TDDD38/726G82 - Advanced programming in C++

Introduction STL

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Introduction
2 IO
3 Sequential Containers

Introduction

What is the STL?

- Library accessible everywhere
- Solving common problems
- Modular design
- Efficiency

Introduction

What is the STL?

- Library accessible everywhere
  - Same behaviour independent of platform
  - Shipped with the compiler itself
  - ISO C++ requires the full library to be accessible
- Solving common problems
- Modular design
- Efficiency

Introduction

What is the STL?

- Library accessible everywhere
- Solving common problems
  - Having to reinvent the wheel is costly
  - There are problems most programmers face
  - Designed to be as widely usable as possible
- Modular design
- Efficiency

Introduction

What is the STL?

- Library accessible everywhere
- Solving common problems
- Modular design
  - Don’t pay for what you don’t use
  - Only import the parts that you need
  - All modules are compatible with each other
- Efficiency

Introduction

What is the STL?

- Library accessible everywhere
- Solving common problems
- Modular design
- Efficiency
  - Library writers are very skilled
  - Components are highly optimized
  - Maintenance is not your responsibility

Introduction

Standard **T**emplate **L**ibrary

Introduction

Design principles of STL

- Should be as general as possible

Introduction

Design principles of STL

- Should be as general as possible
- Solves common problems

Introduction

Design principles of STL

- Should be as general as possible
- Solves common problems
- The common case should be convenient

Introduction

Design principles of STL

- Should be as general as possible
- Solves common problems
- The common case should be convenient
- Must work together with user-defined code

Introduction

Design principles of STL

- Should be as general as possible
- Solves common problems
- The common case should be convenient
- Must work together with user-defined code
- Efficient enough to replace hand-written alternatives

Introduction

Design principles of STL

- Should be as general as possible
- Solves common problems
- The common case should be convenient
- Must work together with user-defined code
- Efficient enough to replace hand-written alternatives
- Should have robust error handling

Introduction

Components

- Algorithms
- Containers
- Functions
- Iterators

Introduction

Components

- Algorithms
  - General facilities for solving common problems
  - A large amount of algorithms exist in the STL
  - Highly optimized for both speed and memory
- Containers
- Functions
- Iterators

Introduction

Components

- Algorithms
- Containers
  - General data structures
  - Based on high level abstractions
  - Should not be required to understand the underlying implementation
- Functions
- Iterators

Introduction

Components

- Algorithms
- Containers
- Functions
  - General utility functions
  - Should be usable for as many types as possible
  - Solves all manner of problems
- Iterators

Introduction

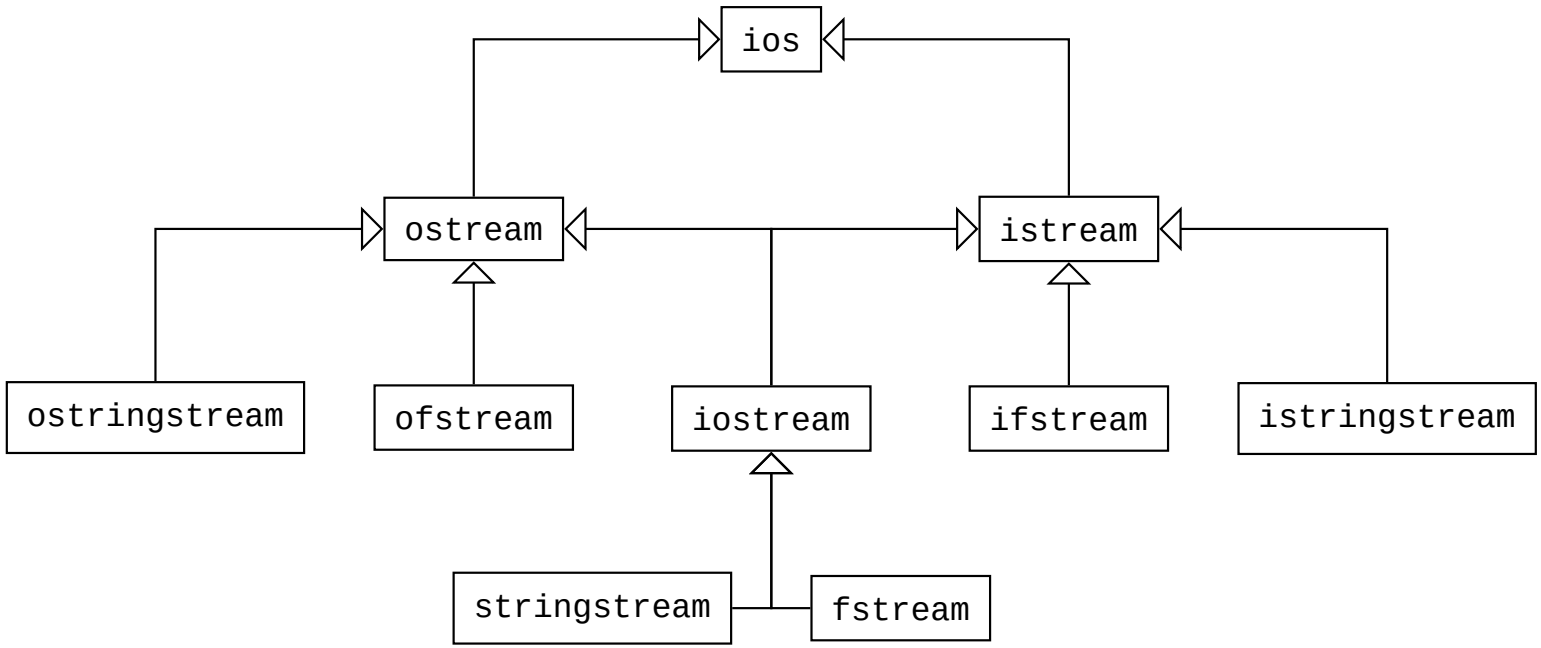
Components

- Algorithms
- Containers
- Functions
- Iterators
  - Abstraction which allows for general traversal of data
  - Used in conjunction with algorithms
  - An interface that works with all containers without the need to specify the container type

LINKÖPING UNIVERSITY

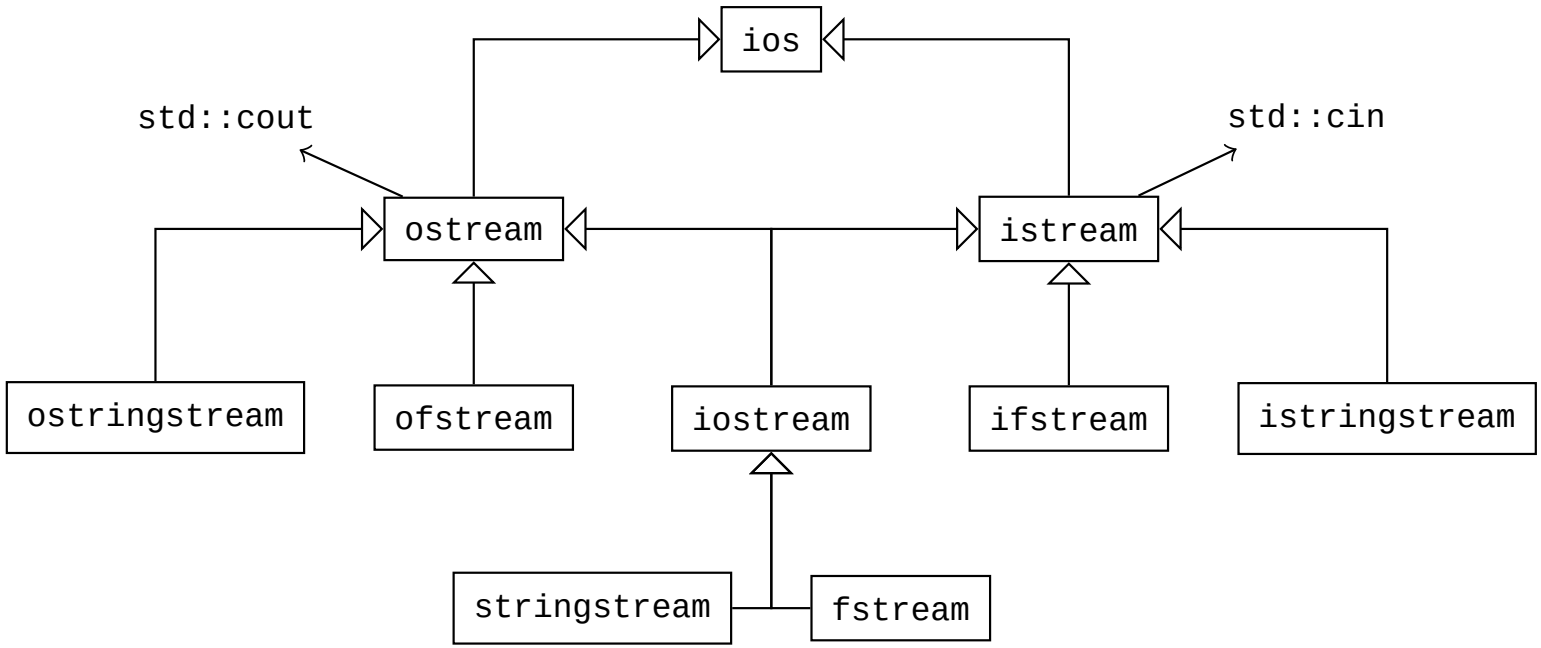
IO

Streams


ios
ostream
istream
ostringstream
ofstream
iostream
ifstream
istringstream
stringstream
fstream

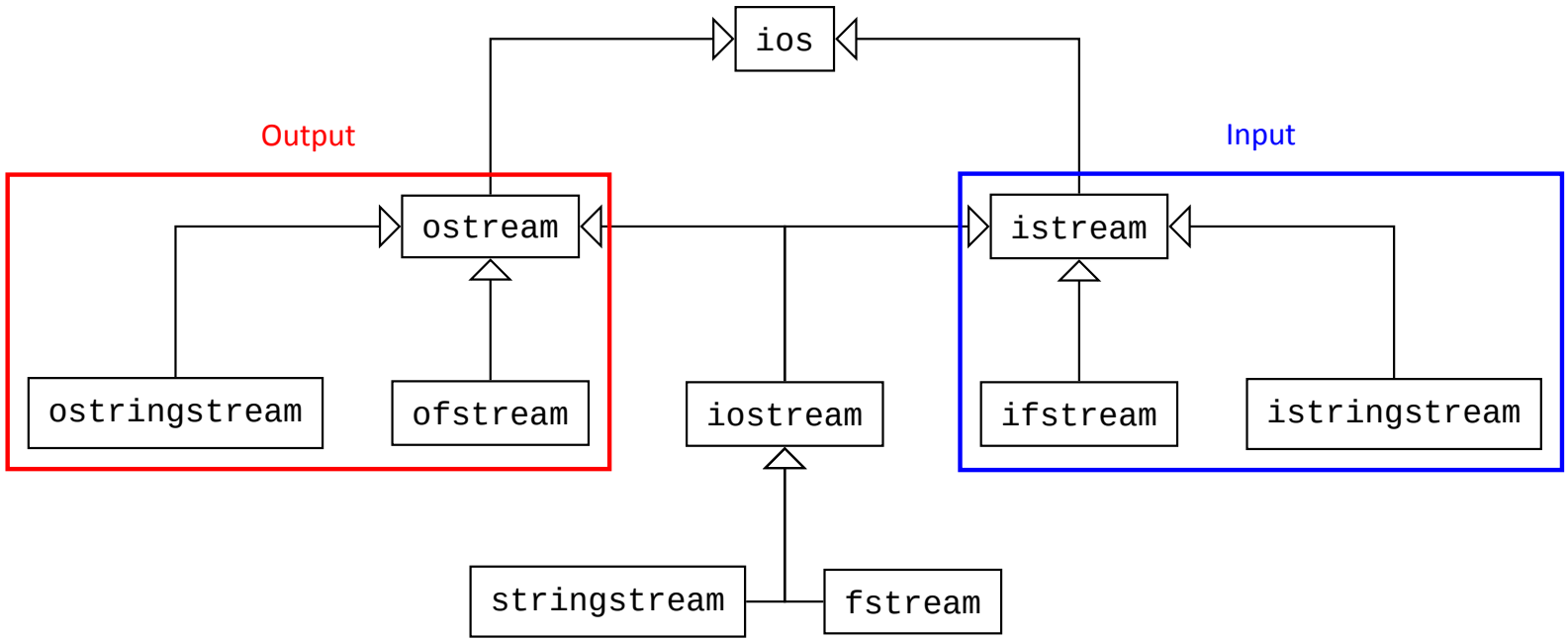
IO

Streams


ios
std::cout
std::cin
ostream
istream
ostringstream
ofstream
iostream
ifstream
istringstream
stringstream
fstream

IO

Streams


ios
Output
Input
ostream
istream
ostringstream
ofstream
iostream
ifstream
istringstream
stringstream
fstream

IO

Streams

- Represent reading and writing data to some *device*
- Example of devices;
  - a terminal
  - a file
  - a chunk of memory
  - sockets
- `operator>>` to read
- `operator<<` to write

IO

Stream operators

```
template <typename T>
ostream& operator<<(ostream& os, T&& data)
{
  // write data to the device
  return os;
}
// ...
cout << 1 << 2;
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
cout << 1 << 2;
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
(cout << 1) << 2;
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
operator<<(cout, 1) << 2;
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
cout << 2;
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
(cout << 2);
```

IO

Stream operators

```
ostream& operator<<(ostream& os, T&& data);
```

```
cout;
```

IO

Chaining operators

- Stream operators return a reference to the stream
- This is done to enable *chaining*
- Since << and >> are *left associative* this will allow us to make several calls to the stream in one expression

