Standard Template Library

Шаблоны. Контейнеры

2018

CQG

О чем эта лекция?

1. **STL**
   - Определения
   - Шаблоны – flashback
   - Архитектура STL
   - Откуда появляются итераторы?
   - Автора!
   - STL и OOP

2. **Контейнеры STL**
   - Классификация
   - Подробно о последовательных контейнерах.
   - Управление памятью.
   - vector, deque, list - кто кого?

О чем эта лекция?

«...На нем не было ленточки!
Не было ярлыка!
Не было коробки
и не было мешка!»

Доктор Зюсс, «Как Гринч украл Рождество»

Эпиграф к книге
Scott Meyers “Effective STL”
Addison-Wesley, 2001

О чем эта лекция?

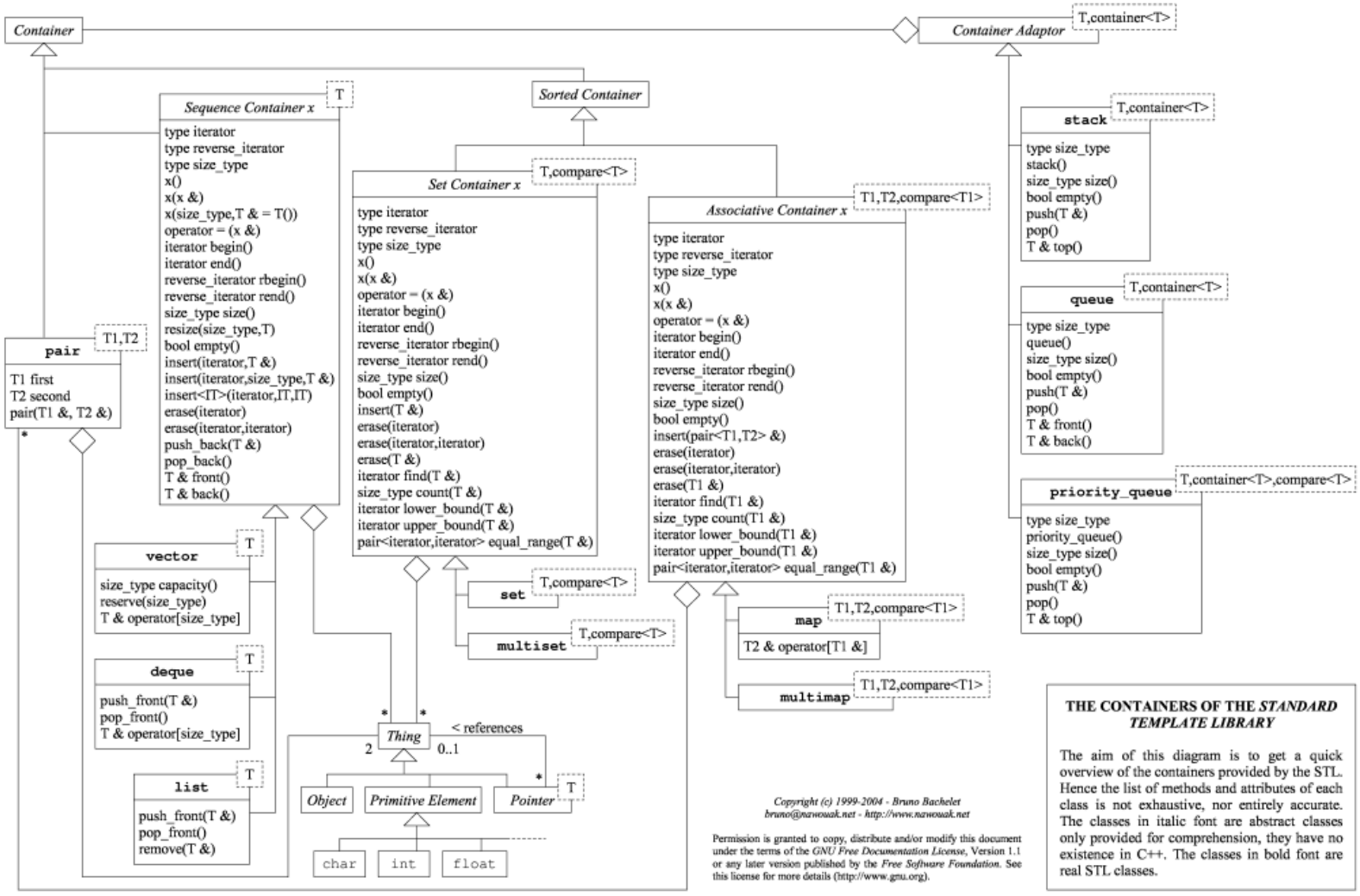
«У STL не существует официального определения, и разные авторы вкладывают в этот термин разный смысл...
...Термин STL относится к **компонентам Стандартной библиотеки C++, работающим с итераторами»**

Скотт Мейерс, «Эффективное использование STL»

«Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальных частей Стандартной Библиотеки (таких, как потоки ввода-вывода (*iostream*), подраздел *Си* и др.)»

