容器，ranges与算法
Container, Ranges and Algorithm

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

Overview

- In the following two lectures, we will cover containers, algorithms (so-called “STL”, which is a legacy name) and ranges.
  - For containers (including iterators), you’ve learnt basics of them before.
    - We will not teach advanced memory-related techniques here, e.g. `std::pmr`, allocator. We’ll assume memory is allocated by `new`.
  - For ranges, we’ll mainly teach their usage.
  - For algorithm, it’s just `<algorithm>`, `<numeric>` and their extensions.
- You’ve already learnt most of these, so we’ll cover some details about how they are implemented.
  - This is sometimes useful for you to know what is efficient and what is not.
  - All implementations of containers and algorithms are Microsoft-version.
  - The standard only regulates the complexity, but not implementation details (so that library providers can optimize in their own platforms).
  - But they are quite similar, and you may read other versions e.g. libc++/libstdc++ (i.e. gcc/clang) yourself after this lecture.

Overview

- In this lecture, we’ll talk about containers.
- First, I admit that these things are all from cppreference and containers code in MS-STL I read.
  - I don’t and may not read Hou-STL because it’s 2000s and SGI-STL may be slightly too old.
  - So correct me if there are anything wrong!
- There are some general principles about containers:
  - Containers are designed by template, which are efficient for general cases but not for all. In real cases, you may use some other libraries that are more suitable for you (this is rare if you’re just a student…).
  - Some containers may have methods that `<algorithm>` has provided, but they are more efficient (otherwise why bother?).
  - Containers are not thread-safe.
    - This will be easy to understand when you know how they’re implemented.

- **Part 1**
- **Iterators**

- **Sequential Containers**
  - **span, mdspan**
  - **Bitset (not in them, but suitable to talk about here)**
    - String is not talked here, and will be covered in the future.

- **Container Adaptors**

- **Associative Containers**

Before that…

- Before all these things, we may introduce two special integral type aliases defined in `<cstddef>`:
  - `std::size_t`: the type of `sizeof()`; this means the size of object cannot exceed the representable value of `std::size_t`.
    - So, the maximum size of array is within `std::size_t`, so index is also within it.
    - Thus, in containers, all `.size()`-like thing will return `std::size_t`, and all `operator[]`-like thing for array will accept `std::size_t`.
    - **It's unsigned**, and the signed version `ssize_t` is not standard.
    - Since C++23, you can use `z` as literal suffix for signed `std::size_t`, and `zu` as literal suffix for `std::size_t`.
  - `std::ptrdiff_t`: the return type of subtracting two pointers.
    - **It's signed**.

Before that…

- They exist because they may be different on different platforms.
  - E.g. for x86 & x64, `size_t` is often respectively 32-bit/64-bit.
  - `ptrdiff_t` is needed because on some old platforms, you need segments to represent the array, and pointer can only operate address in a segment.
- Since they're more "cross-platform", some recommend to only use them instead of e.g. `int`.
  - I slightly disapprove with it…
  - Unless you're writing really really basic library (e.g. C++ standard library), it's better to use e.g. `std::uintxx_t` if you want the capability of cross-platform in your target platforms.
  - But for many methods in containers, they may return `std::size_t`, so it's your duty to consider carefully the limit and conversion between integers.

Containers, ranges and algorithms

Iterators

Iterators

- As its name, iterators are an abstraction of way to iterate over the containers and many other iterable things.
- There are 6 kinds of iterators:
  - Input/Output iterator: For output, you can only do `*it = val, it++, ++it` and `it1 = it2`, provides write-only access; for input, you can also use `==, !=` and `->`, provides read-only access.
  - These two iterators are not for containers, but for e.g. algorithm requirement of iterator and other things.

  - Forward iterator: same as input iterator, and can also be copied or default constructed.
    - This is the weakest iterator for containers (e.g. single linked list).
  - Bidirectional iterator: same as forward iterator, and can also do `--it, it--`.
    - Can go in bi-direction one by one, e.g. for double linked list, map.

Iterators

- Random access iterator: same as bidirectional iterator, and can also `+/-/+=/-=/[]` with an integer and be compared by `</>/<=/>=` (But for generality, we usually use `!=` in loop).
  - E.g. deque; This is almost equivalent to pointers in operations.
- Contiguous iterator (since C++17): same as random access iterator, with an additional guarantee that memory occupied by iterators is contiguous.
  - E.g. vector, string.
- These iterators don't have inheritance hierarchy, though the requirements are exactly more and more strict.

- Since C++20, they become *concept*, so that if your iterators don't match some requirements, the error information is more readable.
  - They are applied in algorithms for ranges; We'll cover them later!

Notice that C++20 iterator concept has different constraints compared with C++17 requirements; we don't cover all requirements of them here either. Check them yourself if you like.

Iterators

- **IMPORTANT**: Iterators are as unsafe as pointers.
  - They can be invalid, e.g. exceed bound.
  - Even if they're from different containers, they may be mixed up!
  - Some may be checked by high iterator debug level.

- All containers can get their iterators by:
  - `.begin(), .end()`
  - `.cbegin(), .cend()`: read-only access.
- Except for forward-iterator containers (e.g. single linked list):
  - `.rbegin(), .rend(), .crbegin(), .crend()`: reversed iterator, i.e. iterate backwards.

Iterators

They have ranges-version, e.g. `std::ranges::begin`; use them since C++20 (reasons covered later) !

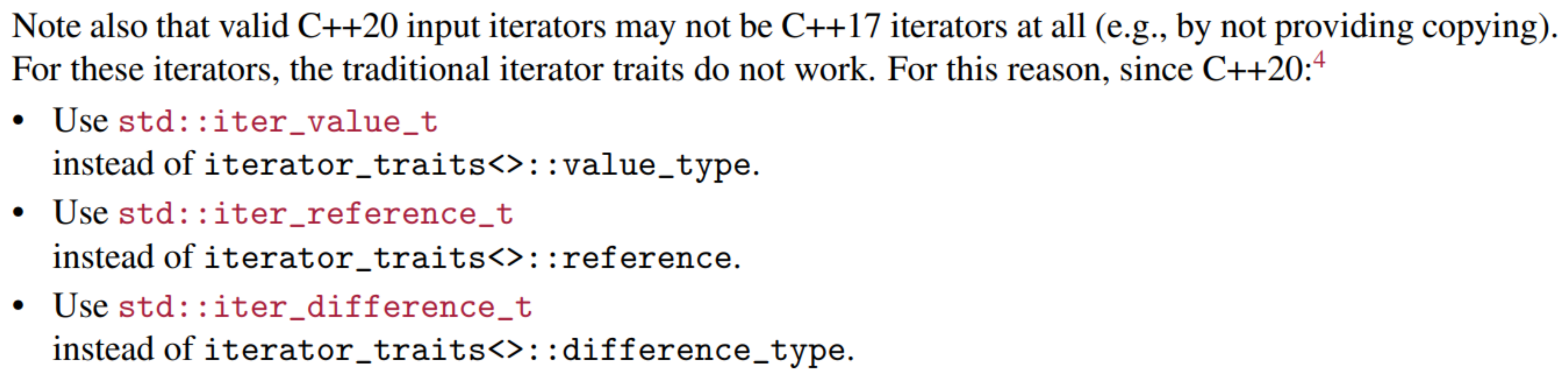
- You can also use global functions to get iterators:
  - E.g. `std::begin(vec), std::end(vec)`.
  - They are defined in any container header.
- Notice that pointers are also iterators!
  - So, for array type (**not pointer type**), e.g. `int arr[5]`, you can also use `std::begin(), std::end()`, etc.
    - We just get two pointers, e.g. here `arr` and `arr + 5`.
- There are also general methods of iterator operations, defined in `<iterator>`.
  - `std::advance(InputIt& it, n)`: `it += n`(for non-random, just increase by `n` times). `n` can be negative, but it should be bidirectional.
  - `std::next(InputIt it, n = 1)`: **return** `it + n`, not change original one.
  - `std::prev(BidirIt it, n = 1)`: **return** `it - n`, not change original one.
  - `std::distance(InputIt it1, InputIt it2)`: return `it2 – it1`(for non-random access, just iterate `it1` until `it1 == it2`).

Iterator traits

- Iterators provide some types to show their information:
  - `value_type`: The type of elements referred to.
  - `difference_type`: The type that can be used to represent the distance between elements (usually `ptrdiff_t`).
  - `iterator_category`: e.g. `input_iterator_tag`. `continuous_iterator_tag` is added since C++20.
    - It’s recommended to use `iterator_concept` when it’s available instead of category in C++20, which has more precise description of the iterator, especially for iterator of C++20 ranges.
  - `pointer`: the type of pointer to the referred element, only available in container iterators.
  - `reference`: the type of reference to the referred element, only available in container iterators.
  - You may use `std::iterator_traits<IteratorType>::xxx` (defined in `<iterator>`) to get them (absent ones will be `void`).

Iterator traits

- Since C++20, you can directly get them by:
  - `std::iter_value_t<IteratorType> / std::iter_reference_t<IteratorType> / std::iter_const_reference_t<IteratorType> / std::iter_difference_t<IteratorType>`
    - Pointer and category are not provided directly.



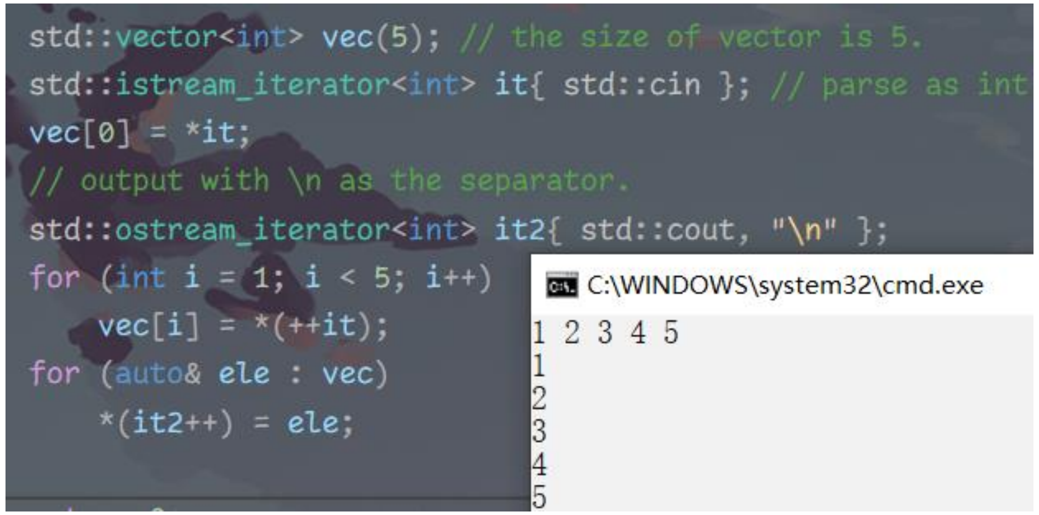
Note also that valid C++20 input iterators may not be C++17 iterators at all (e.g., by not providing copying). For these iterators, the traditional iterator traits do not work. For this reason, since C++20:[4]

- Use `std::iter_value_t` instead of `iterator_traits<>::value_type`.
- Use `std::iter_reference_t` instead of `iterator_traits<>::reference`.
- Use `std::iter_difference_t` instead of `iterator_traits<>::difference_type`.

Credit: C++20 the Complete Guide by Nicolai M. Josuttis

Stream iterator

- Beyond iterators of containers, stream iterators and iterator adaptors are also provided in standard library.
- For stream iterators…
  - When reading from input stream/writing to output stream in a simple and fixed pattern, you can use `std::istream_iterator<T>` and `std::ostream_iterator<T>` (respectively input and output iterator).
  - They are initialized by the stream, e.g. `std::cin/std::cout`.
  - The initialization of `istream_iterator` will cause the first read, `*` will get the value, `++` will trigger the next read.
- For example:



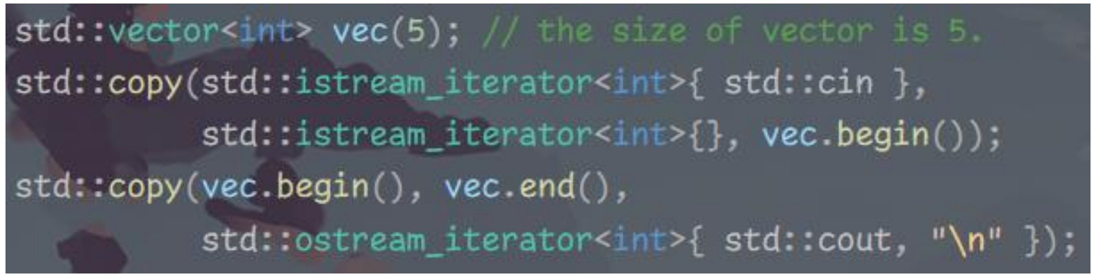
```cpp
std::vector<int> vec(5); // the size of vector is 5.
std::istream_iterator<int> it{ std::cin }; // parse as int
vec[0] = *it;
// output with \n as the separator.
std::ostream_iterator<int> it2{ std::cout, "\n" };
for (int i = 1; i < 5; i++)
    vec[i] = *(++it);
for (auto& ele : vec)
    *(it2++) = ele;
```

C:\WINDOWS\system32\cmd.exe

```
1 2 3 4 5
1
2
3
4
5
```

Stream iterator

- However, they are mostly used with std::(ranges::)copy.

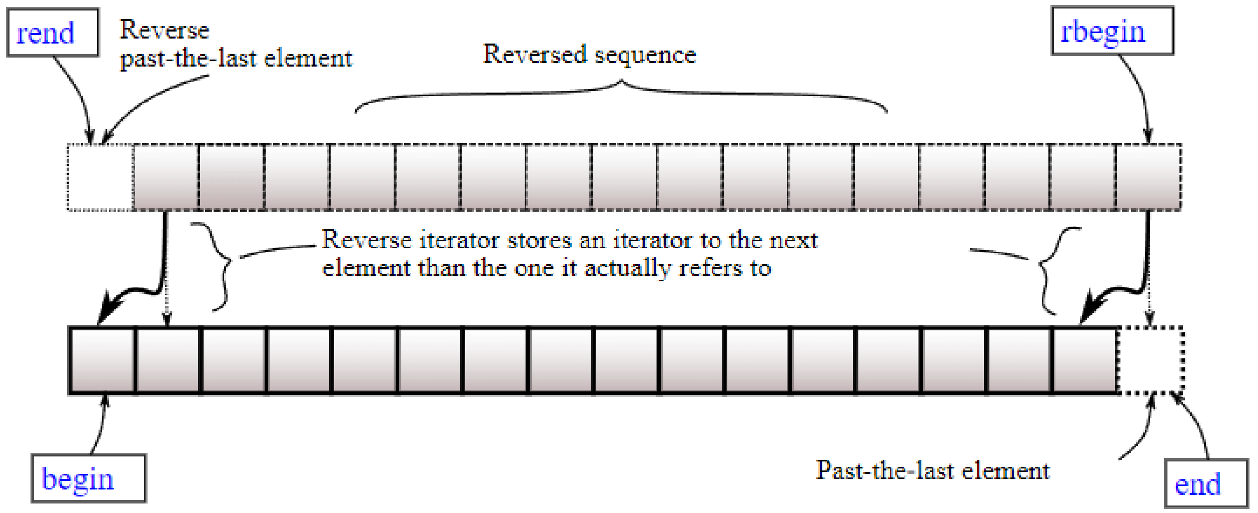


```
std::vector<int> vec(5); // the size of vector is 5.
std::copy(std::istream_iterator<int>{ std::cin },
          std::istream_iterator<int>{}, vec.begin());
std::copy(vec.begin(), vec.end(),
          std::ostream_iterator<int>{ std::cout, "\n" });
```

- The default constructed `istream_iterator` is “end”; this means to terminate until the input stream cannot be parsed as `int` (e.g. input a non-digit character) or encounter with stream error (e.g. end of file for file stream).
  - So it’s dangerous because you may not assume the input of users, and the vector iterator may exceed its bound…
  - Some may hope to use `std::copy_n(std::istream_iterator<int>{std::cin}, vec.size(), vec.begin())`, but if the input stream reaches its end, the dereference is invalid too…
    - There is no `copy_until`!

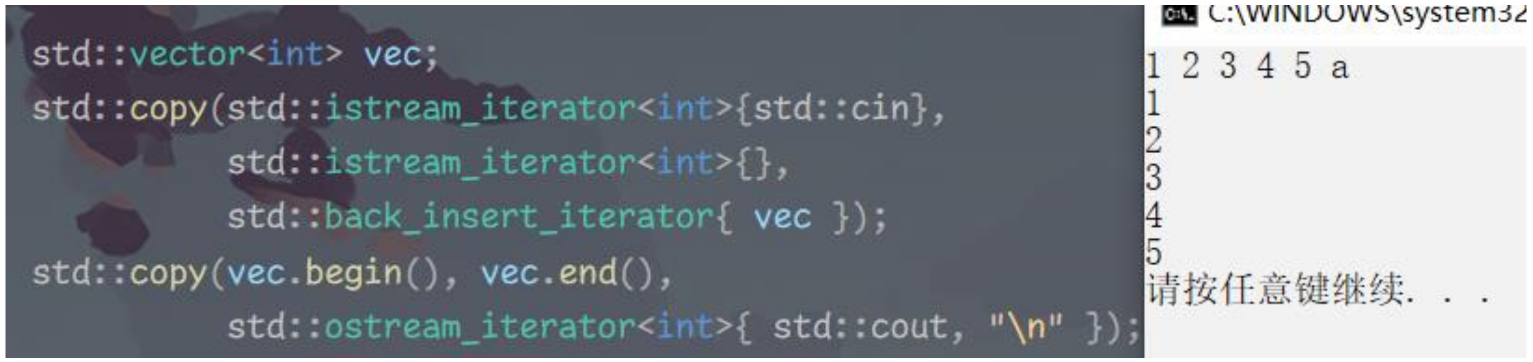
Iterator adaptor

- There are two kinds of iterator adaptors:
- One is created from iterators to perform different utilities:
  - E.g. reversed iterators, i.e. the underlying type is also iterators, while `++` is in fact `--`.
    - You can construct from an iterator, i.e. `std::reverse_iterator r{ p.begin() }`.
  - You can get the underlying iterator by `.base()`, which actually returns the iterator that points to the elements after the referred one.
    - E.g. `rbegin().base() == end()`
  - There is another adaptor called move iterator, which will be covered in *Move Semantics*.


rend
Reverse past-the-last element
Reversed sequence
rbegin
Reverse iterator stores an iterator to the next element than the one it actually refers to
begin
Past-the-last element
end

Iterator adaptor

- Another is created from containers to work more than "iterate".
  - std::back_insert_iterator{container}: *it = val will call push_back(val) to insert.
  - std::front_insert_iterator{container}: call push_front(val) to insert.
  - std::insert_iterator{container, pos}: call insert(pos, val) to insert, where pos should be an iterator in the container.
  - They are all output iterators, and val is provided by assignment.
- For example:



```
std::vector<int> vec;
std::copy(std::istream_iterator<int>{std::cin},
          std::istream_iterator<int>{},
          std::back_insert_iterator{ vec });
std::copy(vec.begin(), vec.end(),
          std::ostream_iterator<int>{ std::cout, "\n" });
```

```
1 2 3 4 5 a
1
2
3
4
5
请按任意键继续. . .
```

Iterator adaptor

- Notice that inserting/assigning a range directly is usually better than inserting one by one (as done in `inserter`) for `vector/deque`.
  - Or at least “reserve” it before (we’ll learn them sooner).
- Final word: there are methods like `std::make_xxx` or `std::xxx` (e.g. `std::back_inserter(), std::make_reverse_inserter()`); You can also use these methods to get the corresponding iterator adaptors.
  - Before C++17, you have to specify the type parameter for template of class, so if you don’t use these functions, you have to write tedious `std::back_inserter_iterator<std::vector<int>>(vec)`.
  - Since C++17, CTAD (*Class Template Automatic Deduction*) will deduce it, so methods are generally not shorter than object initializations.

Containers, ranges and algorithms

Sequential Containers

Containers

- Sequential Containers
    - array
    - vector
    - bitset
    - span
    - deque
    - list
    - forward_list

Array

- We’ve learnt C-style array, e.g. `int a[5]`;
  - However, it will decay to `int*` when passing to function, and the size information is dropped.
    - i.e. the first dimension of the array parameter is meaningless, `void func(int a[5])` is just same as `void func(int a[])` or `void func(int* a)`.
    - So, `sizeof(a)` is different inside and outside the function…
    - The return type cannot be `int[5]`, too…
  - For `int a[5], b[5]`, `a = b` is invalid.
  - The bound is never check so invalid memory may be accessed…
- All in all, we need a safer array!
- `std::array<T, size>` is for you.
  - It’s same as `T[size]`, except that it always preserves size, can copy from another array, and can do more things like bound check.

Array

- So, it’s allocated on stack!
  - Err, but if you `new std::array`, then it’s still allocated on heap.
- For ctor: just initialize `std::array` in the same way as C-style array (may need adding an additional pair of paratheses).
  - For example, `struct S {int i; int j;};`, `std::array<S,2> arr{{ {1,2}, {3,4} }}`.
- For member accessing:
  - `operator[]/at()`: accessing by index; `at()` will check the bound, i.e. if the index is greater than size, `std::out_of_range` will be thrown.
  - `front()/back()`: get the first/last element of vector.
  - Contiguous iterators, as we stated.
  - If you want to get the raw pointer of array content, you can use `.data()`.

Array

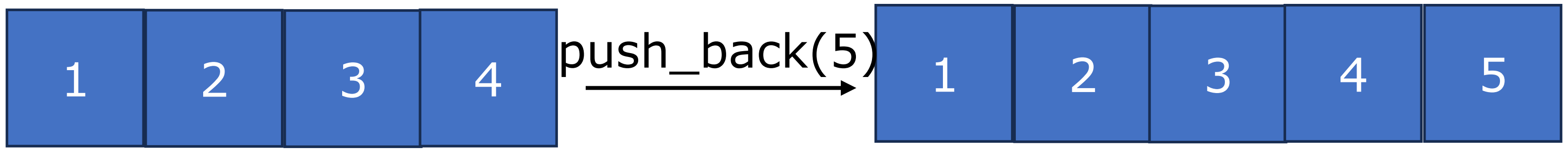
- You can also use some additional methods:
  - `.swap()`: same as `std::swap(arr1, arr2)`.
  - `operator=`, `operator<=>`.
  - All these methods need same array size!

  - `.fill(val)`: fill all elements as val.
  - `std::to_array(C-style array)`: since C++20, get a `std::array` from a C-style array.
- For size operations:
  - `.size()`: get size (return `size_t`).
  - `.empty()`: get a `bool` denoting whether `size == 0`.
  - `.max_size()`: get maximum possible size in this system(usually useless).
- Very easy, so we quickly skim it…

Basics

- Sequential Containers
  - array
  - vector
  - deque
  - list
  - forward_list

vector



- To be exact, vector is dynamic array which can be resized.
  - It supports random access and occupies contiguous space.
  - When inserting and removing elements at the end (i.e. pushing / popping back), the complexity is **amortized** $O(1)$.
    - If not at end, it'll be $O(n)$.
- This container is so important that we'll spend lots of time on it.
  - It's the most commonly used containers in all containers.
  - Though sometimes its theoretical complexity may be higher than other containers (e.g. list), it may still be a better choice since it significantly **utilizes cache**.
    - We've learnt in ICS that a cache-friendly program may consume hundreds or even thousands times less time than a bad one.
    - Use a profiler if you cannot determine which one is better!

vector

- First, let’s review what “amortized” means.
  - Sometimes we talk about the worst complexity.
  - Sometimes we may assume the distribution of cases (e.g. the most commonly is uniform), so that we get average/expected complexity.
    - E.g. For quick sort, the worst complexity is $O(n^2)$ while the average complexity is $O(n\log n)$.
  - For data structures, amortized complexity is also usually used.
    - That is, if we do some operations continuously, $\frac{total\ complexity}{operation\ number}$ is amortized complexity.
    - So “inserting at end is amortized $O(1)$” means that if you push lots of elements from end, some operations may take great time (usually $O(n)$ for vector), but it happens “**rarely**” so that in a total view, one operation takes only $O(1)$ time.

vector

- So, the most naïve version of vector is:
  - When pushing back, allocate continuous space with one more, copy* all and add the new one to the new space, finally freeing the original.
    - You cannot directly allocate one new space since it cannot guarantee the property of array - “contiguous space”.
  - When popping back, shrink the space by one and copy all rest to the new one, finally freeing the original.
- Obviously, you need $O(n)$ on every pushing or popping…
  - So, what if we “prepare” more space than needed in allocation, so that pushing will only construct new object at back?
  - This is $O(1)$, and we just need to control **reallocation** to happen only rarely so that copying will be amortized $O(1)$.
  - The element number is called **size**; total space is called **capacity**.

It’s usually “move” instead of copy, but this doesn’t influence theoretical complexity here. We’ll cover “move” in *Move Semantics*.

Reallocation strategy of vector

- The easiest strategy is increasing space linearly.
  - E.g. 0->4->8->12->16…
  - Every k operations will trigger reallocation and copy $n = km$ elements.
    - So, the amortized complexity is $\Theta\left(\frac{\sum_{i=1}^{m} ki}{km}\right) = \Theta(m) = \Theta(n/k)$.
    - Considering that k is an constant, this is still $O(n)$.
  - This means "linear" is not "rare"!
- So, what about exponentially?
  - E.g. 1->2->4->8->16->32…
  - Every 2^k operations will trigger reallocation and copy $n = 2^k$ elements.
    - So, the amortized complexity is $\Theta\left(\frac{\sum_{i=1}^{k} 2^i}{2^k}\right) = \Theta(1)$.
    - Summation of arithmetic progression v.s. geometric progression.

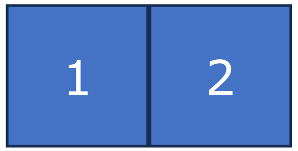
Here we assume the case is "reallocation happens in the last step" (which is the worst!).
You may analyze more general case or watch it on THU DSA course.