IO

Devices

```
ostream& operator<<(ostream& os, T&& data);

int main()
{
  ostringstream oss{};
  ofstream ofs{"my_file.txt"};
  cout << 1; // write to terminal
  oss  << 1; // write to string
  ofs  << 1; // write to file
  oss.str(); // access string
}
```

IO

Devices

```
istream& operator>>(istream& is, T& data);

int main()
{
  int x;
  istringstream iss{"1"};
  ifstream ofs{"my_file.txt"};
  cin >> x; // read from terminal
  oss >> x; // read from string
  ofs >> x; // read from file
}
```

IO

Devices

- The interface of streams are general
- Underlying devices are abstracted away
- all streams are within a (polymorphic) hierarchy
- so we can write general code that operates on arbitrary streams if we take `ostream&` or `istream&`

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

unable to read as `int`

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

- unable to read as `int`
- found end of file character

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

- unable to read as `int`
- found end of file character
- file is corrupt

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

`fail:` unable to read as `int`

found end of file character

file is corrupt

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

`fail:` unable to read as `int`

`eof:` found end of file character

file is corrupt

IO

Error handling

```
int x;
ifstream ifs{"file"};
while (ifs >> x)
{
  // do stuff
}
```

Exits loop if:

`fail:` unable to read as `int`

`eof:` found end of file character

`bad:` file is corrupt

IO

Error handling

```
int x;
ifstream ifs{"file"};
ifs >> x;
if (ifs.fail())     // unable to read as int
// ...
else if (ifs.eof()) // reached end of file
// ...
else if (ifs.bad()) // device is corrupt
// ...
```

IO

Error flags

```
istream& operator>>(istream& is, T& t)
{
  // try to read from is
  if (/* unable to read as T */)
  {
    is.setstate(ios::failbit);
  }
  return is;
}
```

IO

Error flags

`ios::failbit`	stream operation failed
`ios::eofbit`	device has reached the end
`ios::badbit`	irrecoverable stream error
`ios::goodbit`	no error

IO

Error flags

- Multiple flags can be set at once
- except `goodbit`; it is set when no other flag is set
- This means that several errors can occur at once
- Do note that these flags are set *after* a stream operation failed
- The stream does not magically detect an error if no operation has been performed

IO

Converting from strings

```
int main(int argc, char* argv[])
{
  int x;
  istringstream iss{argv[1]};
  if (!(iss >> x))
  {
    // error

    // reset flags
    iss.clear();
  }
  // continue
}
```

```
int main(int argc, char* argv[])
{
  int x;
  try
  {
    x = stoi(argv[1]);
  }
  catch (invalid_argument& e)
  {
    // error
  }
  // continue
}
```

IO

Converting from strings

`istringstream` version

- \+ More general
- \+ Cheaper error path
- − Requires a stream
- − Must check flags

`stoi` version

- \+ No extra objects
- \+ Easier error handling
- − Expensive error path
- − Only works for `int`

Prefer the `istringstream` version because of generality, but as always; there are no universal solutions

IO

What will be printed?

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
  stringstream ss{};
  ss << "123a bc hello";
  int    n{};
  char   c{};
  string str{};

  if (ss >> n >> n >> c) cout << n << " ";
  ss.clear();
  if (ss >> c >> c) cout << c << " ";
  ss.clear();
  if (ss >> str) cout << str << " ";
}
```

1 Introduction
2 IO
3 Sequential Containers

Containers

Containers

- Sequential Containers
- Associative Containers
- Container Adaptors

Sequential Containers

Important concepts

- Memory allocations
  - Different containers have different models of allocation.
  - Calling `new` is **very** slow,
  - So the number of memory allocations is an important factor in the effectiveness of a container
- CPU caching
- Pointer invalidation

Sequential Containers

Important concepts

- Memory allocations
- CPU caching
  - Modern CPU:s perform what is known as caching.
  - Whenever the CPU fetch data from the RAM it will fetch a block of data and store that in the cache.
  - Accessing data in the CPU cache is several magnitudes faster than accessing data in the RAM.
- Pointer invalidation

Sequential Containers

Important concepts

- Memory allocations
- CPU caching
  - We always read data in blocks, so we know that the element after the data we just read is almost guaranteed to be in the cache.
  - So containers that read data in sequence is a lot faster than those that do not.
- Pointer invalidation

Sequential Containers

Important concepts

- Memory allocations
- CPU caching
  - On the flip side: if the elements of a container is spread all around the RAM, then it will be a lot slower since we almost always have to read the data from RAM rather than cache.
  - Usually we talk about the *cache locality* of a container: how much of the cache it can leverage for speedups.

Sequential Containers

Important concepts

- Memory allocations
- CPU caching
- Pointer invalidation
  - If we have pointers or references to data in containers we have to know whenever these gets *invalidated*.
  - A pointer (or reference) points to a specific address in memory,

Sequential Containers

Important concepts

- Memory allocations
- CPU caching
- Pointer invalidation
  - So if the container for some reason moves the element to another address in memory, then the pointer doesn't refer to the same element (and chances are it doesn't even point to a valid object)
  - This can prove to be a big impact in how we use containers.

Sequential Containers

What is a sequential container?

- Data stored in sequence
- Accessed with indices
- Ordered but not (necessarily) sorted

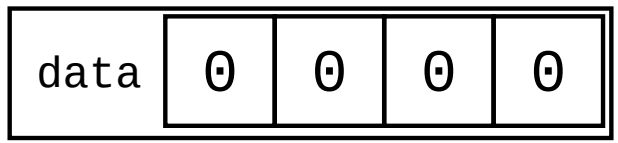
Sequential Containers

Which sequential containers are there?

- `std::array`
- `std::vector`
- `std::list`
- `std::forward_list`
- `std::deque`

Sequential Containers

`std::array`

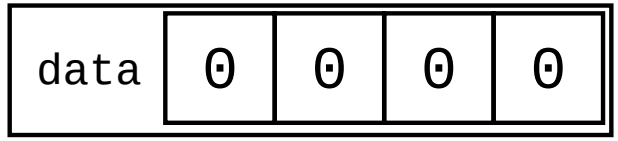


`std::array<int, 4>`

```
std::array<int, 4> array{};
```

Sequential Containers

`std::array`

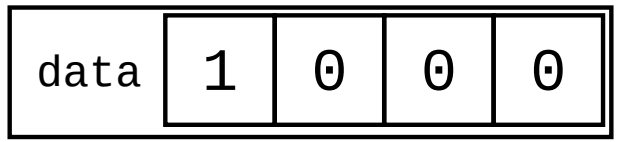


`std::array<int, 4>`

```
array[0] = 1;
```

Sequential Containers

`std::array`

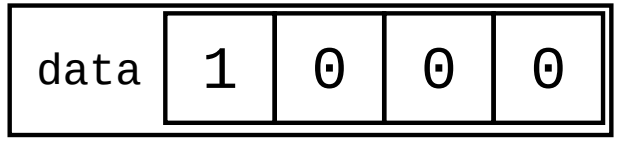


`std::array<int, 4>`

```
array[0] = 1;
```

Sequential Containers

`std::array`

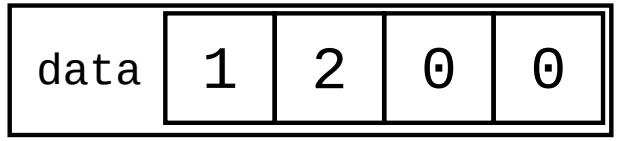


`std::array<int, 4>`

```
array[1] = 2;
```

Sequential Containers

`std::array`

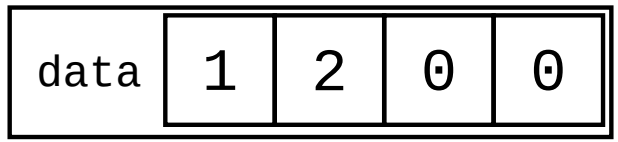


`std::array<int, 4>`

```
array[1] = 2;
```

Sequential Containers

`std::array`

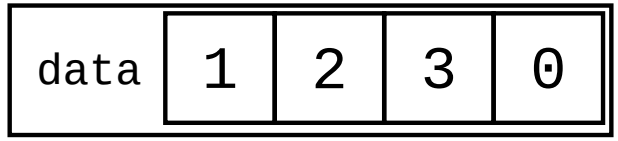


`std::array<int, 4>`

```
array[2] = 3;
```

Sequential Containers

`std::array`

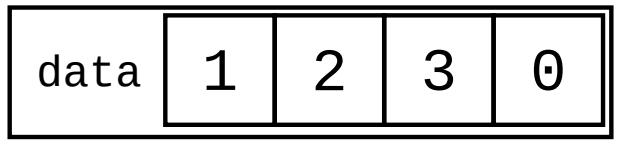


`std::array<int, 4>`

```
array[2] = 3;
```

Sequential Containers

`std::array`

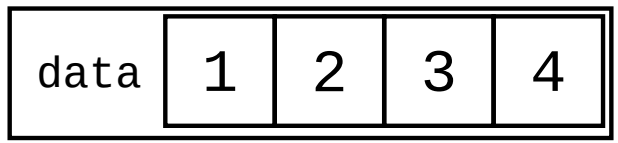


`std::array<int, 4>`

```
array[3] = 4;
```

Sequential Containers

`std::array`



`std::array<int, 4>`

```
array[3] = 4;
```

Sequential Containers

`std::array`

- insertion: *not applicable*
- deletion: *not applicable*
- lookup: $O(1)$

Sequential Containers

`std::array`

- \+ No memory allocations
- \+ Data never move in memory
- – Fixed size
- – Size must be known during compilation

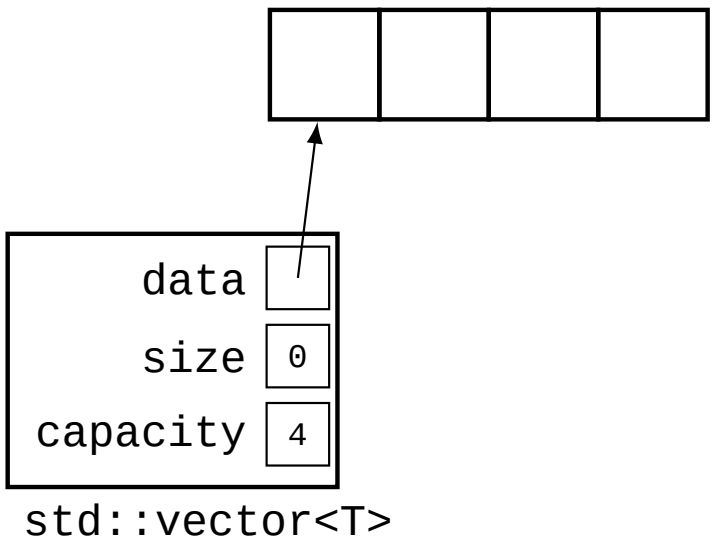
Sequential Containers

Example

```
#include <array>
// ...
int main()
{
  std::array<int, 5> data{};
  for (unsigned i{}; i < data.size(); ++i)
  {
    cin >> data.at(i);
  }
  for (auto&& i : data)
  {
    cout << i << endl;
  }
}
```

Sequential Containers

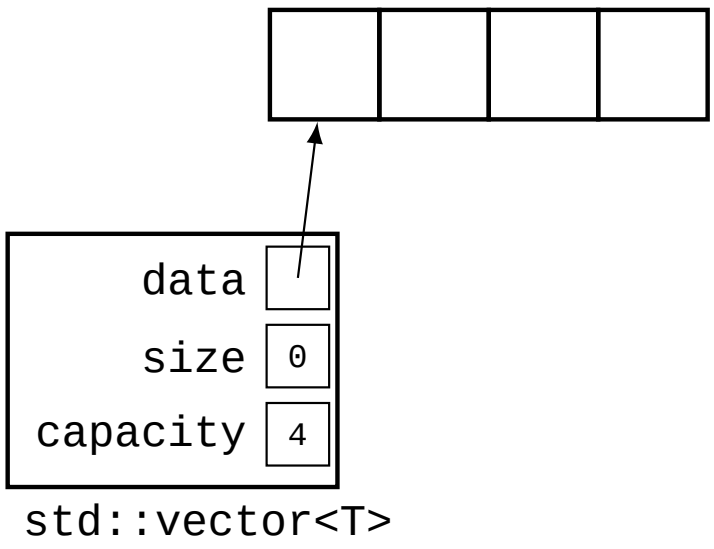
`std::vector`




```
std::vector<int> vector{};
```

Sequential Containers

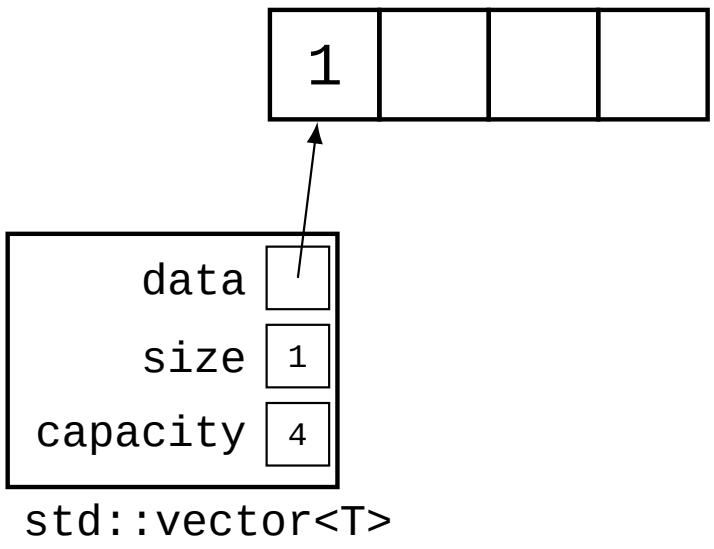
`std::vector`