Wikipedia

О чем эта лекция?



Templates: «Шаблоны? Мм... Что-то слышал»

Типичный пример, с которого начинают все: функция ***Max()*** для разных типов аргументов

```
double Max( double a, double b )
{
    return (a > b) ? a : b;
}

int Max( int a, int b )
{
    return (a > b) ? a : b;
}

my_type Max( my_type a, my_type b )
{
    return (a > b) ? a : b;
}
```

CQG

Templates: «Шаблоны? Что-то припоминаю...»

```
double Max( double a, double b );
int Max( int a, int b );
my_type1 Max( my_type1 a, my_type1 b );
my_type2 Max( my_type2 a, my_type2 b );
```

Проблемы с решениями такого типа

- **трудно** сопровождать (модифицировать, исправлять)
- **невозможно** заранее знать все типы, для которых понадобится функция

Templates: «Шаблоны? Что-то припоминаю...»

```
double Max( double a, double b );
int Max( int a, int b );
my_type1 Max( my_type1 a, my_type1 b );
my_type2 Max( my_type2 a, my_type2 b );
```

Заставим компилятор
делать за нас тупую работу:

```
template< typename T >
T Max(T a, T b)
{
   return (a > b) ? a : b;
}
```

CQG

Templates: Как это работает

Шаблон

```
template< typename T >
T Max(T a, T b)
{
    return (a > b) ? a : b;
}
```

Вызов функции

```
z = Max( x + 0.5, y - 2.5);
```

Компилятор берет **типы реальных** аргументов вызова

Инстанцирование шаблонной функции

Сгенерированная компилятором функция

```
double Max_double(double a, double b)
{
    return (a > b) ? a : b;
}
```

Сгенеренная компилятором функция для конкретного типа, с расширенным именем

Сгенеренный вызов

```
z = Max_double(x + 0.5, y - 2.5);
```

STL: Components

Стандартная библиотека шаблонов (*Standard Template Library*):

1. **Набор** согласованно работающих вместе обобщённых (шаблонных) ***компонентов*** C++.

2. **Набор *соглашений*** (требований, предъявляемых к интерфейсу компонентов), позволящих ***расширять*** этот набор.

STL: Components

Контейнеры
(структуры хранения данных)

Алгоритмы
(методы работы с данными)

Ортогональная архитектура:

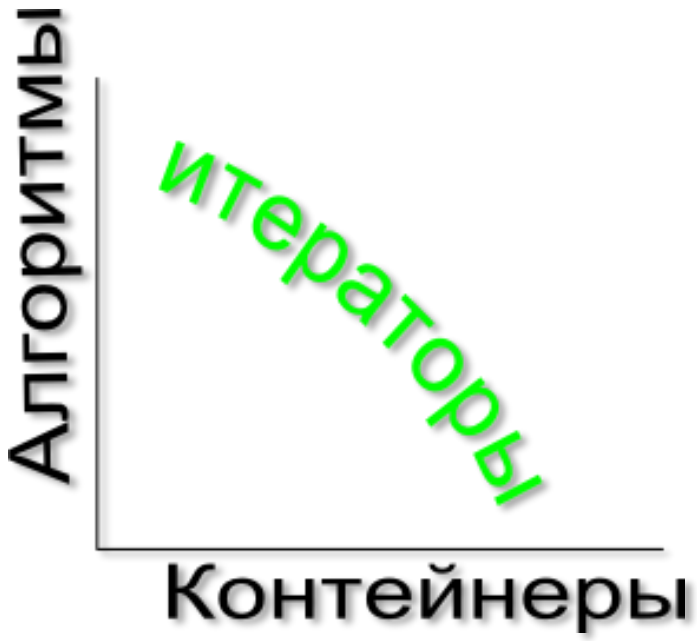
Контейнеры **не знают** об алгоритмах.
Алгоритмы **не знают** о контейнерах.

STL: Components

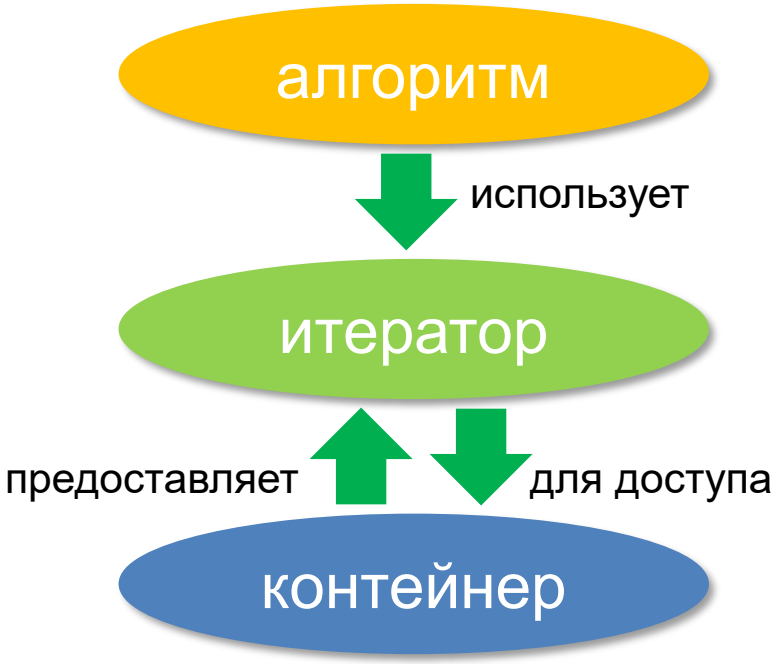
Контейнеры
(структуры хранения данных)