Reallocation strategy of vector

- vector also supports insertion of a range.
    - So more than one elements may be inserted.
- Considering that some insertion will make >2x growth.
    - You may calculate the smallest capacity that is larger than needed.
    - In MS, it directly allocates needed space, which is cheaper.
- Now let me show you process above…

Reallocation strategy of vector

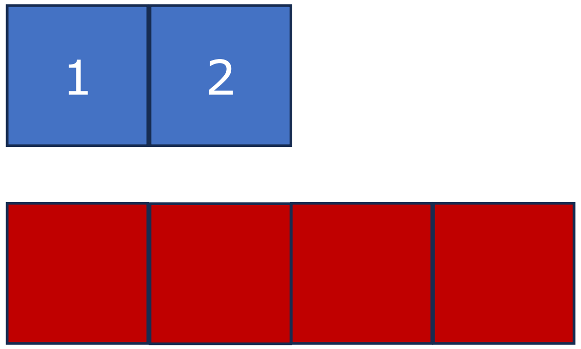
- Case1: `push_back(3)`
    - Full, so need reallocation



capacity = 2
size = 2

Reallocation strategy of vector

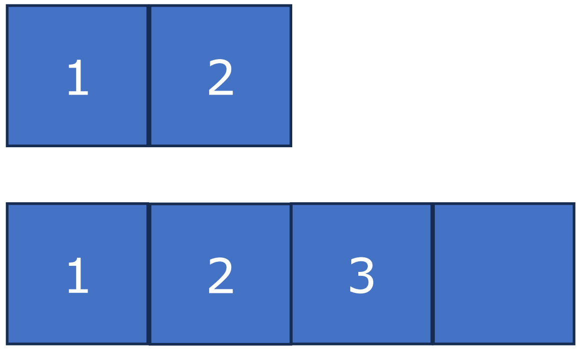
- Reallocation exponentially…



capacity = 4
size = 2

Reallocation strategy of vector

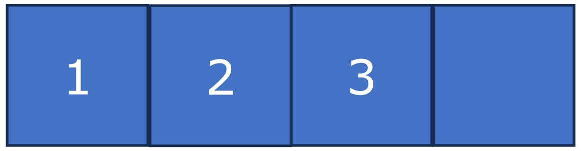
- Copy all elements, push back 3



capacity = 4
size = 3

Reallocation strategy of vector

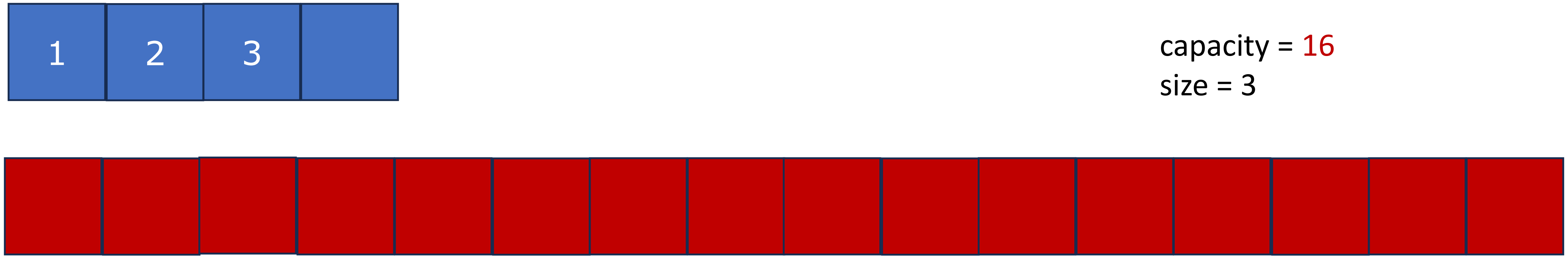
- Free original space…



capacity = 4
size = 3

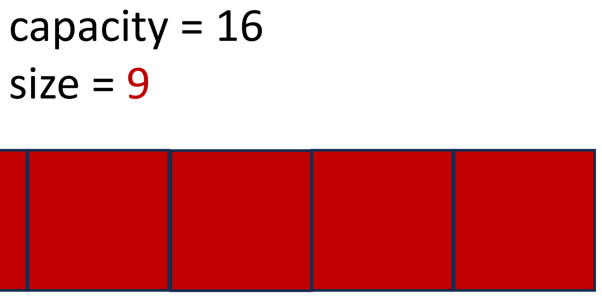
Reallocation strategy of vector

- Case2: `insert(vec.end(), {4,5,6,7,8,9})`
  - Final size is 3+6=9 > 4*2=8
  - For normal exponential increment…



Reallocation strategy of vector

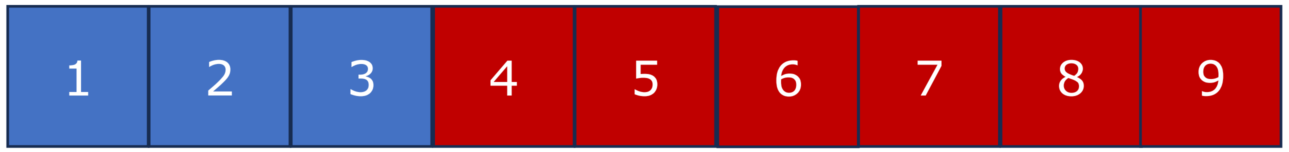
- Case2: `insert(vec.end(), {4,5,6,7,8,9})`
  - Final size is 3+6=9 > 4*2=8
  - For normal exponential increment (copying, inserting and freeing)…



Reallocation strategy of vector

- Case2: `insert(vec.end(), {4,5,6,7,8,9})`
  - Final size is 3+6=9 > 4*2=8
  - For MS implementation(to simplify, we merge all procedures here)…

capacity = 9
size = 9

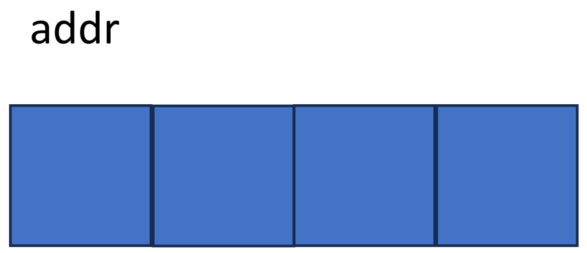


Reallocation strategy of vector

- Finally, why is the exponent 2?
  - You can prove that for any exponent > 1, the amortized complexity is $O(1)$.
  - This is a trade-off between space and time.
    - If the exponent is too low, reallocation will happen more frequently so that the constant of $O(1)$ is larger.
    - If it's too high, the space will be consumed quickly so that you may waste a lot.
  - This is just one practical choice (e.g. in gcc).
- In MS, it's 1.5.
  - This considers more than trade off.
  - Facebook Folly Doc: *Despite other compilers reducing the growth factor to 1.5, gcc has staunchly maintained its factor of 2. This makes std::vector cache-unfriendly and memory manager unfriendly.*
  - Let's show you what it means!

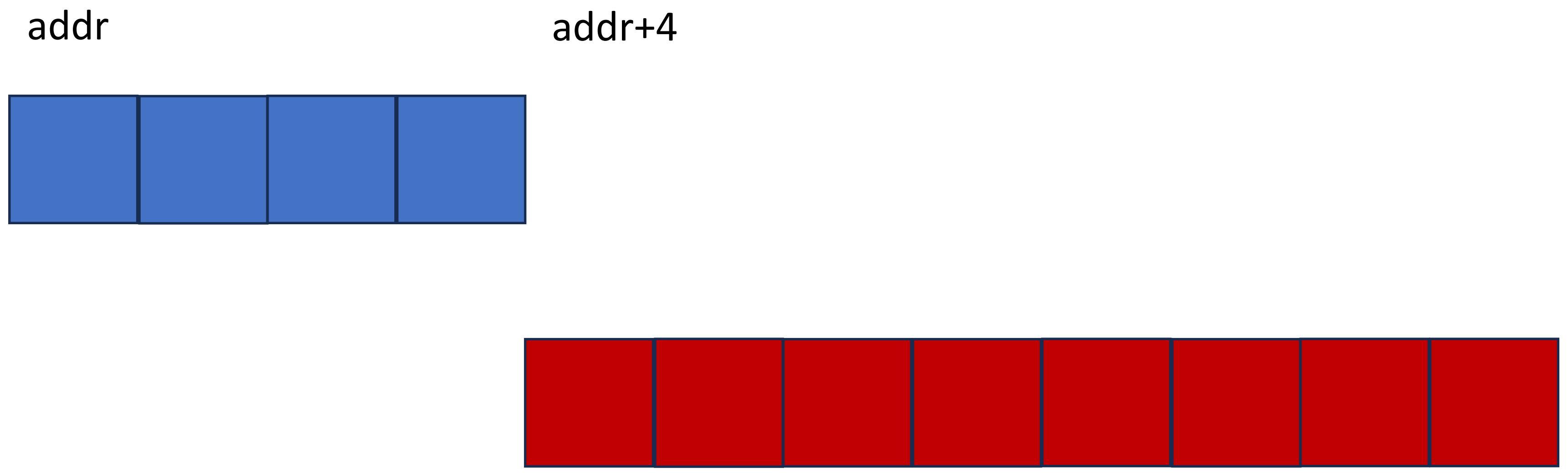
Reallocation strategy of vector

- We assume the begin position is addr, and only vector needs memory allocation…



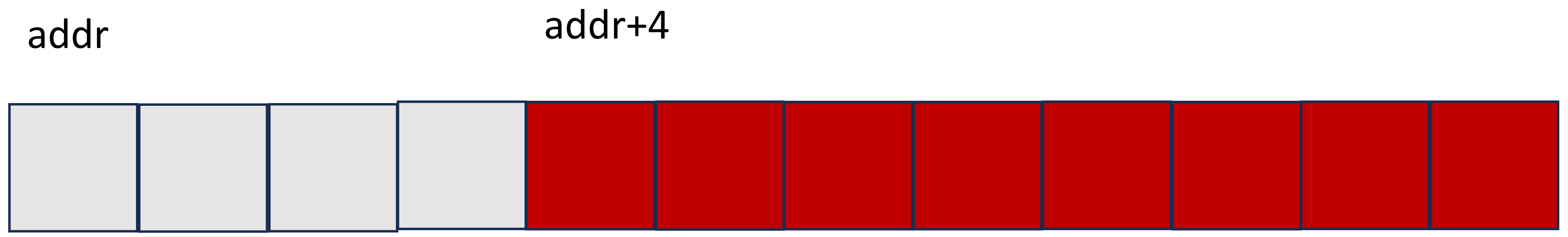
Reallocation strategy of vector

- For x2 strategy…



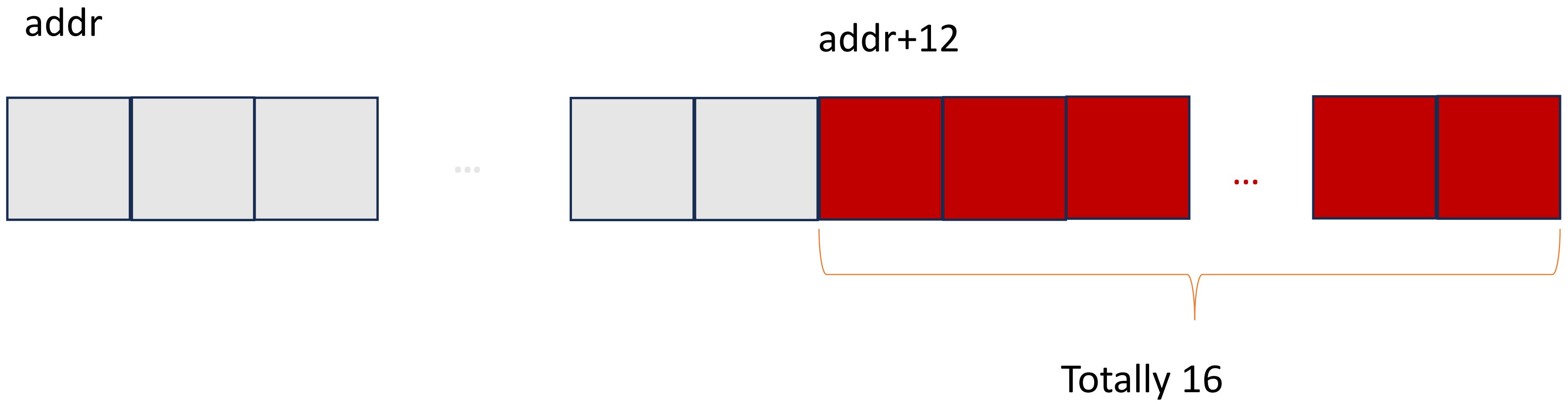
Reallocation strategy of vector

- For x2 strategy…



Reallocation strategy of vector

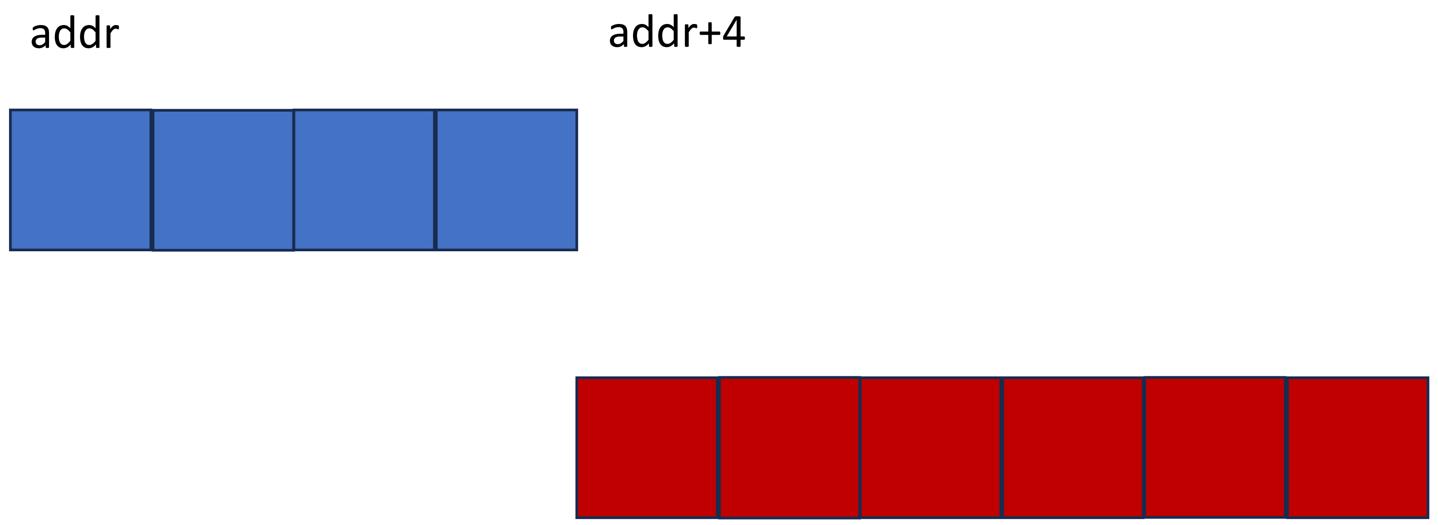
- Next reallocation is 16, cannot utilize `addr – addr + 4`…



- You will find that reallocation will never utilize the freed space!
  - $\left(1 + 2 + \cdots + 2^{k-1}\right) < 2^{k+1}$.

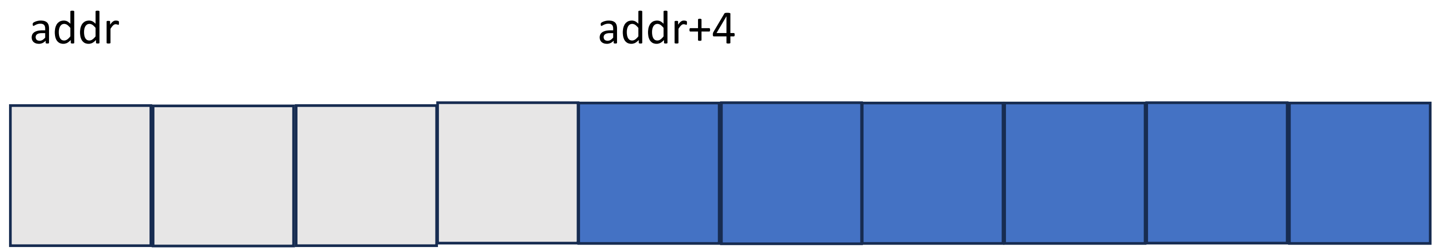
Reallocation strategy of vector

- For x1.5 strategy…



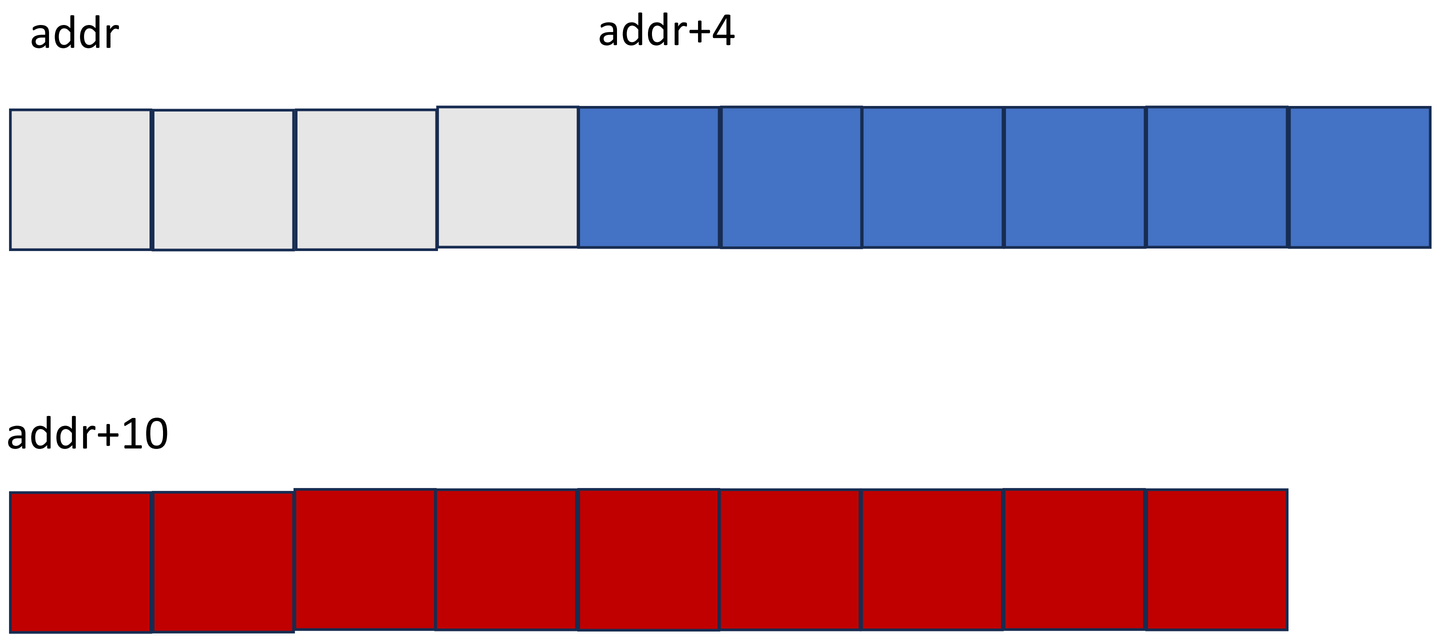
Reallocation strategy of vector

- For x1.5 strategy…



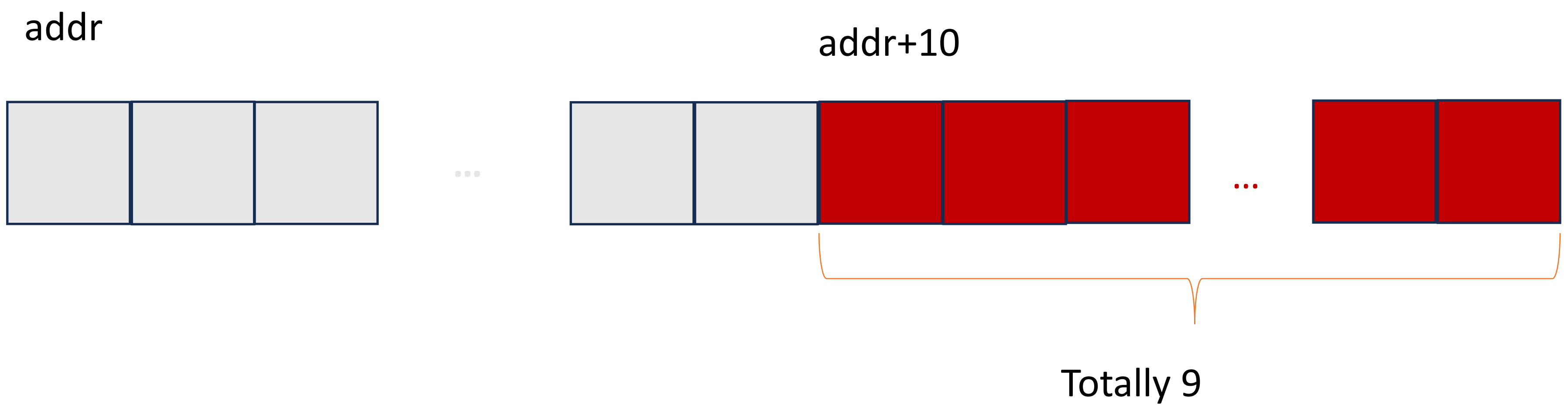
Reallocation strategy of vector

- For x1.5 strategy…



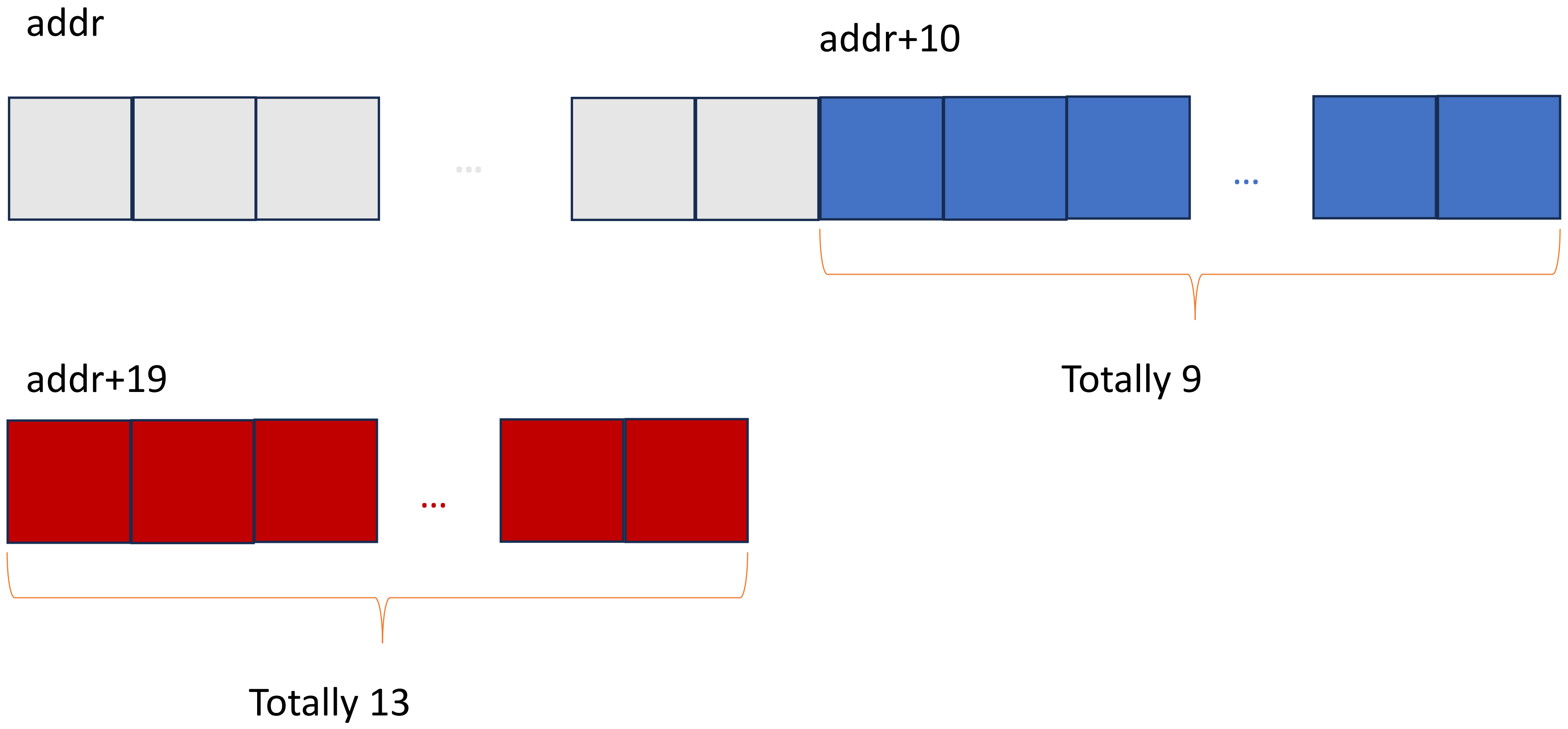
Reallocation strategy of vector

- For x1.5 strategy…



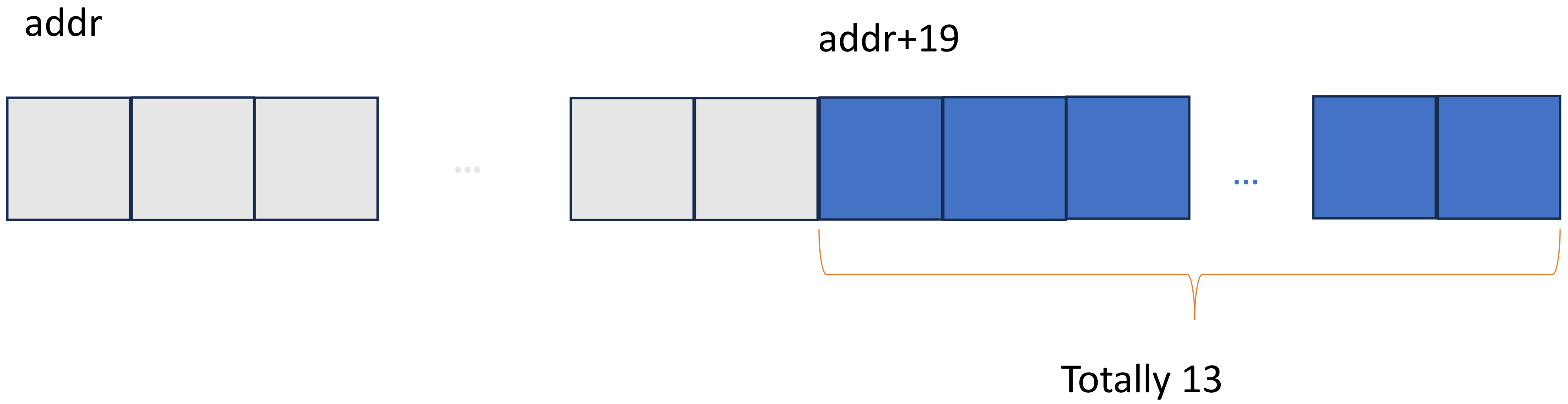
Reallocation strategy of vector

- For x1.5 strategy…



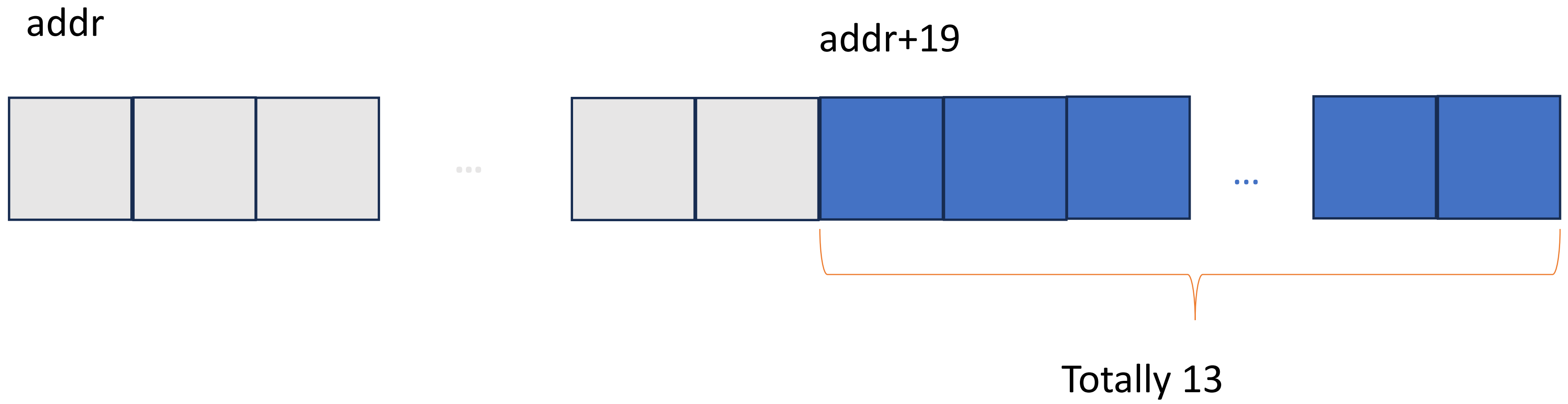
Reallocation strategy of vector

- For x1.5 strategy…



Reallocation strategy of vector

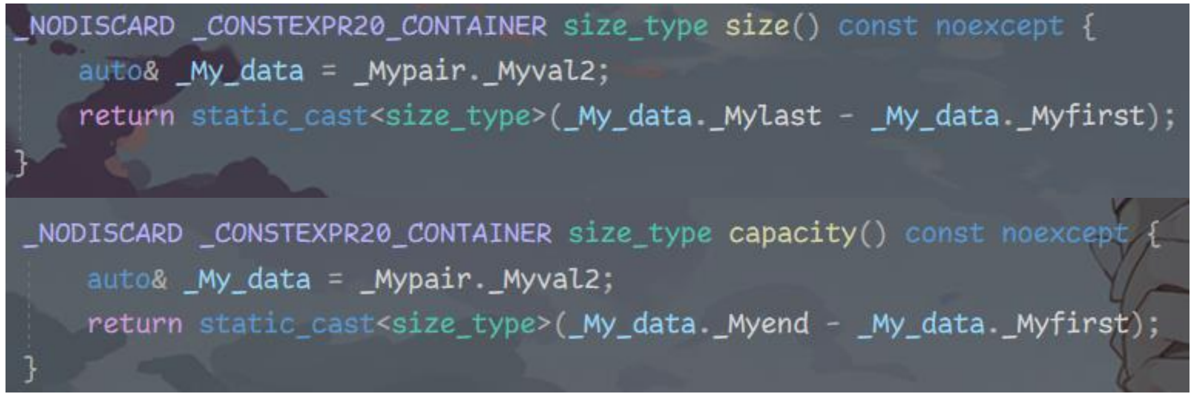
- Next allocation is 13 * 1.5=19, so you can utilize `addr – addr+19`!



