Understand std::vector

Huỳnh Ngọc Thi

so61pi.re@gmail.com

Why?

- std::vector is a fundamental data structure in C++.
- It’s used everywhere.
- It’s array-based, good for performance.
  - Maybe not the best, but still.
- Its iterator’s category is RandomAccessIterator, which is required by some algorithms
  - shuffle, sort, stable_sort, nth_element, make_heap, sort_heap, …

Placement new

- How can we construct std::string on a given buffer?
  - Placement new

```
T* = new (buffer) T;
char buffer[sizeof(string)] = {};
auto pstr = new (buffer) string{"abc"};
```

- How can we destruct it?
  - Placement delete?
  - We must call destructor ourselves.

```
pstr->~string();
```

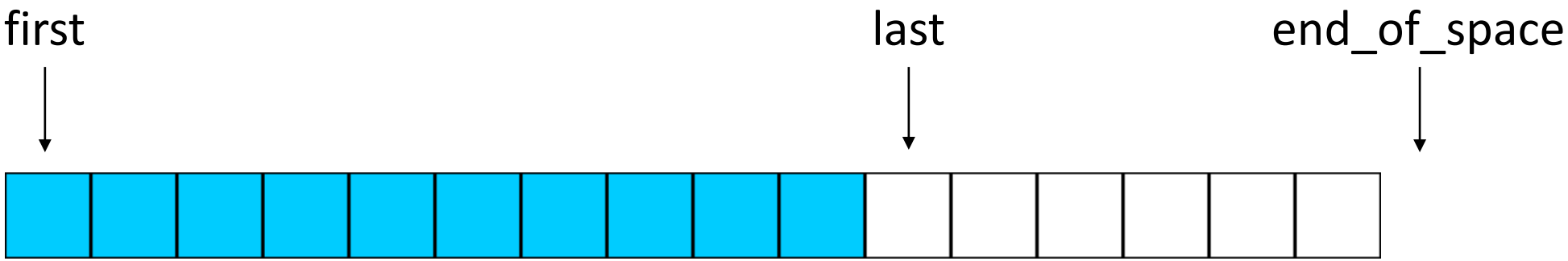
- https://isocpp.org/wiki/faq/dtors#placement-new

std::vector memory layout

- Why do we have extra “unused” space?
  - push_back/emplace_back can have amortized constant complexity.
- Amortized constant complexity
  - “The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus “amortizing” its cost.”
  - https://en.wikipedia.org/wiki/Amortized_analysis

std::vector memory layout

- Basically, std::vector is constructed from 3 pointers.



push_back complexity analysis

- If we don’t have that “unused” space
  - Each push_back
    - allocate new memory
    - move/copy old elements
    - construct new element
    - destruct old elements
    - free old memory.
  - O(n)

push_back complexity analysis

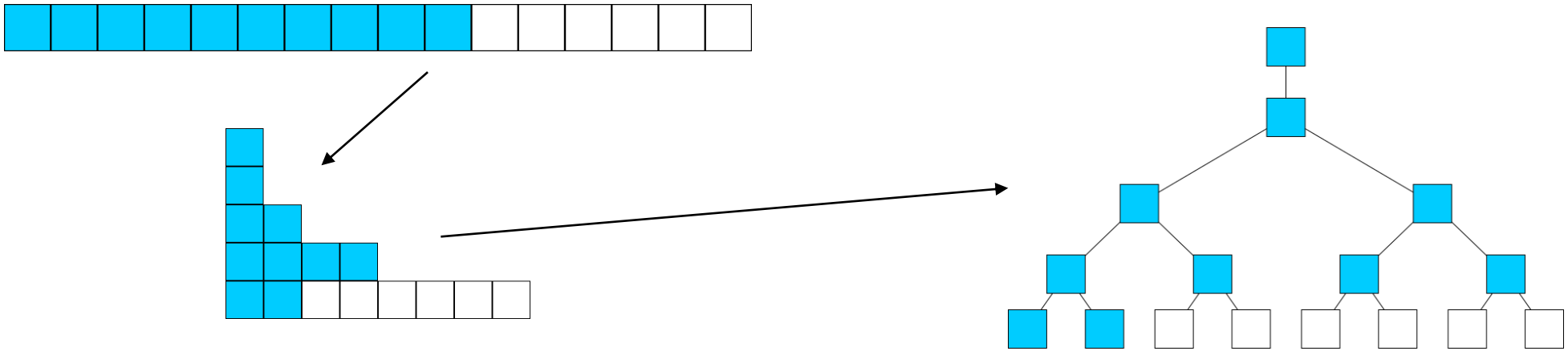
- Else,
  - New element can be constructed without reallocating memory.
- We have a growth factor, k.
  - Each time reallocation happens, new size = max(size() + 1, size() * k).
- MSVC uses 1.5 (_Grow_to), libstdc++ 2 (_M_check_len).

