Сибирь I

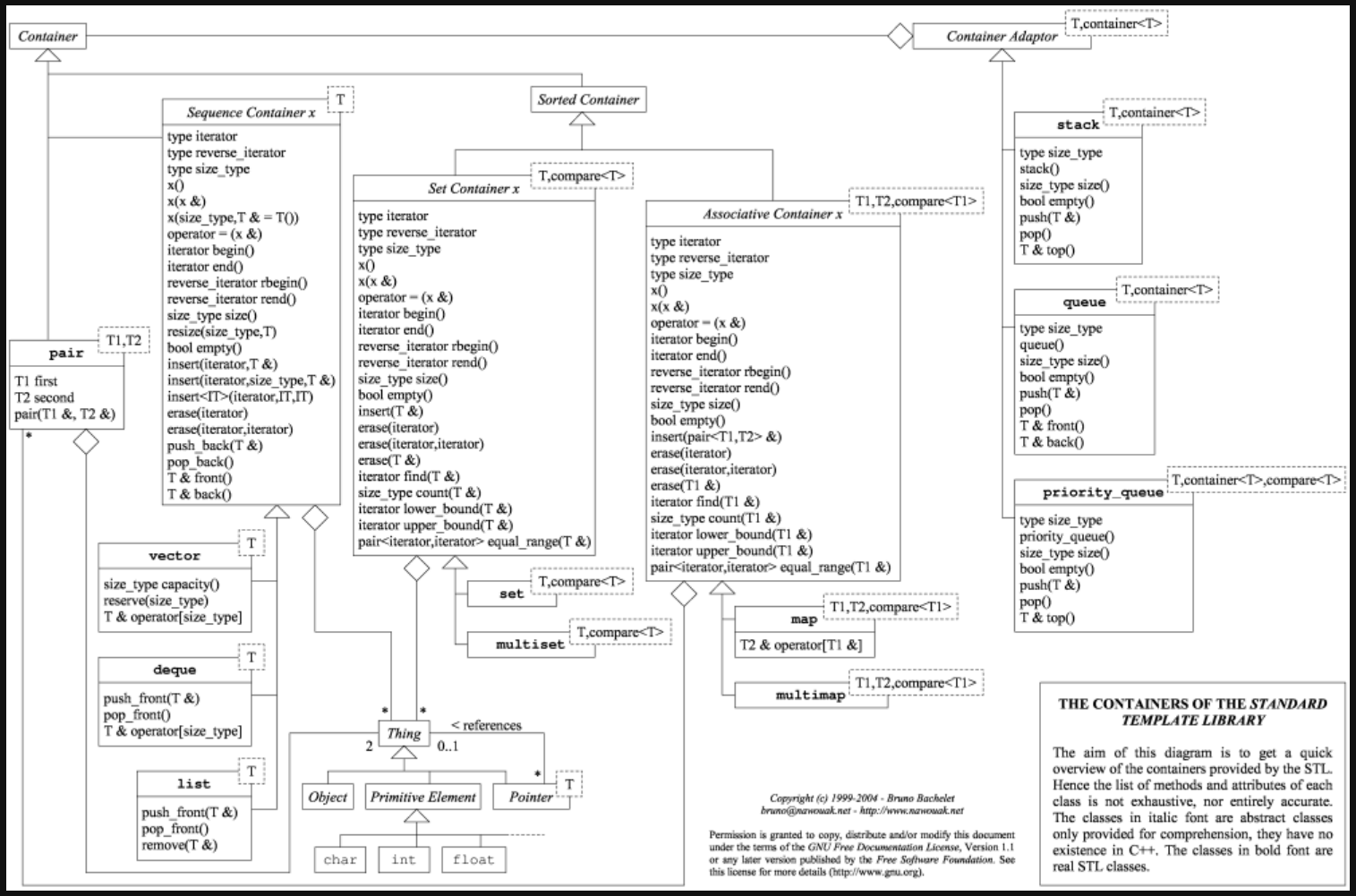
STL Алгоритмы

STL – Standard Template Library

Библиотека STL содержит набор шаблонов, представляющих контейнеры, итераторы, объекты функций и алгоритмов

Зачем оно вообще нужно?