- $\left(1.5 + 1.5^2 + \cdots + 1.5^{k-1}\right) = 2 * (1.5^k - 1.5) > 1.5^{k+1}$ may be true.
- Practically friendly to memory management and cache.

vector

- To sum up,
  - Vector is just a dynamic array.
    - It occupies contiguous space and can be random accessed by `[]`.
    - It has members as: pointer to content, size and capacity.
      - In implementation, they are first pointer, last pointer and end pointer.



```
_NODISCARD _CONSTEXPR20_CONTAINER size_type size() const noexcept {
    auto& _My_data = _Mypair._Myval2;
    return static_cast<size_type>(_My_data._Mylast - _My_data._Myfirst);
}
```

```
_NODISCARD _CONSTEXPR20_CONTAINER size_type capacity() const noexcept {
    auto& _My_data = _Mypair._Myval2;
    return static_cast<size_type>(_My_data._Myend - _My_data._Myfirst);
}
```

  - When the vector is full, it's basically reallocated exponentially so that push_back is $O(1)$.

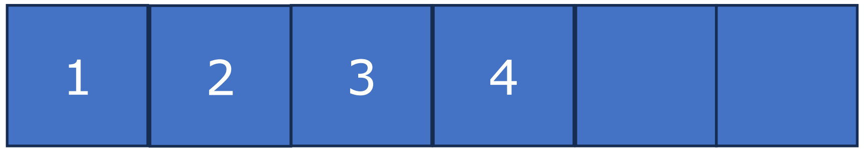
vector

- Obviously, popping back is $O(1)$.
  - You may think the vector will shrink when $\frac{size}{capacity}$ is too low!
    - The analysis is similar to appending, you can prove it's amortized $O(1)$.
  - However, practically, vector doesn't shrink automatically for efficiency, but it gives you ways to shrink it manually.
    - Besides, automatic shrink will violates regulation on iterator invalidation, which will be covered sooner.

- For insertion, implementation you may have learnt is:
  - Move backwards (prevent overwriting) from the final element.
  - Insert into the empty positions.

Insertion

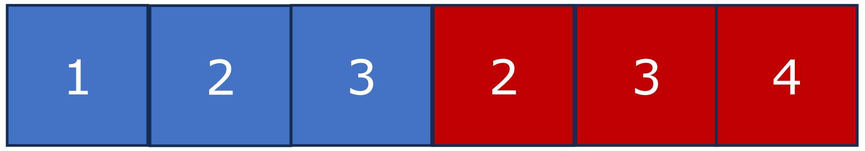
- `insert(vec.begin() + 1, {4, 5})`



capacity = 6
size = 4

Insertion

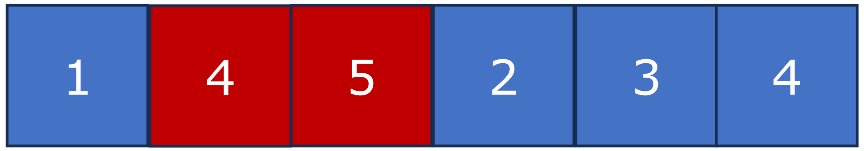
- Move backwards…



capacity = 6
size = 6

Insertion

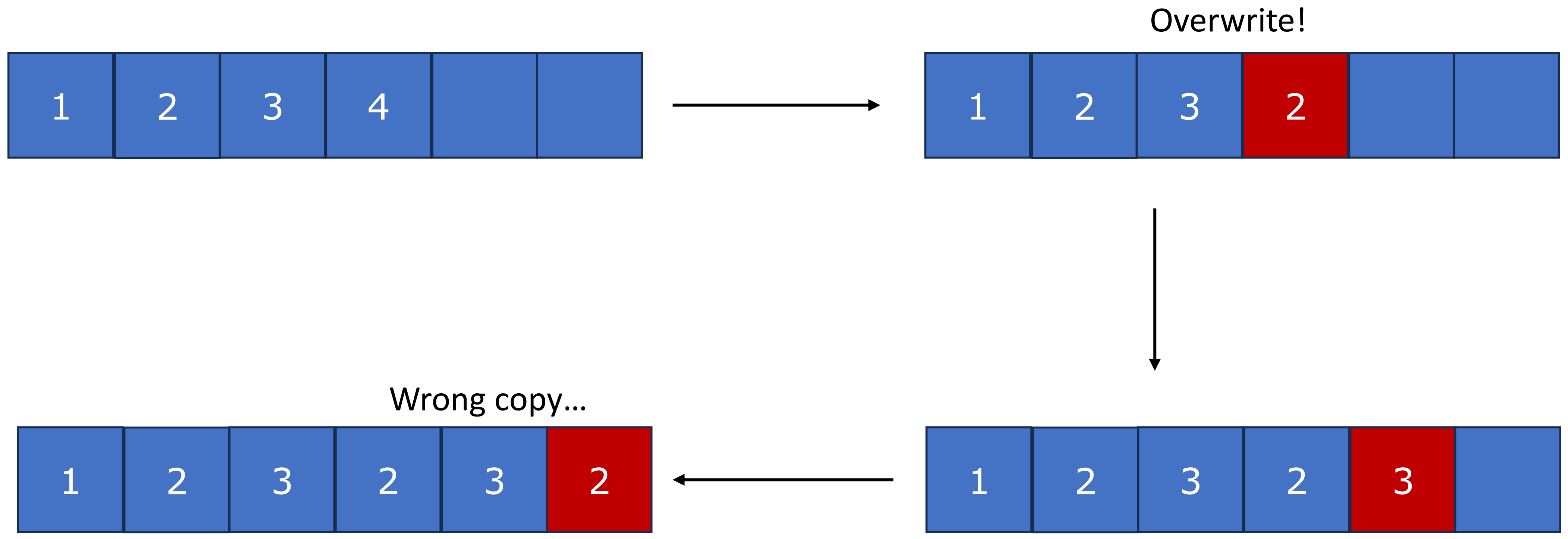
- Insertion



capacity = 6
size = 6

Insertion

- If we move forward from begin…



vector

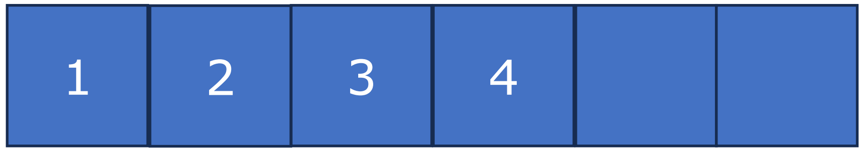
- Removal is similar, but move forwards from the end of deletion to the deletion point, and finally destruct the last several elements.
    - You may draw it yourself.
- If reallocation is needed, operations of copying from old to new and copying in insertion can be merged.

- However, MS-STL’s implementation of insertion doesn’t use this way (we’ll tell you why in the future); It is:
    - Reallocate if needed (same as normal insertion).
    - push_back all elements one by one.
    - **Rotate** them to the insertion point.

But removal is same as what we say.

Insertion

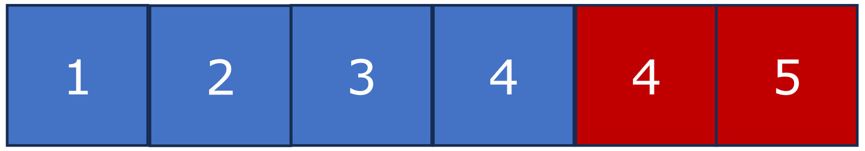
- `insert(vec.begin() + 1, {4, 5})`



capacity = 6
size = 4

Insertion

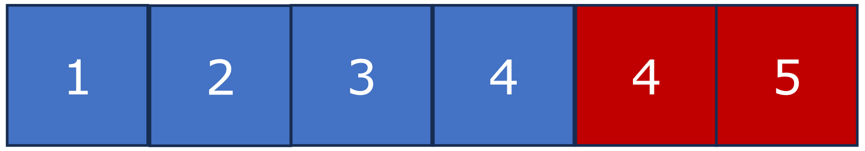
- push_back all elements one by one…



capacity = 6
size = 6

Insertion

- Rotation(This is $O(n)$ and we'll cover it in the next lecture…)



capacity = 6
size = 6

vector

- So, let’s have a look on methods provided by vector (return `void` if unspecified):
- For ctor:
  - Default ctor.
  - Copy ctor & Move Ctor.
  - `(size_t count, const T& elem = T{})`: construct a vector with `count` copies of `elem`.
  - `(InputIt first, InputIt last)`: copy elements from `[first, last)` into the vector.
  - `(std::initializer_list<T>)`: copy all elements from `initializer_list` into the vector.
  - All ctors have an optional allocator parameter.

Initializer list

- Then what is `std::initializer_list`?
  - In list initialization, we may use { 1, 2 } to pass params to ctor/function.
  - However, how is it possible to unify the initialization of vector and C array?
    - All in all, we need to pass "a list of elements" to ctor! How is it represented?
    - By `std::initializer_list`.
  - So, when a class accepts `std::initializer_list<T>`, { … } **whose elements are of type** `T` **or can be converted to** `T` will be regarded as `std::initializer_list<T>` rather than separate params!
    - Then, you can use `std::vector<int> v{1,2,3,4}` to initialize the vector (**Uniform**).
    - So, when you actually want to use `(size_t count, const T& elem = T{})`, when `T` is e.g. `int`, you cannot use `v{10, 1}` to construct a vector with 10 elements that are all 1.
      - You have to use `v(10, 1)`.
  - Finally, `std::initializer_list<T>` can be roughly seen to have underlying `const T[N]`, with methods `begin`, `end` and `size`.

- For member accessing (same as array):
  - `operator[]/at()`: accessing by index; `at()` will check the bound, i.e. if the index is greater than size, `std::out_of_range` will be thrown.
  - `front()/back()`: get the first/last element of vector.
  - Contiguous iterators, as we stated.
  - If you want to get the raw pointer of vector content, you can use `.data()`.
- For capacity operations (i.e. adjust memory):
  - `.capacity()`: get capacity (return `size_t`).
  - `.reserve(n)`: expand the memory to make `capacity = n` if it's greater than the current capacity (else do nothing); but the size is not changed.
    - You may prevent reallocation over and over again (especially `push_back` many times) by `reserve` first!
      - This is dramatically important in some parallel programs because of iterator invalidation; we'll talk about this sooner.
  - `.shrink_to_fit`: request to shrink the capacity so that `capacity == size`.
    - This is the general way for you to shrink; request may or may not be accepted.
    - For mainstream implementation (libc++/libstdc++/MS STL), shrink will happen basically as long as your class can be copied **or** moved without *exception* and space is enough for a new vector. This is because of exception guarantee, which will be covered in the following lectures!

vector

- For size operations (i.e. operate on elements, possibly influence capacity implicitly)
  - `.size()`: get size, return `size_t`.
  - `.empty()`: get a `bool` denoting whether `size == 0`.
  - `.max_size()`: get maximum possible size in this system (usually useless).
  - `.resize(n, obj=Object{})`: make the `size = n`;
    - If the original size is `n`, nothing happens.
    - If greater than `n`, elements in `[n, end)` will be removed.
    - If less than `n`, new elements will be inserted, and their values are all `obj`.
  - `.clear()`: remove all things; size will be 0 after this.
    - But the capacity is usually not changed! You need to use capacity-related operations explicitly if you want to clear memory as well.

vector

- `.push_back(obj)`: insert an element at the end.
- `.emplace_back(params)`: insert an element **constructed** by `params` at the end.
  - Since C++17, it returns reference of inserted element (before it's `void`).
- `.pop_back()`: remove an element from the end.
- `.insert(const_iterator pos, xxx)`: insert element(s) into pos, so that `vec[pos – begin]` is the first inserted element. `xxx` is similar to params of ctor:
  - `(value)`: insert a single element.
  - `(size_t count, value)`: insert `count` copies of `value`.
  - `(InputIt first, InputIt last)`: insert contents from `[first, last)`.
  - `(std::initializer_list<T>)`: insert contents of initializer list.
- `.emplace(const_iterator pos, params)`: insert an element constructed by `params` into pos.

vector

  - `.erase(const_iterator pos)/.erase(const_iterator first, const_iterator last)`: erase a single element/ elements from `[first, last)`. `first, last` should be iterators of this vector.
    - insert/erase will return next valid iterator of inserted/erased elements, so you can continue to iterate by `it = vec.erase(…)`. We'll tell you reason sooner.
- Interact with another vector:
  - `.assign`: also similar to ctor
    - `(vec)`: same as `operator=`, assign another vector
    - `(count, const T& value)`
    - `(InputIt first, InputIt last)`
    - `(std::initializer_list<T>)`.
  - `.swap(vec)`: swap with another vector, same as `std::swap(vec1, vec2)`.
- Since C++23, ranges-related methods are added.
  - `.assign_range(Range)`: can copy any range to the vector.
  - `.insert_range(const_iterator pos, Range)`
  - `.append_range(Range)`: insert range from the end.

Iterator Invalidation

- Obviously, the iterator is designed as a wrapper of the pointer to the element.
  - All operations are just for pointers, e.g. +/- is just moving pointers.
  - But pointers are unsafe!
- An iterator may be unsafe because it may not correctly represent the state of the object it is iterating. This may be caused by:
  - reallocation, the original pointer is dangling; dereferencing the iterator will access unknown memory.
  - On insertion & removal, so the original pointer points to an element that it may not intend to refer.
    - e.g. `1 2 3 4` and `it` points to `3`; after removing `2`, `it` points to `4`, which is in fact `it + 1` in the original context!
  - This is called **iterator invalidation**.

Iterator Invalidation

- For vector:
  - If the capacity changes, all iterators are invalid.
  - If the capacity doesn’t change, but some elements are moved, iterators after the changed points are invalid.
    - i.e. inserting/removing will make iterators after the insertion/removal point invalid.
    - That’s why `insert/emplace` will return a new iterator referring to the inserted element, and `erase` will return one after the final removed element.
      - You may use them to continue to iterate the vector.
- We say vector is thread-unsafe because:
  - It is only safe when two threads are reading the vector.
  - If one is writing, the other may read inconsistent content (e.g. for a vector of pair, you may read an old first and a new second…).
  - When the internal structure of the container changes (e.g. vector reallocated when inserting), another thread will access invalid memory (i.e. unexpected iterator invalidation)…

vector

- But if you can ensure that threads are just writing different elements, it's basically OK.
  - Particularly, since `vector<bool>` is still dangerous if two bits are in e.g. the same byte (you'll understand it in the next page)…

- Final words:
  - vector supports comparison, as we stated in `operator<=>`.
  - If you just want to remove all elements that equals to XXX in a vector, it's costly to use erase repeatedly ($O(n^2)$ obviously)…
    - We'll teach you $O(n)$ method in the next lecture.
    - You may just use `std::erase(vec, val)/std::erase_if(vec, func)` since C++20; they return number of removed elements.

vector<bool>

- `vector<bool>` is a weird specialization of vector…
- Boolean can be represented by only 1 bit, so `vector<bool>` is regulated to be compacted as "dynamic array of bit".
- However, the smallest unit that can be directly operated is byte, and you cannot return `bool&` for `operator[]` here!
  - What is returned is a **proxy class** of the bit.
    - For const method, it still returns `bool`.
  - You can get/set the bit through this proxy, just like normal reference.
  - This may be confusing sometimes, e.g.
    - For `vector<int>`, `auto a = vec[1]; a = 1;` will not change the vector since `auto` will not deduce reference.
    - However, for `vector<bool>` , `auto` is proxy, and this proxy holds the reference of the bit, so this will change the vector!
    - Range-based for may use `auto`, so pay attention if you're doing so!

vector<bool>

- Besides, since the returned proxy is a value type instead of reference, so the returned object is temporary!
  - Then, you cannot use `auto&` when iterating `vector<bool>`, though it's right for other types…
  - To sum up, use `auto` rather than `auto&` if you want to change elements of `vector<bool>`, use `const auto&` or `bool` if you don't want to change.
- Specialization also brings more methods…
  - proxy supports `operator~` and `flip()`, which will flip the referred bit;
  - `vector<bool>` supports `flip()`, which will flip all elements in the vector.
  - `vector<bool>` is supported by `std::hash`, which will be covered in the unordered map(i.e. hash table).
- Final word
  - For its unfriendly properties for generic code and novices, `vector<bool>` is discouraged by many. Besides, operating bits is also slower than bytes, and reducing the memory seems unnecessary in the modern computers.
  - Its iterator is also not seen as contiguous.
  - So, be cautious and careful if you want to use/process this type!

Containers

- Sequential Containers
  - array
  - vector
  - bitset
  - span
  - deque
  - list
  - forward_list

Bitset

- `bitset` is in fact not a container, and we cover it here just because it also has many bits like `vector<bool>`…
  - However, the size is determined at compile time, i.e. you need to specify `bitset<size>`.
  - `vector<bool>` to `bitset` is similar (**not same**) to `vector` to `array`!
- Difference:
  - `bitset` doesn’t provide iterators.
  - `bitset` provides more methods, which makes it a more proper way to manipulate bits.
    - You may use `&,|,^,~,<<,>>`, just like operating binary series.
    - You can use `set(), set(pos, val = true), reset(), reset(pos), flip(), flip(pos)` to make all bits 1/0/flipped or set bit at pos `val`/0/flipped.
      - `pos` is index (`size_t`) since `bitset` doesn’t support iterator.
    - You can use `all(), any(), none(), count()` to check whether all/if any/whether none of bits are set / get number of set bits.
    - It can also be input/output by `>>/<<`.

Bitset

- Besides, `bitset` can be converted to `std::string/unsigned long long` by `.to_string(zero = ‘0’, one = ‘1’)/to_ullong()`.
  - The former may throw `std::bad_alloc` for allocation error on string
  - The latter may throw `std::overflow_error` if the value is unrepresentable by `unsigned long long`.
  - `bitset` can also be constructed by a `string/unsigned long long`, i.e. `(str, start_pos = 0, len = std::string::npos, zero = ‘0’, one = ‘1’)`.
    - No need to remember; check in cppreference when needed.
- Similarity:
  - You can access the bit by `operator[]`, which returns a proxy class too (for const methods, `bool` too).
    - There is no `at(pos)` in `bitset`, but a `bool test(pos)`, which will do bound check.
  - You can compare two bitsets (only in the same size, and only `==` and `!=`)
  - You can get size by `.size()`.
  - You can hash it by `std::hash`.

Containers

- Sequential Containers
  - array
  - vector
  - bitset
  - span
  - deque
  - list
  - forward_list

span

- Since C++17, more and more view-like things are provided.
  - View means that it doesn't actually hold the data; it observes the data.
  - So, their construction and copy are much cheaper than traditional containers.
- Span is a view for contiguous memory (e.g. vector, array, string, C-style array, initializer list, etc.).
- Before, you may have written things like `void func(int* ptr, int size)`.
  - Even if there are things like vector and you may use their references as parameters, what if you only want to operate on e.g. a sub-vector?
    - You have to copy it to a new container, which is costly…
  - Span is for this case; you can code `void func(std::span<int> s);`.

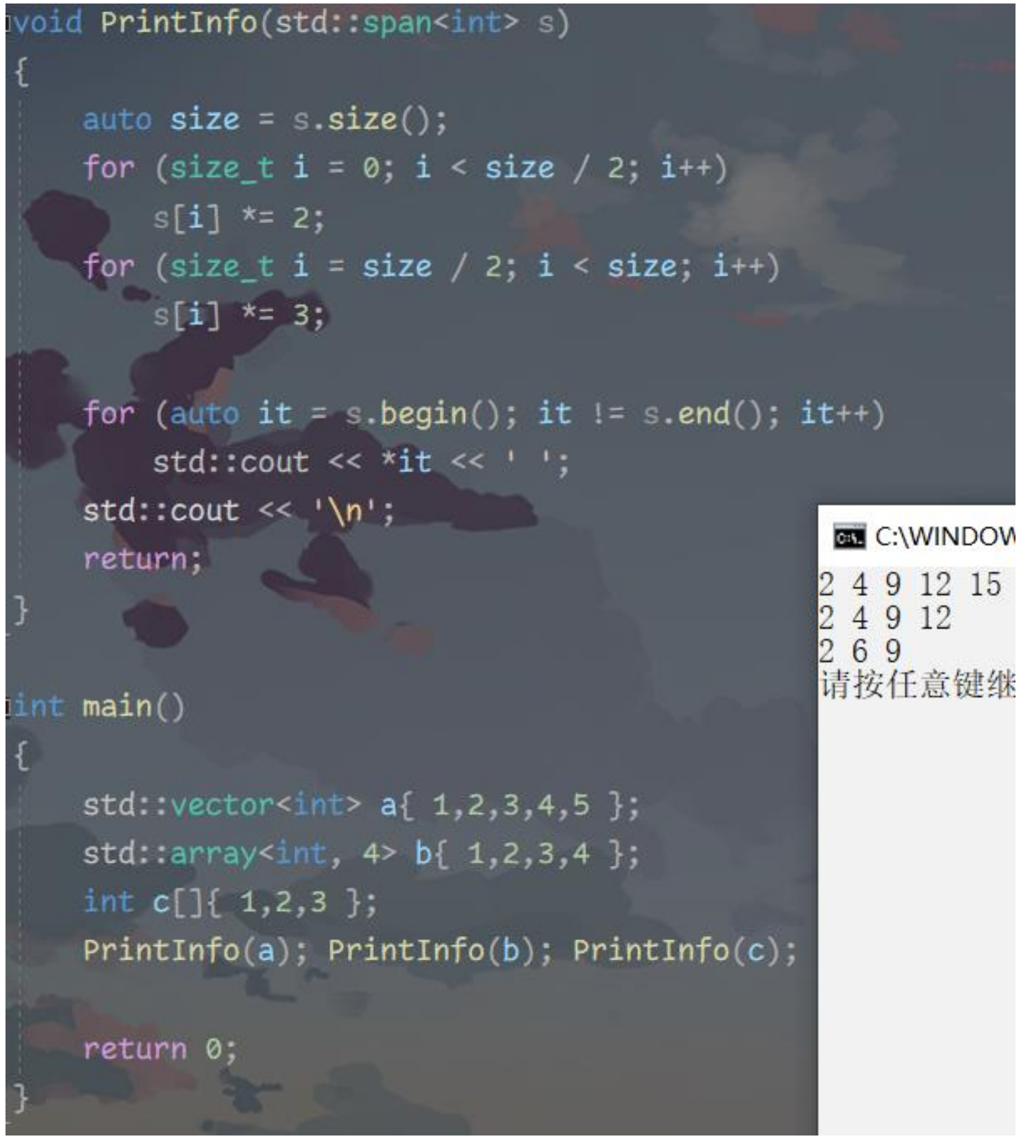
span

- You can just operate span almost as if operate on an array.
  - You can use random access iterators.
  - You can use `front()/back()/operator[]/data()`.
  - You can use `size()/empty()`.
- You can also use `size_bytes()` to get size in byte.
- You can create new sub-spans in span cheaply:
  - `.first(N)/.last(N)`: make a new subspan with the first `N`/the last `N` elements.
  - `.subspan(beginPos(, size))`: make a new subspan begin at `beginPos` with `size` (by default until the last one).
- Remember: span is just a pointer with a size! All copy-like operations are cheap.

You can also use `std::as_bytes` and `std::as_writable_bytes` that convert a span to a span of bytes.

span

- For example:



```
void PrintInfo(std::span<int> s)
{
    auto size = s.size();
    for (size_t i = 0; i < size / 2; i++)
        s[i] *= 2;
    for (size_t i = size / 2; i < size; i++)
        s[i] *= 3;

    for (auto it = s.begin(); it != s.end(); it++)
        std::cout << *it << ' ';
    std::cout << '\n';
    return;
}

int main()
{
    std::vector<int> a{ 1,2,3,4,5 };
    std::array<int, 4> b{ 1,2,3,4 };
    int c[]{ 1,2,3 };
    PrintInfo(a); PrintInfo(b); PrintInfo(c);

    return 0;
}
```

C:\WINDOW

```
2 4 9 12 15
2 4 9 12
2 6 9
请按任意键继
```

Notice that if you create a span from a vector, and the vector is reallocated (i.e. capacity changed), then the original span will be dangling!

Always remember that it's as unsafe as a pointer.

- You can also create a span with a `[begin, end)` iterator pair or `(begin, size)` pair.
- Notice that spans will ~~never~~ (C++26 adds `.at()`) check whether the accessed position is valid!
  - E.g. You can use out-of-range index for `operator[]`.
  - You should carefully manage it!
- Span is in fact `std::span<T, extent>`, but the `extent` is `std::dynamic_extent` by default.
  - For fixed extent, it's even more dangerous since you can assign a range that in fact doesn't have `extent` elements (but they need `explicit` construction).
    - Only C-style array (i.e. `T[extent]`) and `std::array<T, extent>` can implicitly construct it.
    - Also, you need `.first/last<N>()`, `.subspan<offs, N>()` to create subspan with fixed extent (`.subspan<offs>()` will create fixed/dynamic one for fixed/dynamic span).
  - You can get the extent by the static member `extent`.

For non-dynamic extent, object will only store a pointer because extent is just size.

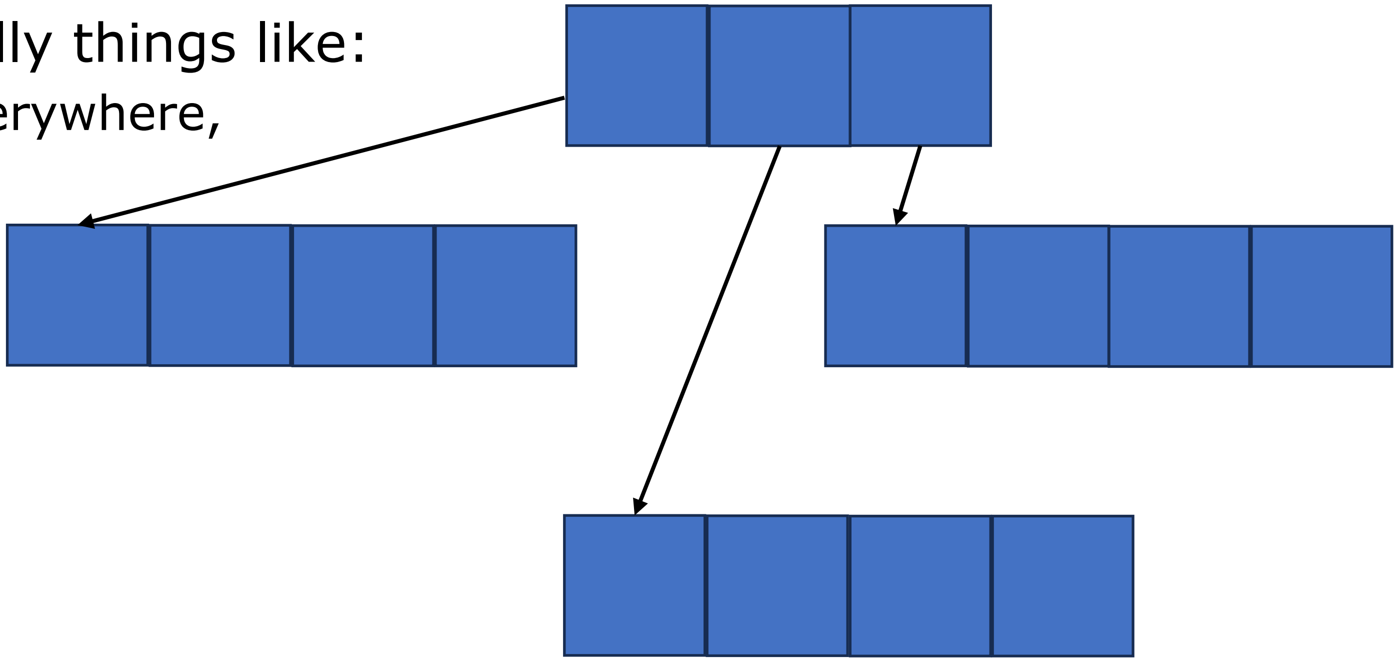
span

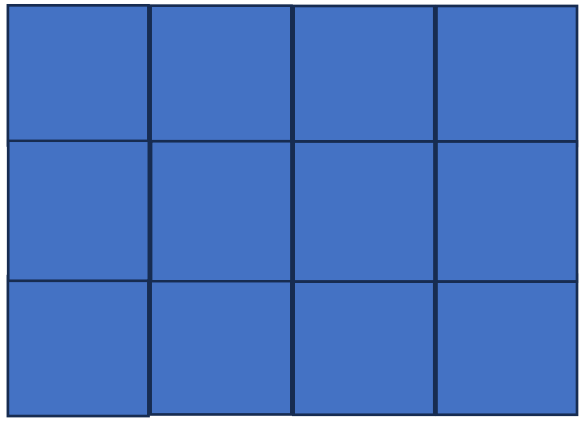
- Notice that since C++17/20, `std::data()/empty()/size()/ssize()` can be used to get the raw pointer, etc., just like `std::begin` to extract the iterator.
  - However, in C++20, you should prefer `std::range::data()/begin()/…`, which is safer and has many other advantages; we'll cover ranges in the next lecture.