push_back complexity analysis

k = 2				
before push_back		after push_back		reallocation
size()	capacity()	size()	capacity()	
0	0	1	1	true
1	1	2	2	true
2	2	3	4	true
3	4	4	4	false
…	…	…	…	…
127	128	128	128	false
128	128	129	256	true
129	256	130	256	false
…	…	…	…	…

push_back complexity analysis

- If we push_back N elements
  - How many reallocations will happen?
  - How many moves/copies will happen?
  - Is it amortized constant complexity?
    - That means the ratio between number of moves/copies and N is constant.



push_back complexity analysis

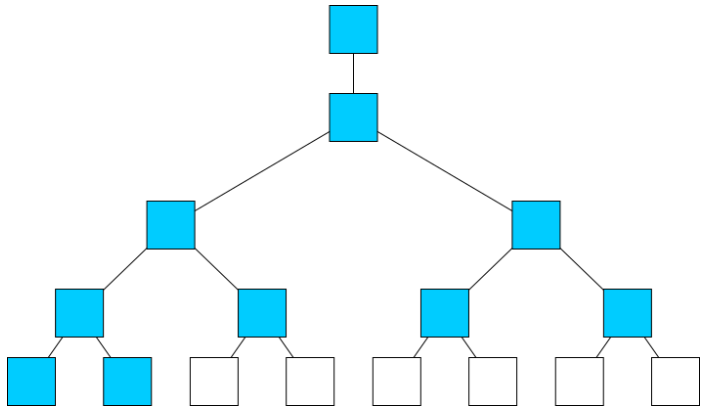
$$numRealloc = \lceil \log_k N \rceil$$

$$N = 10, k = 2 \Rightarrow numRealloc = 4$$

$$numMovCp = k^0 + k^1 + k^2 + \cdots + k^{numRealloc-1}$$

$$= \frac{k^{numRealloc} - 1}{k - 1} = \frac{k^{\lceil \log_k N \rceil} - 1}{k - 1}$$