Вот они (контейнеры) слева направо



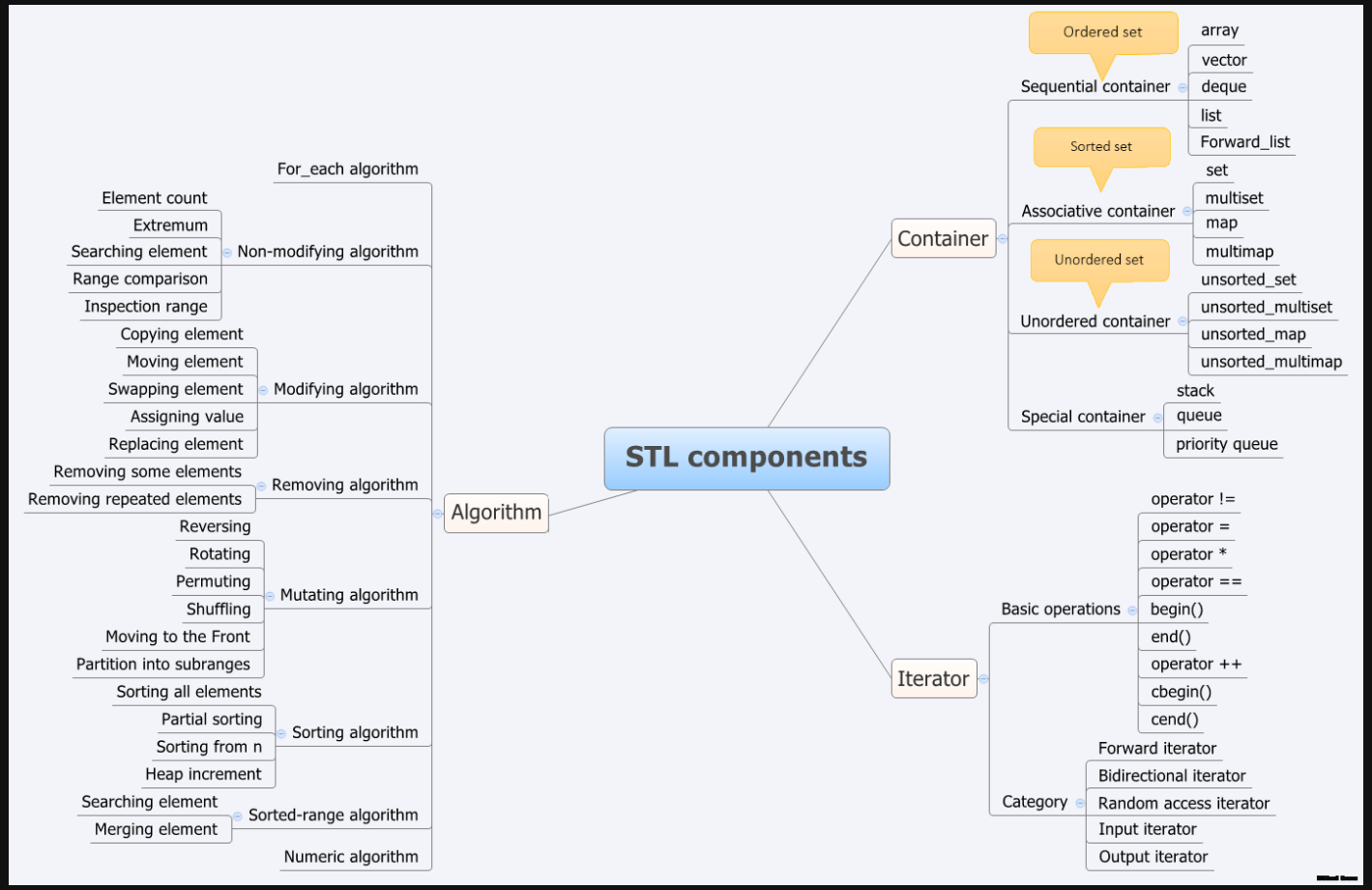
Возьмём std::vector

Часто ли приходится его сортировать, искать в нем элементы?

A в std::list?