- Final word: if you hope the span has read-only access, you need to use `std::span<const T>`.
  - This actually makes the pointer `const T*`, so it's read-only.
  - However, for containers, you need to specify as `const std::vector<int>`.
    - That's because spans are observer, and containers are the owner!

mdspan*

- You may find it's annoying to declare multi-dimensional array by containers…
  - `vector<vector<int>>` is actually things like:
    - The memory is scattered everywhere, which is unfriendly to cache.
  - We want multi-dimensional array to have contiguous memory like:



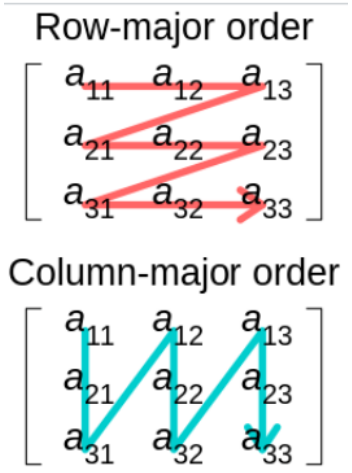


mdspan*

- When talking about multi-dimensional index `operator[]`, we've implemented a really simple multi-dimensional array, with a one-dimensional array as buffer and overloaded operator to access it as if a multi-dimensional array.
- Since C++23, a multi-dimensional span is also provided.
  - It's still a non-owning view, providing a multi-dimensional `operator[]`!
    - For owning one, `mdarray` may be provided in C++26…
  - It is rather complex to cover all contents, and the current library support is not sufficient, so we'll only introduce it.
    - This part is **optional**, and it's **totally** acceptable if you don't understand all of it!

mdspan*

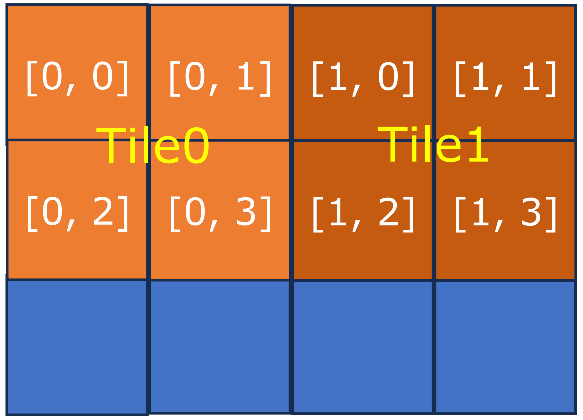
- There are three components for mdspan:
  - Extent: We need a multi-dimensional extent, too; so it's written as `std::extent<IndexType, sizes…>`.
    - E.g. `std::extent<std::size_t, 3, 2, 4>`.
    - Similarly, you may hope to make some dimensions dynamic, then you can still use `std::dynamic_extent`, e.g. `std::extent<std::size_t, std::dynamic_extent, 2, std::dynamic_extent>`.
    - Obviously, the most frequently used one is that all dimensions are dynamic, so you can abbreviate it as `std::dextent<IndexType, dimensionNum>`.
  - Layout: By default `std::layout_right`.
    - You may know that Fortran and C/C++ have different layouts in array.
      - In C/C++, the last dimension(i.e. the rightmost one) is contiguous.
        - Row major, i.e. rows are stored one by one.
      - In Fortran, the first dimension(i.e. the leftmost one) is contiguous.
        - Column major, i.e. columns are stored on by one.


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

mdspan*

- So, layout is used to regulate how you view the memory.
  - All in all, so-called multi-dimension is just a **mapping**: $(i_1 \cdots i_n) \rightarrow index$, where index is used to access the memory.
  - You can also use `std::layout_left` to use Fortran order or `std::layout_stride` to use special strides.
  - Sometimes, you may need even more dedicate control, e.g. as a sliding window (then, the mapping is possibly not injective, i.e. two md-index may map to the same index), as a tiled array, as a triangle matrix, etc.
    - Then you can customize your own layout policy!
    - You need to define a template struct, i.e. `LayoutPolicy::mapping<Extent>`, which specify lots of properties of the mapping.
      - Uniqueness: i.e. if it’s injective（单射）.
      - Exhaustion: i.e. if it’s surjective（满射）.
      - Stride: i.e. if there is a fixed stride in a dimension.
      - Methods are `is_always_xxx` (e.g. `is_always_unique, is_always_exhaustive, is_always_strided`) and `is_xxx`.

mdspan*



- For example, for a tiled array:
    - Notice: tiled array has many possible access way, and this is just a possible one.
  - It’s always unique, so `is_always_unique` and `is_unique` should `return true`;
  - It’s possibly not exhaustive, so `is_always_exhaustive` should `return false`; but when the tile can exactly cut the array, it’s definitely exhaustive, so `is_exhaustive` should `return (extents[0] % tile[0] == 0) && ...`;
  - It’s possibly not strided, because if tiles are in the same row, then the stride is fixed; if the tile goes into the next row, then the stride is different.
    - You can check stride by observing stride between [0, 0], [1, 0], …(for row stride) and [0, 0], [0, 1], …(for column stride), etc.
    - So `is_always_strided` should `return false`, and since basically tiled stride cannot be fixed, so `is_strided` should `return false`.
  - These properties exist for optimization, so you can `return false` even if there is some corner case to make it `return true`.
- Besides, you need to add some other functions and types in mapping. Since there is no official example temporarily, you can see an example here.

mdspan*

- The final component is accessor, which is `std::default_accessor` by default.
  - That is, when you get index, how do you access the memory?
    - The default one is just `return mem[i]`, but E.g. you may want to add a lock…
  - So, you need to define `access(pointer, index)` that returns a reference to the element and `offset(pointer, index)` that returns a pointer to the element.
  - This is rarely used, and default one is enough.
- You may think it too complex!
- **But all in all, the most frequently-used is just `std::mdspan<T, std::dextent<IndexType, DimNum>>`.**

Containers

- Sequential Containers
  - array
  - vector
  - bitset
  - span
  - deque
  - list
  - forward_list

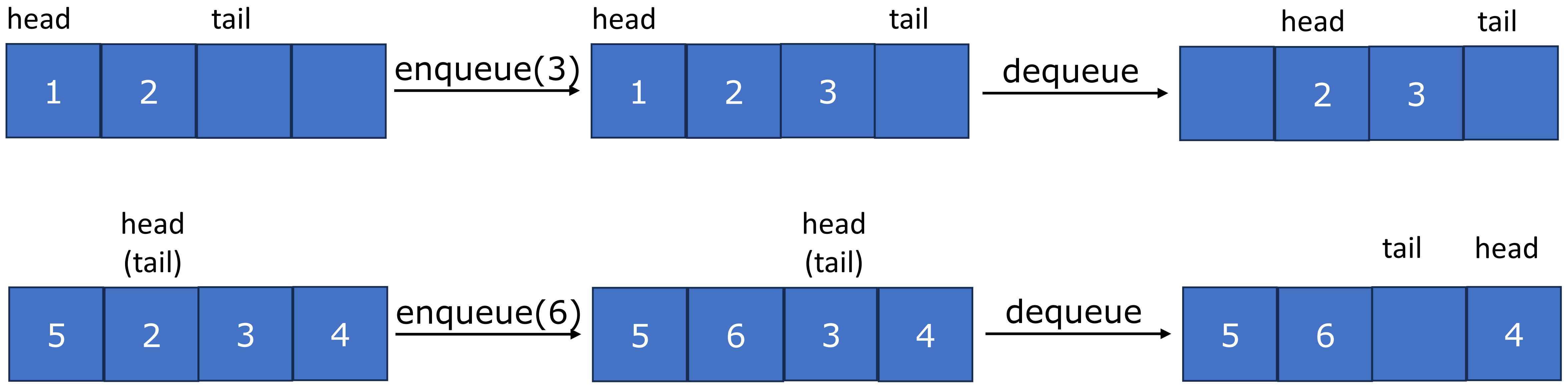
Deque

- **D**ouble-**E**nded **Que**ue
- The most significant requirement of deque is:
  - $O(1)$ on insertion & removal of elements at the front or the back.
  - Random access.
- Other properties are just like vector, e.g. $O(n)$ insertion & removal.
  - For methods, except that it provides `push_front, emplace_front, pop_front` (and `prepend_range` since C++23), all things (including ctors) are the same as a normal vector, so we'll not repeat it.
- What is important is how it's implemented.

Deque

- Before introducing the implementation of deque, we need a new data structure called ***circular queue***.
    - Queue is FIFO.
    - But in practice, usually the space is limited so we cannot always push into queue…
    - Circular queue allocates a fixed-size buffer, records a head and a tail.
        - When enqueue, tail moves forward.
        - When dequeue, head moves forward.
        - If tail == head i.e. the queue is full, overwrite the element at head, both tail and head move forward.
    - This is used widely, e.g. for prefetching prediction in hardware. If there are too many predictions, the oldest will be dropped.

Circular queue



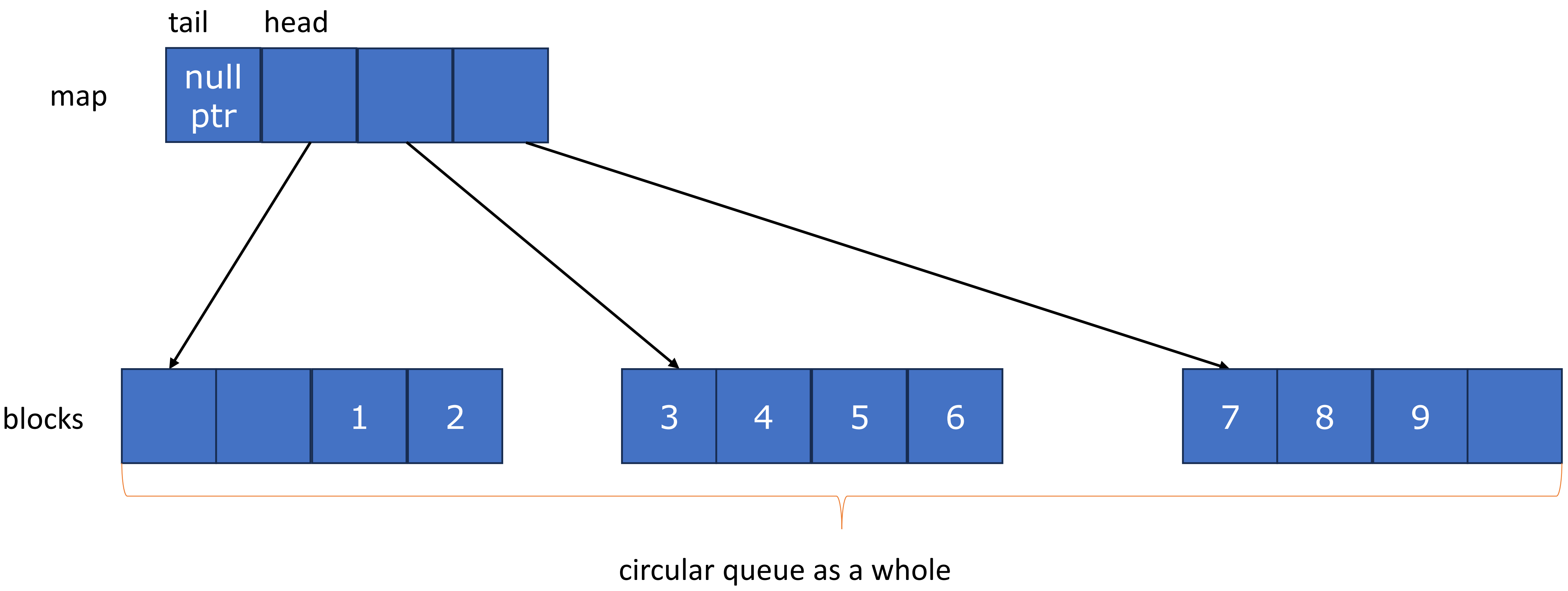
Deque

- So, can we use circular queue to emulate deque?
  - We've said how to enqueue from tail; you can easily know how to enqueue from head.
  - So obviously $O(1)$ insertion and removal!
  - However, deque shouldn't drop elements when full.
- So we can use **dynamic** circular queue; when it's full, the space should be enlarged and used to make a new circular queue.
  - Similar to vector, you need to expand space exponentially.
  - When you enqueue continuously, the amortized complexity is $O(1)$; dequeue is obviously $O(1)$.
  - You can also random access, e.g. `deque[i]` is just `vec[(head + i) % size]`.

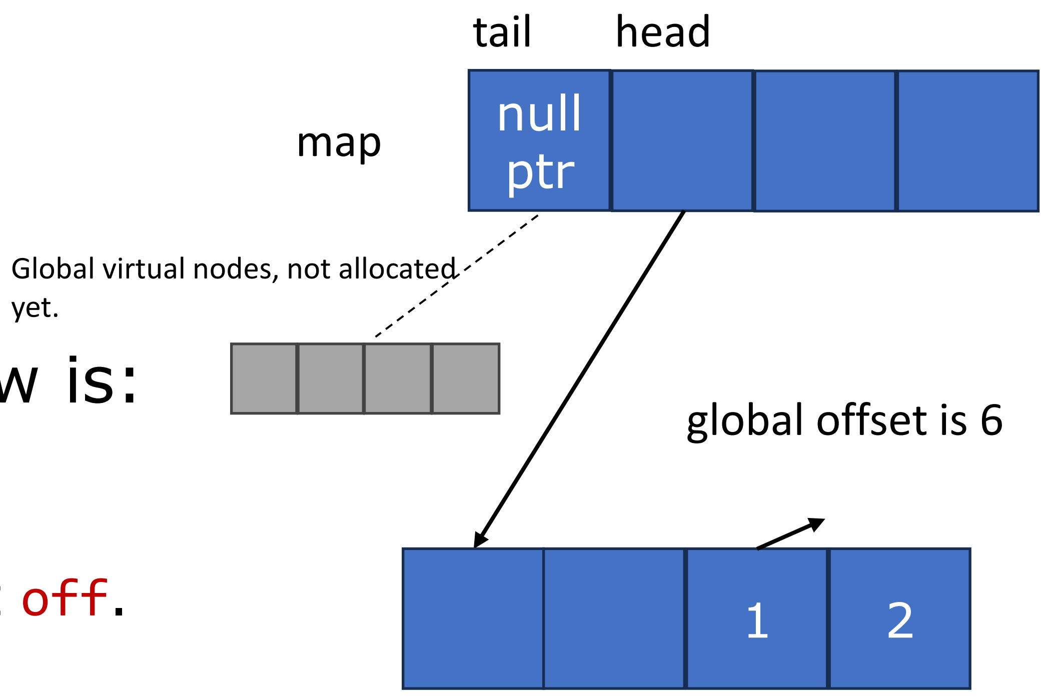
Deque

- But deque expects true $O(1)$ rather than amortized $O(1)$…
  - That seems quite impossible!
  - What deque implementation does is "For expensive copy of objects, the complexity is approximately $O(1)$".
  - If we only use dynamic circular queue, we need to copy all elements when resizing; that won't satisfy it.
  - The solution is to lower down the copy cost…But how is it possible?
- The typical implementation is using a dynamic circular queue (called **map**) whose elements are pointers.
  - Each pointer points to a block, with many objects stored there.
    - The block size is fixed, e.g. in libc++, that's `max(16*sizeof(obj), 4096)`; in libstdc++, that's `8*sizeof(obj)`.
  - You may think it as a big circular queue as a whole!

Deque


tail
head
map
null ptr
blocks
1
2
3
4
5
6
7
8
9
circular queue as a whole

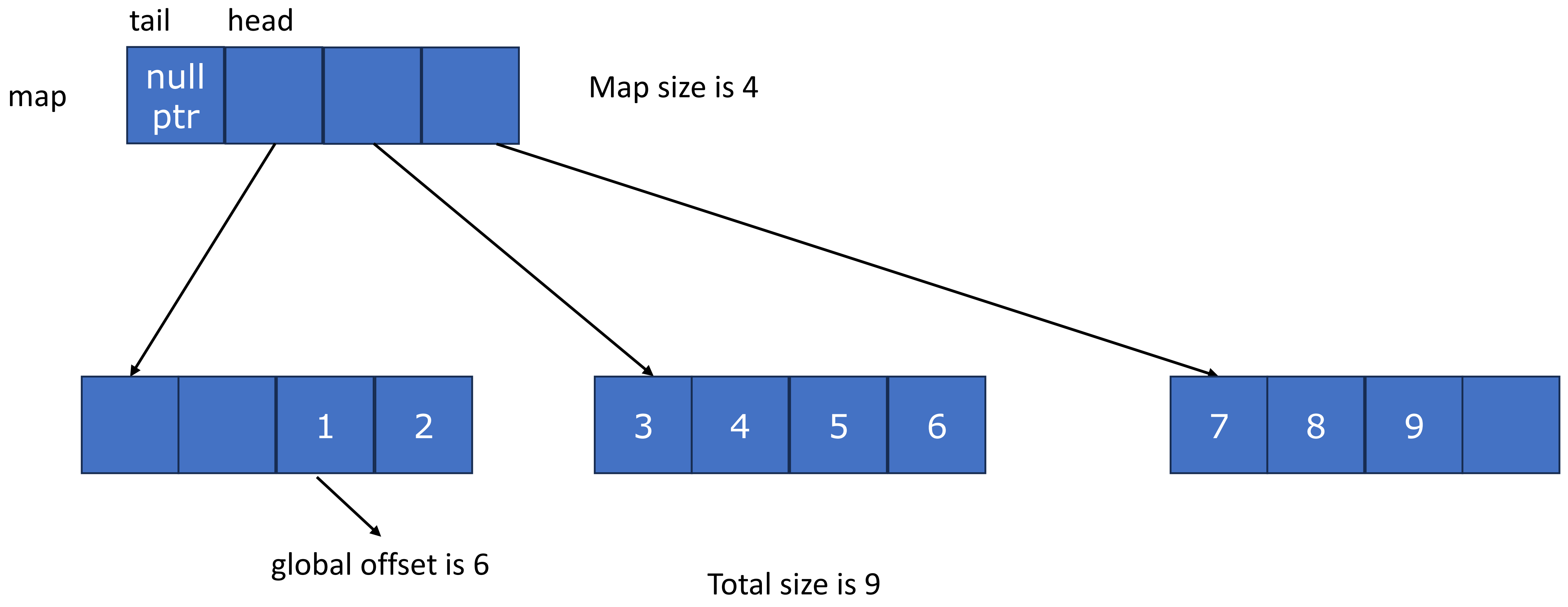
Deque



- What deque needs to record/know is:
  - The map and its size.
  - The block size.
  - The global offset of the first element `off`.
  - Element numbers.
- We can use `off / block_size` to know the position of head.
- When resizing, we just need to copy all pointers!
  - The number of pointers is $n\ /\ k$, and copying them is very cheap…
  - If object copy is expensive, this cost can be approximately seen $O(1)$.
  - Even if in the context of amortized complexity, it is $O(1) + O(pointer\ copy\ cost\ /\ k)$ rather than $O(1) + O(object\ copy\ cost)$, which is still cheaper.
  - Let me show you different cases…

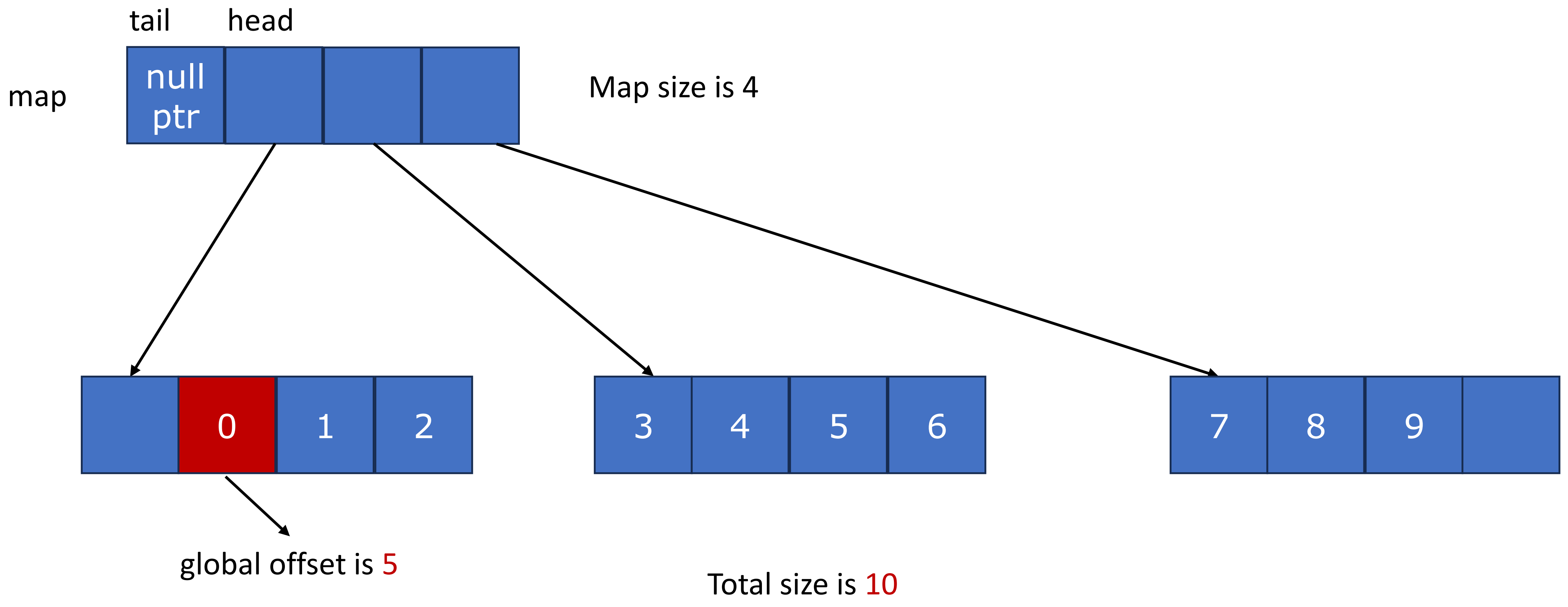
Deque

- Original:



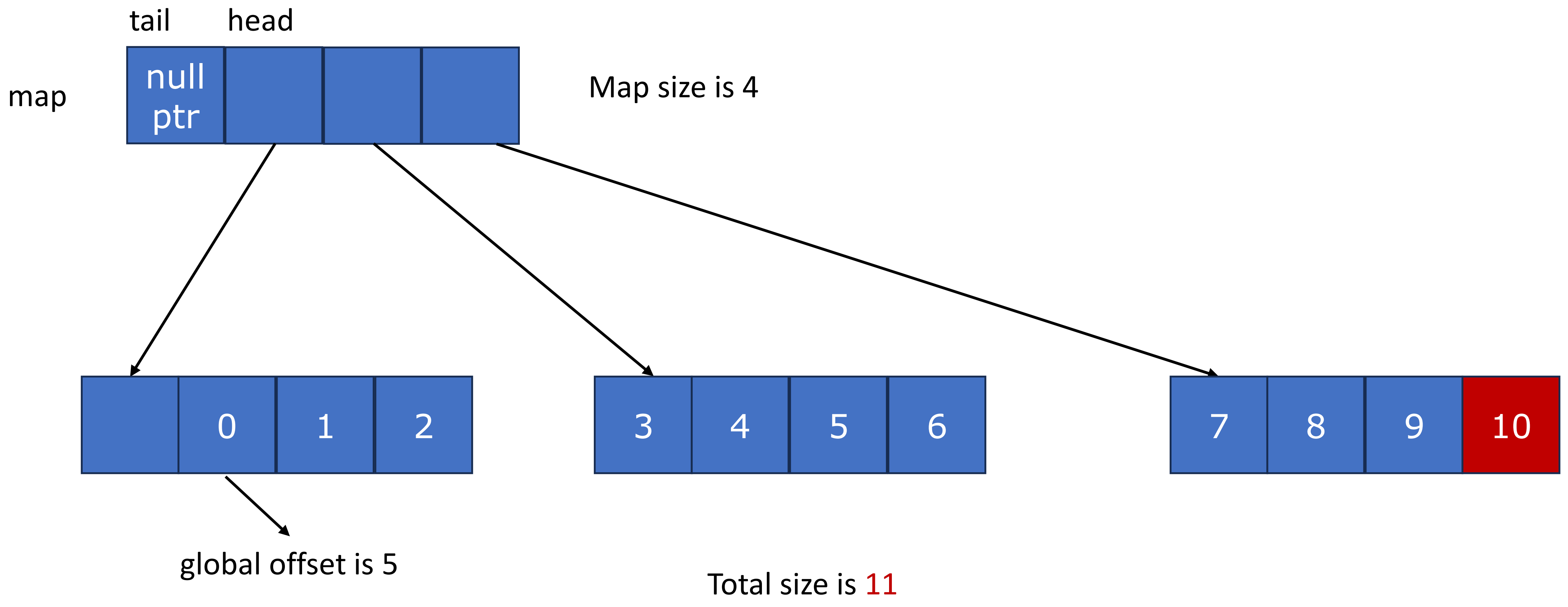
Deque

- Case1： push_front

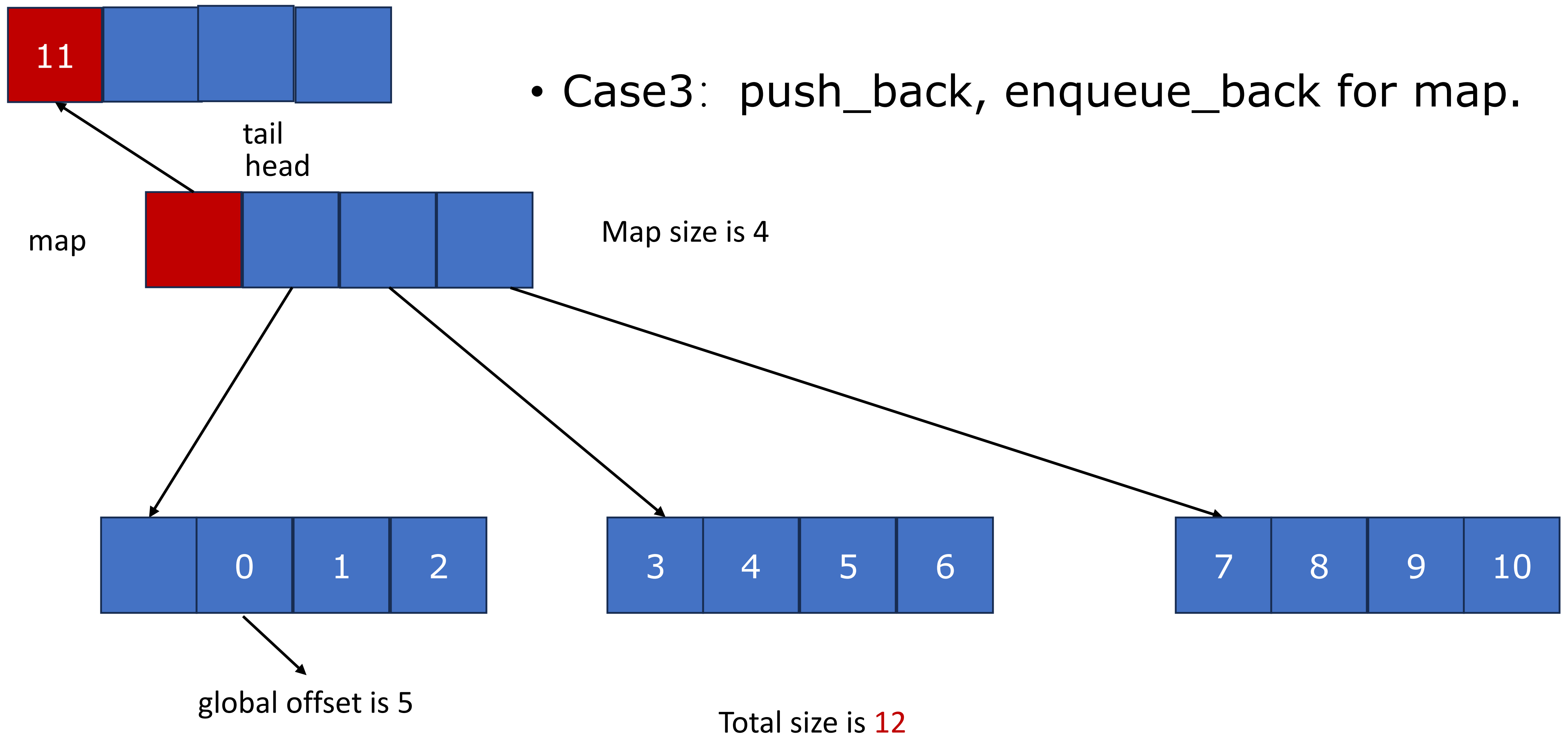


Deque

- Case2: push_back



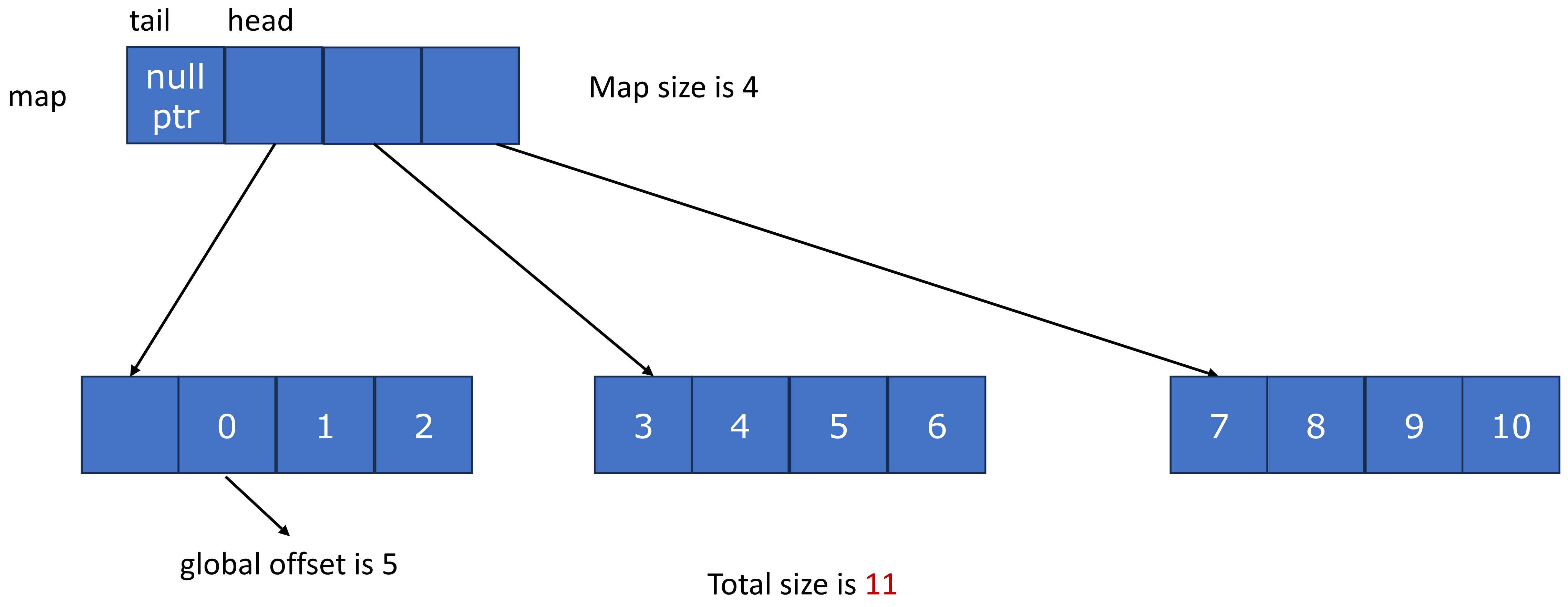
Deque



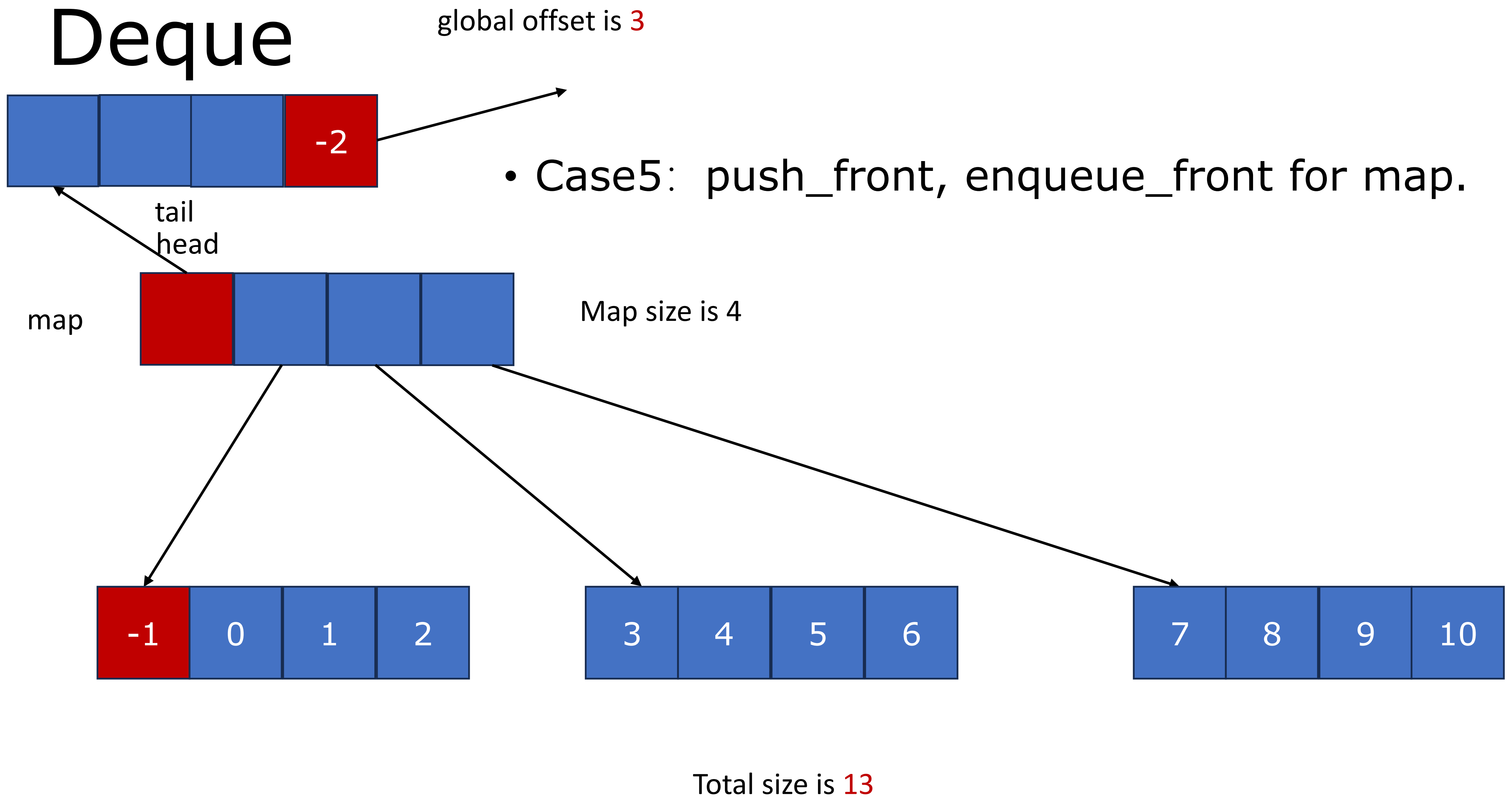
- Case3：push_back, enqueue_back for map.

Deque

- Case4: pop_back, also dequeue from back of map.



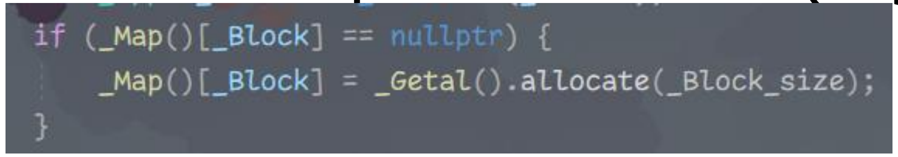
Deque



- Case5: push_front, enqueue_front for map.

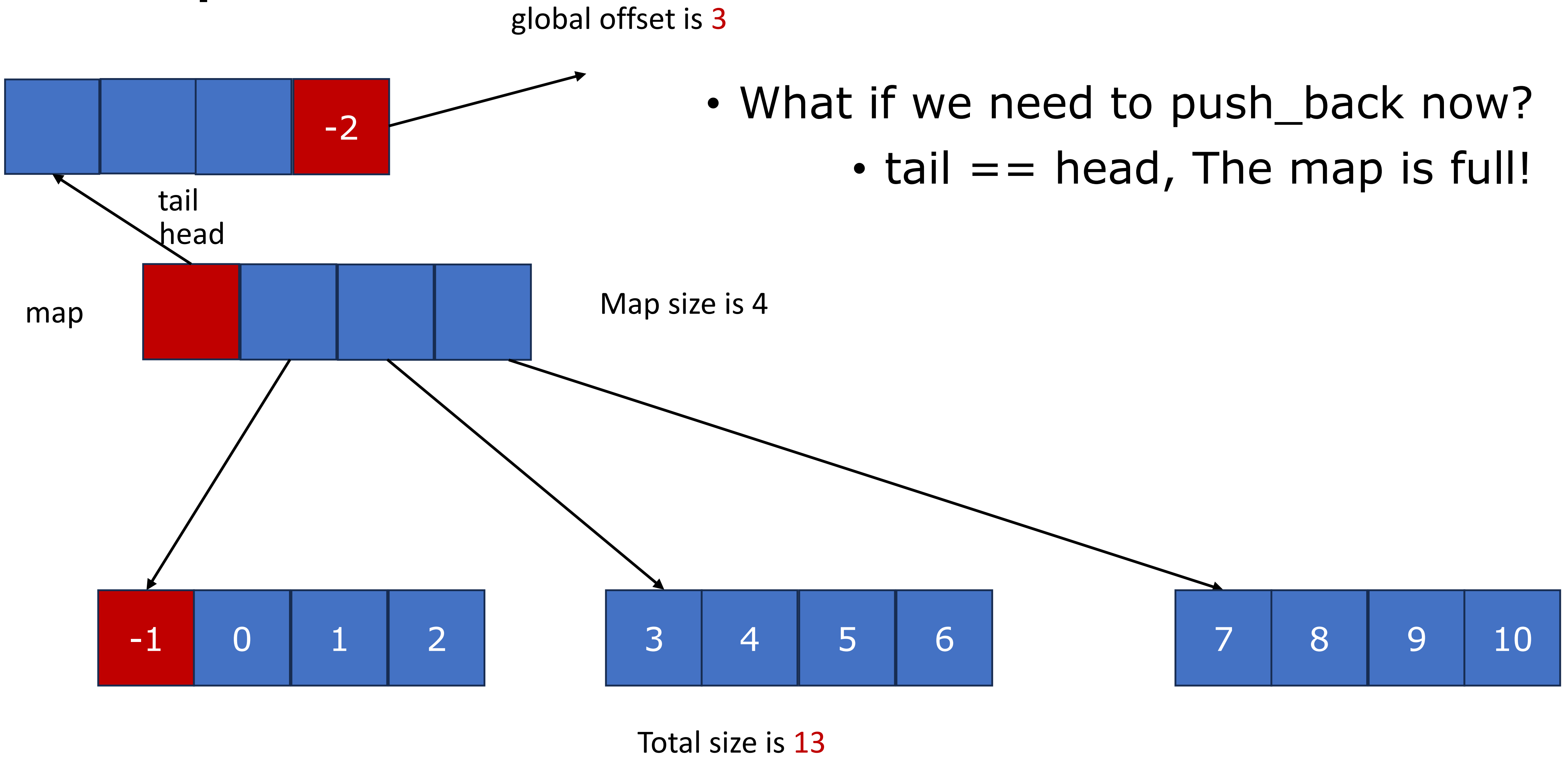
Deque

- Normal pop_front/back (i.e. without dequeue of map) is easy, you may think it yourself!
- You can also simulate pop_front when dequeuing map.
- Notice that it’s not forced to free the space of block (e.g. in MS implementation).
  - We just move tail & head.
  - This is kind of “lazy load”; resources are allocated only when we need, but may not be released.
  - When you find an element of map is not `nullptr`, you can directly use the block it points to.
    - Our strategy guarantees that valid data will not be overwritten.
  - You may use `shrink_to_fit`, just like vector, to free those unused blocks.
    - It may also shrink map, just like vector.



```
if (_Map()[_Block] == nullptr) {
    _Map()[_Block] = _Getal().allocate(_Block_size);
}
```

Deque



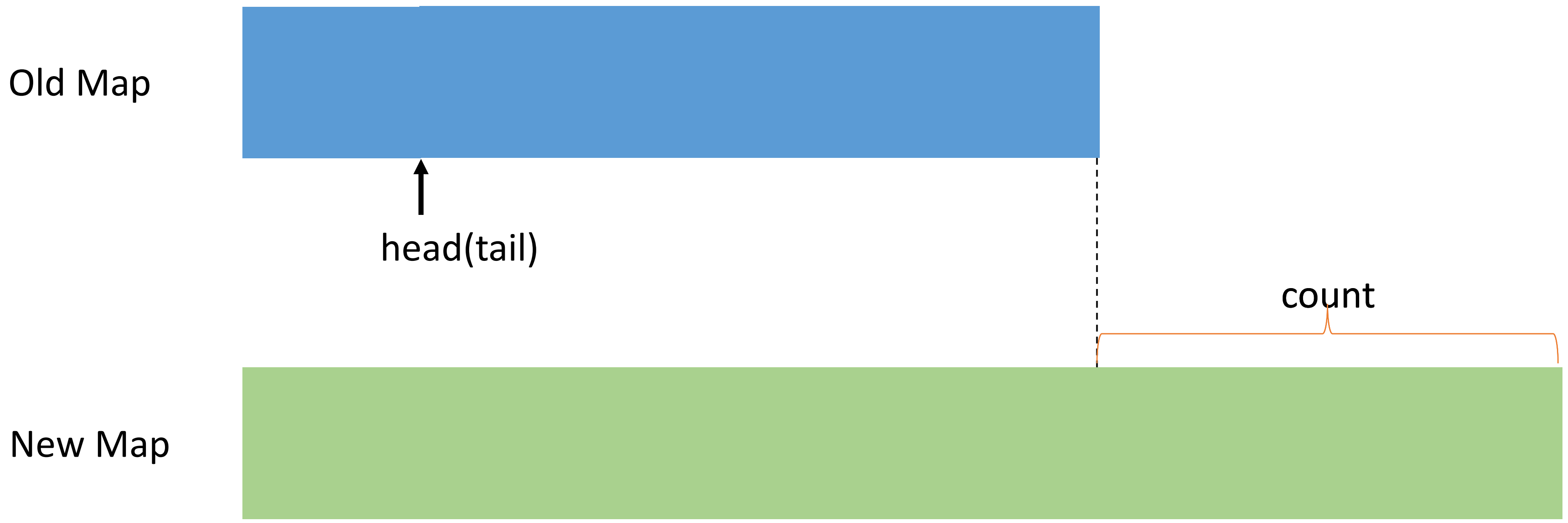
- What if we need to push_back now?
  - tail == head, The map is full!

Map Reallocation In Deque

- When the map is full, it should be reallocated…
  - We assume that newly added block number is `count`.
- Now, we just need to make the circular queue still continuous in the new vector.
- Procedures:
  - First, copy all elements from `vec[head, vecEnd)` to `newVec[head, currEnd)`;
    - Then, if `head <= count`, copy `[0, head)` to `[currEnd, …)`.
    - Else, copy after `currEnd` as much as possible, and the rest is arranged to the `newVecBegin`.
  - Finally, set all the rest to `nullptr`.
  - It’s kind of abstract, let me show you…

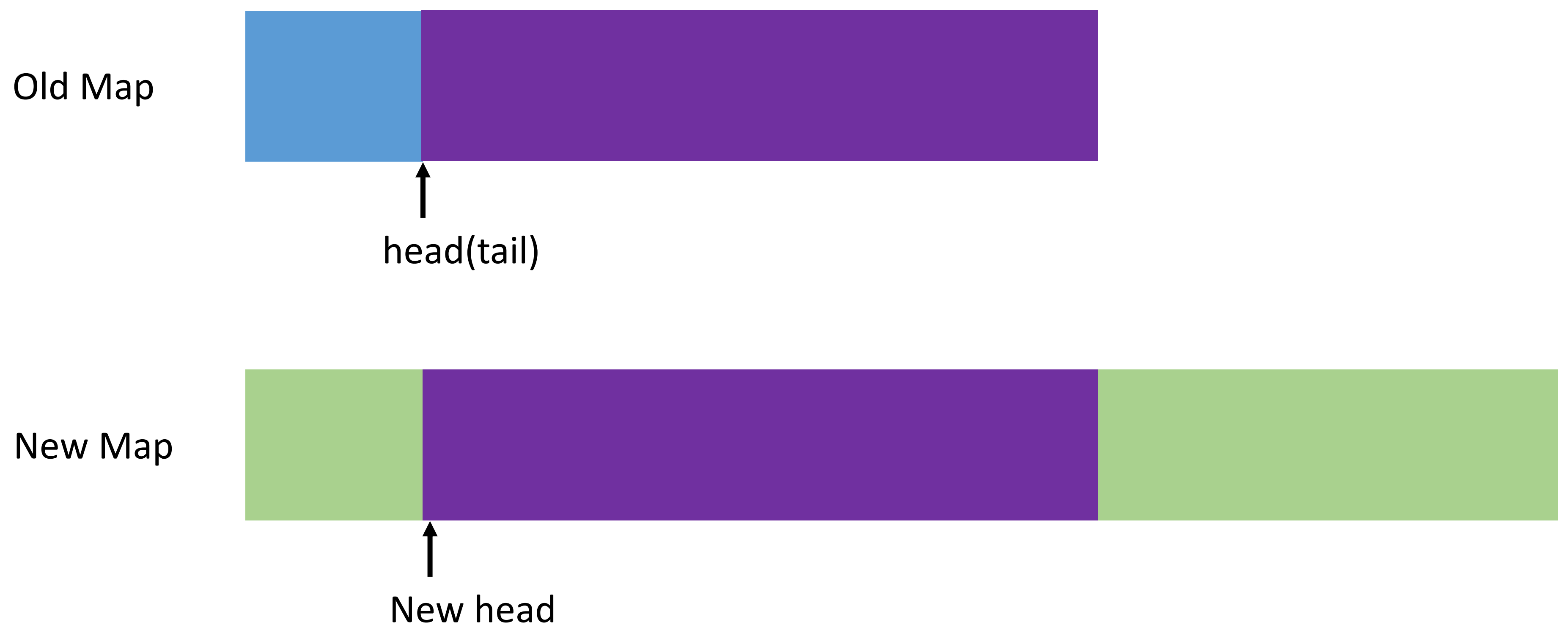
Map Reallocation In Deque

- Allocate new space…



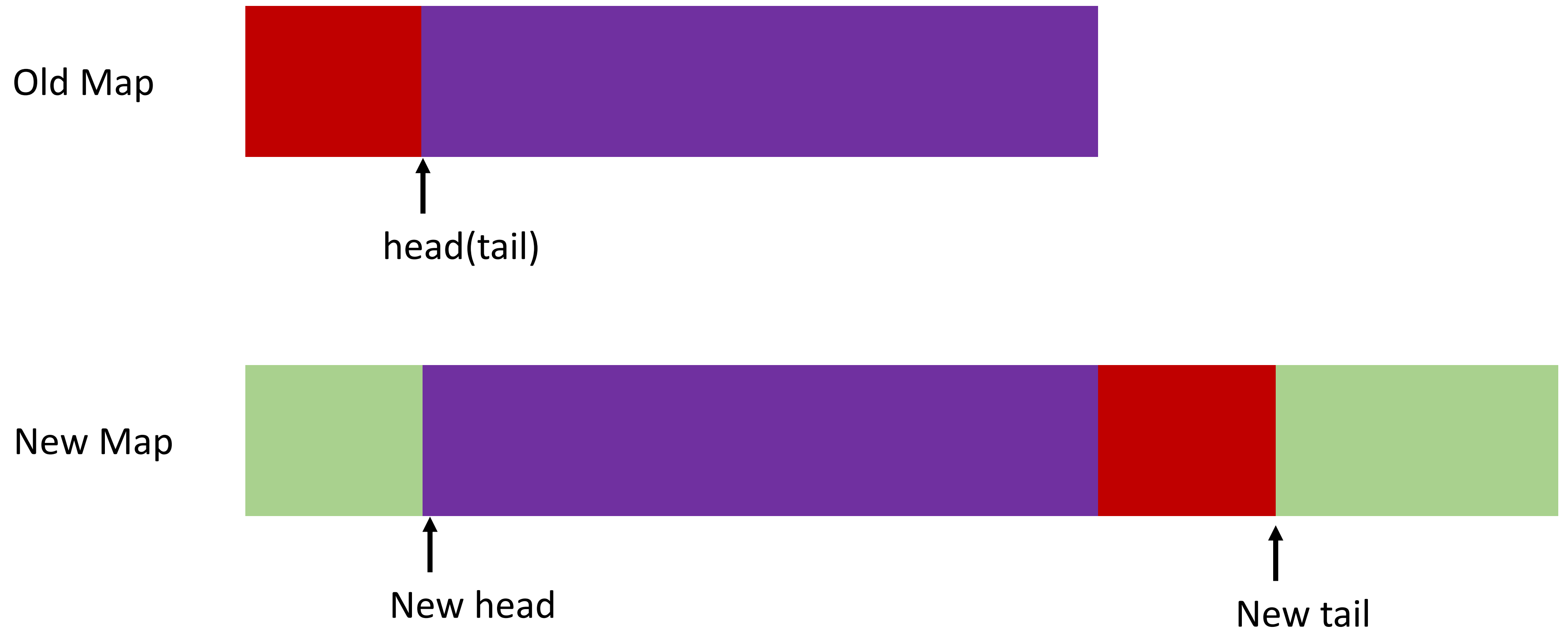
Map Reallocation In Deque

- Copy from old head to new head…



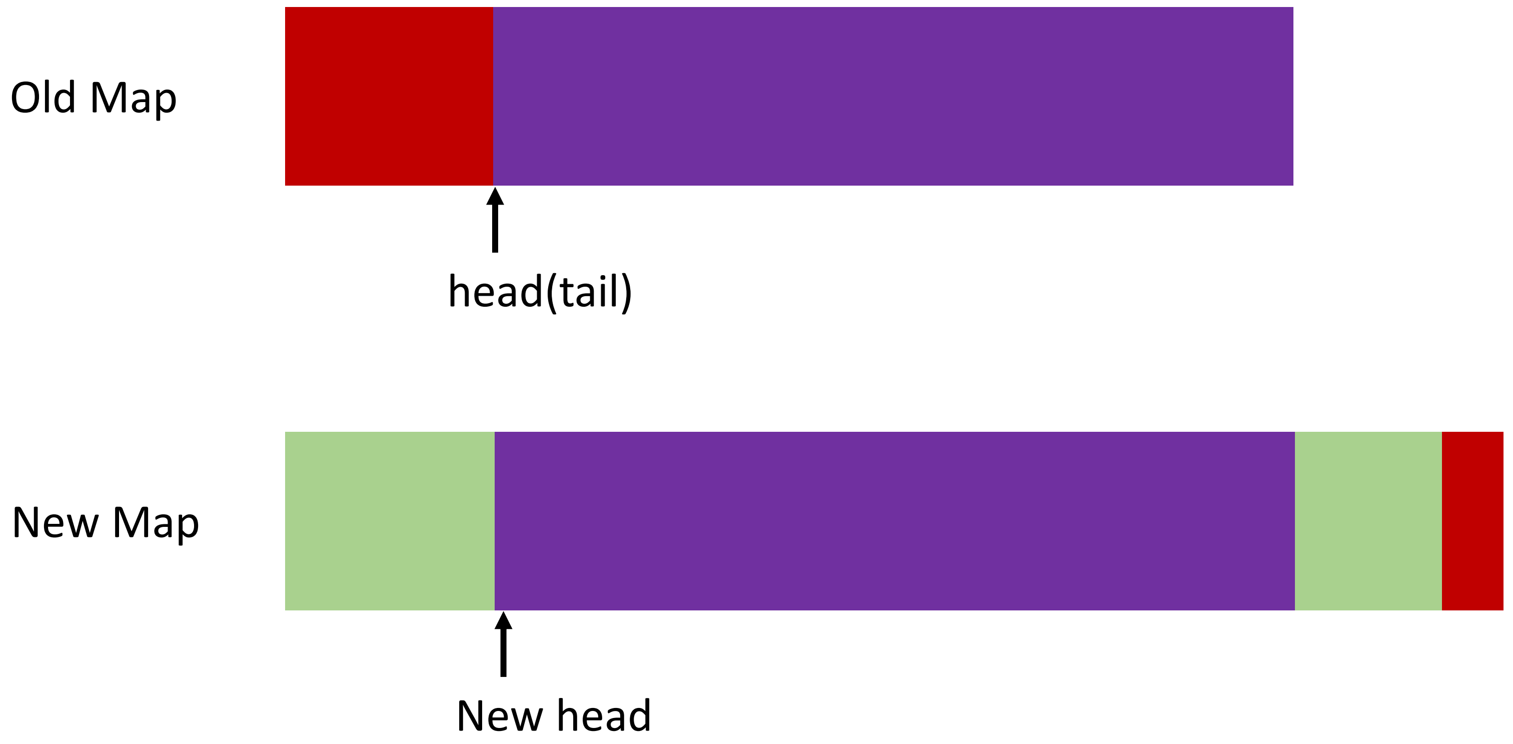
Map Reallocation In Deque

- Case1: `head <= count`



Map Reallocation In Deque

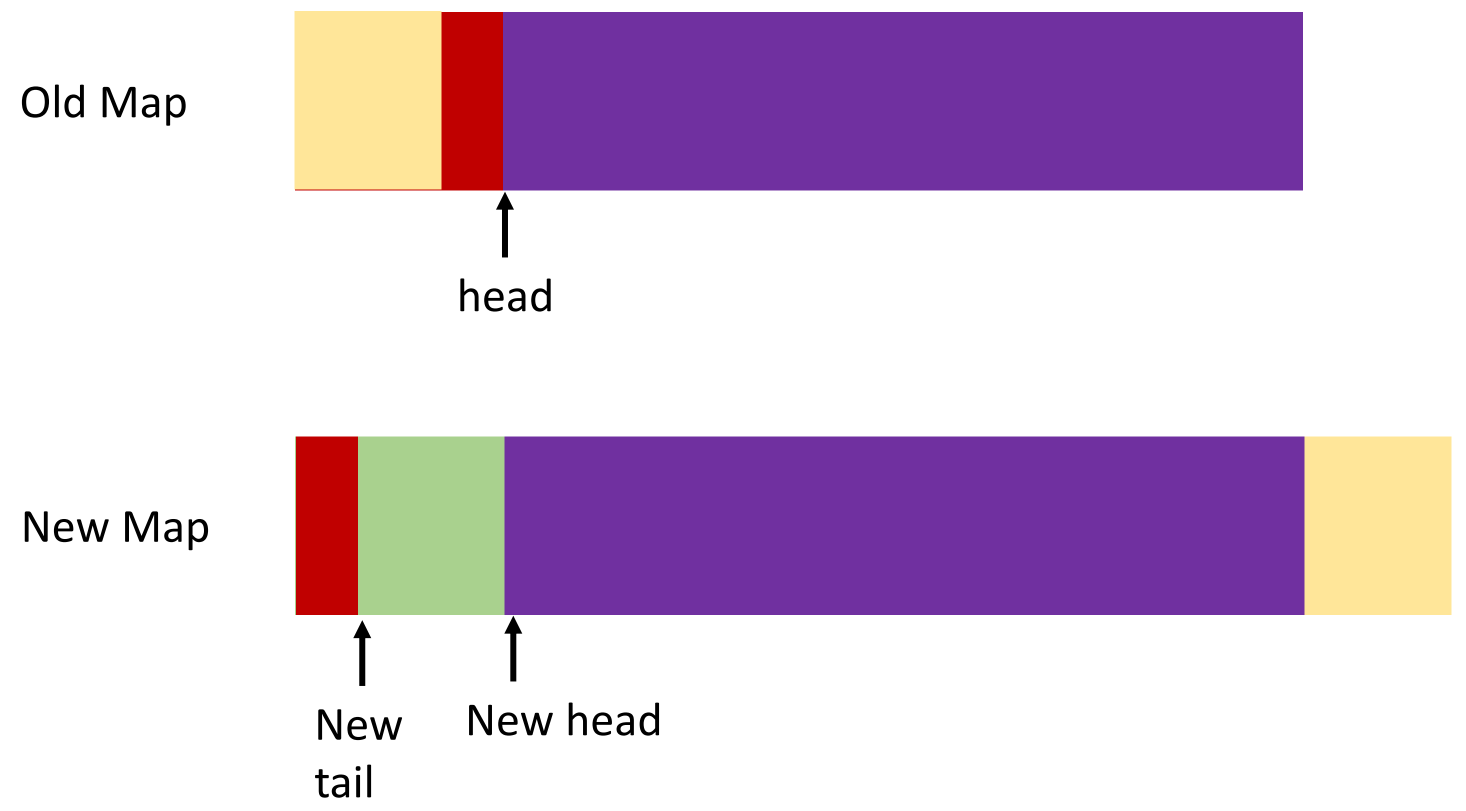
- Case2: `head > count`



Too small to fill in all...

