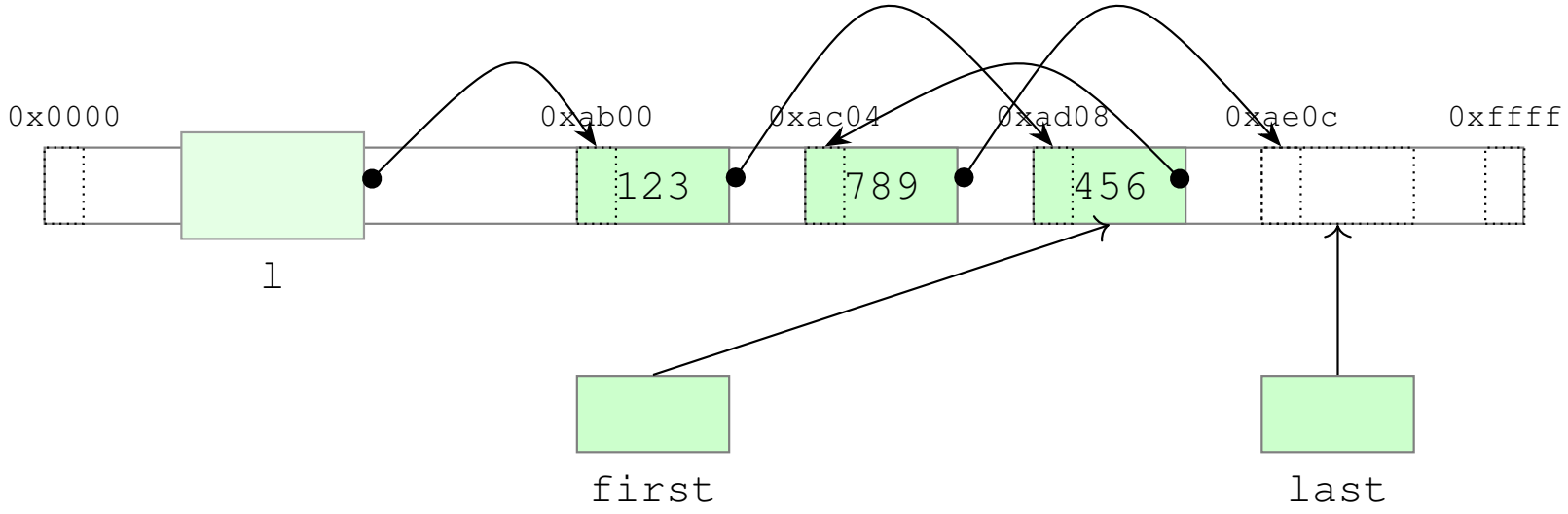
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

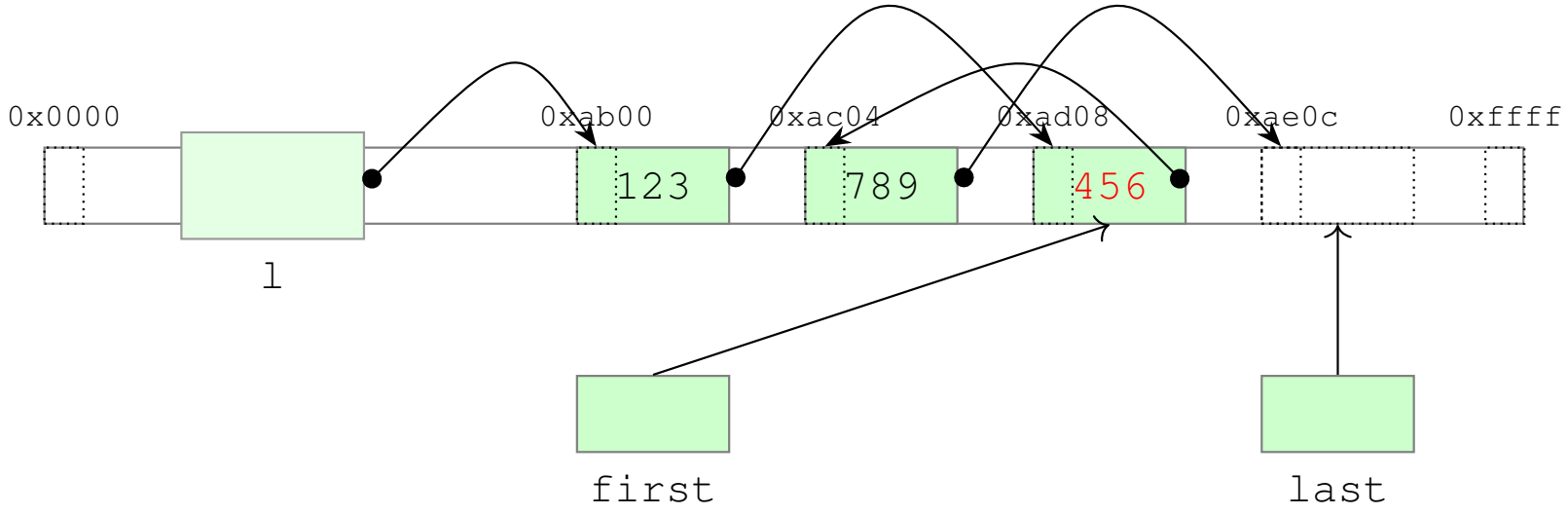
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