Алгоритмы
(методы работы с данными)

Итераторы
(с их помощью контейнеры предоставляют алгоритмам доступ к своим данным)



STL: Iterators



* Итератор – это ***тип.***

* Любой объект может играть роль итератора, если он удовлетворяет ***определенному набору условий*** .

* Итераторы можно рассматривать как ***обобщение указателей*** .

* Итераторы являются ***интерфейсом взаимодействия между контейнером и алгоритмом***, который им манипулирует .

STL: Intro to Concept of Iterator

Пример: Напишем функцию, которая ищет первый элемент массива, равный заданному числу:

```
int* find0(int* array, int n, int x)
{
    int* p = array;
    for( int i = 0; i < n; ++i )
    {
        if ( *p == x ) return p; // success
        ++p;
    }
    return 0; // fail
}
```

Каковы ограничения, которые накладывает наша реализация?

STL: Intro to Concept of Iterator

```
int* find0(int* array, int n, int x)
{
    int* p = array;
    for( int i = 0; i < n; ++i)
    {
        if ( *p == x ) return p; // success
        ++p;
    }
    return 0; // fail
}
```

1 3 4 2

Ограничения алгоритма:

1. Он ищет int
2. Он сканирует массив int-ов
3. Мы должны передать указатель на первый элемент
4. Мы должны передать число элементов в массиве

Как ослабить ограничения?

STL: Intro to Concept of Iterator

Шаг 1: Обобщим ТИП элементов массива:

```
template< typename T >
T* find1( T* array, int n, const T& x)
{
   T* p = array;
   for( int i = 0; i < n; ++i)
   {
      if ( *p == x ) return p; // success
      ++p;
   }
   return 0; // fail
}
```

Функция стала более общей, однако мы неявно требуем, чтобы тип T имел **`operator==()`**

STL: Intro to Concept of Iterator

Основной недостаток нашего алгоритма (с точки зрения общности) – мы завязаны на специфическую структуру хранения данных: С-массив.

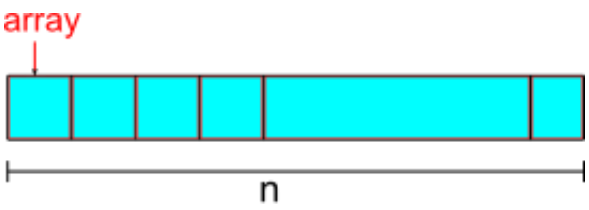
1. Мы знаем **адрес первого элемента** массива.
2. Мы используем **operator++() для перемещения** от одного элемента массива к следующему.
3. Мы используем **информацию о количестве элементов** массива для выхода из цикла.

Попытаемся избавиться от этих ограничений.

STL: Intro to Concept of Iterator

Шаг 2: Избавимся от размера массива:

```
template< typename T >
T* find2( T* array, T* beyond, const T& x)
{
    T* p = array;
    while( p != beyond )
    {
        if ( *p == x ) return p; // success
        ++p;
    }
    return beyond; // fail
}
```



STL: Intro to Concept of Iterator

Шаг 3: Упростим и немного оптимизируем алгоритм:

```
template< typename T >
T* find3( T* first, T* beyond, const T& x)
{
    T* p = first;
    while( p != beyond && *p != x )
    {
        ++p;
    }
    return p; // результат
}
```

Изменения:

1. Мы объединили две проверки вместе, инвертировав условие.
2. Мы убрали один **return**, функция стала короче.
3. Мы переименовали **array** в **first**

STL: Intro to Concept of Iterator

```
template< typename T >
T* find3( T* first, T* beyond, const T& x)
{
    T* p = first;
    while( p != beyond && *p != x )
    {
        ++p;
    }
    return p; // результат
}
```

Что мы получили на этом шаге :

1. Алгоритм ***find3*** производит поиск в структуре данных, состоящей из элементов типа T.
2. Предположение 1: Тип T поддерживает **operator!=()**.
3. Мы получаем доступ к данным с помошью указателей **T***.
4. Предположение 2: Применяя **operator++()** к **T***, мы получаем *указатель на следующий элемент*.