```
vector.push_back(1);
```

Sequential Containers

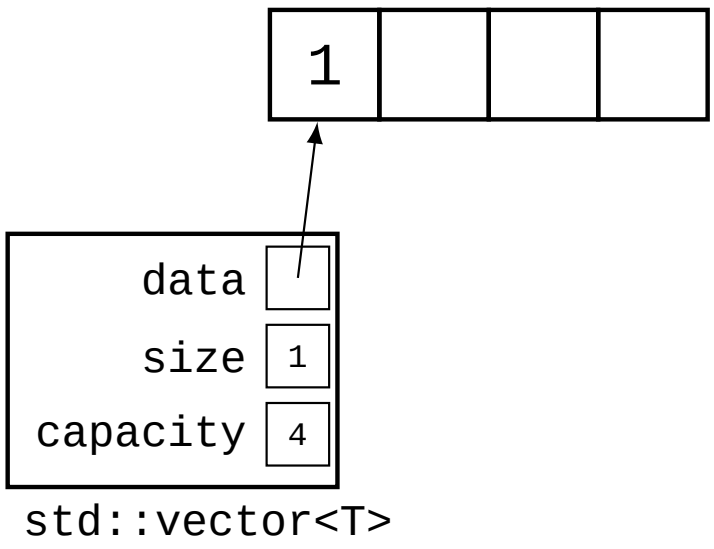
`std::vector`




```
vector.push_back(1);
```

Sequential Containers

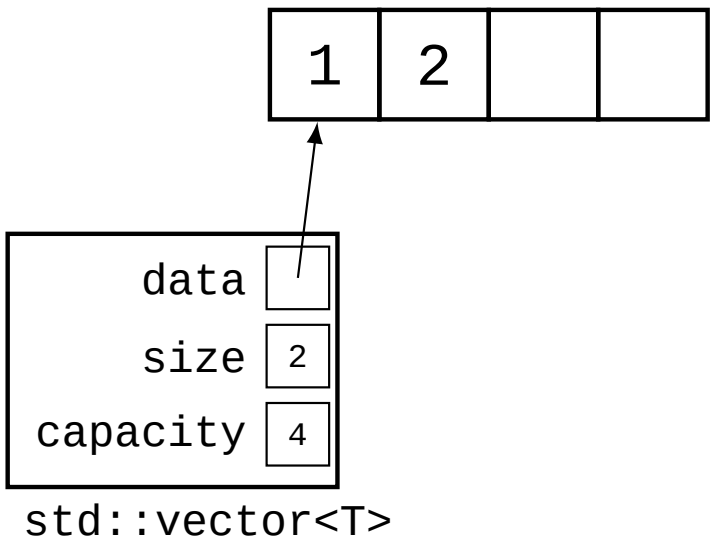
`std::vector`




```
vector.push_back(2);
```

Sequential Containers

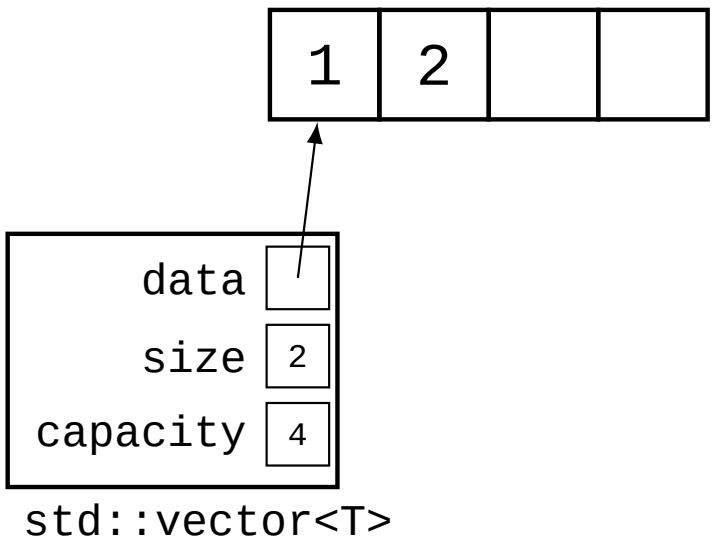
`std::vector`




```
vector.push_back(2);
```

Sequential Containers

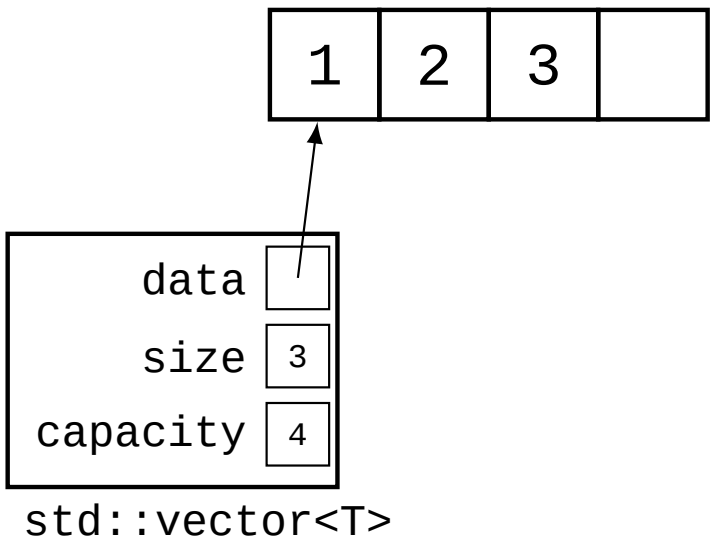
`std::vector`




```
vector.push_back(3);
```

Sequential Containers

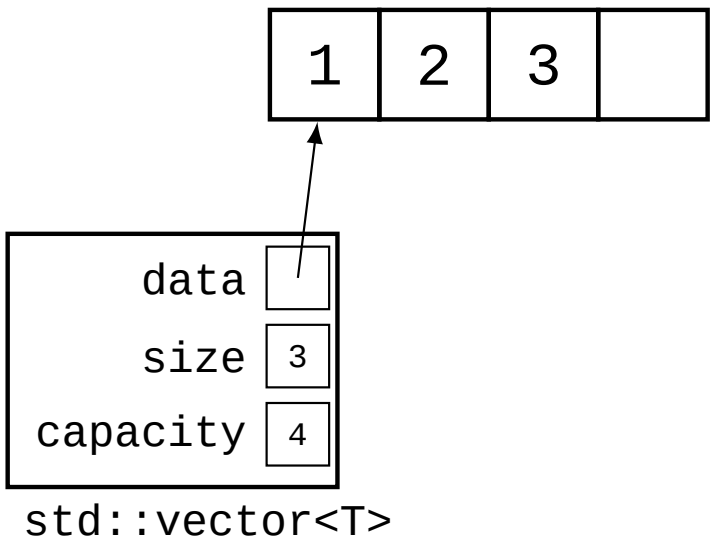
`std::vector`




```
vector.push_back(3);
```

Sequential Containers

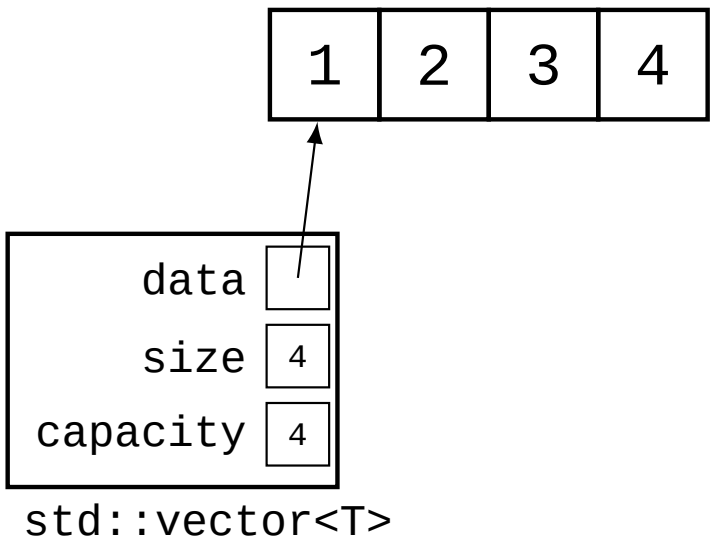
`std::vector`




```
vector.push_back(4);
```

Sequential Containers

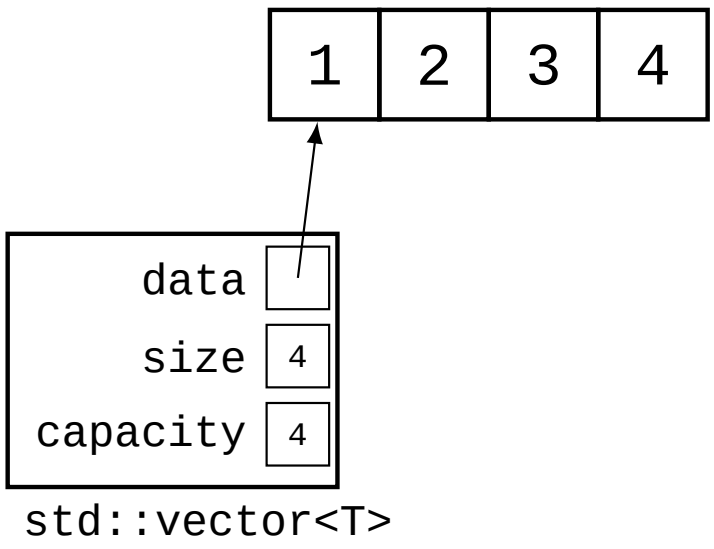
`std::vector`




```
vector.push_back(4);
```

Sequential Containers

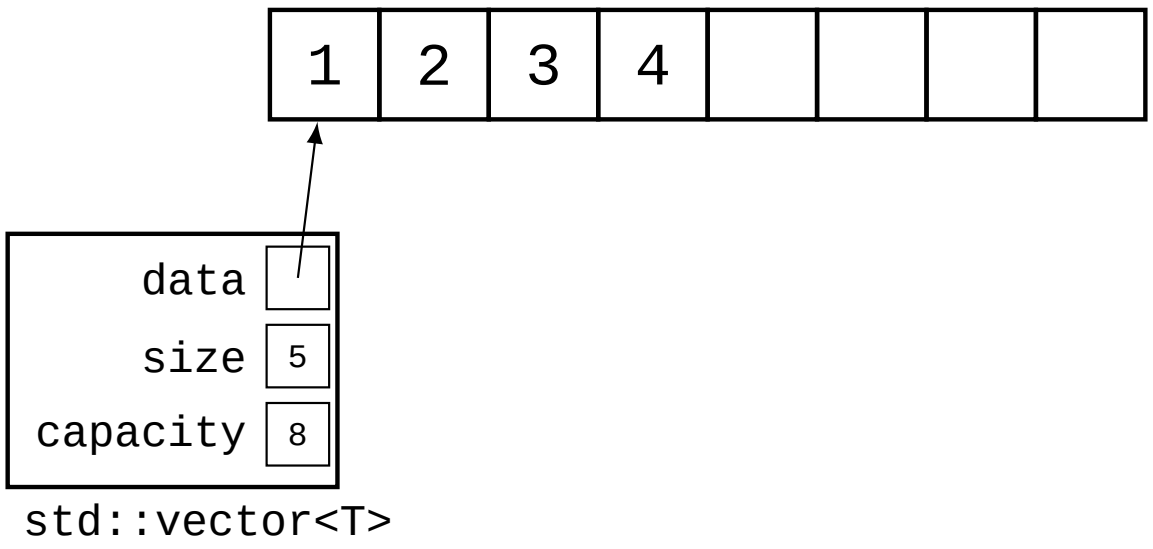
`std::vector`




```
vector.push_back(5);
```

Sequential Containers

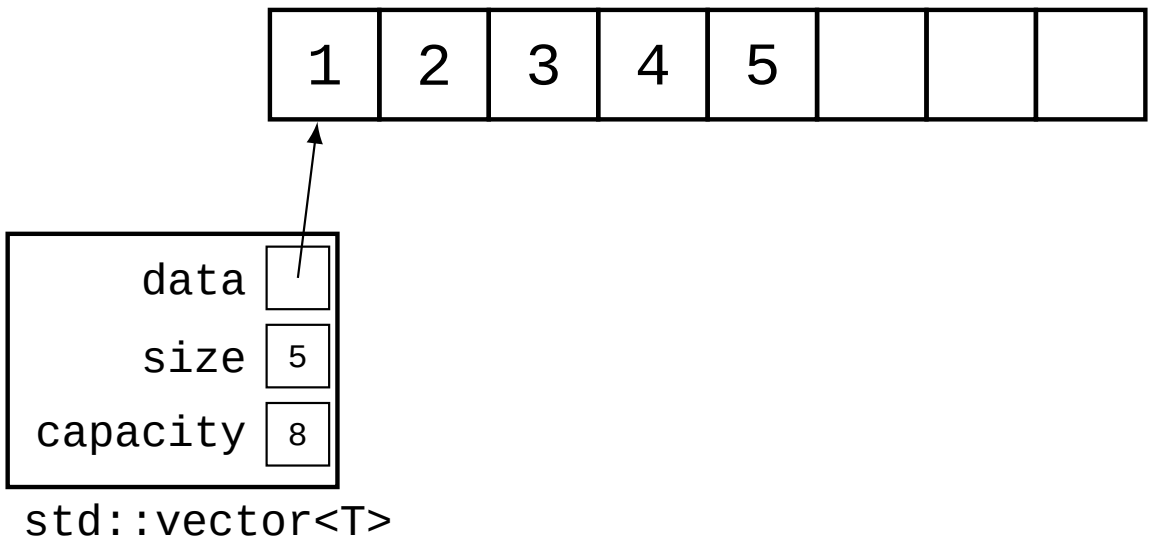
`std::vector`