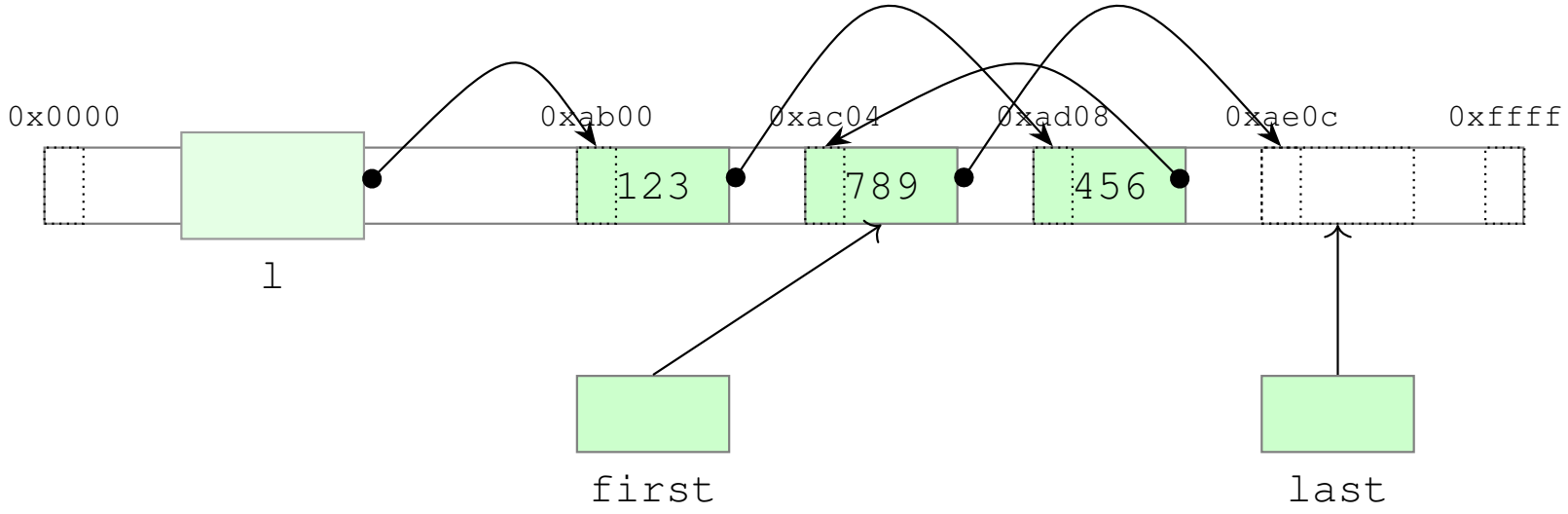
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