STL: Intro to Concept of Iterator

Шаг 4, последний: Откажемся от указателей совсем!

```
template< typename T, typename P >
P find4( P first, P beyond, const T& x)
{
    P p = first;
    while( p != beyond && *p != x)
    {
        ++p;
    }
    return p; // результат
}
```

STL: Intro to Concept of Iterator

```
template< typename T, typename P >
P find4( P first, P beyond, const T& x)
{
    P p = first;
    while( p != beyond && *p != x)
    {
        ++p;
    }
    return p; // результат
}
```

- Алгоритм **find4** осуществляет поиск в структуре с данными типа **T**
- Доступ к данным осуществляется с помощью объектов типа **P**

Требования/Предположения:

1. Тип **T** должен поддерживать **operator!=()**
2. Тип **P** должен поддерживать **operator*()**, возвращающий значение типа **T**.
3. Тип **P** должен поддерживать **operator++()**, возвращающий значение типа **P**, ассоциированное со *следующим элементом в структуре*.
4. Тип **P** должен поддерживать **operator!=()**

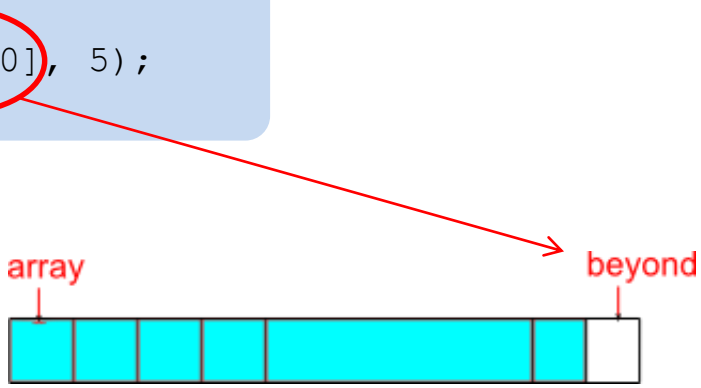
CQG

STL: Intro to Concept of Iterator

```
template< typename T, typename P >
P find5( P first, P beyond, const T& x)
{
   P p = first;
   while( p != beyond && *p != x)
   {
      ++p;
   }
   return p; // result
}
```

Использование с массивами:

```
int A[100];
// initialization...

int* pResult = find4(A, &A[100], 5);
```