```
vector.push_back(5);
```

Sequential Containers

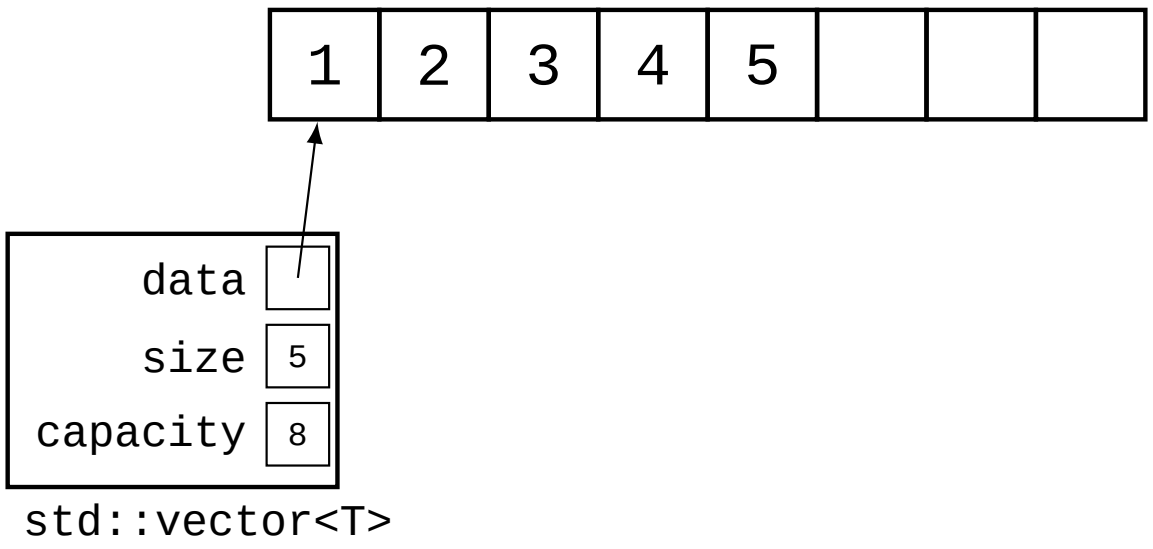
`std::vector`




```
vector.push_back(5);
```

Sequential Containers

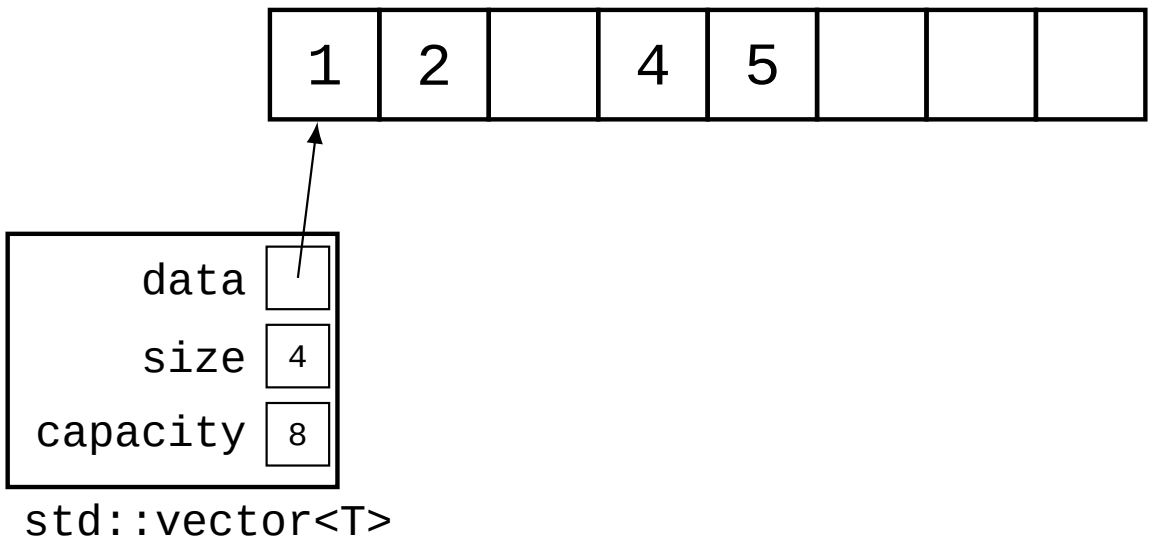
`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

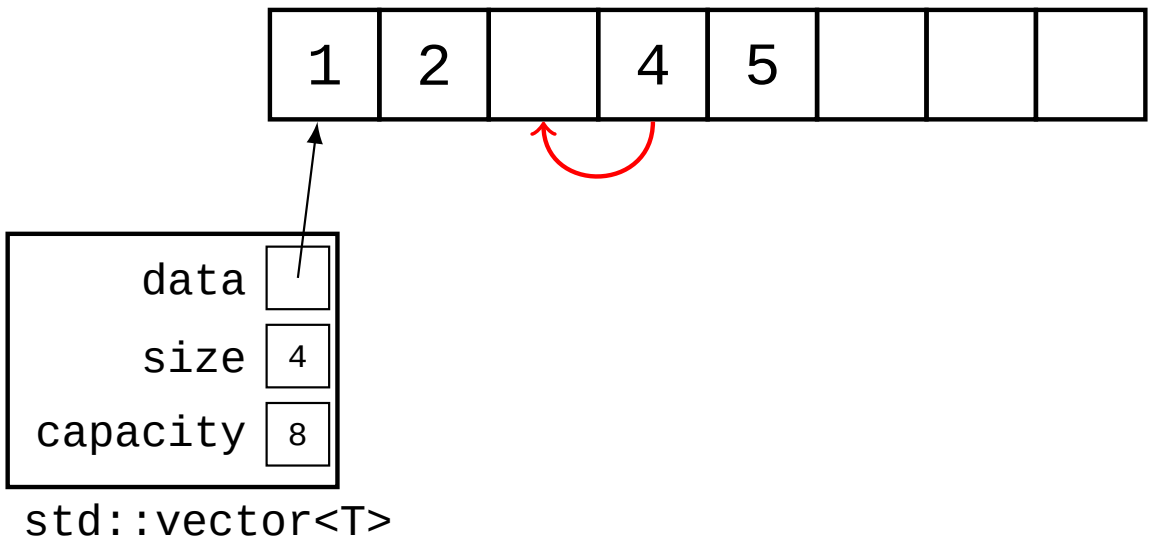
`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

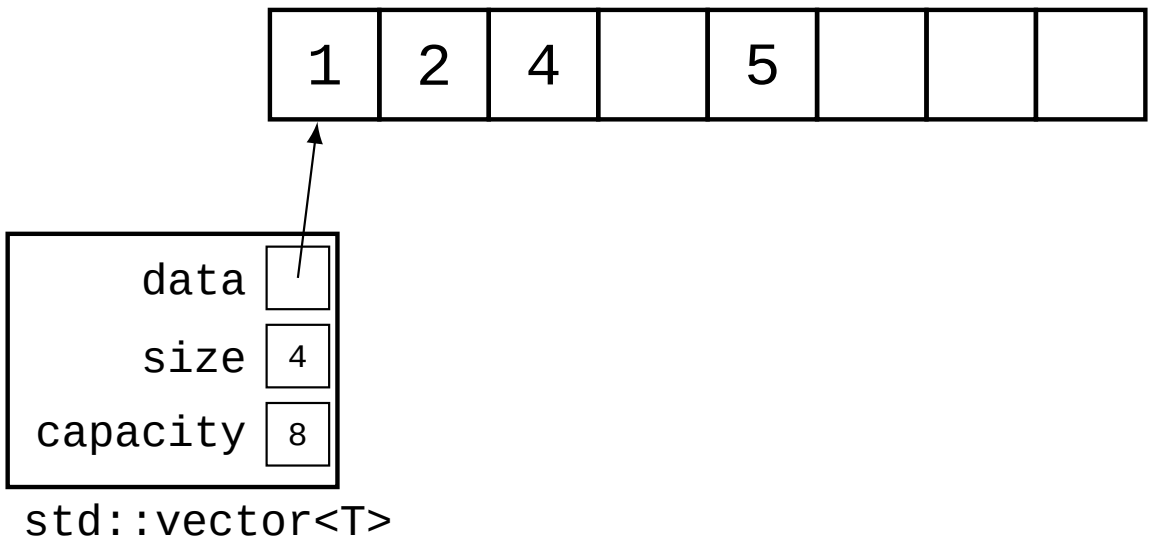
`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

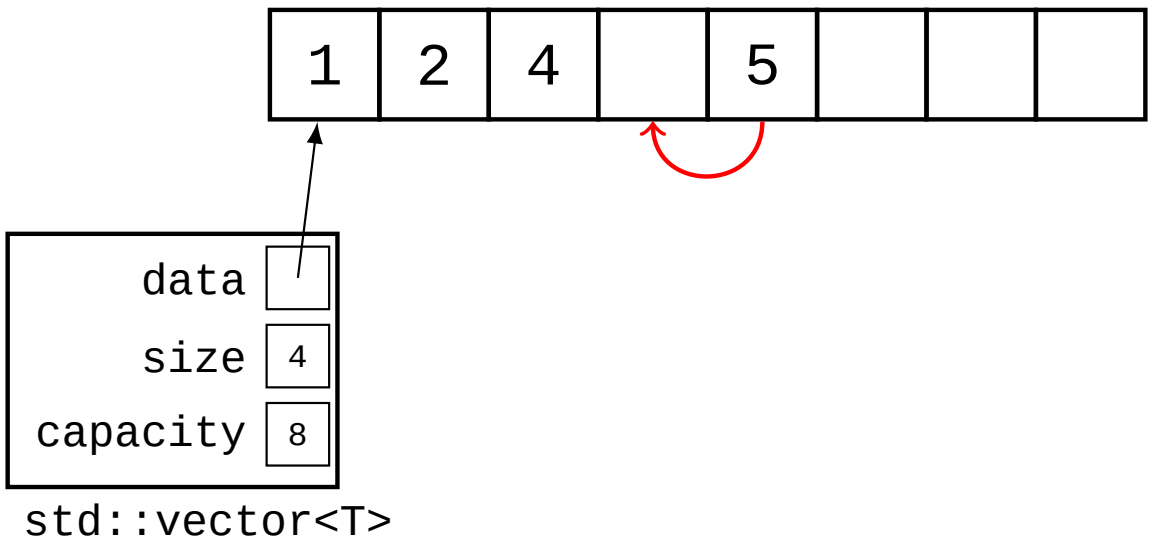
`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

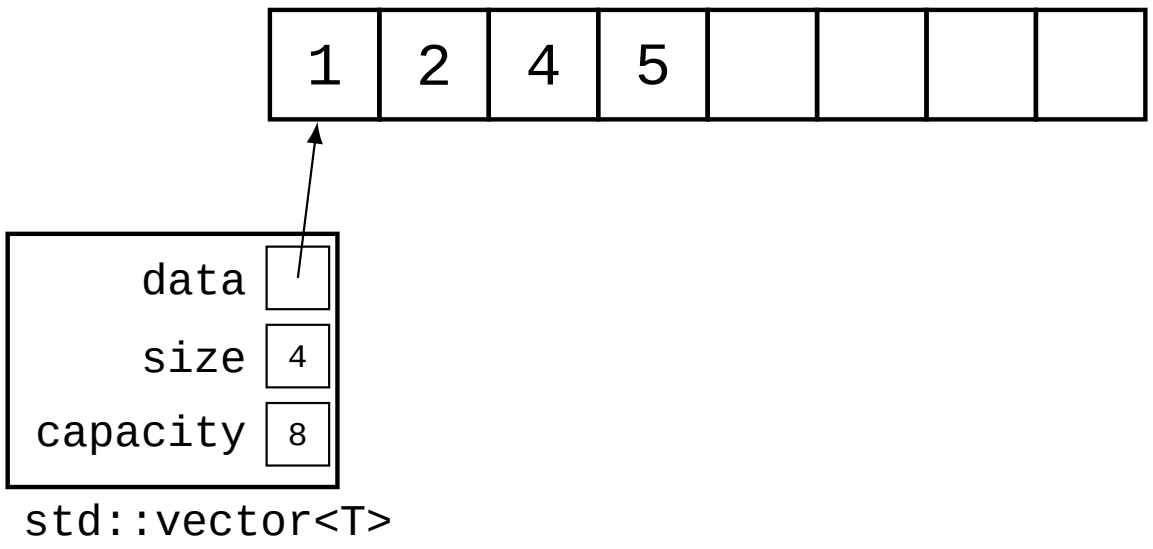
`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

`std::vector`




```
vector.erase(vector.begin() + 2);
```

Sequential Containers

`std::vector`

- insertion:
  - at end: $O(1)$
  - otherwise: $O(n)$
- deletion:
  - last element: $O(1)$
  - otherwise: $O(n)$
- lookup: $O(1)$

Sequential Containers

`std::vector`

+ Data is sequential in memory
+ Dynamic size
– Entire data range can move in memory
– Dynamic allocations are slow

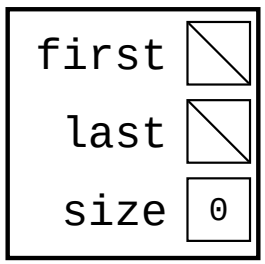
Sequential Containers

Example