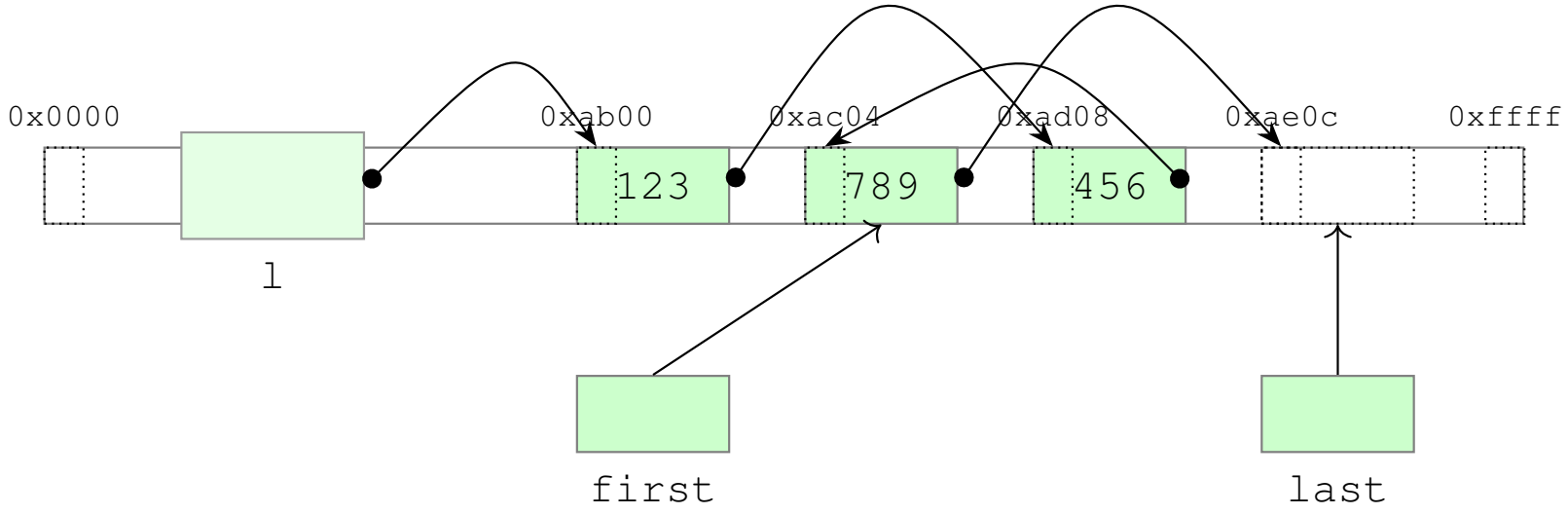
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

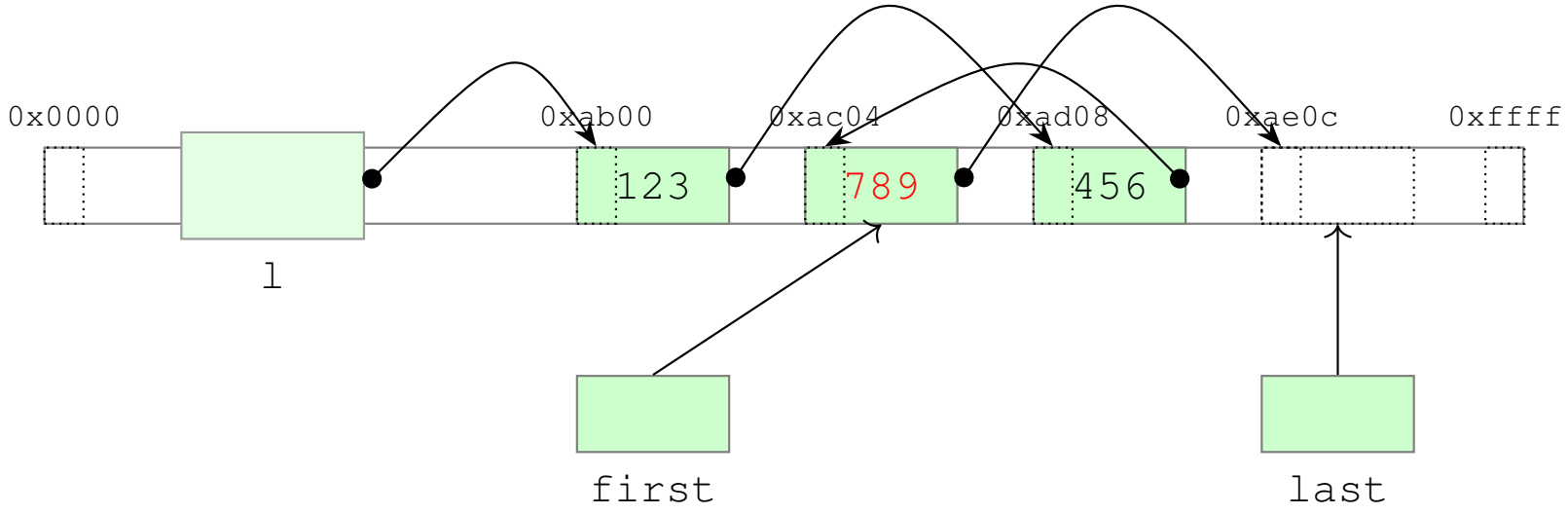
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