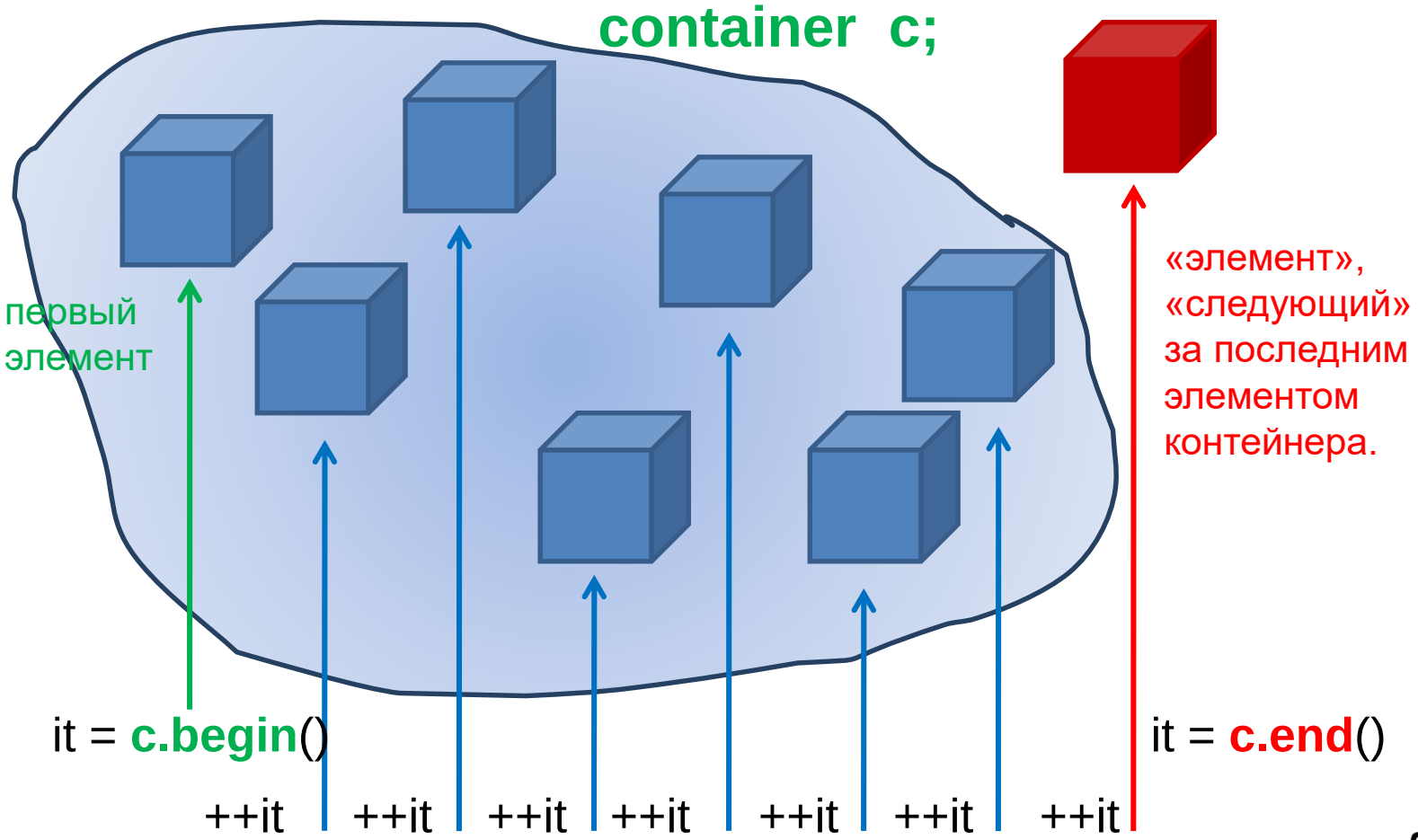


Явное инстанцирование:

T <=> int

P <=> int*

STL: Iterators



STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

Collection::iterator iter1 =  data.begin();
Collection::iterator iter2 =  data.end();

for( ; iter1 != iter2;  ++iter1 )
{
      std::cout << *iter1 << std::endl;
}
```

CQG

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

Collection::iterator iter1 =  data.begin();
Collection::iterator iter2 =  data.end();

iter1 = std::find( iter1, iter2, 4 );   // we pass iterators, NOT the container itself!
if( iter1 != data.end() )
{
      std::cout << *iter1 << std::endl;
}
```

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

Collection::iterator iter1 =  data.begin();
Collection::iterator iter2 =  data.end();

const int sum = std::accumulate( data.begin(), data.end(), 0 );
```

CQG

STL: Iterators

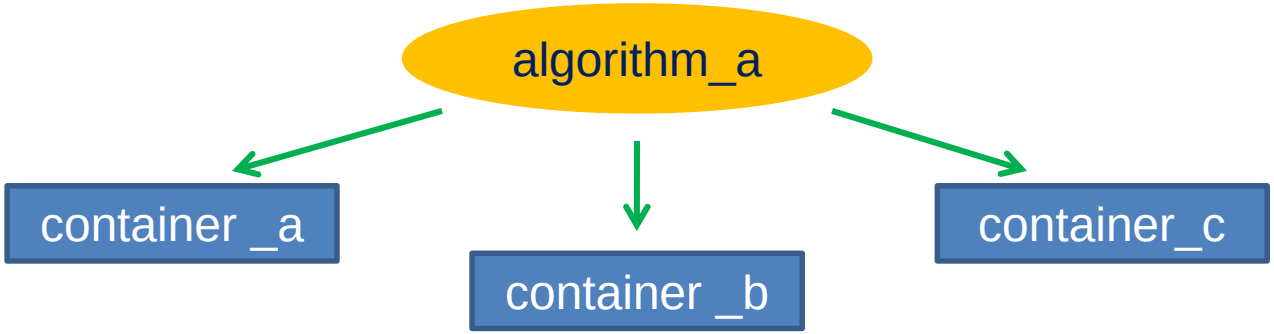
```
using Collection = std::vector<int>;

Collection data;

Collection::iterator iter1 =  data.begin();
Collection::iterator iter2 =  data.end();
Collection::iterator iter3 =  data.find(5);

int sum = std::accumulate( data.begin(), data.end(), 0 );