```
#include <vector>
// ...
int main()
{
  std::vector<int> data{};
  int x{};
  while (cin >> x)
  {
    data.push_back(x);
  }
  for (auto&& i : data)
    cout << i << endl;
}
```

Sequential Containers

`std::list`

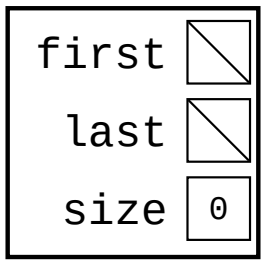


`std::list<T>`

```
std::list<int> list{};
```

Sequential Containers

`std::list`

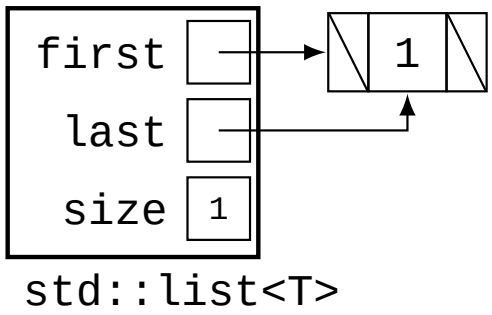


`std::list<T>`

```
list.push_back(1);
```

Sequential Containers

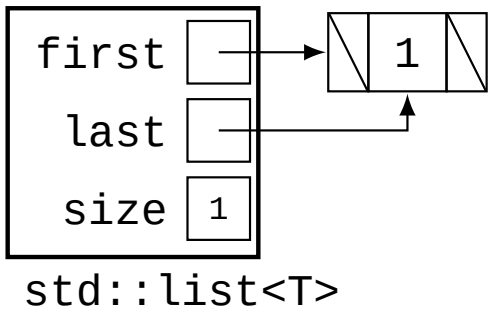
`std::list`




```
list.push_back(1);
```

Sequential Containers

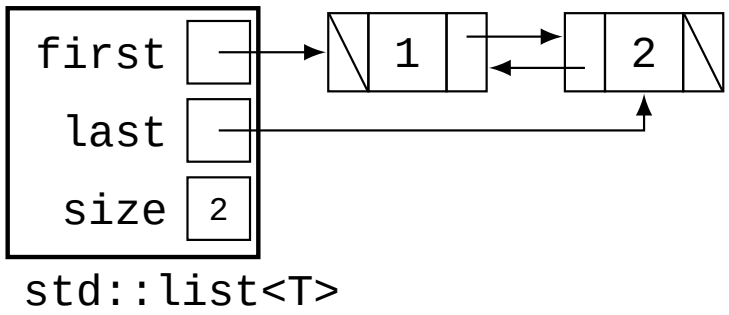
`std::list`




```
list.push_back(2);
```

Sequential Containers

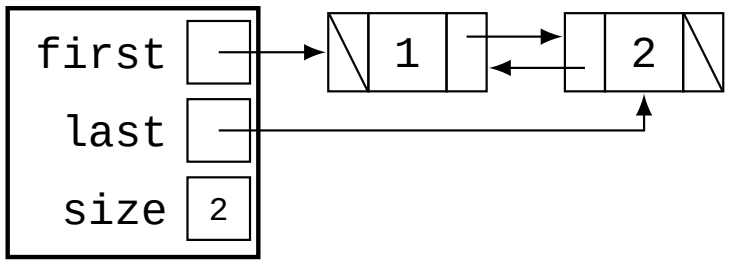
`std::list`




```
list.push_back(2);
```

Sequential Containers

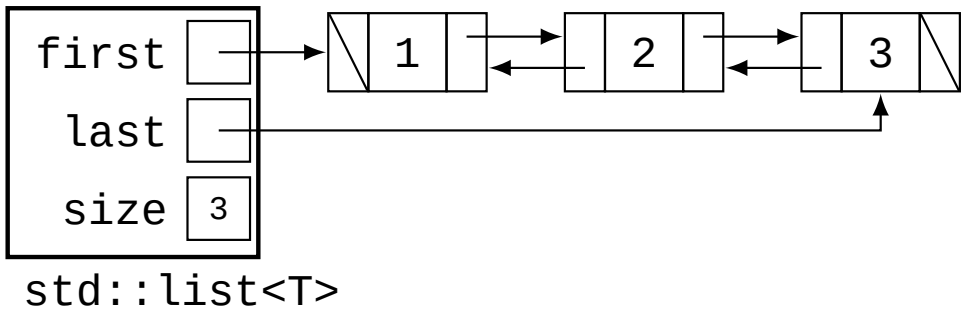
`std::list`




```
list.push_back(3);
```

Sequential Containers

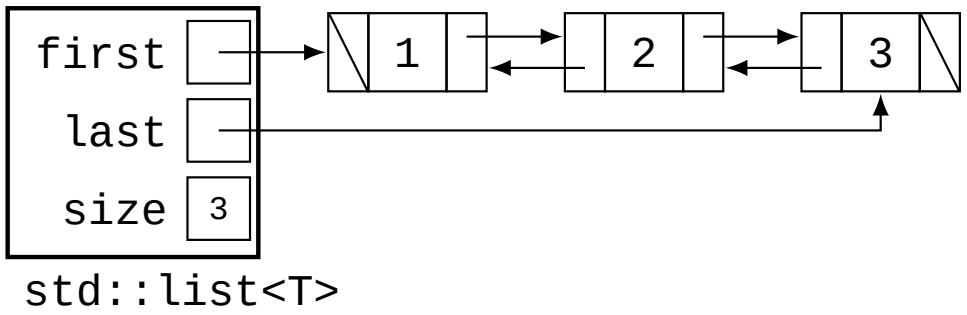
`std::list`




```
list.push_back(3);
```

Sequential Containers

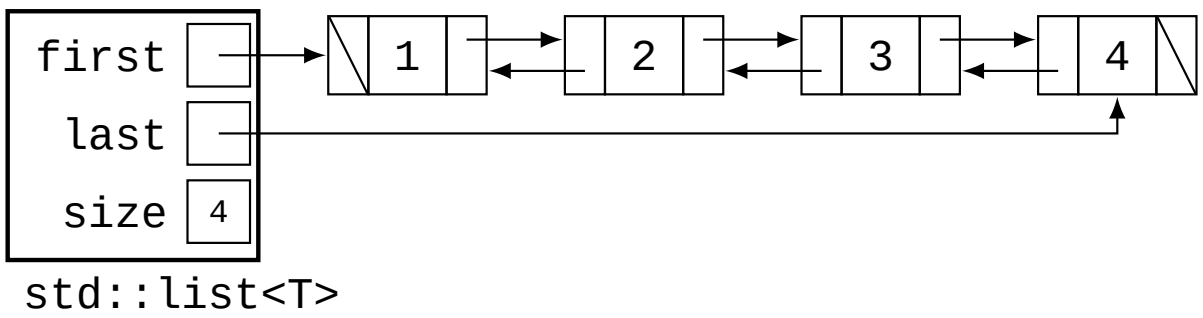
`std::list`




```
list.push_back(4);
```

Sequential Containers

`std::list`




```
list.push_back(4);
```

Sequential Containers

`std::list`

- insertion:
  - at the ends: $O(1)$
  - otherwise: $O(n)$
- deletion:
  - first or last element: $O(1)$
  - otherwise: $O(n)$
- lookup: $O(n)$

Sequential Containers

`std::list`

- \+ elements never move in memory
- \+ Operations around a specific element is $O(1)$
- − Many allocations (one for each element)
- − Linear lookup

Sequential Containers

`std::list`

- \+ elements never move in memory
- \+ Operations around a specific element is $O(1)$
- − Many allocations (one for each element)
- − Linear lookup
- − Makes the CPU cache very sad :(

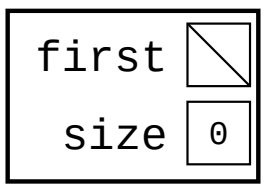
Sequential Containers

Example

```
#include <list>
#include <vector>
// ...
int main()
{
  std::list<int> data{};
  std::vector<int*> order{};
  int x;
  while (cin >> x)
  {
    data.push_back(x);
    order.push_back(&data.back());
  }
  data.sort();
  int i{0};
  for (auto&& val : data)
  {
    cout << val << ", " << *order[i++] << endl;
  }
}
```

Sequential Containers

`std::forward_list`

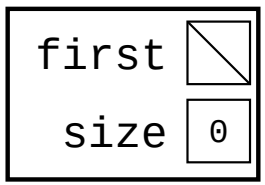


`std::forward_list<T>`

```
std::forward_list<int> list{};
```

Sequential Containers

`std::forward_list`

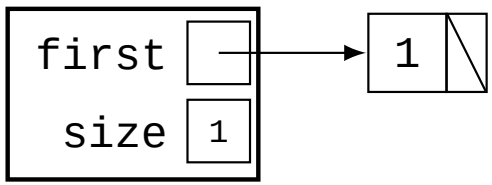


`std::forward_list<T>`

```
list.push_front(1);
```

Sequential Containers

`std::forward_list`

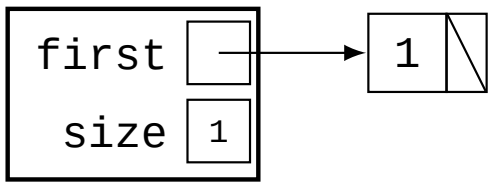


`std::forward_list<T>`

```
list.push_front(1);
```

Sequential Containers

`std::forward_list`

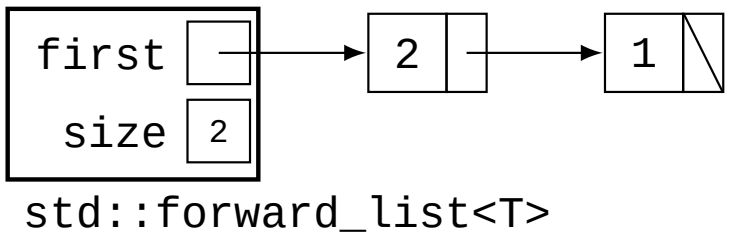


`std::forward_list<T>`

```
list.push_front(2);
```

Sequential Containers

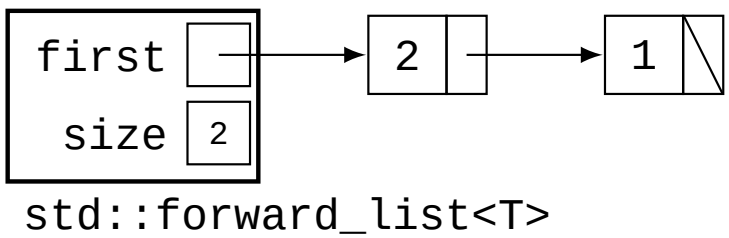
`std::forward_list`




```
list.push_front(2);
```

Sequential Containers

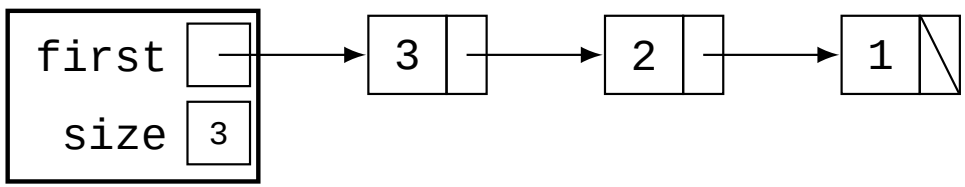
`std::forward_list`




```
list.push_front(3);
```

Sequential Containers

`std::forward_list`

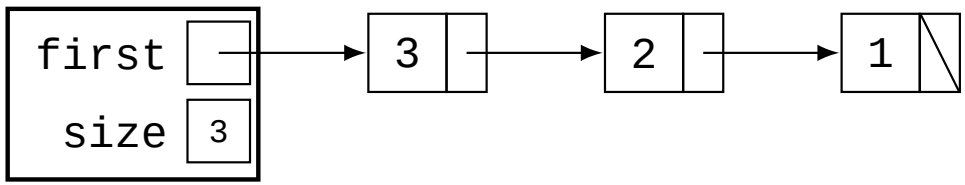


`std::forward_list<T>`

```
list.push_front(3);
```

Sequential Containers

`std::forward_list`

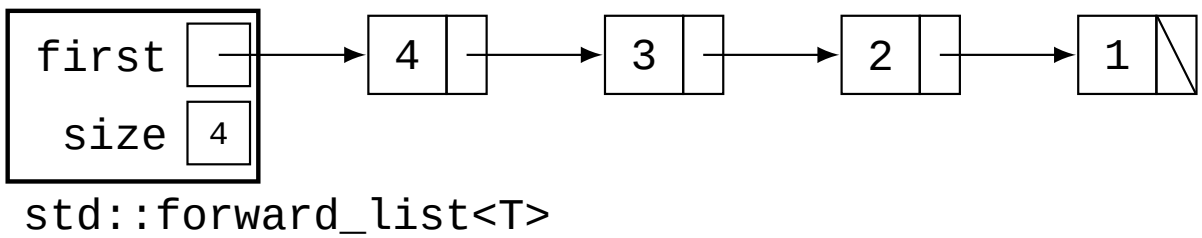


`std::forward_list<T>`