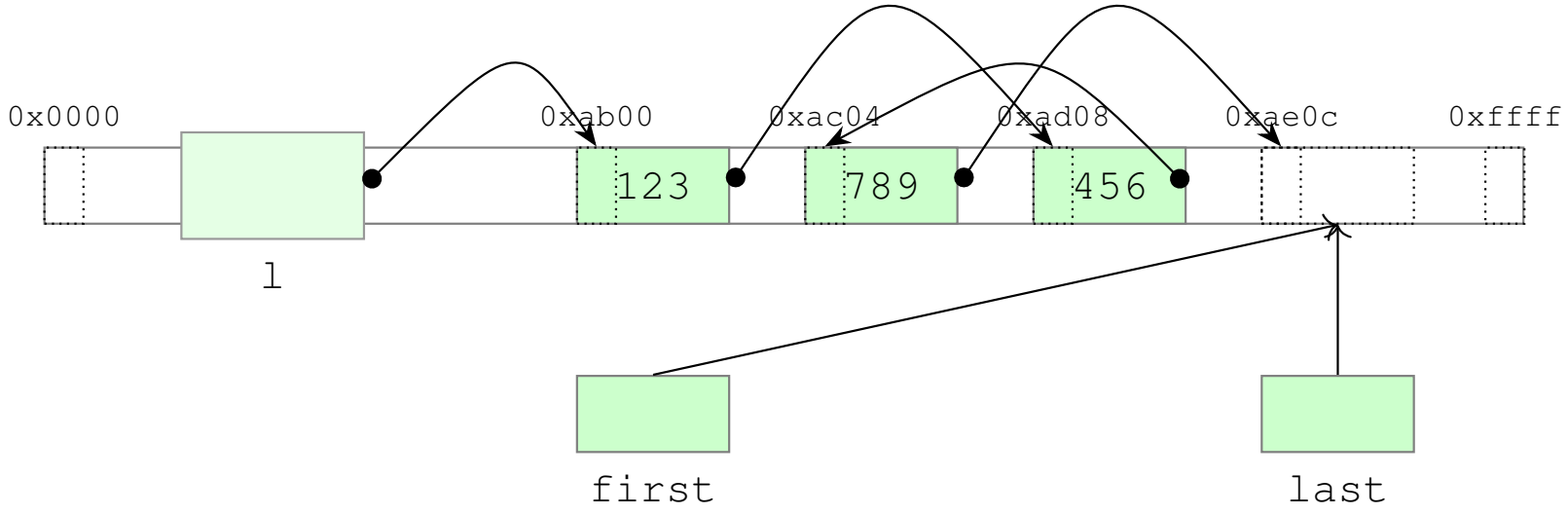
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