sum = std::accumulate( data.begin(), iter3, 0 );
```

STL: Iterators

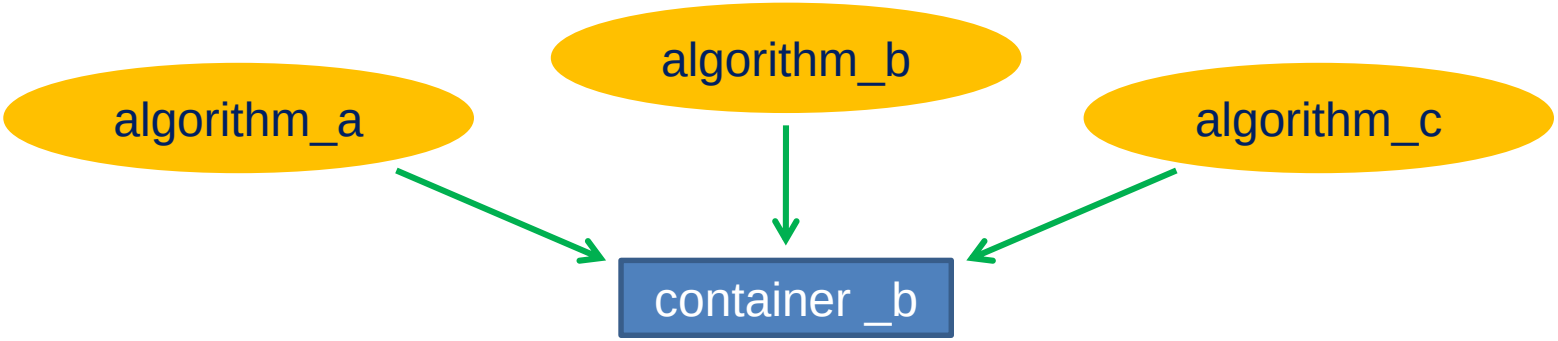
Категория	Поддерживаемые операции
Выходные	operators: **++, *,** copy ctor
Входные	operators: **++, *, ->, =, ==, !=** copy ctor
Однонаправленные	operators: **++, *, ->, =, ==, !=** copy ctor, default ctor
Двунаправленные	operators: **++, --, *, ->, =, ==, !=** copy ctor, default ctor
Произвольного доступа	operators: **++, --, *, ->, =, ==,!=,** **+, -, +=, -=, <, >, <=, >=, []** copy ctor, default ctor

CQG

STL: Гибкость



мы можем применить один и тот же **алгоритм** почти ко всем **контейнерам**



мы можем применить разные **алгоритмы** к одному и тому же **контейнеру**

STL: Гибкость

- **Существующие компоненты STL ортогональны**:
  - ✓ мы можем применить **один и тот же алгоритм** к **разным контейнерам**
  - ✓ мы можем применить **разные алгоритмы** к **одному и тому же контейнеру**

- **STL расширяема «по обеим осям»** :
  - ✓ Программист может определить **свои контейнеры**. Библиотечные алгоритмы будут работать и с ними (при соблюдении соглашений)
  - ✓ Программист может определить **свои алгоритмы**. Они будут работать с библиотечными контейнерами (при соблюдении соглашений)

STL: История

Александр Степанов

- **Александр Степанов** родился в 1950 в Москве. Учился в Московском государственном университете.
- В 1977 Александр уехал в США и начал работать в *General Electric Research Center*. Получил грант для работы над реализацией своих идей обобщённого программирования в виде библиотеки алгоритмов на языке Ada.
- В 1987 получил предложение поработать в *Bell Laboratories*, чтобы реализовать свой подход в виде библиотеки на языке C++. Однако стандарт языка в это время ещё не позволял в полном объёме осуществить задуманное.
- В 1992 он вернулся к работе над алгоритмами. В конце 1993 он рассказал о своих идеях Эндрю Кёнигу, который, высоко оценив их, организовал ему встречу с членами Комитета ANSI/ISO по стандарту C++.
- Весной **1994** библиотека STL, разработанная Александром Степановым при помощи Менг Ли стала частью официального стандарта языка C++

CQG

STL: OOP vs GP

STL (почти) не использует **ООП**
STL – это **GP** (generic programming)

ООП

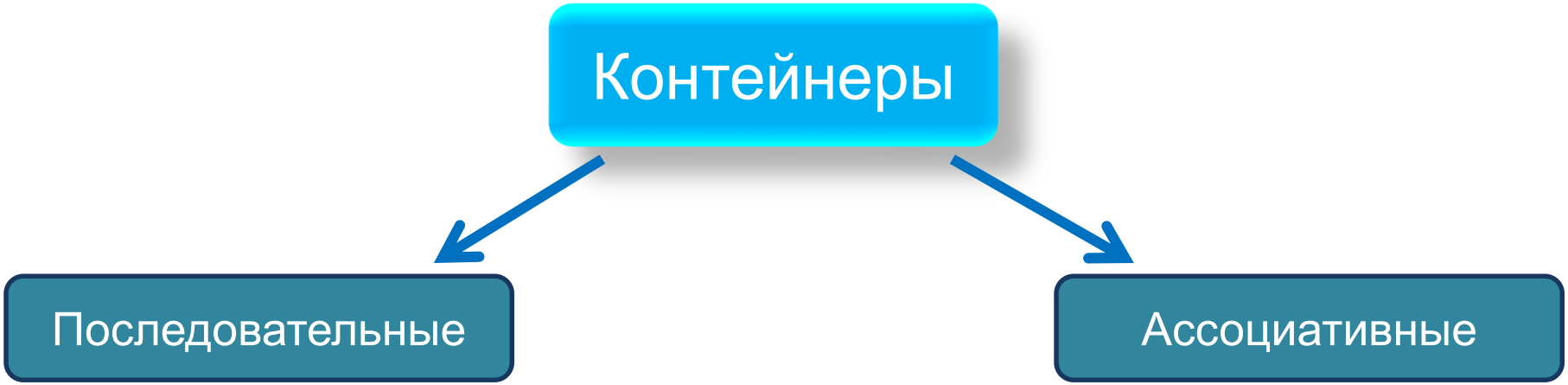
Класс:
- Данные класса
- Операции над данными

STL

Обобщенные контейнеры
(независимые от алгоритмов)

Обобщенные алгоритмы
(независимые от контейнеров)

STL: Containers



STL: Containers

Последовательные

ОСНОВНЫЕ:

vector (расширяемый массив)
deque (двусторонняя очередь)
list (двусвязный список)

ИХ АДАПТЕРЫ:

stack
queue
priority_queue

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

multiset (множество с дубликатами)
multimap (словарь с дубликатами)

unordered_set (hashed)
unordered_map (hashed)

unordered_multiset (hashed)
unordered_multimap (hashed)

CQG

STL: Containers

Последовательные

vector
(расширяемый массив)