```
list.push_front(4);
```

Sequential Containers

`std::forward_list`




```
list.push_front(4);
```

Sequential Containers

`std::forward_list`

- insertion:
  - in beginning: $O(1)$
  - otherwise: $O(n)$
- deletion:
  - first element: $O(1)$
  - otherwise: $O(n)$
- lookup: $O(n)$

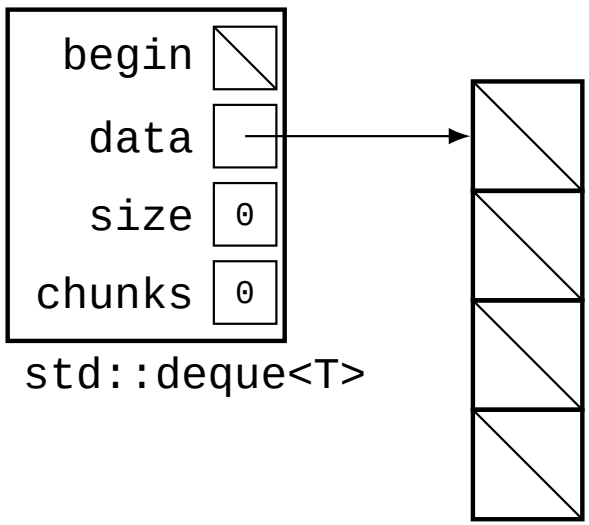
Sequential Containers

`std::forward_list`

- \+ Less memory per element compared to `std::list`
- − No $O(1)$ operations on last element
- − Unable to go backwards

Sequential Containers

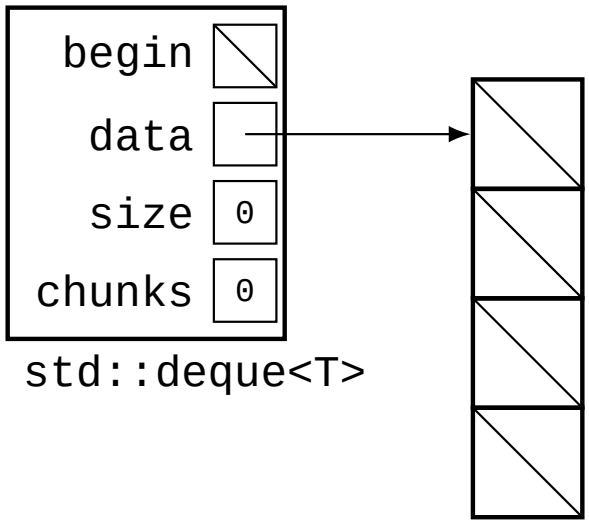
`std::deque`




```
std::deque<int> deque{};
```

Sequential Containers

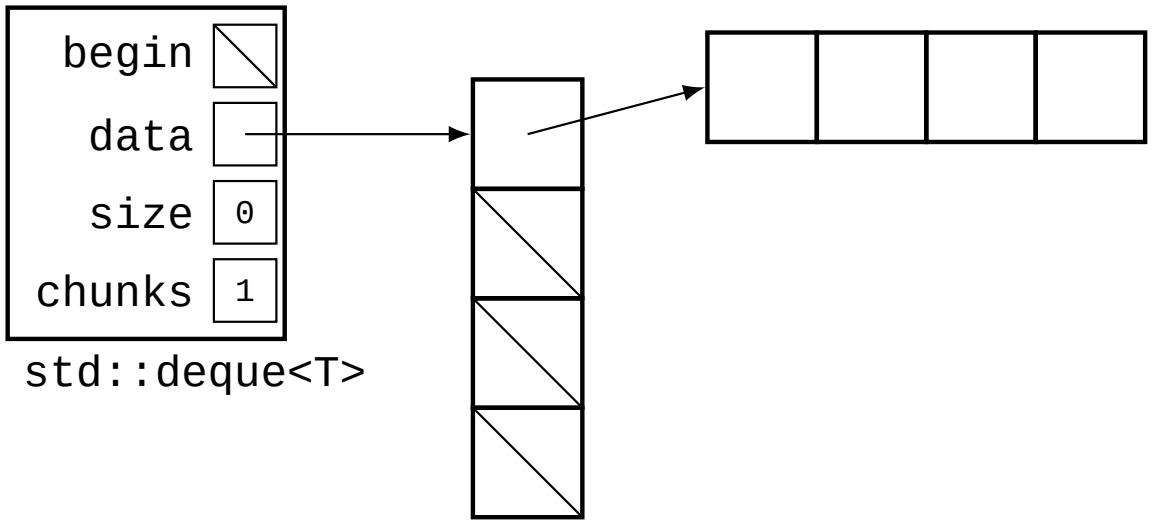
`std::deque`




```
deque.push_back(1);
```

Sequential Containers

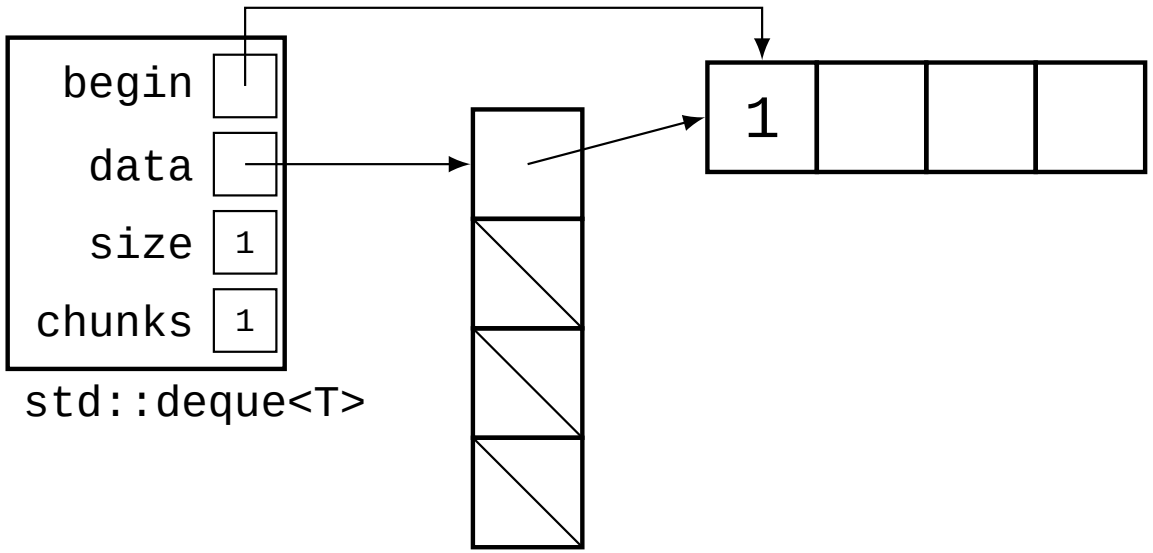
`std::deque`




```
deque.push_back(1);
```

Sequential Containers

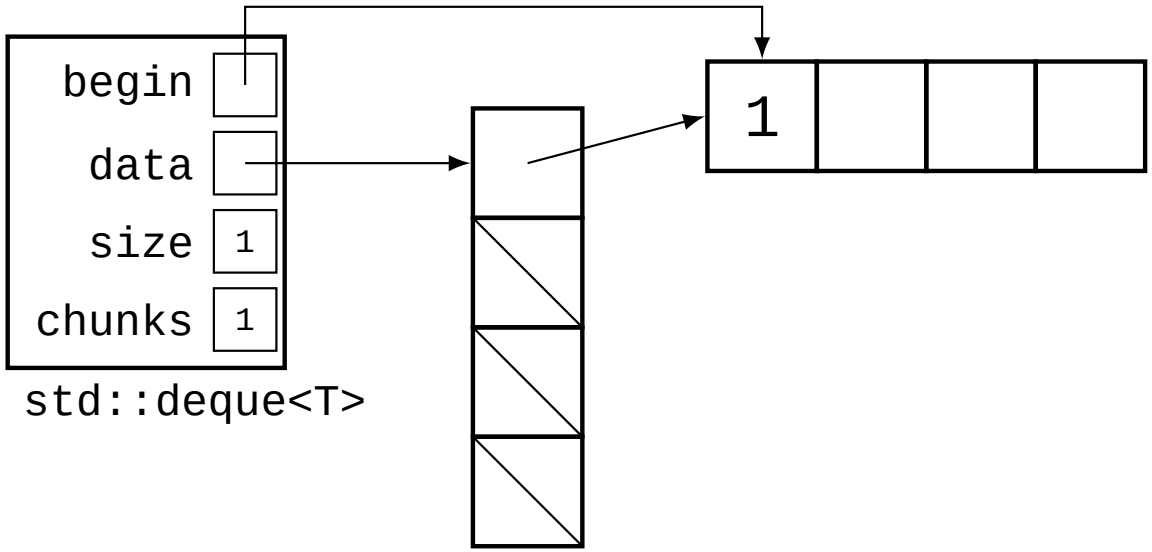
`std::deque`




```
deque.push_back(1);
```

Sequential Containers

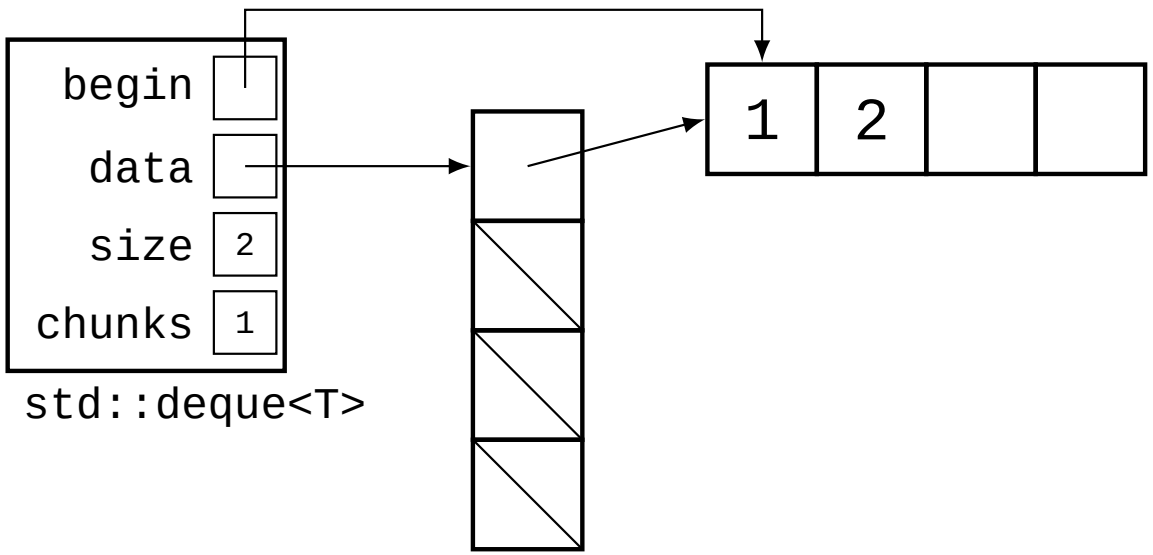
`std::deque`




```
deque.push_back(2);
```

Sequential Containers

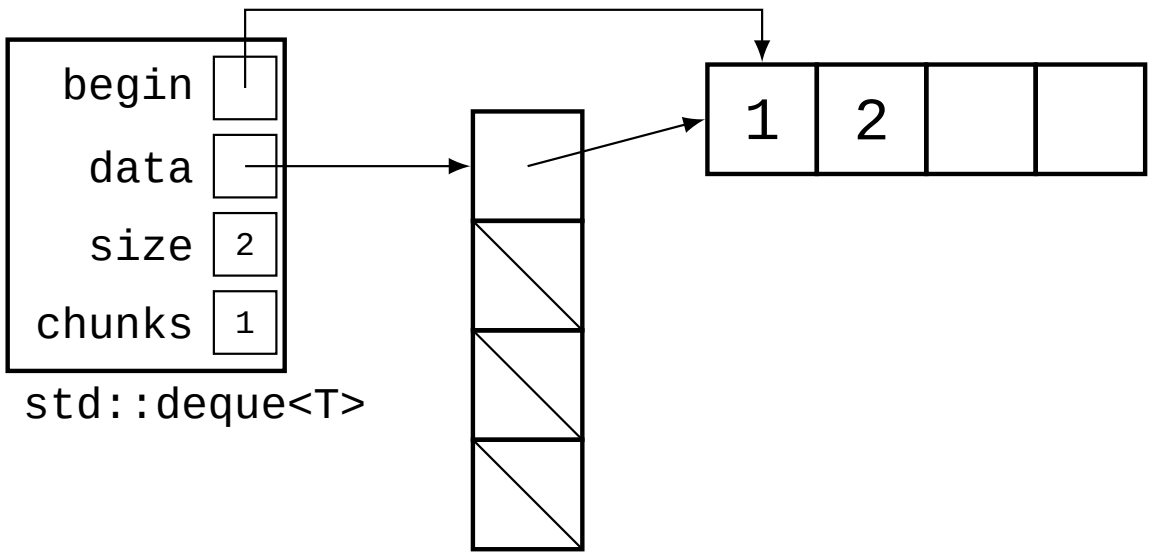
`std::deque`




```
deque.push_back(2);
```

Sequential Containers

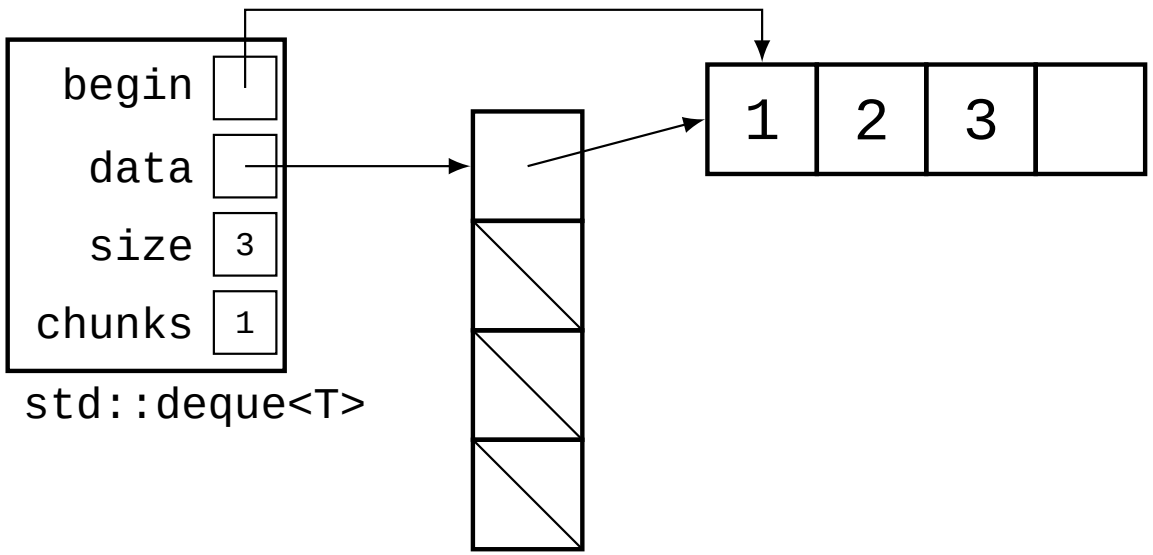
`std::deque`