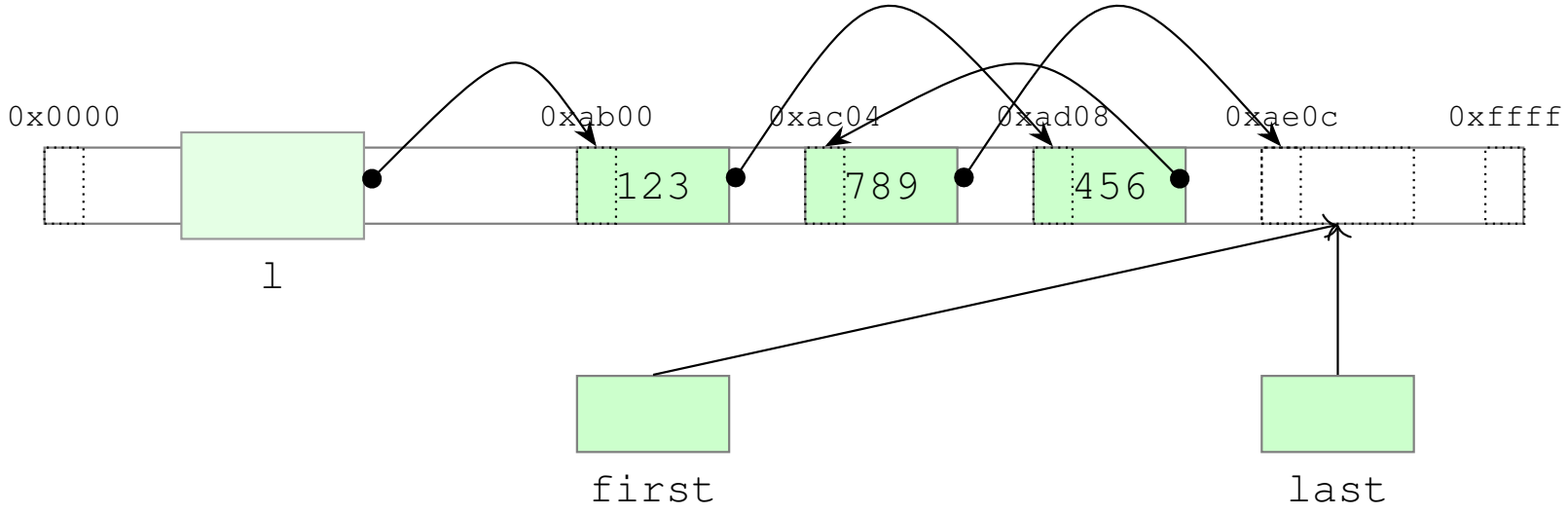
Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

Using the standard library

```
#include <array>      // note the angular brackets
                      // for standard headers


std::array<int,5> a;
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers


std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
```

Using the standard library

```
#include <array>     // note the angular brackets
#include <random>    // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ return dist(e); });
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ return dist(e); });
```

Using the standard library

```
#include <array>      // note the angular brackets
#include <random>     // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ return dist(e); });
std::sort(a.begin(), a.end());
```

Using the standard library

```
#include <array>     // note the angular brackets
#include <random>    // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ return dist(e); });
std::sort(a.begin(), a.end());
    std::find(a.begin(), a.end(), 123)
```

Using the standard library

```
#include <array>     // note the angular brackets
#include <random>    // for standard headers
#include <algorithm>

std::array<int,5> a;
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(a.begin(), a.size(), [&](){ return dist(e); });
std::sort(a.begin(), a.end());
if (std::find(a.begin(), a.end(), 123) == a.end()) {
  // 123 not found in a
}
```

std::vector

- The *default* container
- Similar to an array, but adjusts its size dynamically

```
std::vector<int> v; // size is 0
v.push_back(123);   // size is 1
...
v[42];
```

```
std::array<int,5> a;
std::vector<int> v(5);
```

std::vector

- The *default* container
- Similar to an array, but adjusts its size dynamically
- Inserting/removing at the end is $\mathcal{O}(1)$
- Inserting/removing in the middle is $\mathcal{O}(n)$
- Access to the i^{th} element is $\mathcal{O}(1)$

```
std::vector<int> v; // size is 0
v.push_back(123);   // size is 1
...
v[42];
```

```
std::array<int,5> a;
std::vector<int> v(5);
```