```
#include <vector>

std::vector<int> vec;
std::vector<int>::iterator iter;

vec.push_back( 10 );
```

STL: Containers

Последовательные

vector
(расширяемый массив)

```
#include <vector>

using Collection = std::vector<int>;

Collection vec;
Collection::iterator iter;

vec.push_back( 10 );
```

STL: Containers

Последовательные

vector
(расширяемый массив)

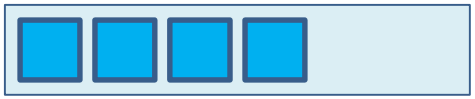
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- возможность управления выделением памяти
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector
(расширяемый массив)



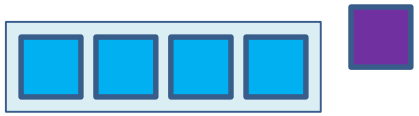
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector
(расширяемый массив)



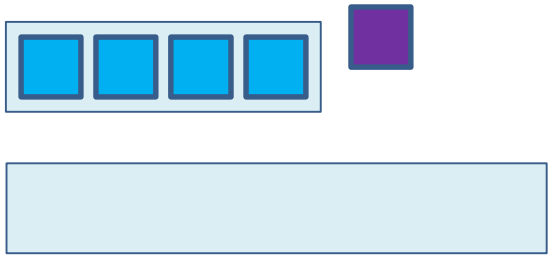
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector
(расширяемый массив)



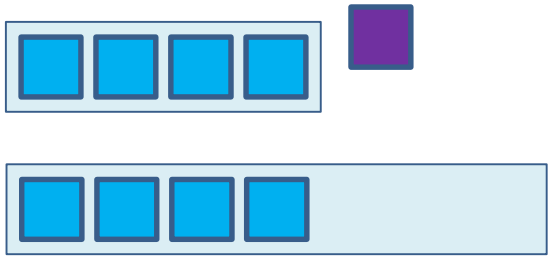
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector
(расширяемый массив)



vector:

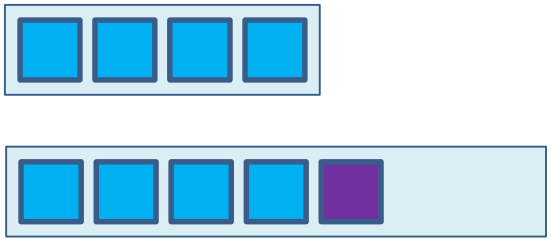
- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

vector
(расширяемый массив)



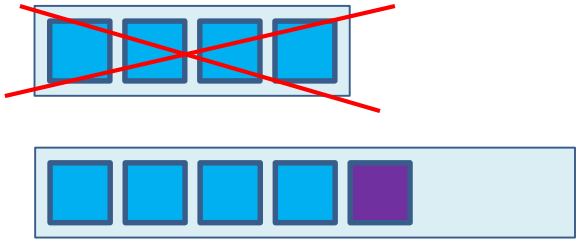
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector
(расширяемый массив)



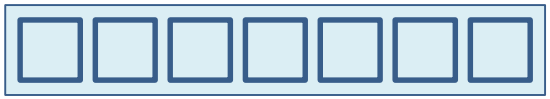
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
```

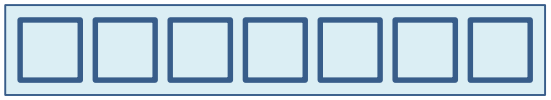
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == 0

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
```

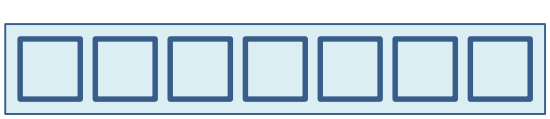
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
```

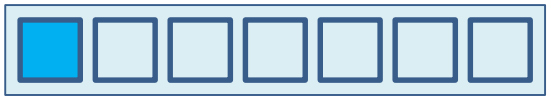
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == **1**