```
deque.push_back(3);
```

Sequential Containers

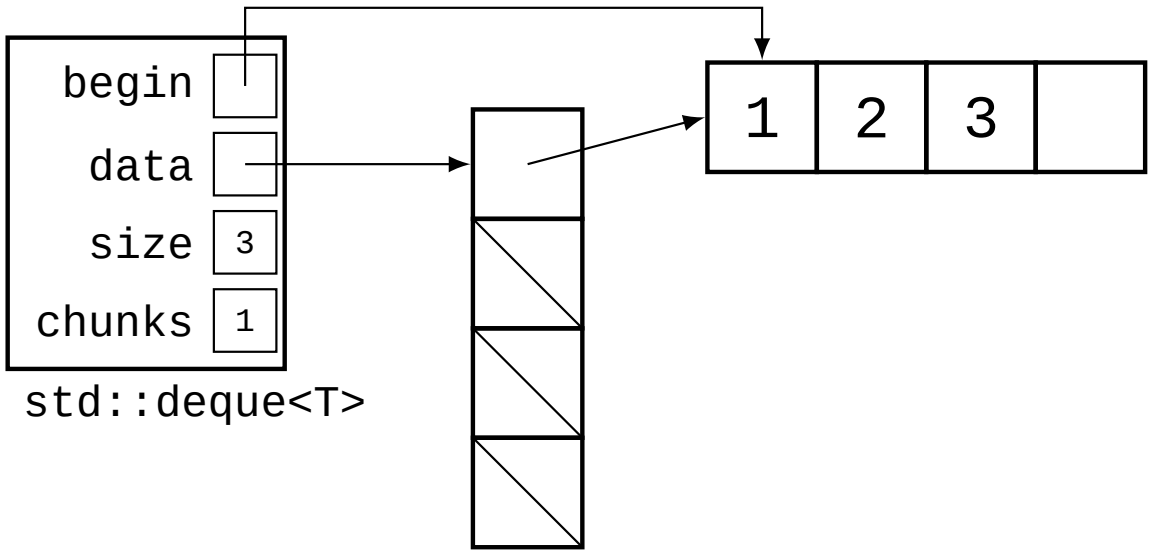
`std::deque`




```
deque.push_back(3);
```

Sequential Containers

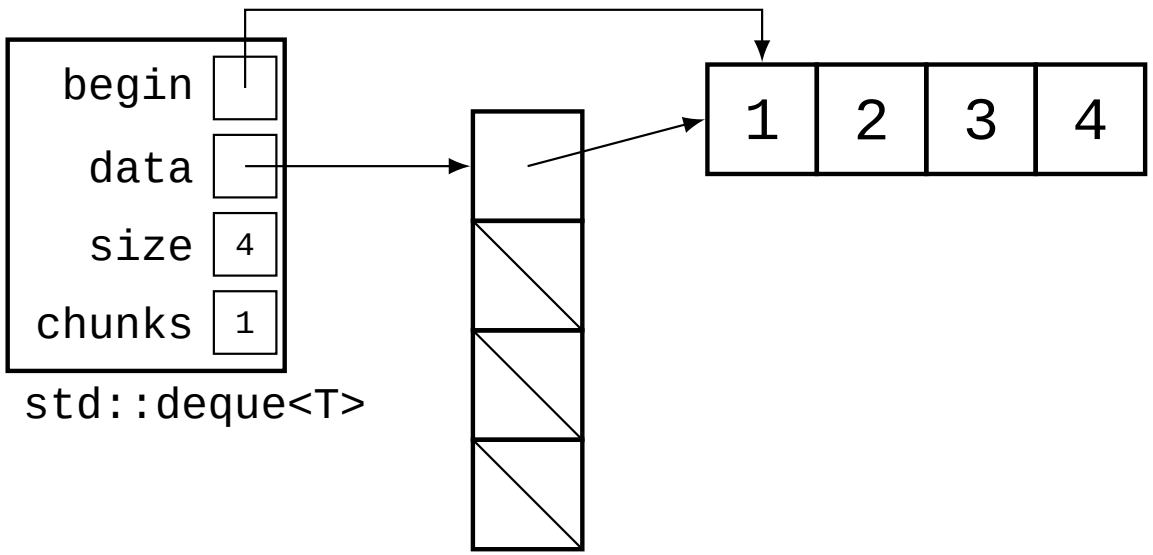
`std::deque`




```
deque.push_back(4);
```

Sequential Containers

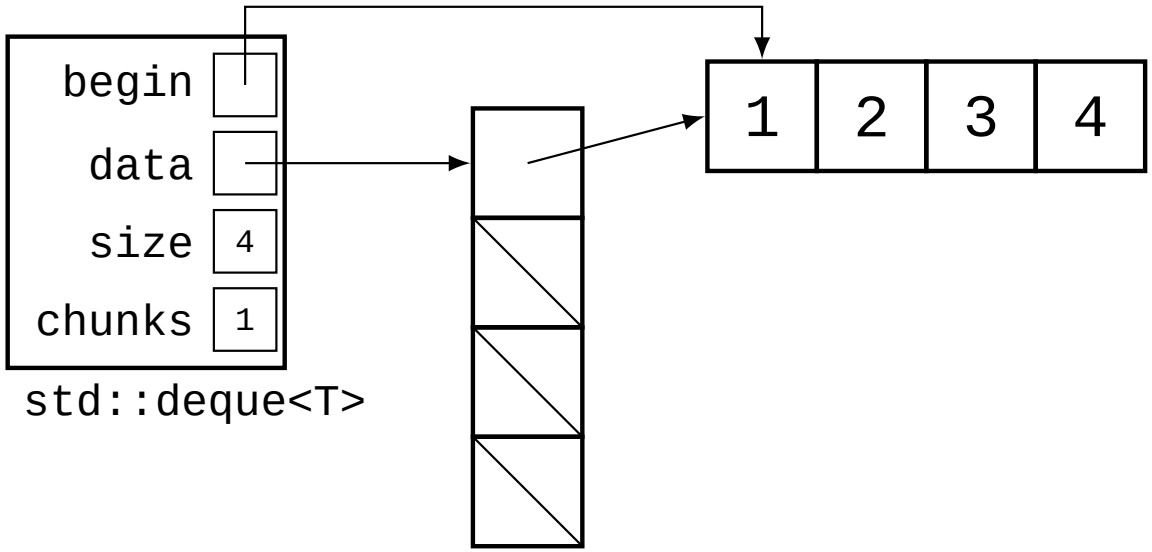
`std::deque`




```
deque.push_back(4);
```

Sequential Containers

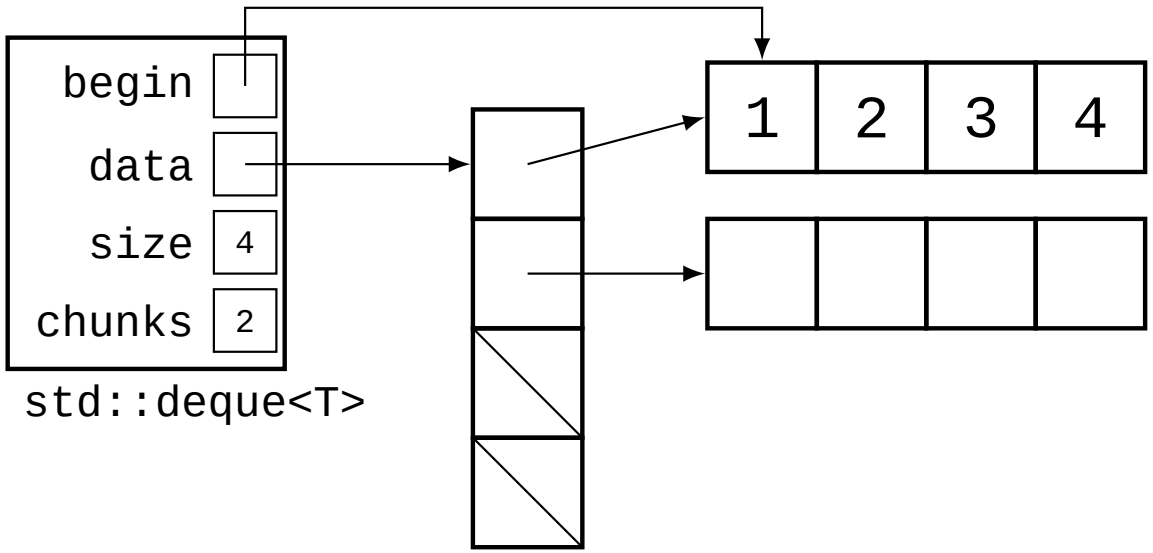
`std::deque`




```
deque.push_back(5);
```

Sequential Containers

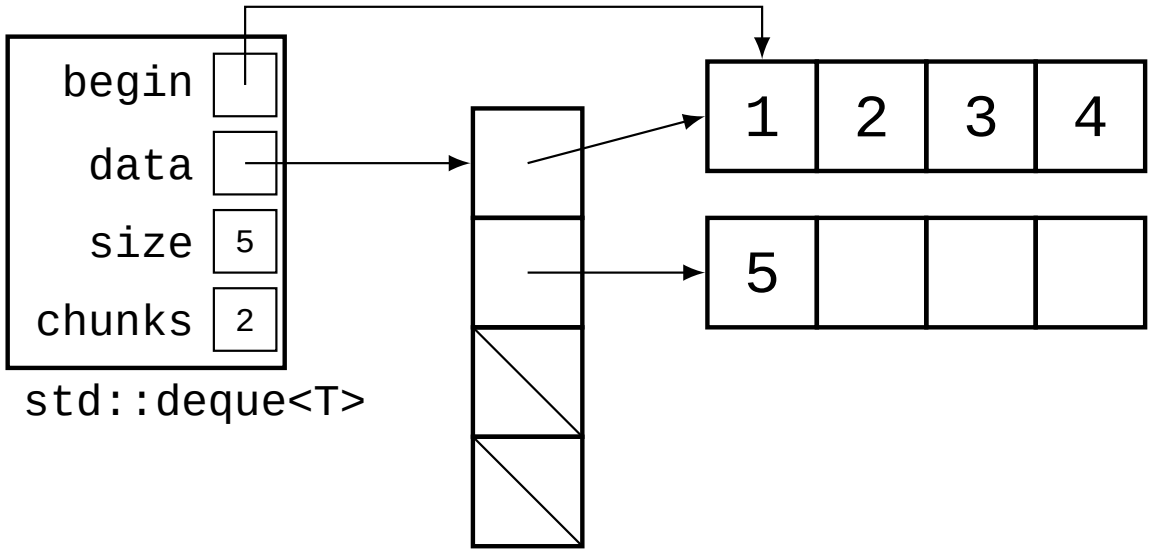
`std::deque`




```
deque.push_back(5);
```

Sequential Containers

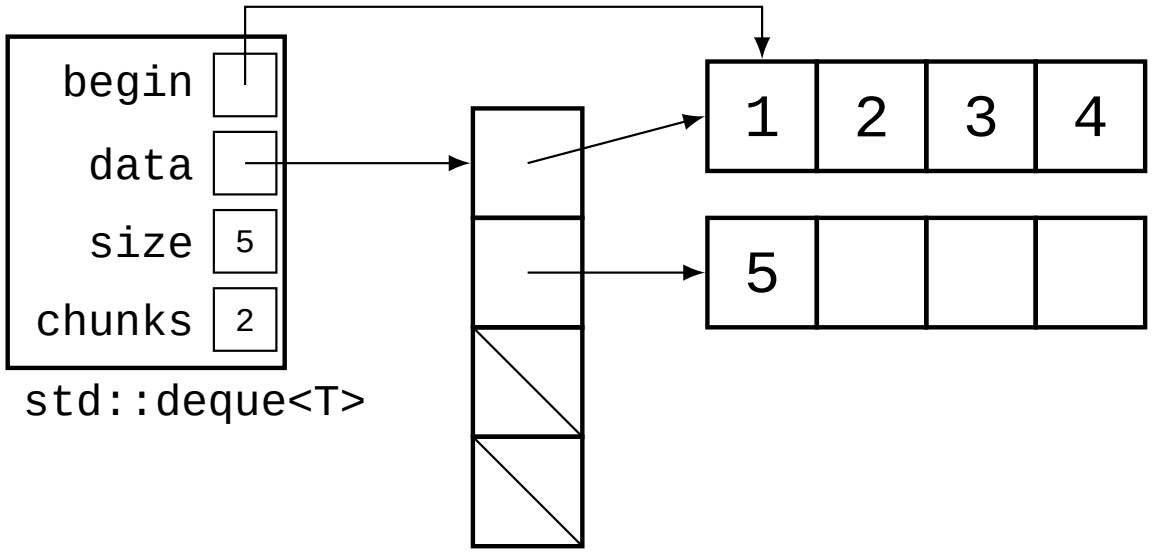
`std::deque`