std::vector

- The *default* container
- Similar to an array, but adjusts its size dynamically
- Inserting/removing at the end is $\mathcal{O}(1)$
- Inserting/removing in the middle is $\mathcal{O}(n)$
- Access to the i^{th} element is $\mathcal{O}(1)$

```
std::vector<int> v; // size is 0
v.push_back(123);   // size is 1
...
v[42];
```

```
std::array<int,5> a;
std::vector<int> v(5);
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(v.begin(), v.size(), [&](){ return dist(e); });
std::sort(v.begin(), v.end());
if (std::find(v.begin(), v.end(), 123) == v.end()) {
  // 123 not found in v
}
```

std::list

- Bi-directional list
- Node-based, scattered in memory

```
std::list<int> l;
v.push_back(123);
v.push_front(456);
...
*std::next(l.begin(), 42);
```

std::list

- Bi-directional list
- Node-based, scattered in memory
- Inserting/removing is always $\mathcal{O}(1)$
- Access to the i^{th} element is $\mathcal{O}(n)$

```
std::list<int> l;
v.push_back(123);
v.push_front(456);
...
*std::next(l.begin(), 42);
```

std::list

- Bi-directional list
- Node-based, scattered in memory
- Inserting/removing is always $\mathcal{O}(1)$
- Access to the i^{th} element is $\mathcal{O}(n)$

```
std::list<int> l;
v.push_back(123);
v.push_front(456);
...
*std::next(l.begin(), 42);
```

```
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(l.begin(), l.size(), [&](){ return dist(e); });
std::sort(l.begin(), l.end());
if (std::find(l.begin(), l.end(), 123) == l.end()) {
  // 123 not found in l
}
```

std::list

- Bi-directional list
- Node-based, scattered in memory
- Inserting/removing is always $\mathcal{O}(1)$
- Access to the i^{th} element is $\mathcal{O}(n)$

```
std::list<int> l;
v.push_back(123);
v.push_front(456);
...
*std::next(l.begin(), 42);
```

```
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(l.begin(), l.size(), [&](){ return dist(e); });
std::sort(l.begin(), l.end()); // error
if (std::find(l.begin(), l.end(), 123) == l.end()) {
  // 123 not found in l
}
```

std::list

- Bi-directional list
- Node-based, scattered in memory
- Inserting/removing is always $\mathcal{O}(1)$
- Access to the i^{th} element is $\mathcal{O}(n)$

```
std::list<int> l;
v.push_back(123);
v.push_front(456);
...
*std::next(l.begin(), 42);
```

```
std::default_random_engine e;
std::uniform_int_distribution<int> dist(1, 6);
std::generate_n(l.begin(), l.size(), [&](){ return dist(e); });
l.sort();
if (std::find(l.begin(), l.end(), 123) == l.end()) {
  // 123 not found in l
}
```

Function objects

A mechanism to define *something-callable-like-a-function*

Function objects

A mechanism to define *something-callable-like-a-function*

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}


auto s = cat("XY", 5); // XY-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat{};
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = cat("XY", 5); // XY-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = Cat{}("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = Cat{}("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, Cat{}
); // XY-2-3-5
```

Function objects

A function object, being the instance of a class, can have state

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = cat2("XY", 5); // XY+5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = cat2("XY", 5); // XY+5

vector<int> v{2,3,5};
auto s3 = accumulate(..., cat1); // XY-2-3-5
auto s4 = accumulate(..., cat2); // XY+2+3+5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = CatWithState{'-'}("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = CatWithState{'+'}("XY", 5); // XY+5

vector<int> v{2,3,5};
auto s3 = accumulate(..., CatWithState{'-'}); // XY-2-3-5
auto s4 = accumulate(..., CatWithState{'+'}); // XY+2+3+5
```

Function objects (cont.)