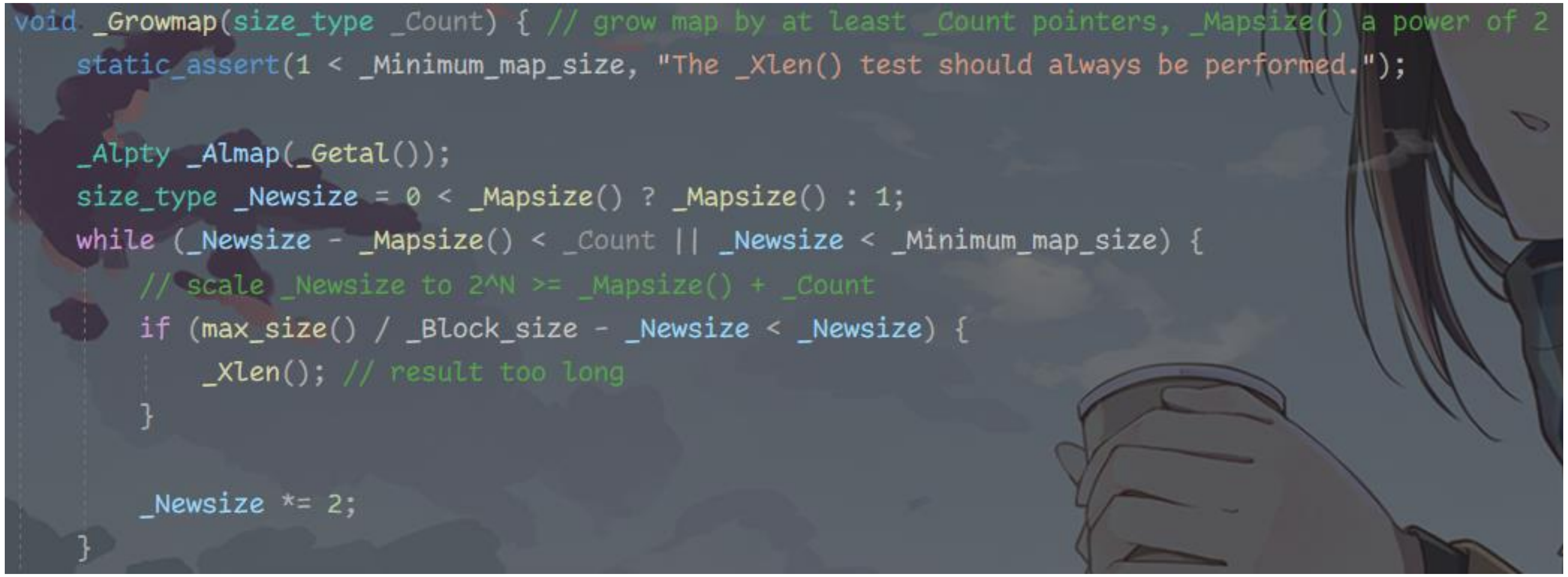
Map Reallocation In Deque

- Case2: `head > count`



Map Reallocation In Deque

- Let’s see code!
- Find a new size…



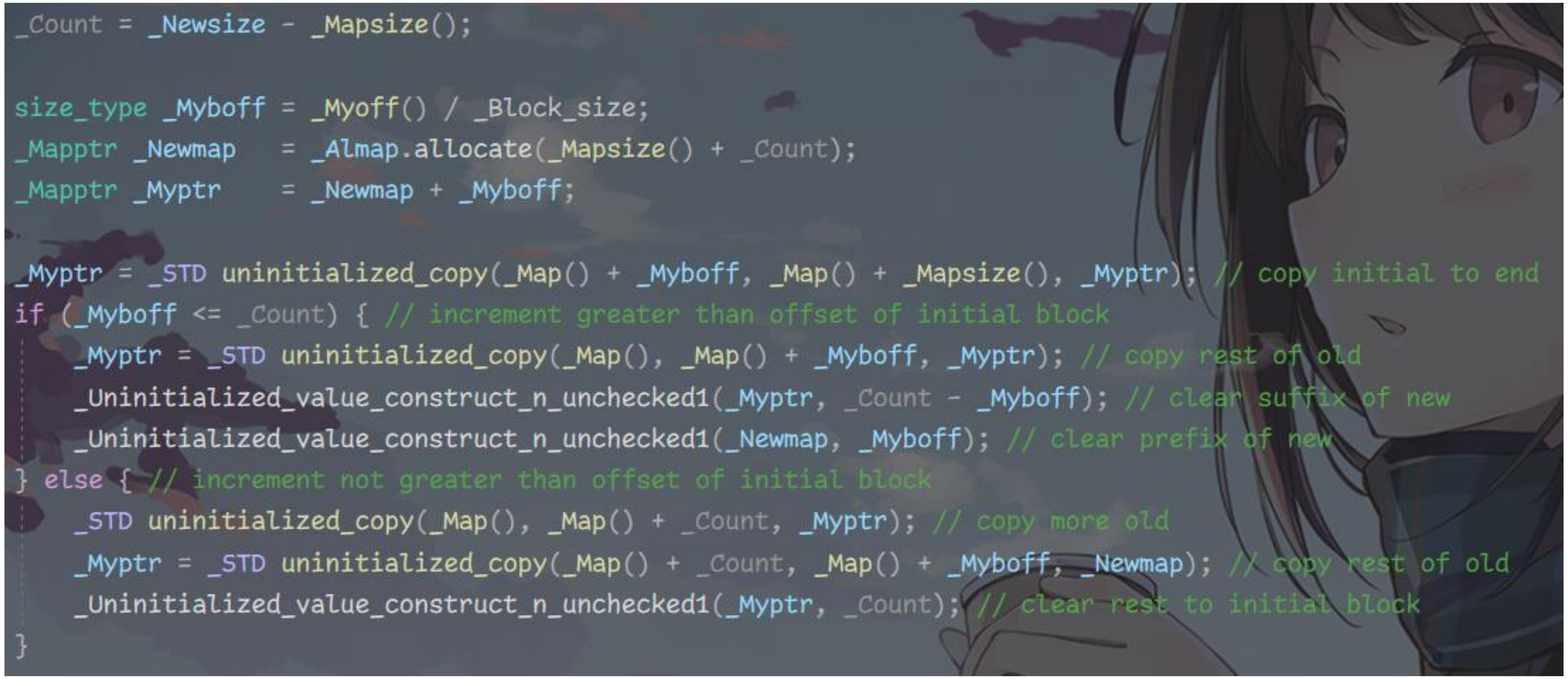
```
void _Growmap(size_type _Count) { // grow map by at least _Count pointers, _Mapsize() a power of 2
    static_assert(1 < _Minimum_map_size, "The _Xlen() test should always be performed.");

    _Alpty _Almap(_Getal());
    size_type _Newsize = 0 < _Mapsize() ? _Mapsize() : 1;
    while (_Newsize - _Mapsize() < _Count || _Newsize < _Minimum_map_size) {
        // scale _Newsize to 2^N >= _Mapsize() + _Count
        if (max_size() / _Block_size - _Newsize < _Newsize) {
            _Xlen(); // result too long
        }

        _Newsize *= 2;
    }
```

Map Reallocation In Deque

- Copy old to new…



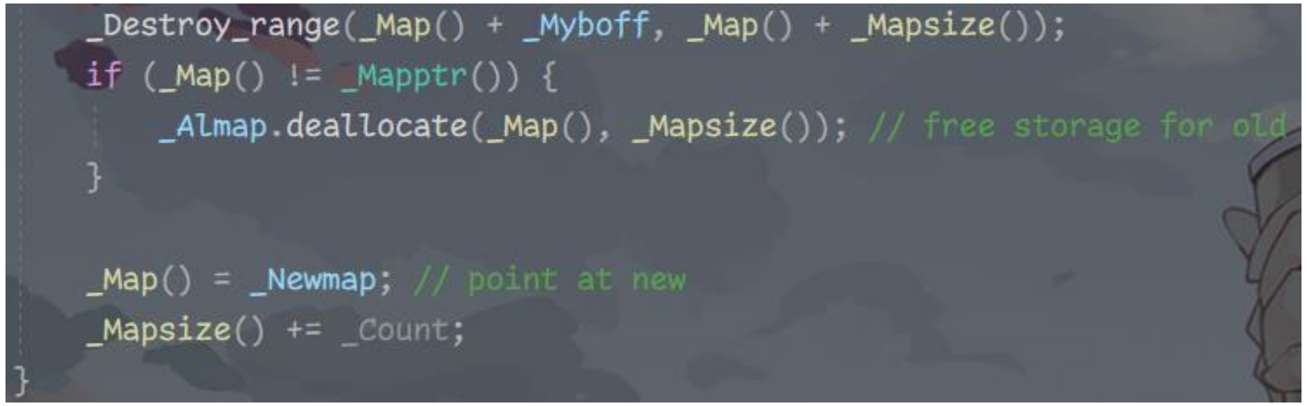
```
_Count = _Newsize - _Mapsize();

size_type _Myboff = _Myoff() / _Block_size;
_Mapptr _Newmap   = _Almap.allocate(_Mapsize() + _Count);
_Mapptr _Myptr    = _Newmap + _Myboff;

_Myptr = _STD uninitialized_copy(_Map() + _Myboff, _Map() + _Mapsize(), _Myptr); // copy initial to end
if (_Myboff <= _Count) { // increment greater than offset of initial block
    _Myptr = _STD uninitialized_copy(_Map(), _Map() + _Myboff, _Myptr); // copy rest of old
    _Uninitialized_value_construct_n_unchecked1(_Myptr, _Count - _Myboff); // clear suffix of new
    _Uninitialized_value_construct_n_unchecked1(_Newmap, _Myboff); // clear prefix of new
} else { // increment not greater than offset of initial block
    _STD uninitialized_copy(_Map(), _Map() + _Count, _Myptr); // copy more old
    _Myptr = _STD uninitialized_copy(_Map() + _Count, _Map() + _Myboff, _Newmap); // copy rest of old
    _Uninitialized_value_construct_n_unchecked1(_Myptr, _Count); // clear rest to initial block
}
```

Map Reallocation In Deque

- Destroy old map and change members to new map!



```
    _Destroy_range(_Map() + _Myboff, _Map() + _Mapsize());
    if (_Map() != _Mapptr()) {
        _Almap.deallocate(_Map(), _Mapsize()); // free storage for old
    }

    _Map() = _Newmap; // point at new
    _Mapsize() += _Count;
}
```

Deque

- Insertion and erasure are all $O(n)$:
  - Their implementation is also similar to vector.
    - Insert by pushing and rotating.
    - Erase by moving and popping.
    - Particularly, since deque can be pushed in both sides, the closer one will be chosen to push/pop. So to be exact, the complexity is $O(closer\_distance)$.
- The iterator is just a deque pointer with an offset.
  - * is `deque->map_[offset / block_size][offset % block_size]`.
  - +/-/++/-- is just operating offset (need to round back to 0 when reaching the total end). It may also be checked whether it exceeds tail/head to see the validity of iterator.

Deque Iterator Invalidation

- From the view of vector, insertion will only invalidate elements "after" the insertion point.
  - However, that's because vector is always inserted with `push_back`; deque may use `push_front` to reduce complexity. You cannot assume the invalidation happens before or after the insertion point.
  - Besides, the map may be resized, so even if only `push_back/front`, the original offset is also not guaranteed to be corrected.
    - E.g. tail < head before, but after resizing, the tail is copied beyond head so that tail > head then.
  - Thus, **all iterators are seen as invalid after insertion**.
    - This includes `resize` when the size is growing; also includes `shrink_to_fit` and `clear` since it may change map size (e.g. in MS, `clear` will drop both elements and map)…
- Erasing from the front and back will only invalidate the erased elements, otherwise all iterators are also invalidated.
  - This includes `resize` when the size is reducing.

Final word

- References to elements are not invalidated when operating from front/back (including e.g. insert(end)) since blocks always remain unchanged. The only changed part is map.
  - Of course, references to removed elements are invalidated; this is necessary and obvious.
  - vector cannot keep references since the buffer itself has changed.

Containers

- Sequential Containers
  - array
  - vector
  - bitset
  - span
  - deque
  - list
  - forward_list

List

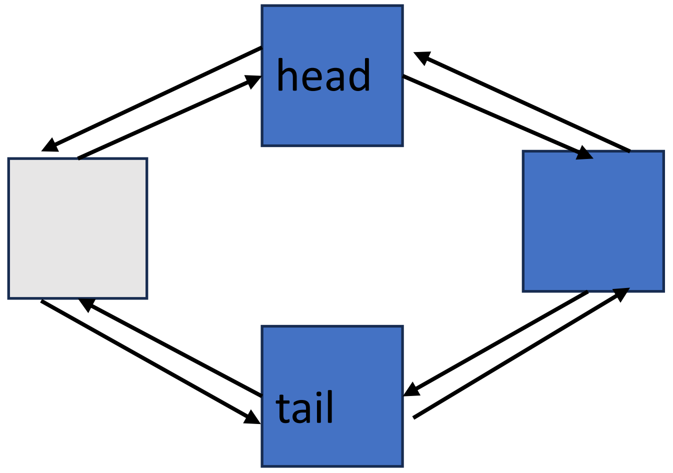
- Double linked list
- Though list is an important data structure, personally I really hardly ever use it.
    - `forward_list` is even more rare. I never use it before.
- List has these properties:
    - $O(1)$ insertion and removal.
    - $O(1)$ splice.
    - No random access.
- The implementation is just similar as we've learnt, so we mainly cover APIs.

List

- We know that double linked list is consisted of nodes.
  - Each node has a `T data`, a pointer to the previous node `prev` and a pointer to the next node `next`.
  - Particularly, `prev` of the first node is `nullptr`, and `next` of the last node is `nullptr`.
    - If you've written a double linked list before, you'll find that it's really annoying to process corner case…
- So in MS implementation, list is implemented as a circular list.
  - That is, we introduce **a sentinel node**, which is `prev` of the first node and `next` of the last node.
  - This will unify corner case and reduce code difficulty hugely, because it's not `nullptr`, but a virtual node that can be operated.
  - So totally, list stores the sentinel node(or its pointer) and size(to make `size()` $O(1)$, though you can count it in $O(n)$). Other nodes are dynamically allocated and linked together.

List

sentinel



- For example:

```
void addAtHead(int val) {
    if (head == nullptr)
    {
        head = new LinkedListNode();
        head->next = head->prev = nullptr;
        head->value = val;
    }
    else
    {
        head->prev = new LinkedListNode();
        head->prev->next = head;
        head = head->prev;
        head->prev = nullptr;
        head->value = val;
    }
    return;
}
```

No-sentinel version

```
void addAtHead(int val) {
    // We omit possible exception thrown by new.
    LinkedListNode* oldHead = m_sentinel.next;
    LinkedListNode* newHead = new LinkedListNode(&m_sentinel, oldHead, val);
    m_sentinel.next = newHead;
    oldHead->prev = newHead;

    return;
}
```

Sentinel version
(here it’s only int, so we use val of sentinel to store size)

https://leetcode.cn/problems/design-linked-list/description/

List

- The iterator is just a wrapper of the node.
  - `--/++` is going to prev/next.
  - `end()` is the sentinel node.
- So, it’s easy to understand iterator invalidation in list!
  - Only erasure, and it only invalidates the erased node.
  - However, it’s still thread-unsafe, e.g. erase two adjacent nodes.

List

- For member accessing:
  - `front()/back()`: get the first/last element of list.
  - Bidirectional iterators, as we stated.
- For size operations:
  - `.size()`: get size, return `size_t`.
  - `.empty()`: get a `bool` denoting whether `size == 0`.
  - `.max_size()`: get maximum possible size in this system (usually useless).
  - `.resize(n, obj=Object{})`: make the `size = n`;
    - If the original size is `n`, nothing happens.
    - If greater than `n`, elements in `[n, end)` will be removed.
    - If less than `n`, new elements will be inserted at back, and they are all `obj`.
  - `.clear()`: remove all things; size will be 0 after this.

List

- Here is same as deque, all $O(1)$:
  - `.push_back(obj)`: insert an element at the end.
  - `.emplace_back(params)`: insert an element **constructed** by `params` at the end.
    - Since C++17, it returns reference of inserted element(before it’s `void`).
  - `.pop_back()`: remove an element from the end.
  - `.push_front(obj)`
  - `.emplace_front(params)`
  - `.pop_front()`.
- Since C++23, $O(len(range))$
  - `.assign_range(Range)`
  - `.append_range(Range)`
  - `.prepend_range(Range)`
  - `.insert_range(const_iterator pos, Range)`

List

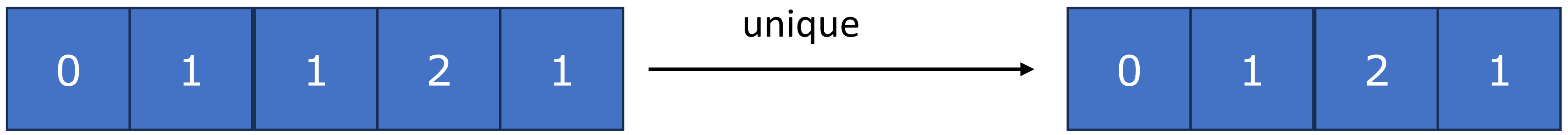
- Same as vector:
  - `.insert(const_iterator pos, xxx)`: `xxx` is similar to params of ctor:
    - `(value)`: insert a single element.
    - `(size_t count, value)`: insert `count` copies of `value`.
    - `(InputIt first, InputIt last)`: insert contents from `[first, last)`.
    - `(std::initializer_list<T>)`: insert contents of initializer list.
  - `.erase(const_iterator pos)`/`.erase(const_iterator first, const_iterator last)`: erase a single element/ elements from `[first, last)`. `first, last` should be iterators of this vector.
  - Insertion returns the iterator referring to the first inserted element, and removal returns the one for the next element of the erased element, too.

List

- Ctor:
  - Default ctor.
  - Copy ctor & Move Ctor.
  - `(size_t count, const T& elem = T{})`: construct a vector with `count` copies of `elem`.
  - `(InputIt first, InputIt last)`: copy elements from `[first, last)` into the vector.
  - `(std::initializer_list<T>)`: copy all elements from `initializer_list` into the vector.
  - All elements have an optional allocator parameter.
- Interact with another list:
  - `.assign`: also similar to ctor
    - `(count, const T& value)`
    - `(InputIt first, InputIt last)`
    - `(std::initializer_list<T>)`.
  - `.swap(list)`: swap with another list, same as `std::swap(list1, list2)`.
  - `operator<=>`.

List

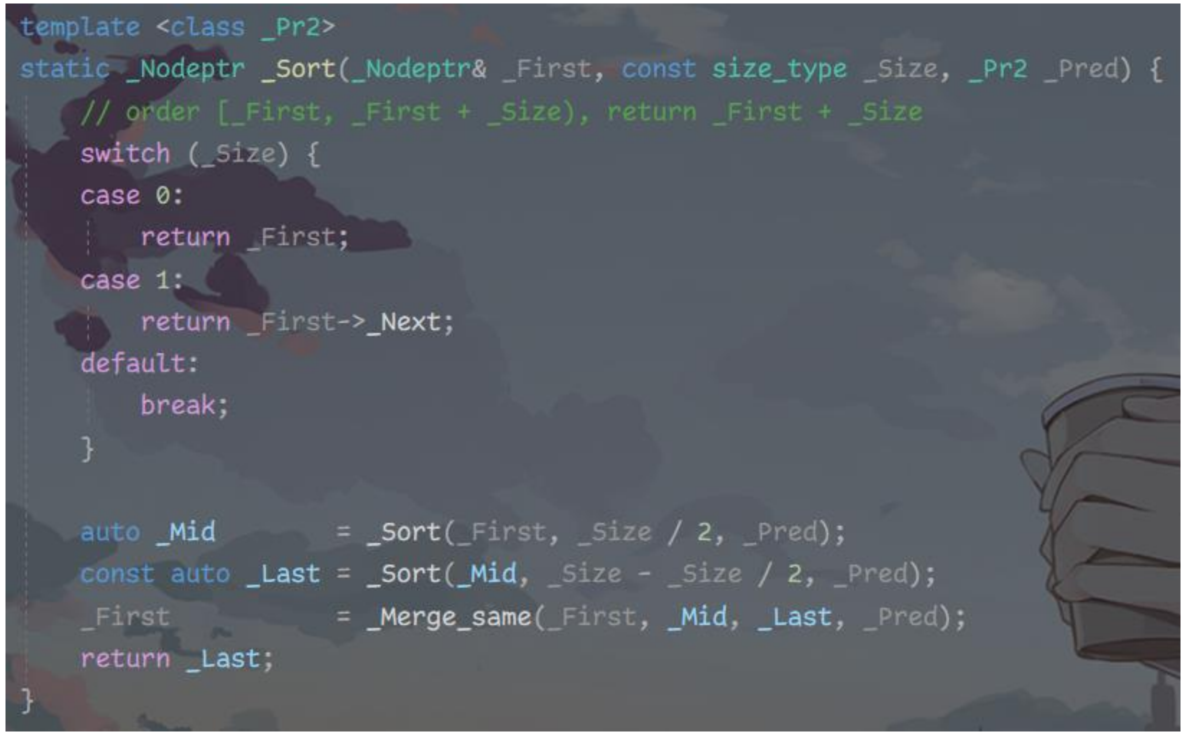
- Now we finally introduce some unique APIs for list…
  - `.remove(val)/.remove_if(func)`: remove all elements that is `val` or can make `func` return `true`.
  - `.unique()/.unique(func)`: remove equal (judged by `==/func`) **adjacent** elements.



  - They will return number of removed elements since C++20.
  - `.reverse()`: reverse the whole list.
  - `<algorithm>` also has these methods, but they don't erase nodes from the list, and are less efficient.
    - We'll tell you why in the next lecture…

List

- `.sort/.sort(cmp)`: stable sort.
  - MS implementation is merge sort, $O(1)$ space complexity and $O(n \log n)$ time complexity.
  - `sort` in `<algorithm>` needs random iterator, and usually not merge sort because its bad space complexity for vector ($O(n)$).



```
template <class _Pr2>
static _Nodeptr _Sort(_Nodeptr& _First, const size_type _Size, _Pr2 _Pred) {
    // order [_First, _First + _Size), return _First + _Size
    switch (_Size) {
    case 0:
        return _First;
    case 1:
        return _First->_Next;
    default:
        break;
    }

    auto _Mid        = _Sort(_First, _Size / 2, _Pred);
    const auto _Last = _Sort(_Mid, _Size - _Size / 2, _Pred);
    _First           = _Merge_same(_First, _Mid, _Last, _Pred);
    return _Last;
}
```

List

- There are two methods to **move** nodes from another list:
  - That means, another list will not own these nodes anymore.
    - `.insert(pos, it1, it2)` will not change the ownership of nodes; it’s just copy!
  - `.merge(list2)/.merge(list2, cmp)`: same procedure of merge in merge sort, so usually used in sorted list.
    - Two sorted list will be merged into a single sorted list.
  - `.splice(pos, list2, …)`:
    - `()`: insert the total `list2` to `pos`.
    - `(it2)`: insert `it2` to `pos` (and remove it from `list2`).
      - `it2` should come from `list2`.
    - `(first, last)`: insert `[first, last)` to `pos` (and remove them from `list2`).
      - `first, last` should come from `list2`.
    - You may notice that just removing some nodes from a list and moving them to another doesn’t need `list` itself!
      - Then why the last two need to provide a `list2`?
      - Because `list2` records size!

List

```cpp
int main ()
{
    std::list<int> list1{1, 2, 3, 4, 5};
    std::list<int> list2{10, 20, 30, 40, 50};

    auto it = list1.begin();
    std::advance(it, 2);

    list1.splice(it, list2);

    std::cout << "list1:" << list1 << '\n';
    std::cout << "list2:" << list2 << '\n';

    list2.splice(list2.begin(), list1, it, list1.end());

    std::cout << "list1:" << list1 << '\n';
    std::cout << "list2:" << list2 << '\n';
}
```

```
list1: 1 2 10 20 30 40 50 3 4 5
list2:
list1: 1 2 10 20 30 40 50
list2: 3 4 5
```

Containers

- Sequential Containers
  - array
  - vector
  - bitset
  - span
  - deque
  - list
  - forward_list

Forward list

- Single linked list.
- The purpose of forward list is reducing space, so it doesn't record an additional size and doesn't provide `.size()`.
    - If you really need it, you can use `std::(ranges::)distance(l.begin(), l.end())`, but remember it's $O(n)$.
- It's implemented in the same way as we've learnt; the node has `T data` and pointer to the next node `next`.
    - It doesn't have a sentinel; `next` of the last node is `nullptr`.
- Its APIs are almost same as `list`.
    - But, considering that `prev` doesn't exist in `forward_list`, **you cannot go to the previous node (forward iterator)**, so many APIs are changed to `xxx_after`!
- Iterator invalidation is same, too.

Forward list

- Difference:
  - There is no `.back()/.pop_back()/.push_back()/.append_range()` in forward list.
  - `.insert_after()/.erase_after()/.emplace_after()/.insert_range_after()/.splice_after()`: the position parameter accepts **the iterator before the insertion position**, so you can insert after it.
    - Particularly, iterators from another list in `.splice_after()` is also (first, last) instead of [first, last)!
      - But `.insert_after()` is still [first, last), since it doesn't need to go back and may come from any container.
    - It's impossible for you to provide the position instead of the one before, since it cannot go backward…
  - So, there is a `.before_begin()/.cbefore_begin()` iterator in forward list, so that you can insert at the head.
  - Finally, since forward list doesn't record size separately, `list2` in `splice_after` is in fact redundant…
    - It may be used in debug mode to check whether it's a node in the `list2`.

Containers, ranges and algorithms

Container Adaptors

Container adaptors

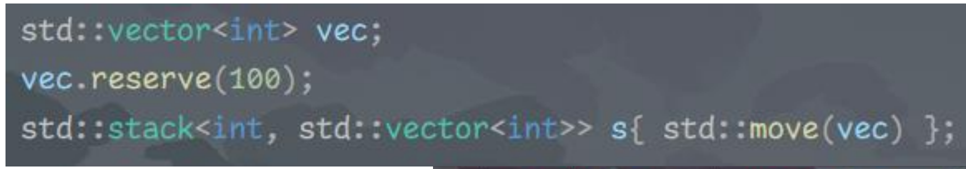
- Container adaptors are a wrapper of existing containers; it transforms their interfaces to a new interface of another data structure.
  - They usually don't provide iterators.
- So, their template parameters are always `<T, Container, …>`.
- `flat_set/flat_map/flat_multiset/flat_multimap` are provided since C++23, but they are very similar to associative containers, so we'll cover them after the next section.

Containers

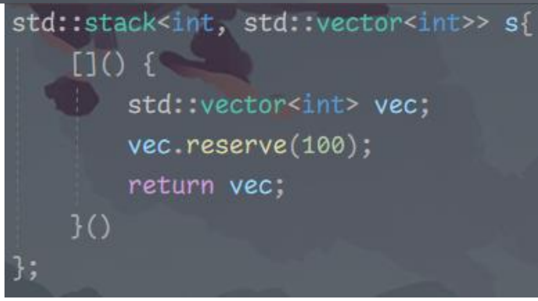
- Container adaptors
  - stack
  - queue
  - priority_queue
  - flat_set/flat_map/flat_multiset/flat_multimap

Stack

- Stack is a LIFO data structure.
  - And that's all!
- The provided container should have `push_back`, `emplace_back`, `pop_back`, so vector, deque and list are all OK.
  - The default one is deque, but personally I think vector is better.
    - As we stated, cache efficiency of vector is better; but deque may win when the data size is huge because its reallocation efficiency is better (anyway, profile it if it's important).
    - So you may hope to reserve the capacity of stack when using vector, but you can't do it easily.
      - We'll tell you the meaning in *Move Semantics*…
- For ctor, besides default ctor & copy ctor, you can only construct from a container.
  - Since C++23, you can construct from [first, last).
  - You can optionally provide an allocator as the last param, too.



```
std::vector<int> vec;
vec.reserve(100);
std::stack<int, std::vector<int>> s{ std::move(vec) };
```



```
std::stack<int, std::vector<int>> s{
    []() {
        std::vector<int> vec;
        vec.reserve(100);
        return vec;
    }()
};
```

Stack

- APIs provided by stack:
  - `.pop()`: pop from back, **return void**;
  - `.push(val)/.emplace(params)`: push to back.
    - Since C++17, emplace will return the reference to the inserted element.
  - `.push_range(Range)`: since C++23.
  - `.top()`: the element at back.
- These are basic functionality of stack; some others are:
  - `.empty()/.size()`
  - `.swap(s2)`
  - `operator=`
  - `operator<=>`
  - These should be provided by the container, too.
- That's all, quite easy, isn't it?