```
deque.push_back(5);
```

Sequential Containers

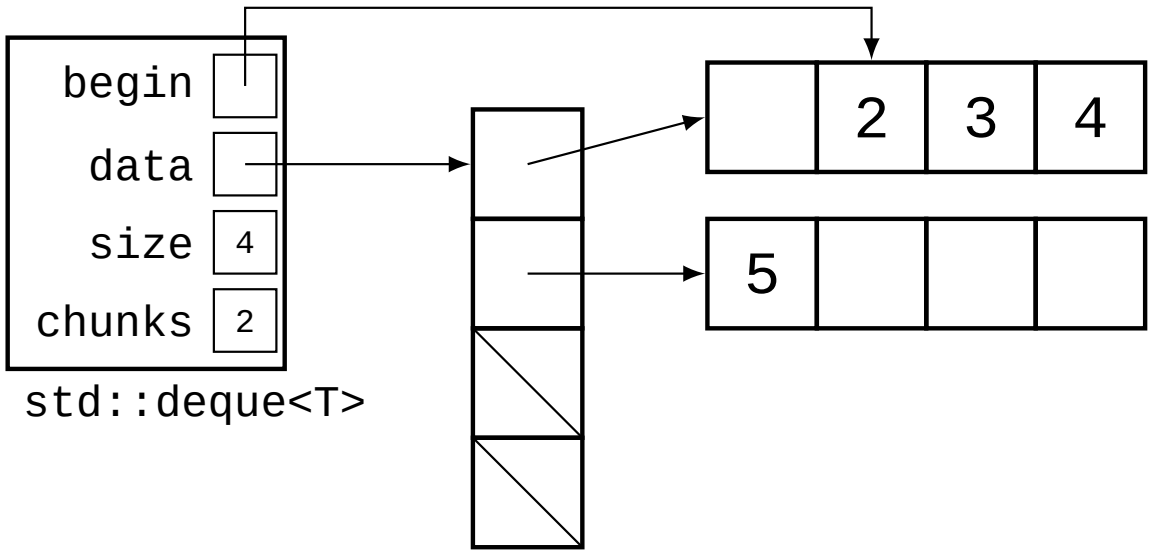
`std::deque`




```
deque.pop_front();
```

Sequential Containers

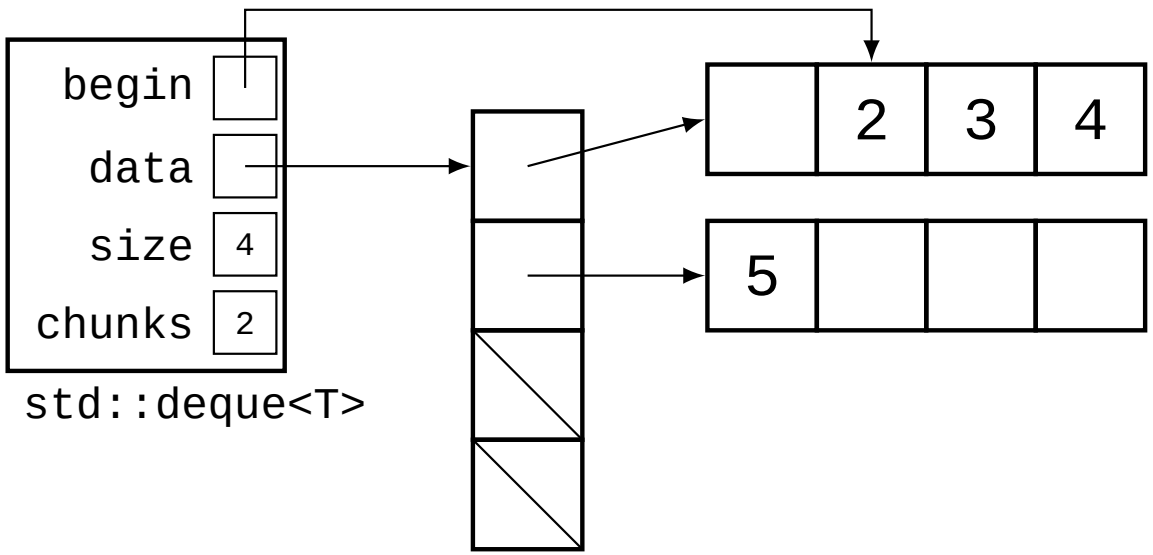
`std::deque`




```
deque.pop_front();
```

Sequential Containers

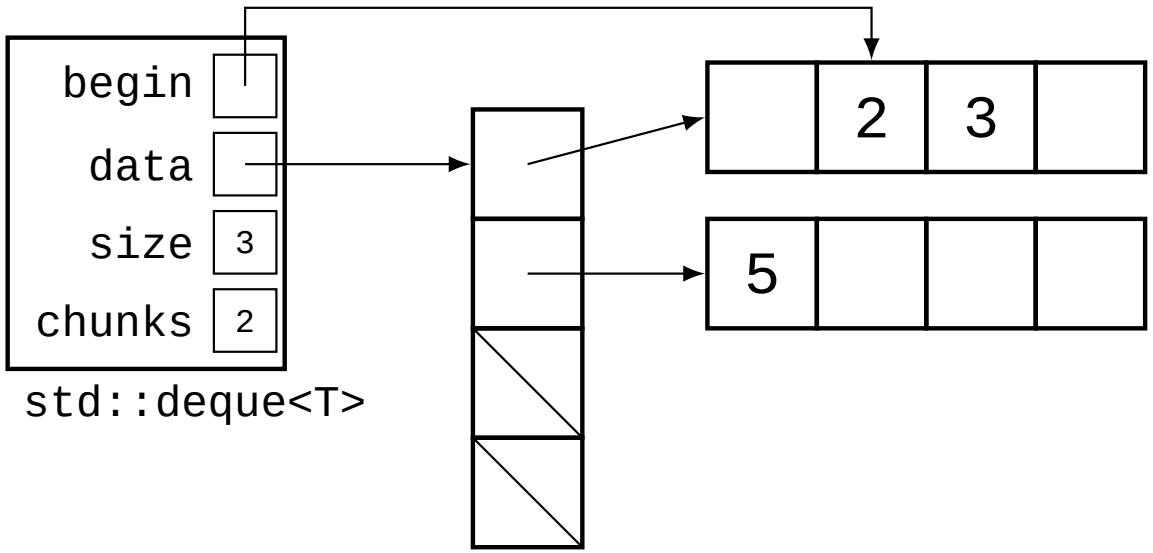
`std::deque`




```
deque.erase(deque.begin() + 2);
```

Sequential Containers

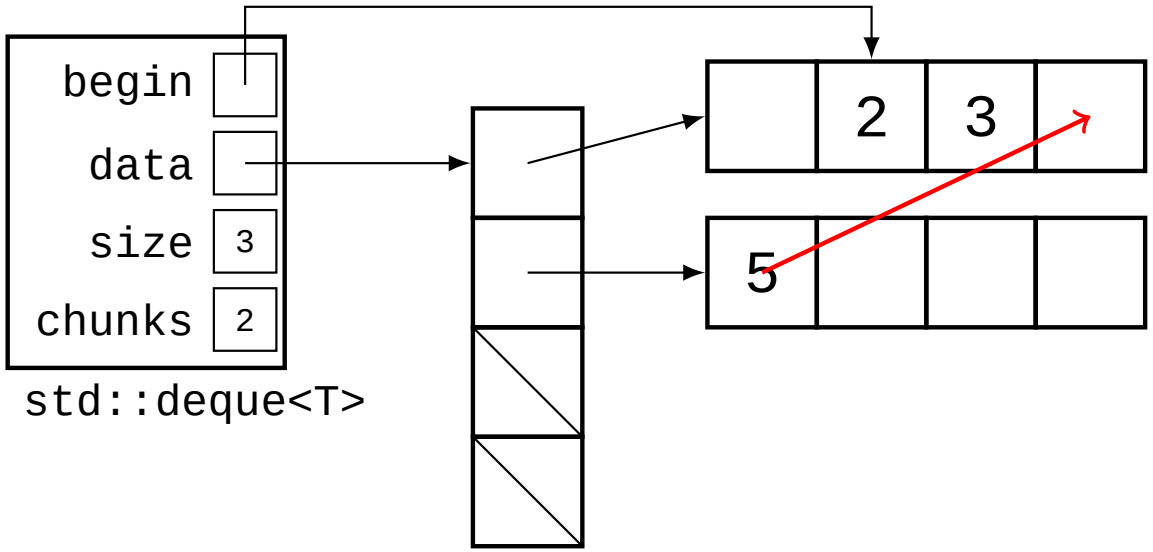
`std::deque`




```
deque.erase(deque.begin() + 2);
```

Sequential Containers

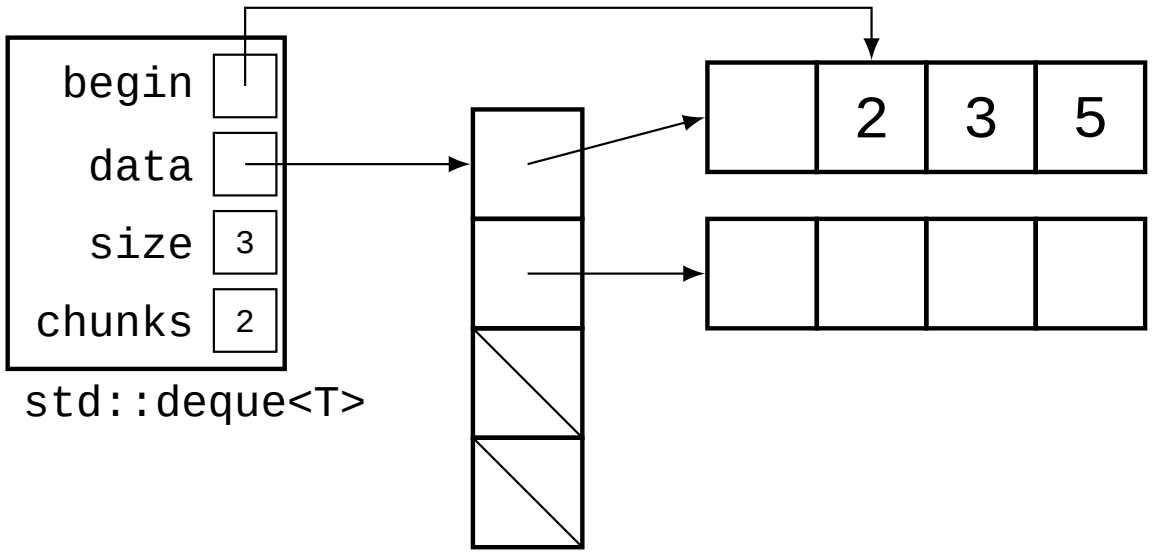
`std::deque`




```
deque.erase(deque.begin() + 2);
```

Sequential Containers

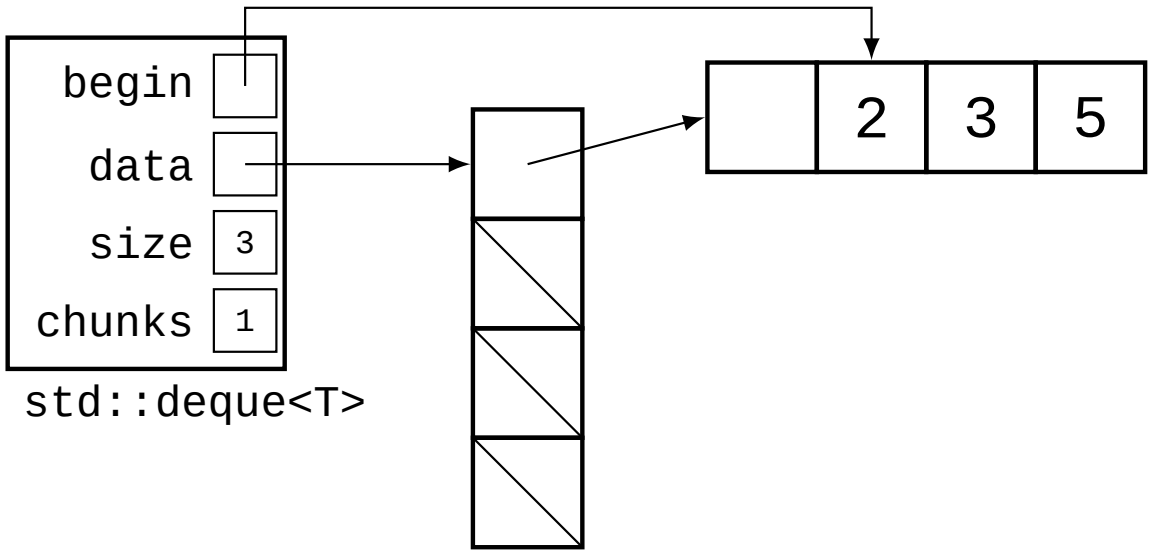
`std::deque`




```
deque.erase(deque.begin() + 2);
```

Sequential Containers

`std::deque`




```
deque.erase(deque.begin() + 2);
```

Sequential Containers

`std::deque`

- insertion:
  - at ends: $O(1)$
  - otherwise: $O(n)$
- deletion:
  - at ends: $O(1)$
  - otherwise: $O(n)$
- lookup: $O(1)$

Sequential Containers

`std::deque`

- \+ Elements rarely move in memory
- \+ Fast operations at ends
- \+ More cache friendly than `std::list`
- – Not contigous in memory
- – Additional complexity gives slighly worse performance

Sequential Containers

Uses

- Great for queues and stacks!
- Will automatically shrink the container so use it when there are a lot of insertions and deletions

www.liu.se

LINKÖPING UNIVERSITY