An example from the standard library

```
#include <random>

// random bit generator (mersenne twister)
std::mt19937 gen;

// generate N 32-bit unsigned integer numbers
for (int n = 0; n != N; ++n) {
  std::cout << gen() << '\n';
}

// generate N floats distributed normally (mean: 0., stddev: 1.)
std::normal_distribution<float> dist;
for (int n = 0; n != N; ++n) {
  std::cout << dist(gen) << '\n';
}

// generate N ints distributed uniformly between 1 and 6 included
std::uniform_int_distribution<> roll_dice(1, 6);
for (int n = 0; n != N; ++n) {
  std::cout << roll_dice(gen) << '\n';
}
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});
```

```
accumulate(...,CatWithState{'-'});
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});

accumulate(...,
 [](string const& s, int i) {
   return s + '-' + to_string(i);
 }
);

accumulate(...,CatWithState{'-'});
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});

accumulate(...,
 [](string const& s, int i) {
   return s + '-' + to_string(i);
 }
);

accumulate(...,CatWithState{'-'});

char c{'-'};
accumulate(...,
  [=](string const& s, int i) {
    return s + c + to_string(i);
  }
);
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = []
```

```
class SomeUniqueName {

 public:
  explicit SomeUniqueName(      )
                {}
  auto operator()

};


auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [](int i)
{ return i + v; }
```

```
class SomeUniqueName {

 public:
  explicit SomeUniqueName(      )
                {}
  auto operator()(int i)
  { return i + v ; }
};


auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [](int i)
{ return i + v; }
```

```
class SomeUniqueName {

 public:
  explicit SomeUniqueName(      )
                {}
  auto operator()(int i)
  { return i + v ; }
};

int v = 3;
auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [=](int i)
{ return i + v; }
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [=](int i)
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [v = 3](int i)
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [v = 3](int i) -> int
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  int  operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [v] {};
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_v {}
  ...
};

auto l = SomeUniqueName{v};
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [&v] {};
```

```
class SomeUniqueName {
  int& v_;
 public:
  explicit SomeUniqueName(int& v)
    : v_v {}
  ...
};

auto l = SomeUniqueName{v};
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [&v] {};
```

```
class SomeUniqueName {
  int& v_;
 public:
  explicit SomeUniqueName(int& v)
    : v_v {}
  ...
};

auto l = SomeUniqueName{v};
```

- Global variables are available without being captured

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variable captured by value are not modifiable

```
[] {};
```

```
struct SomeUniqueName {
  auto operator()() const {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variable captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable {};
```

```
struct SomeUniqueName {
  auto operator()() {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variable captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable -> void {};
```

```
struct SomeUniqueName {
  void operator()() {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variable captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable -> void {};
```

```
struct SomeUniqueName {
  void operator()() {}
};
```

- Variables captured by reference can be modified
  - There is no way to capture by `const&`

```
int v = 3;
[&v] { ++v; }();
assert(v == 4);
```

Lambda: dangling reference

Be careful not to have dangling references in a closure

- It's similar to a function returning a reference to a local variable

```
auto make_lambda()
{
  int v = 3;
  return [&] { return v; }; // return a closure
}

auto l = make_lambda();
auto d = l(); // the captured variable is dangling here
```

```
auto start_in_thread()
{
  int v = 3;
  return std::async([&] { return v; });
}
```

std::function

- Type-erased wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function
- Some space and time overhead, so use only if a template parameter is not satisfactory

std::function

- Type-erased wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function
- Some space and time overhead, so use only if a template parameter is not satisfactory

```
#include <functional>

int sum_squares(int x, int y) { return x * x + y * y; }

int main() {
  std::vector<std::function<int(int, int)>> v {
    std::plus<>{},        // has a compatible operator()
    std::multiplies<>{}, // idem
    &sum_squares)
  };
  for (int k = 10; k <= 1000; k *= 10) {
    v.push_back([k](int x, int y) -> int { return k * x + y; });
  }

  for (auto const& f : v) { std::cout << f(4, 5) << '\n'; }
}
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    int(*)(FILE*)
  >{
  std::fopen(...),
  &std::fclose
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    decltype(&std::fclose)
  >{
  std::fopen(...),
  &std::fclose
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<int(FILE*)>
  >{
  std::fopen(...),
  std::fclose
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  std::fclose
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  [](FILE* f){ std::fclose(f); }
};
```

Smart pointers and custom deleters

- `unique_ptr` and `shared_ptr` can be used as general-purpose resource handlers
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  [](auto f){ std::fclose(f); }
};
```

Hands-on

Starting from `algo.cpp`, write code to

- sum all the elements of the vector
- multiply all the elements of the vector
- sort the vector in ascending and descending order
- move the even numbers to the beginning
- move the three central numbers to the beginning
- erase from the vector the elements that satisfy a predicate
- ...