Containers

- Container adaptors
  - stack
  - queue
  - priority_queue
  - flat_set/flat_map/flat_multiset/flat_multimap

Queue

- Queue is a FIFO data structure.
  - And that's all, too.
- The provided container should have `push_back`, `emplace_back`, `pop_front`, so deque, list are all OK.
  - The default one is deque.
  - If you want to use list, forward_list is obviously better; but it doesn't provide `size()` and `push_back()`, so you may write a small wrapper yourself.
    - Notice that you can choose to not provide `size()` if you don't use it in queue; this is benefited from *selective instantiation*, which will be covered in *Template*.
- All member functions are same as stack, except that:
  - You should use `.front()` to check the next-popped object (i.e. the oldest data) instead of using `.top()`.
  - `.back()` is also provided to check newly inserted elements.

Containers

- Container adaptors
  - stack
  - queue
  - priority_queue
  - flat_set/flat_map/flat_multiset/flat_multimap

Priority queue

- It's defined in `<queue>`!
- It's in fact max heap; that is, it can always provide the current maximum element in $O(\log n)$ after inserting a new element/ popping the last max one.
  - Constructing the heap needs only $O(n)$, which is cheaper than sort.
  - It only needs the container to be random-access, so vector (default) and deque is all OK.
- Since comparing is needed, you can provide a template parameter as comparison method.
  - E.g. if you want the min heap, you can use `std::priority_queue<T, std::vector<T>, std::greater<T>>` (yes, min heap needs `greater`!).
  - There is a weird thing in ctor of priority queue; that is, the method is the first param, so if you want to provide a vector that has been `reserve`d, you have to use e.g. `std::priority_queue q{ std::less<int>{}, vec }`.
    - CTAD will help you to fill in all template parameters.

Priority queue

    - And then, you can provide input iterator pair since C++11 (instead of C++23 for stack & queue).
        - i.e.(first, last, cmp = CMP{}, container = Container{}).
- Let’s quickly review algorithms about heap.
    - It’s in fact a binary tree, with restriction that children are less than the parent(in array, that is $arr[i] \leq arr[(i-1)/2]$).
    - Percolation is the core algorithm.
        - When a new element is inserted at the back of array, we need to **percolate up** to maintain the restriction of heap.
            - That is, swap with the parent until it’s not greater than parent; obviously $O(\log n)$.
        - When the max element is removed from the top of array, we need to **percolate down**.
            - That is, use the last element to fill in the top, and swap with the bigger child until it’s greater than both children.
        - For constructing the heap, Floyd algorithm; that is, percolate down from the last non-leaf node to the root.
            - The higher it is, the higher percolation complexity is, but the less nodes it need to percolate.
            - So the total complexity is $\sum O(\log n - \log i) = O(\sum \log \frac{n^n}{n!})$; Stirling approx. tells us it’s $O(n)$.

Any question?

- You may take a small rest here ☺.

Containers, ranges and algorithms

Associative Containers

Associative containers

- They're called associative because they *associate key with value*.
  - The value can also be omitted, so that you can only check whether the key exists.
- There exist ordered one and unordered one.
  - Ordered one needs a compare function for "less than".
    - Iterating over the ordered one will also get elements in "ascending order".
      - Similarly, you can designate compare function.
    - It's BBST (balanced binary search tree), and search, insertion, removal are all $O(\log n)$.
      - It's usually RB tree, though e.g. AVL tree can also satisfy the requirement.
        - RB tree will cause less reordering than AVL tree when removing in topology.
  - Unordered one needs a hash function and a compare function for "equal".
    - It's (open) hash table, and search, insertion, removal are all expected $O(1)$ while the worst is $O(n)$, if all keys are hashed as the same key.
  - They're all node-based containers, i.e. every element is stored in a node separately (similar to linked list).
    - Thus, nodes can be extracted and inserted to reduce complexity of moving between containers.

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

Map

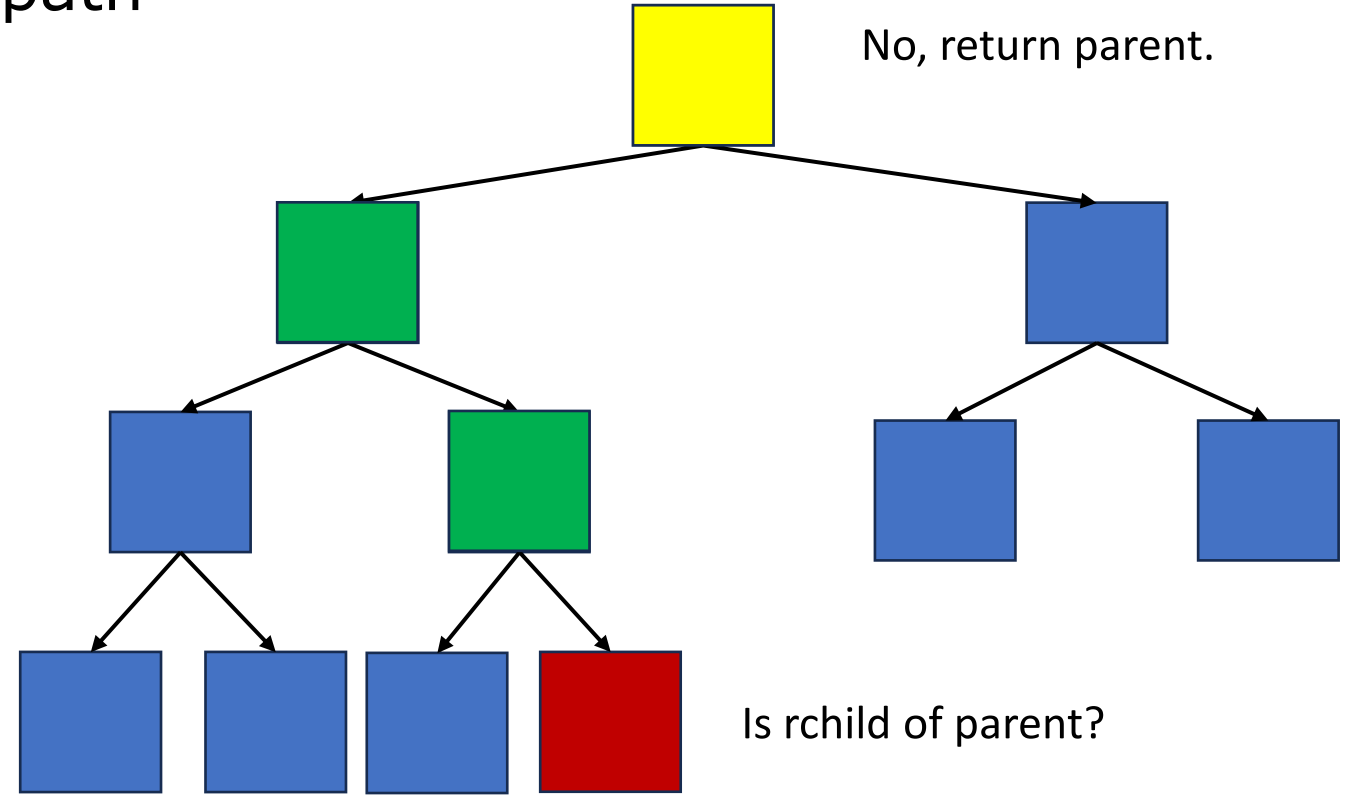
- We don’t review RB tree here and don’t deeply analyze the MS implementation because it’s a little bit too complex and you’ve already learnt the theory before.
- Map is a container that maps key to value.
  - The key is unique; a single key cannot be mapped to multiple values.
- `std::map<Key, Value, CMPForKey = std::less<Key>>`.
  - You should implement an `operator<`(or `<=>`) or provide a compare function.
  - `CMPForKey` should be able to accept `const Key`;
    - If it’s a member function, it should be `const`.
    - For example, you may define `auto operator<=>(const Key&) const`.

Map

- For member accessing:
  - `operator[]/at()`: accessing by index; `at()` will check whether the index exists; if not, `std::out_of_range` will be thrown.
  - Bidirectional iterators, as we stated.
    - Notice that the worst complexity of `++/--` is $O(\log N)$ since you may go from the rightmost child of the left subtree to the right subtree.
    - You can think of a BBST; `++/--` is just go to the next/previous node in ***inorder traversal***.
      - So, `.begin()` is just the leftmost node and `.rbegin()` is just the rightmost node.
        - Every node has a color, parent, left child and right child; to reduce complexity, empty node is also instantiated with a "isnil" member to check whether it's a virtual node.
        - To prevent low efficiency, the implementation stores a virtual "header" node, whose parent is root, left child is the leftmost node and right child is the rightmost node, and `isnil=true`.
          - It also acts as the parent of root and `end()`, which can unify code like linked list sentinel.
        - The RB tree members are just the header and node number of the tree (i.e. `.size()`).
      - `++` is just to find the next element that is bigger than self.
      - So, you may go up along parent chain, until it's not the right.
    - However, iterating from begin to end is $O(N)$, so `++/--` is still $O(1)$ on average.

Map

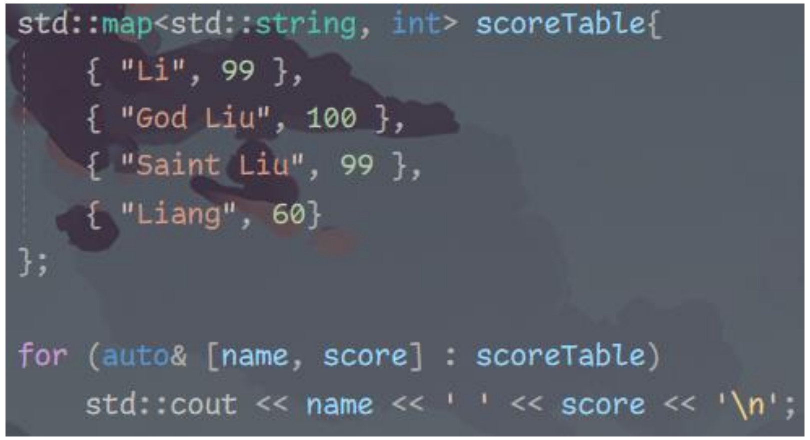
- $O(\log N)$ path



++ here

Map

- Note1: `operator[]` will insert a default-constructed value if the key doesn't exist, so:
  - If the key doesn't exist and the default value isn't needed, `insert_xxx` (covered later) is more efficient (construct + assign v.s. construct only).
  - It may insert new values, so it's not a `const` method, and you cannot use it in `const map`.
  - You cannot use it if the value cannot be default-constructed (i.e. no default ctor for class).
- Note2: key-value pair is stored in RB tree, so iterator also points to the pair.
- Note3: you can use *structured binding* (since C++17) to facilitate iteration.



```
std::map<std::string, int> scoreTable{
    { "Li", 99 },
    { "God Liu", 100 },
    { "Saint Liu", 99 },
    { "Liang", 60}
};

for (auto& [name, score] : scoreTable)
    std::cout << name << ' ' << score << '\n';
```

Structured binding

- As you've seen, structured binding is just `auto& […]{xx}`.
  - `{xx}` can be `(xx)` or `=xx`.
  - `auto&` can be anything, as long as it has `auto`.
    - E.g. `auto` means copy by value; `const auto&` is also valid; etc.
- We've seen that `xx` can be a pair; it can also be:
  - An object with all public data members, which will be bound on them.
  - A C-style array or `std::array`, which will be bound on elements `arr[i]`.
  - A **tuple-like** thing, which will be bound on every element.
- Pair is just a tuple-like thing; so what's tuple?
  - It's a generalization of pair; you can have multiple instead of two members.
  - E.g. `std::tuple<int, float, double> t{1, 2.0f, 3.0};`

Tuple

`std::tuple<int, float, double> t{1, 2.0f, 3.0};`

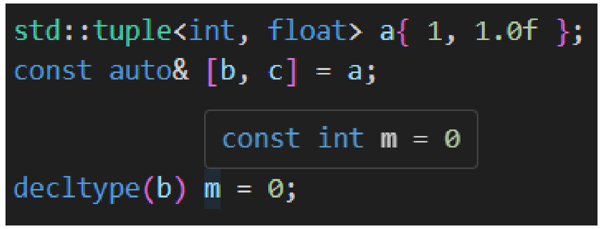
- It can only be accessed by an index that can be determined in compile time (unlike tuple in python)!
  - So, you can use `std::get<0>(tuple)` to get the `int`.
  - Since C++14, you can also use type-based index when the type occurs only once in the tuple, e.g. `std::get<int>(tuple)`.
  - You can use `operator=, swap` and `operator<=>`, just like pair.
  - You can also use `std::tuple_size<YourTupleType>::value` to get the size and `std::tuple_element<index, YourTupleType>::type` to get the type of `index`th element, `std::tuple_size<YourTupleType>::value` to get the size of tuple.
    - The type of tuple can be got by e.g. `decltype(t)`, which will be further discussed in the future!
  - You can concatenate two tuples to get a new tuple by `std::tuple_cat`.
  - NOTICE: `tuple` is discouraged to use in normal programming because the elements don't specify their names and can be obscure for maintenance. It's usually used in ***metaprogramming***, which will be covered in *Template*.

Structured binding

- Note1: pair and `std::array` is also somewhat tuple-like thing and can use some tuple methods, e.g. `std::get`.
  - We'll give precise definition of "tuple-like" in the future.
- Note2: Structured binding is declaration, so you cannot bind on existing variables.
  - Instead, for tuple and pair, you can use `std::tie(name, score) = pair` to assign.
- Personally, I prefer to use _ as variable name to denote "it is dropped and never used except for constructing it".
  - E.g. `auto& [_, score] : scoreTable`.
  - Since C++26, _ is preserved to be "discarded value"; you can **redeclare** _ in a function scope without reporting redefinition error (but you cannot use it after redeclaration; it's discarded.).
  - For `std::tie`, you can use `std::ignore`, e.g. `std::tie(std::ignore, score) = pair`.

Structured binding

- Note3: the structured binding is equivalent to use an anonymous struct, with its members aliased.
  - E.g. `b` is of type `const int` instead of `&`, because it's `const auto& anonymous`, and `anonymous.b` is not reference (i.e. here it's `std::tuple<int,float>&` instead of `std::tuple<int&,float&>`).
  - But, `anonymous.b` still references the original data, since the `anonymous` is reference.



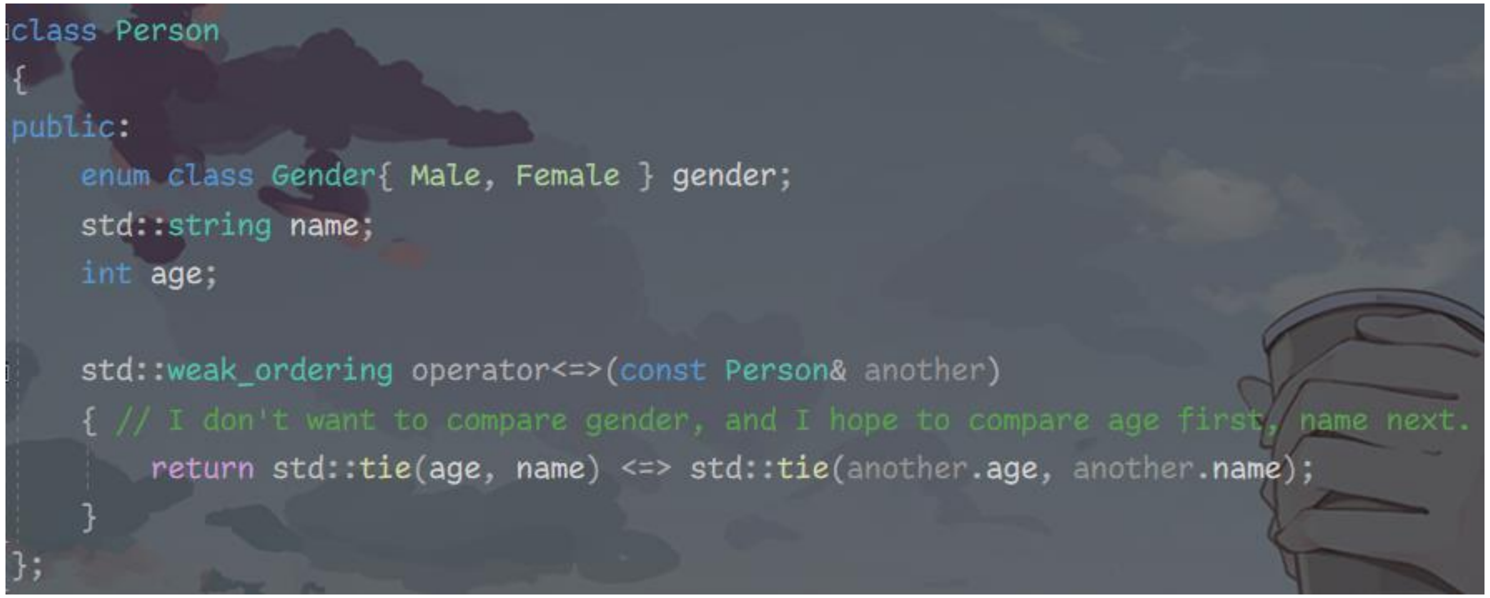
```
std::tuple<int, float> a{ 1, 1.0f };
const auto& [b, c] = a;
                    const int m = 0
decltype(b) m = 0;
```

- Note4: structured binding is usually more efficient than novice/ careless programmers.
  - For example, you can code like: `for(const std::pair<std::string,int>& p : m)`
    - However, map stores `std::pair<const std::string, int>`, so iterate by `std::pair<std::string, int>` will lead to unnecessary copy.
    - Of course, `const auto& p` can eliminate this problem too.

Tips: you should use structured bindings on only if their order is guaranteed to be stable. Otherwise if you swap two members with the same type in definition, then user will get totally wrong result!

Tuple

- Final word: for `operator<=>` that cannot be default while you still want some form of lexicographical comparison, you can utilize `operator<=>` of tuple!
- For example:



```
class Person
{
public:
    enum class Gender{ Male, Female } gender;
    std::string name;
    int age;

    std::weak_ordering operator<=>(const Person& another)
    { // I don't want to compare gender, and I hope to compare age first, name next.
        return std::tie(age, name) <=> std::tie(another.age, another.name);
    }
};
```

Map

- For size operations:
  - `.size()`: get size, return `size_t`.
  - `.empty()`: get a `bool` denoting whether `size == 0`.
  - `.max_size()`: get maximum possible size in this system (usually useless).
  - `.clear()`: remove all things; size will be 0 after this.
- For lookup:
  - `.find(key)`: get the iterator of key-value pair; if key doesn’t exist, return `end()`.
    - Remember `if(auto it = map.find(key); it != map.end()){ … }` !
  - `.contains(key)`: since C++20, return a bool to denote whether the key exist.
  - `.count(key)`: get the number of key-value pair referred by key; in map, it can only be 0 or 1.

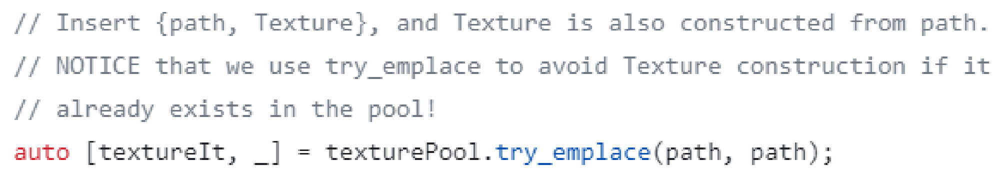
Map

- `.lower_bound(key)`: find `it` that `prev(it)->key < key <= it->key`.
    - As its name, use key as a lower bound to make [it, end) >= key.
    - Return `end()` if key is the biggest.
- `.upper_bound(key)`: find `it` that `prev(it)->key <= key < it->key`.
    - As its name, use key as a upper bound to make [begin, it) <= key.
    - Return `end()` if key is the smallest.
- `.equal_range(key)`: find `it` pair with the same key as `key` in range.
    - It's equivalent to `[lower_bound(key), upper_bound(key))`, but more efficient.
    - Particularly, when this key exists in the map, it will return `[it, it + 1)`; else, it will return `[it, it)`.

- `operator=, operator<=>, swap, std::erase_if` are all available.
    - `operator<=>` compares pair one by one from begin.

Map

- Insertion
  - Since map regulates the uniqueness of key, insertion may fail if the key exists. `pair<iterator, bool>` will be returned;
    - If succeed, `iterator` refers to the inserted element and `bool` is `true`;
    - If fail, `iterator` refers to the element with the same key and `bool` is `false`.
  - But, different methods may process this failure differently:
    - Leave it unchanged:
      - `.insert({key, value})`
      - `.emplace(params)`: same as `insert`, except that the params are used to construct **pair**.
    - Overwrite it (C++17):
      - `.insert_or_assign(key, value)`: return `pair<iterator, bool>`;
        - Difference: **overwrite**; provide key and value separately instead of providing a `std::pair`.
    - Leave it unchanged and even not construct the inserted value (C++17):
      - `.try_emplace(key, params)`: same as `emplace`, except that the params are used to construct **value**, and `emplace` is not forbidden to construct the pair in failure.

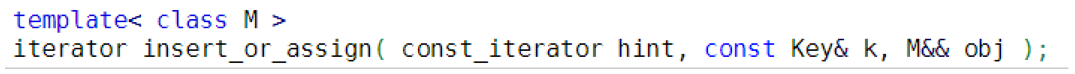


```
// Insert {path, Texture}, and Texture is also constructed from path.
// NOTICE that we use try_emplace to avoid Texture construction if it
// already exists in the pool!
auto [textureIt, _] = texturePool.try_emplace(path, path);
```

It's costly to construct a Texture, and `try_emplace` is perfect!

Map

- Erasure:
  - `.erase(…)`
    - `(key)`
      - It will return the number of erased elements, 0 or 1 in map.
    - `(iterator pos)`: erase pos, requiring that pos is from the map.
    - `(iterator first, iterator last)`: erase [first, last) from map.
      - These two will return the iterator after the last erased iterator.
- You can also provide a hint iterator for insertion.
  - The hint iterator should be after the inserted element to gain efficiency.
    - If it's before the element, the efficiency may be hurt, so use it carefully.
  - The insertion methods are same, except that:
    - It should provide a hint as the first parameter.
    - `emplace()` is replaced by `emplace_hint()`.
    - It will only return the iterator, no `bool`.

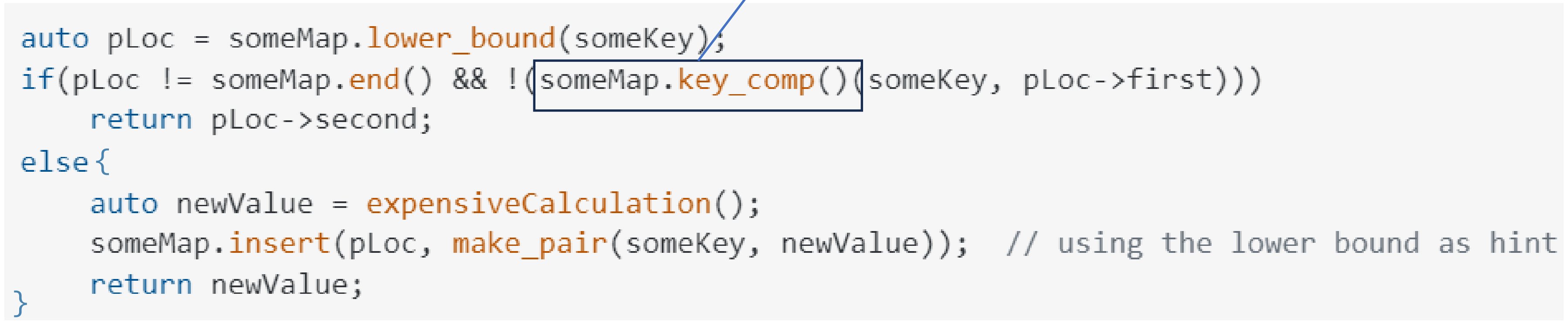


```
template< class M >
iterator insert_or_assign( const_iterator hint, const Key& k, M&& obj );
```

Map

`key_comp()` gets the comparison function.
This means `someKey >= pLoc->first`, while `lower_bound` means `someKey <= pLoc->first`, thus `someKey == pLoc->first`, meaning that this key exists.

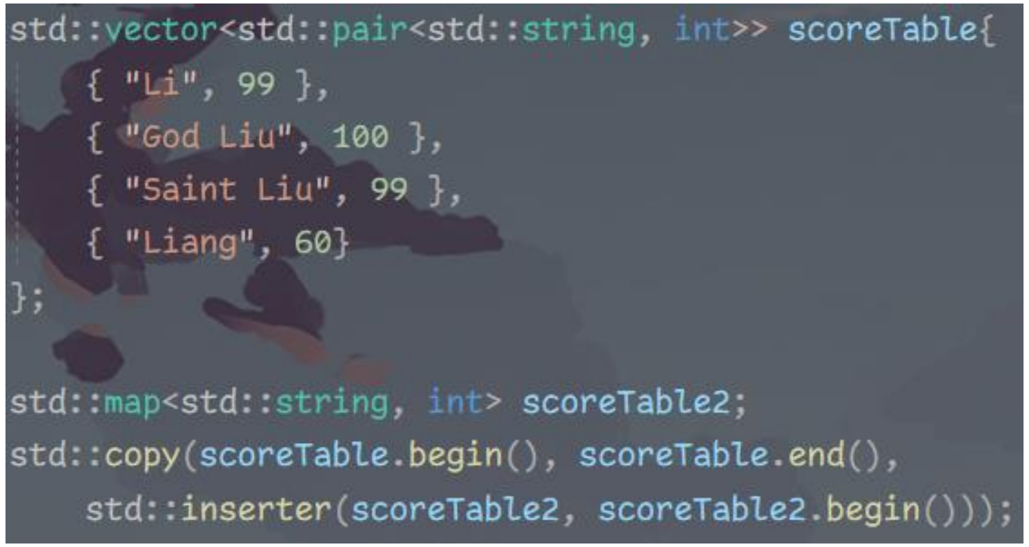
- Hint is often used in idiom below:



```
auto pLoc = someMap.lower_bound(someKey);
if(pLoc != someMap.end() && !(someMap.key_comp()(someKey, pLoc->first)))
    return pLoc->second;
else{
    auto newValue = expensiveCalculation();
    someMap.insert(pLoc, make_pair(someKey, newValue));  // using the lower bound as hint
}   return newValue;
```

- For "`else`", the key doesn't exist, `lower_bound()` will return the exact one that is bigger than key (i.e. iterator after that), which will facilitate the future insertion.
- Notice that they don't return a `bool` for uniformity with e.g. vector's insert.
  - Yes, you see that params of this method are same as vector!
  - Inserter iterator uses it, so you can also insert into map with inserter.

Map



```
std::vector<std::pair<std::string, int>> scoreTable{
    { "Li", 99 },
    { "God Liu", 100 },
    { "Saint Liu", 99 },
    { "Liang", 60}
};

std::map<std::string, int> scoreTable2;
std::copy(scoreTable.begin(), scoreTable.end(),
    std::inserter(scoreTable2, scoreTable2.begin()));
```

- Notice that key of map is `const` to maintain order; you need to erase the original and insert a new one if you want to change key.
  - So, you cannot directly use `std::copy(.., .., scoreTable2.begin())`, since it assigns a whole pair and violates `const` of key.

Map

Except for extraction, insertion and judging empty, you can also use `.key()/.value()` to get the reference to key/value (so that you can modify key/value and then insert!), and dtor will free node automatically if it's not moved away. You can see here for more details if you're interested in other methods.