https://en.wikipedia.org/wiki/Geometric_series



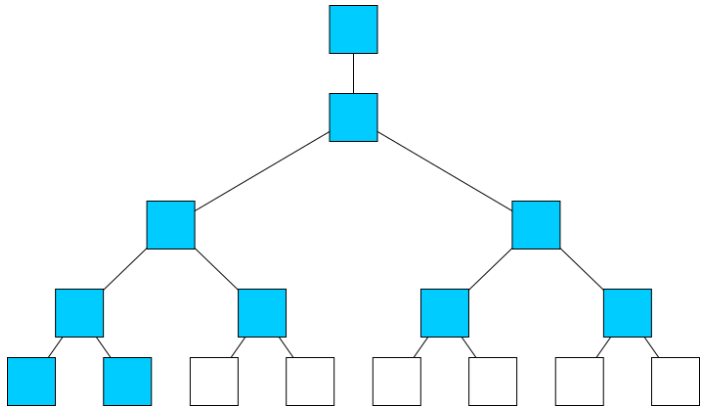
push_back complexity analysis

$$\log_k N \leq \lceil \log_k N \rceil < \log_k N + 1$$

$$\Rightarrow N \leq k^{\lceil \log_k N \rceil} < Nk$$

$$\Rightarrow \frac{N-1}{k-1} \leq numMovCp < \frac{Nk-1}{k-1}$$

$$\Rightarrow \frac{1-\frac{1}{N}}{k-1} \approx C_1 \leq \frac{numMovCp}{N} < \frac{k-\frac{1}{N}}{k-1} \approx C_2$$



Reallocation

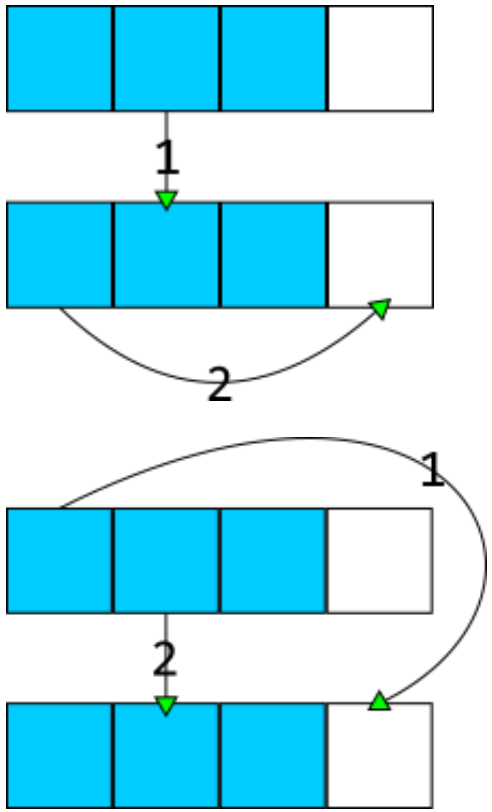
- How can we move/copy old elements?
  - Move
    - Efficient.
    - Not everything can be moved.
    - Move is not always noexcept.
  - Copy
    - Inefficient (imagine we're copying 1'000 std::string).
    - Strong exception guarantee.
- Move when noexcept, copy otherwise.
  - If type is uncopyable, and move is not noexcept, still use move.
  - Dangerous.

emplace_back

- Complexity: Amortized constant.
- Exception: Strong.
- Implementation
  - Use perfect forwarding.
  - emplace_back constructs object inplace.
- Can we emplace_back a self element?

```
vector<int> v{1, 2, 3};
v.emplace_back(v[0]);
```

  - Yes.



push_back

- Complexity: Amortized constant.
- Exception: Strong.
- Implementation
  - Copy or move the input object into vector.
  - Can be costly.
- Can we push_back a self element?
  - Yes.

reserve

- Complexity: Usually O(N)
- Exception: Strong
- Implementation
  - Reallocate new memory.
  - Move/copy old elements.
- Used to enlarge “unused” space.

What's wrong with this?

```
template <class T> struct my_ptr {
    my_ptr(T* p) : m_p{p} {}
    ~my_ptr() { delete m_p; }
};
vector<my_ptr<int>> v;
v.push_back(new int{0});
v.emplace_back(new int{0});
```

- Don't use new, unless we have a reason.

What's wrong with this?

```
vector<int> v;
for (auto i = 0; i < 1000; ++i)
    v.emplace_back(i);
```

- We should do a reserve first.

What’s wrong with this?

```
vector<int> v{1, 2, 3};
auto x = v.at(0);
```

- vector::at is useless, use vector::operator[] instead.

Recommendation

- Design your type so it has noexcept move.
- You should use std::vector by default.
- "Use vectors whenever you can. If you cannot use vectors, redesign your solution so that you can use vectors." - Alexander Stepanov

Thanks for listening.

Question?