Идея STL – обобщенное программирование – создание кода, ***не зависящего от типа данных***

Вот они (алгоритмы) сверху вниз



Небольшой list

- `std::find`
- `std::find_if`
- `std::count_if`
- `std::transform`
- `std::sort`
- `std::any_of`
- `std::for_each`

std::find

Возвращает итератор на первый элемент, равный `value`

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value);
```

std::find_if

Возвращает итератор на первый элемент, удовлетворяющий условию pred

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Pr
```

std::any_of

Проверяет, выполняется ли pred хоть для одного элемента последовательности

```
template< class InputIt, class UnaryPredicate >
bool any_of(InputIt first, InputIt last, UnaryPredicate pred);
```

std::count_if

Возвращает количество элементов, удовлетворяющих условию pred

```
template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
count_if(InputIt first, InputIt last, UnaryPredicate pred);
```

std::for_each

Выполняет функцию для каждого элемента

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunctio
```

std::transform

Возвращает результат применения функции к каждому элементу последовательности

```
template <class InIter, class OutIter, class Func>
OutIter transform(InIter start, InIter end,
      OutIter result, Func unaryfunc);
```

std::sort

Сортирует

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);

template<class RandomAccessIterator, class Predicate>
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          Predicate comp);
```

Хотим отсортировать в порядке убывания

```
bool greater(int a, int b) { return a > b; }

void my_sort() {
  std::vector<int> nums = {1, 2, 3, 0, 4, 5};
  std::sort(nums.begin(), nums.end(), greater);
}
```

lambda functions

aka. анонимные функции

```
[] (параметры) { выражения }
```

Можно сделать предыдущий пример красивее

```
std::sort(s.begin(), s.end(), [](int a, int b) // предикат
                              { return a > b; });
```


LAMBDA
ME
A NORMAL FUNCTION
imgflip.com

Библиотека диапазонов (ranges)

Или немного о C++20

Допустим, мы хотим получить выборку из четных элементов вектора и умножить каждый элемент на 2

При помощи algorithm

```
std::vector<int> nums = {0, 1, 2, 3, 4, 5};

std::vector<int> result;
std::copy_if(nums.begin(), nums.end(), std::back_inserter(result)
             [](int x) { return x % 2 == 0; });
std::transform(result.begin(), result.end(), result.begin(),
               [](int x) { return x * 2; });
```

При помощи ranges:

```
std::vector<int> nums = {0, 1, 2, 3, 4, 5};

auto result =
nums | std::views::filter([](int n) { return n % 2 == 0; })
     | std::views::transform([](int n){ return n * 2; });

// result: 0, 4, 8
```

`range` – диапазон элементов, по которому можно итерироваться. Все контейнеры в STL – это диапазоны.

`view` – определенное представление элементов из диапазона, которое может быть результатом операции. Представление не владеет данными.

Их много

ranges можно применять, когда требуется обработать весь контейнер

```
std::vector<int> very_long_vec;

...

std::ranges::sort(very_long_vec);
// std::sort(very_long_vec.begin(), very_long_vec.end())
```

```
std::unordered_map<std::string, int> freqWord{{"witch", 25}, {"wi
                                              {"tale", 45},  {"do
                                              {"cat", 34},   {"fi
  auto names = std::views::keys(freqWord);
  for (const auto& name : names) {
    std::cout << name << " ";
  }

  auto values = std::views::values(freqWord);
  for (const auto& value : values) {
    std::cout << value << " ";
  }
```

А если мы хотим ключи, отсортированные в обратном порядке?

```
std::map<std::string, int> freqWord{{"witch", 25}, {"wizard", 33}
                                    {"tale", 45},  {"dog", 4},
                                    {"cat", 34},   {"fish", 23}};
  auto names = std::views::reverse(std::views::keys(freqWord));
  for (const auto& name : names) {
    std::cout << name << " ";
  }
```

Композиция функций

Оператор `|` используется в C++20 как *синтаксический сахар* для композиции функций

`R(C)` эквивалентно `C | R`

~~Добро пожаловать в функциональное программирование~~

И выглядит приятно, и читается слева направо

```
std::map<std::string, int> freqWord{{"witch", 25}, {"wizard", 33}
                                    {"tale", 45},  {"dog", 4},
                                    {"cat", 34},   {"fish", 23}};

  for (const auto& name : std::views::keys(freqWord)
                        | std::views::reverse) {
    std::cout << name << " ";
  }
```