- Finally, since map is node-based containers, it can also extract nodes and insert them to another map like `splice` of linked list.
- Since C++17:
  - `extract(key)/extract(iterator pos)`: extract out the node from the map.
    - It will return a `node_type` thing, and you may just use `auto`.
      - In fact `std::map<Key, Value, …>::node_type`, that's too long…
    - You may think it as a pointer; if key doesn't exists, then `ret.empty()` or `operator bool` will return `false`, just like return a `nullptr`. Empty node will do nothing in insertion.
  - `.insert(node_type&&)`: insert the node to the map.
    - We'll tell you what `&&` means in the future; now you just need to know you need to pass `std::move(node)` or directly `xx.extract(yy)` to this param.
    - Return `insert_return_type`, a struct with `{ iter position, bool inserted, node_type };`
      - The first two is same as normal insertion; node will be empty if succeed, else the original node.
        - After `std::move(node)`, the variable `node` is invalid, and you should get it again here.
    - You can also provide a hint as the first param, and the return type is still only iterator.
  - `.merge(another map/multimap)`: merge another map into self, i.e. iff. the key doesn't exist, it will be moved from another to self (existing keys will not be moved).

Map

- Since C++23, you can also use `insert_range`.
- For ctor, besides default ctor, copy ctor, move ctor:
  - `(cmp)`: specify the compare function, utilize CTAD.
  - `(first, last, cmp = Compare())`: construct by iterator pair, which refers to a key-value pair.
  - `(intiailizer_list<pair>, cmp = Compare())`.
- For iterator invalidation, since map is node-based, and iterators are just pointer to the node, so only erasure will invalidate the iterators of erased elements.
  - This is same as linked list.

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

Set

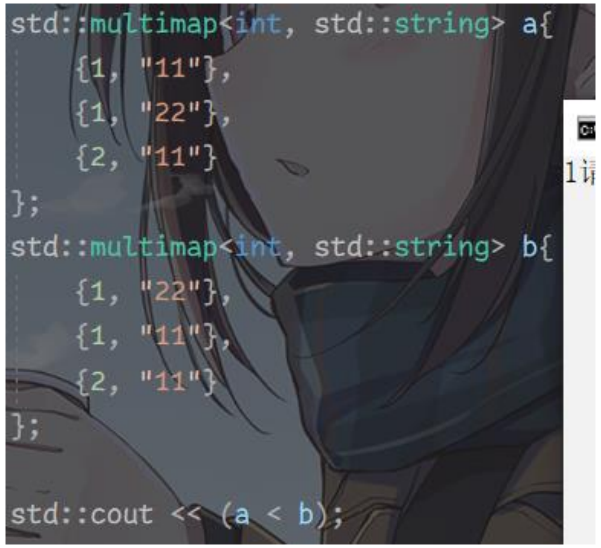
- Set is just a map without value; that is, you can only insert/remove/check whether an element exist in $O(\log N)$.
  - The key is still unique; this is same as the definition of set in math.
- So, the only difference with map is that it doesn't have `operator[]` and `.at()`; the iterator points to only key instead of key-value pair.
  - And that's all!

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

Multimap

- Multimap cancels the uniqueness of key, i.e. a key can be mapped to multiple values.
  - So, you cannot use `operator[]` and `at()` too.
  - These equivalent values are in the same order of inserting sequence.
    - The operator `<=>` check equality one by one, so even when two multimap stores same things with only different orders in equal keys, `==` is `false`.
    - E.g.



```
std::multimap<int, std::string> a{
    {1, "11"},
    {1, "22"},
    {2, "11"}
};
std::multimap<int, std::string> b{
    {1, "22"},
    {1, "11"},
    {2, "11"}
};

std::cout << (a < b);
```

Multimap

- Also, insertion will never fail, so there is no `insert_or_assign/try_emplace`, and `insert/emplace` will only return iterator.
  - Hint is still available.
- For `find()`, only a random element with equal key will be returned.
- For `count()`, return the number of elements (not only 0/1).
  - The complexity is $O(\log N) + O(M)$, where M = count().
- For `equal_range()`, return iterator pair with equal key.
  - This is the often used method to find elements in multimap; `it1 == it2` means the key doesn't exist.
- Finally, nodes of multimap and map can be exchanged, e.g. you can insert node into multimap extracted from map and vice versa.
  - Multimap into map will only reserve the first element in the equal range to maintain uniqueness.

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

Multiset

- Except for only key and no value, same as multimap.
  - That’s all!
- You can also exchange nodes of multiset and set.
- In fact, map is just almost like set<pair>, and multimap is almost like multiset<pair> (with the first element as compare standard).

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

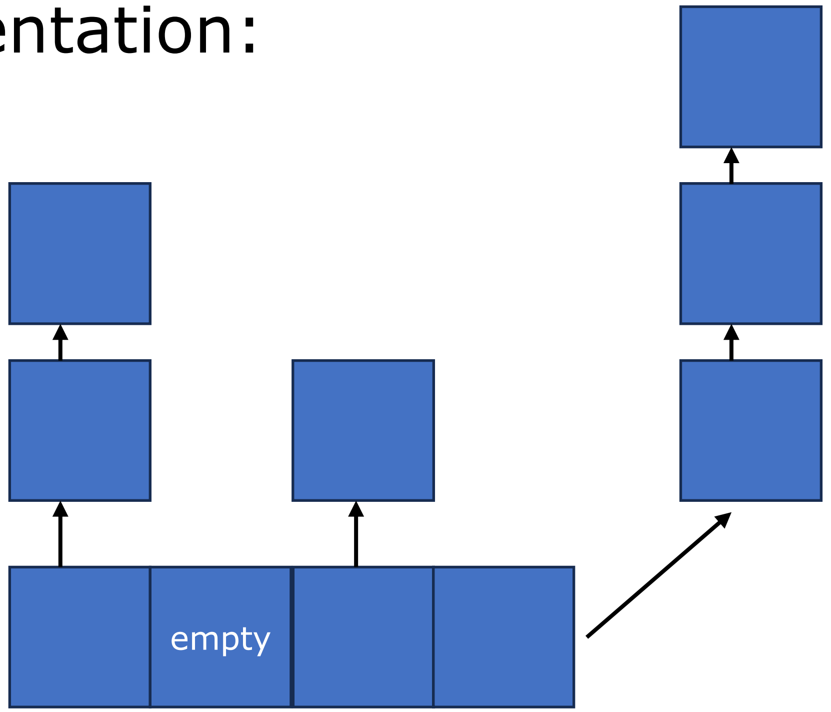
Unordered map

- `std::unordered_map<Key, Value, Hash = std::hash<Key>, Equal = std::equal_to<Key>>`
- Many types have `std::hash<Type>`, e.g. `std::string, float`, etc., so you can use them as key directly.
  - Similar to map, `Hash` and `Equal` should be like `(const Key&) const`.
- The hash value of different keys may be same, so we need `Equal` to judge which key is wanted.
  - In open hash table, elements with the same key is stored in a "bucket".
    - Usually the bucket is stored as single linked list for $O(1)$ random removal.
    - C++ just requires **forward iterator**, so it's possible.
  - Buckets consist of an array, so the final index is `hash value % bucket number` (if 2^n, can be optimized as `&`)

Unordered map

- Possible implementation:

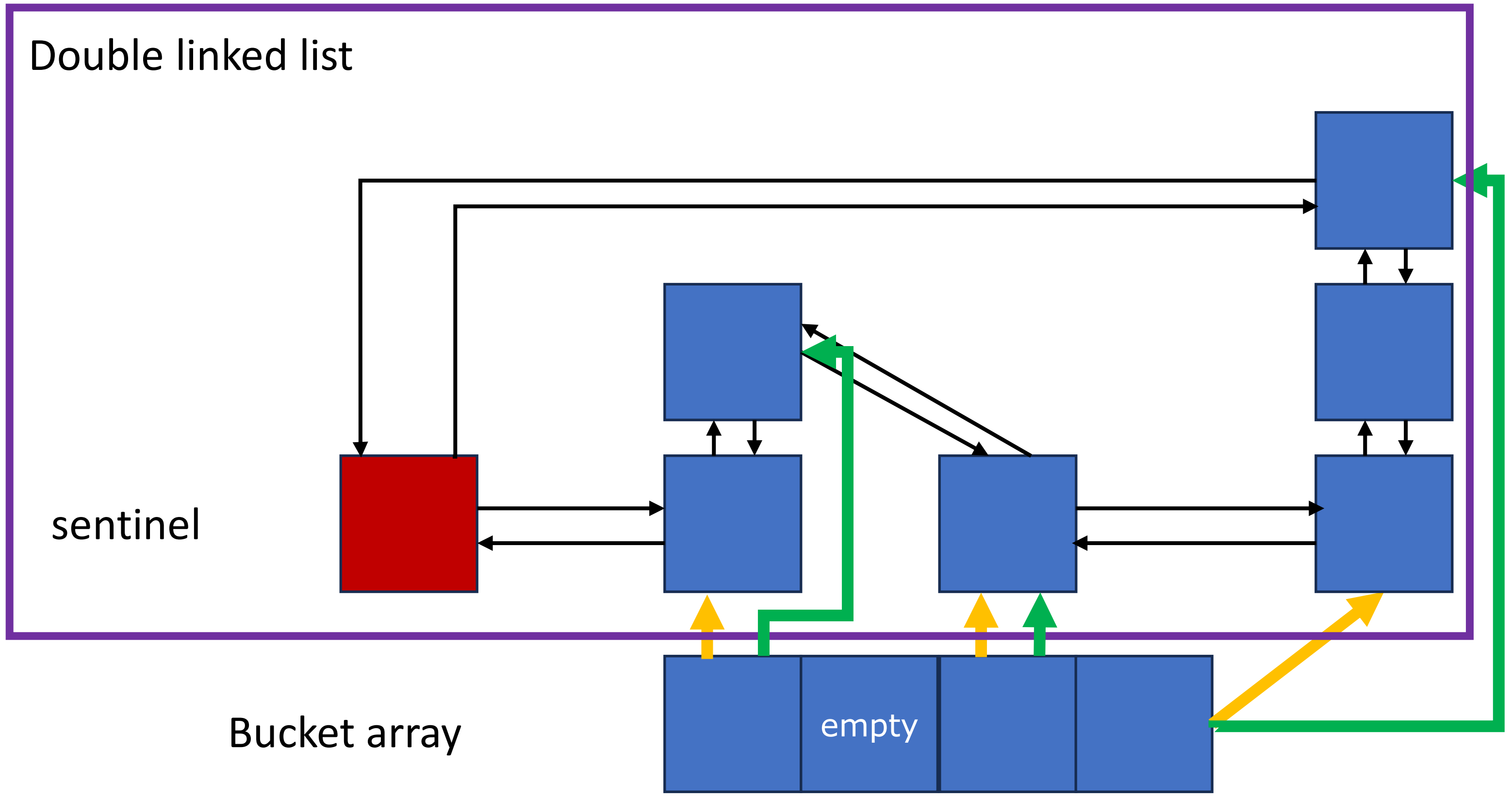
Bucket array



Unordered map

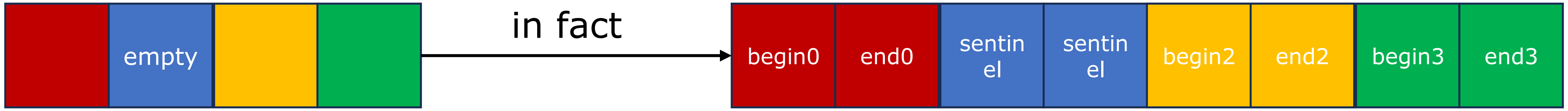
- In MS implementation, it’s slightly different.
  - It uses double linked list instead of single linked list.
  - So we think it as array of double linked list; then from the view of implementation of `list`, we need to allocate a sentinel for each linked list.
    - However, hash table is usually scattered, so each linked list is quite small; a sentinel wastes much memory.
    - So MS makes all linked list “share” the same sentinel.
    - To be exact, all nodes are linked together as a huge linked list.
  - So, begin iterator is just sentinel->next, end is the sentinel.
    - `operator++` is just list iterator’s `operator++`.
    - Insertion, erasure are all operating on the whole linked list.
  - A little abstract, let me show you by animation!

- Each bucket records the begin(↑) and the end(↑), where [begin, end] contains all bucket elements.
    - Notice: close interval!



Unordered map

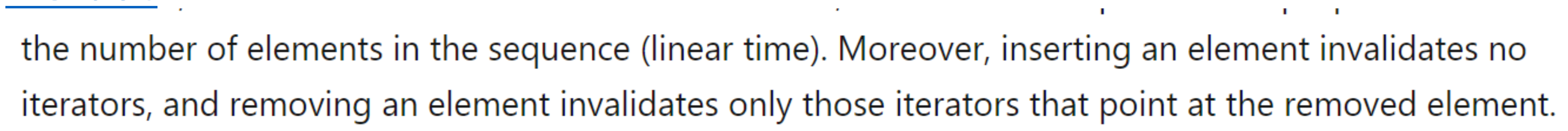
- To be specific, we assume the size of bucket array is `s0`, then in MS implementation, array of pointers with size of `2 * s0` are allocated.



- When the bucket is empty, its begin and end are assigned pointer to sentinel.
  - begin and end is called “bucket high” and “bucket low” there.

Unordered map

- When we insert into too many elements, there will also be too many elements in each bucket!
  - This increases the complexity of finding by key.
  - $\frac{size}{bucket\ num}$ is called **load factor**.
  - So when load factor is high, we need to enlarge the size of bucket array to make elements scattered again.
    - The grow policy is not regulated, e.g. in ms, you can see `_Desired_grow_bucket_count` in `<xhash>`.
  - This is called **rehash**.
  - Though the hash values of elements remain same, the indices after modulus are changed, so the whole list needs rearrangement.
    - **C++ regulates that rehash will invalidate all iterators**, though implementing by list will not invalidate anything(but you may not rely on this when cross-platform).
    - Since it's node-based, references will always be valid.
    - MS doc:



the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators that point at the removed element.

Unordered map

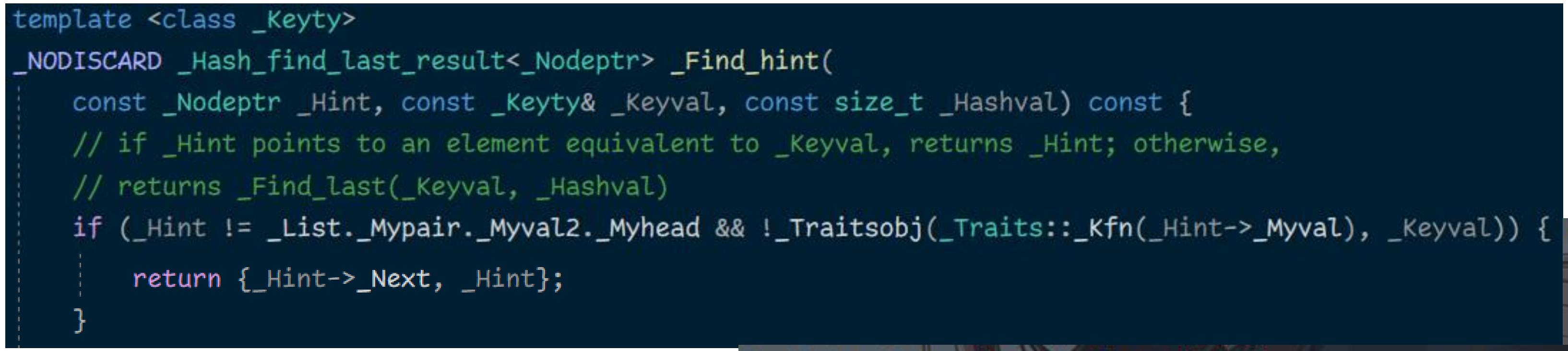
- So, as a hash table, it provides many related methods:
  - `.bucket_count()`: size of bucket **array**.
  - `.max_bucket_count()`: similar to `max_size()` in vector, not that useful usually.
  - `.load_factor()`: `size() / bucket_count()`
  - `.max_load_factor()`: when load factor exceeds this limit, rehash will happen. You can set it by `.max_load_factor(float xx)`.
    - In MS, this is 1 by default, i.e. when every bucket has more than 1 element on average, rehash will happen.
  - `.rehash(n)`: make `bucket_count()=max(n,ceil(size()/max_load_factor()))` and rehash; particularly, `rehash(0)` may be used when you adjust `max_load_factor()` to immediately rehash to meet minimum requirement.
  - `.reserve(n)`: reserve the bucket to accommodate at least n elements, i.e. before `size() > n`, no rehash should happen. Equivalent to `rehash(ceil(n/max_load_factor()))`.
    - This is similar to `vector::reserve(n)`; before `size() > n`, no resizing will happen.

Unordered map

- C++ also provides you interface to get a bucket directly.
  - `.bucket(key)`: get the bucket index of the key.
  - `.begin/cbegin/end/cend(index)`: get the iterator of the bucket at index.
  - `.bucket_size(index)`: get the size of bucket at index.
    - This is just `std::distance(begin(index), end(index))`, the complexity is $O(bucket\ size)$.
- Finally, you can observe functions by `hash_function()` and `key_eq()`.

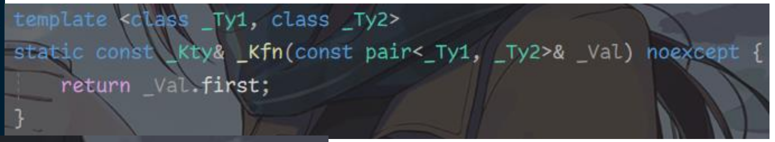
Unordered map

- As a “map”, its methods are almost same as `std::map`, except:
  - It doesn’t have `lower_bound` and `upper_bound`, because of unordered.
  - It can also use hint, but requirement is not the same.
    - The standard doesn’t specify how the hint influences lookup; in MS implementation, it’s only useful when the inserted key and hint key are same.
      - Obviously basically only useful in unordered_multimap.



```cpp
template <class _Keyty>
_NODISCARD _Hash_find_last_result<_Nodeptr> _Find_hint(
    const _Nodeptr _Hint, const _Keyty& _Keyval, const size_t _Hashval) const {
    // if _Hint points to an element equivalent to _Keyval, returns _Hint; otherwise,
    // returns _Find_last(_Keyval, _Hashval)
    if (_Hint != _List._Mypair._Myval2._Myhead && !_Traitsobj(_Traits::_Kfn(_Hint->_Myval), _Keyval)) {
        return {_Hint->_Next, _Hint};
    }
```

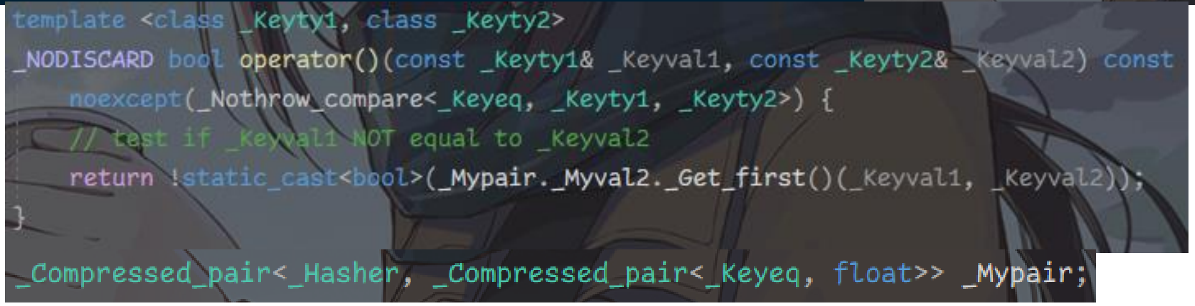
_Hint->_Myval: get key-value pair of hint.
_Traits::Kfn: get key.



```cpp
template <class _Ty1, class _Ty2>
static const _Kty& _Kfn(const pair<_Ty1, _Ty2>& _Val) noexcept {
    return _Val.first;
}
```

_Traitsobj::operator(a, b):

_Mypair._Myval2._Get_first() will get _Keyeq, i.e. the equal function.

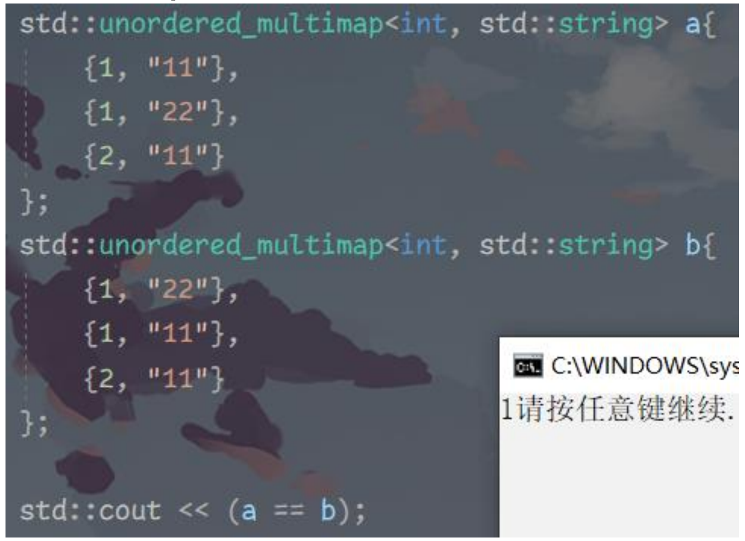


```cpp
template <class _Keyty1, class _Keyty2>
_NODISCARD bool operator()(const _Keyty1& _Keyval1, const _Keyty2& _Keyval2) const
    noexcept(_Nothrow_compare<_Keyeq, _Keyty1, _Keyty2>) {
    // test if _Keyval1 NOT equal to _Keyval2
    return !static_cast<bool>(_Mypair._Myval2._Get_first()(_Keyval1, _Keyval2));
}

_Compressed_pair<_Hasher, _Compressed_pair<_Keyeq, float>> _Mypair;
```

Unordered map

- You can also extract nodes and insert them.
- For comparison, you can only compare `==/!=` for two unordered map.
  - Particularly, for `==/!=` in unordered_multimap, it's only required that they have same values on each key, rather than force an insertion sequence like multimap.
    - That is, equal as long as they are permutation, order is not important.
    - The worst complexity is thus $O(N^2)$ and the average is $O(N)$, because testing whether two ranges are permutation is $O(N^2)$ in worst case.



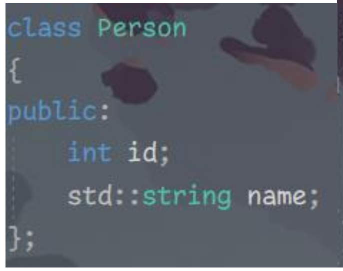
```
std::unordered_multimap<int, std::string> a{
    {1, "11"},
    {1, "22"},
    {2, "11"}
};
std::unordered_multimap<int, std::string> b{
    {1, "22"},
    {1, "11"},
    {2, "11"}
};

std::cout << (a == b);
```

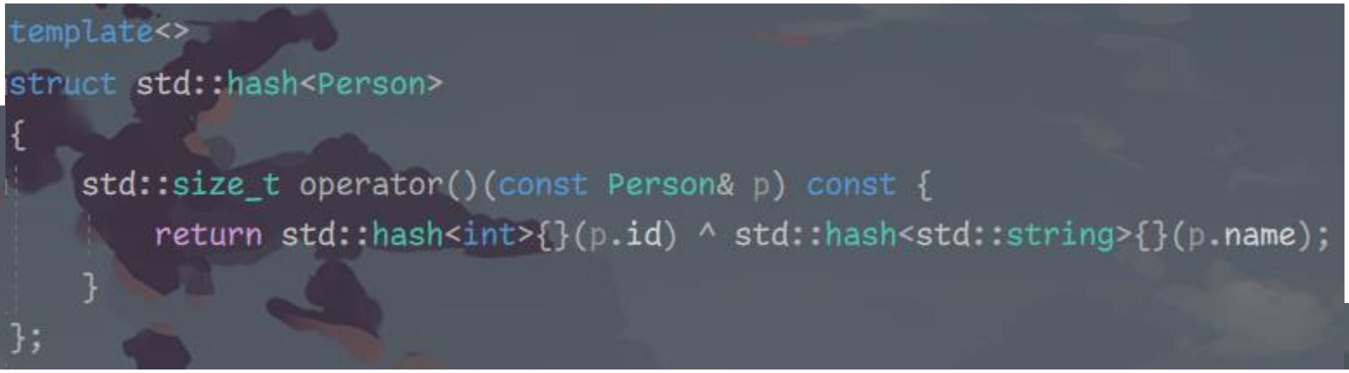
```
C:\WINDOWS\sys
1请按任意键继续.
```

Unordered map

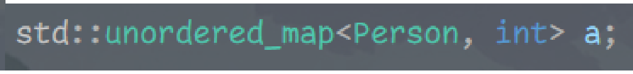
- When you use your own class as key, you need to customize hash.
    - You can of course define a class with `operator()` and return `size_t`.
    - Another frequently-used way is to specialize `std::hash`; we'll cover specialization of template in *Template*, and here you can just have a look at it.



```
class Person
{
public:
    int id;
    std::string name;
};
```



```
template<>
struct std::hash<Person>
{
    std::size_t operator()(const Person& p) const {
        return std::hash<int>{}(p.id) ^ std::hash<std::string>{}(p.name);
    }
};
```



```
std::unordered_map<Person, int> a;
```

- We use xor to combine two hash values of members.
    - This is just a common way; but designing a good hash function to reduce conflict is quite difficult, and we'll not cover them here.

Containers

- Ordered containers
  - map
  - set
  - multimap
  - multiset
- Unordered containers
  - unordered_map
  - unordered_set
  - unordered_multimap
  - unordered_multiset

The relation of unordered ones are same as ordered ones, so we don’t cover them.

Containers

- Container adaptors
  - stack
  - queue
  - priority_queue
  - flat_set/flat_map/flat_multiset/flat_multimap

Flat containers

- The only defect of map/unordered_map/… is that they’re really cache-unfriendly!
  - This is the common problem of node-based containers, including linked list.
  - It will also waste too much memory when `sizeof` data is actually small because a node has many pointers.
  - It’s criticized by many (e.g. Google) and they write their own versions.
    - Even possibly their theoretical complexity is higher, the real efficiency is still higher because of good locality.
- Flat containers are for it.
  - The functionality is same as set/map;
  - But it’s in fact an ordered “vector”!
    - It doesn’t have any redundant data, and is more cache-friendly obviously.
  - For flat map, it’s just two vectors.

Flat containers

- So, the whole definition is
```
std::flat_map<Key, Value,
              Compare = std::less<Key>,
              ContainerForKey = std::vector<Key>,
              ContainerForValue = std::vector<Value>>
```
- You can also choose deque as container.
- Obviously, the complexity is:
  - For lookup, $O(\log N)$, with a really small constant (just a simple binary search, much smaller than RB tree).
  - For insertion/removal, $O(n)$.
    - Though search insertion position is $O(\log N)$, it needs to move the elements, and even possibly resize, and they are all $O(n)$.
  - For `iterator++`, constant $O(1)$.
    - The iterator is also **random-access** iterator!

Flat containers

- But, you need to pay more attention to iterator invalidation.
  - For map/set, only erased elements are invalidated.
  - For unordered ones, though iterators are invalidated after rehashing, at least the reference is still valid.
  - But if you use vector, insertion/removal itself will make more or even all iterators/references invalid.
    - Resizing will also cause iterator & reference invalidation!
- Besides, you cannot store objects that cannot be copied/moved.
  - Still resizing problem!
  - The exception guarantee of vector is also looser than map, which will be covered in *Error Handling*.
  - We may cover more details in the future.

Flat containers

- Finally, since keys and values are stored separately in two containers, the iterator doesn't point to a whole pair; it stores only an index, and has separate first and second.
  - So dereferencing it will get a proxy of pair.

- Err… There is still one thing that I want to cover – heterogeneous container, but this lecture already has too much information, so we'll cover them in the next lecture.

Summary

- Iterators
  - Iterator categories, stream iterator, iterator adaptors (iterator-based/container-based).
- Containers, including design purpose, some implementation and APIs.
  - Iterator invalidation & reference invalidation.
  - Functionality and complexity.
  - Sequential containers
    - array, vector, deque, list, forward_list
    - vector<bool>, bitset
    - span, mdspan (it's just required to know how to use it in the simplest case)
- Container adaptors
  - stack, queue, priority_queue
  - flat_map/set/multimap/multiset
- Associative containers
  - map/set/multimap/multiset
    - Structured binding, tuple.
  - unordered_map/set/multimap/multiset

Next lecture…

- We’ll cover ranges in C++20 first.
- Then understand function more deeply!
  - Next, we’ll cover heterogeneous containers, which are easy to understand.
- Finally, we will introduce algorithms.
  - This part is bit of bored, but we’ll quickly grasp the thorough design.
  - We’ll also tell you how some algorithms are implemented.