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
```

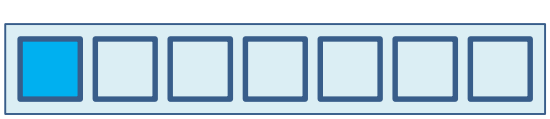
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
data.push_back(item2);
```

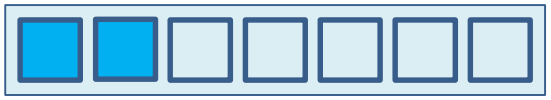
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == **2**

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
data.push_back(item2);
```

vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

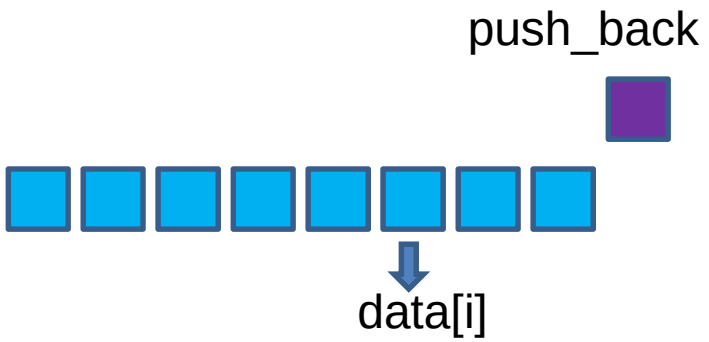
vector
(расширяемый массив)

```
#include <vector>

using Collection = std::vector<int>;

Collection vec;
Collection::iterator iter =  vec.begin();

vec.push_back( 8 );     /// [8]

vec.reserve( 10 );      /// [8xxxxxxxxx]

vec.push_back( 2 );     /// [82xxxxxxxx]

vec.resize( 4, 7 );     /// [8277xxxxxx]

int a = vec[5];
```

STL: Containers

Последовательные

vector (расширяемый массив)



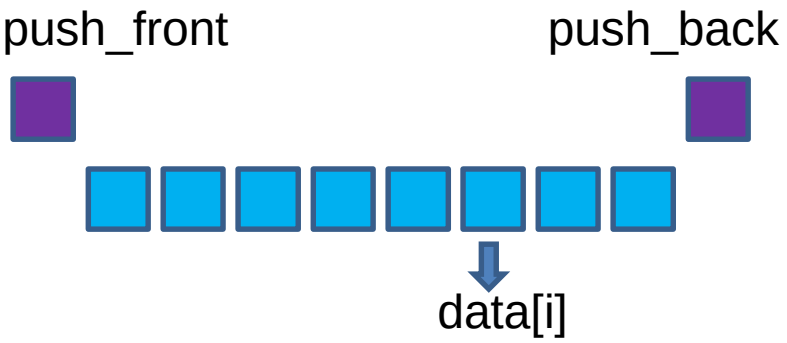
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- возможность управления выделением памяти
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя «очередь»)



deque:

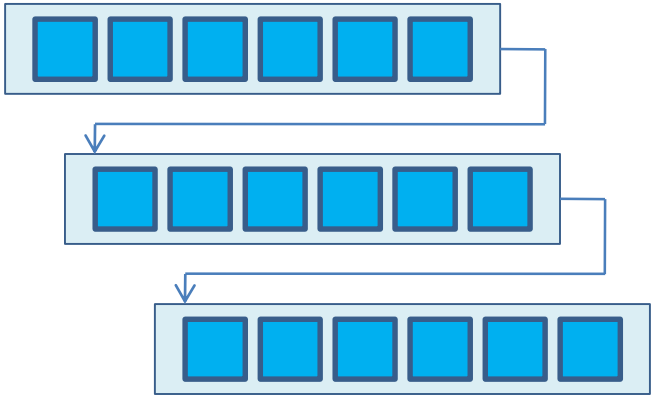
- быстрое добавление **в конец** **и в начало** (push_front , pop_front)
- быстрый доступ к произвольному элементу **[i]** (но медленнее вектора)
- **нет возможности управлять выделением памяти**
- **несовместимость** с С-массивами.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя «очередь»)



deque:

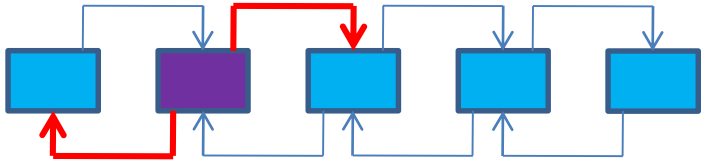
- быстрое добавление **в конец и в начало** (push_front , pop_front)
- быстрый доступ к произвольному элементу **[i]** (но медленнее вектора)
- **нет возможности управлять выделением памяти**
- **несовместимость** с С-массивами.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя «очередь»)
list (двусвязный список)



list:

- **быстрое добавление в любую позицию** (insert, push_back)
- **нет доступа (!)** к произвольному элементу **[i]**
- нет возможности управлять выделением памяти
- несовместимость с C-массивами.
- **итераторы остаются валидны** при любом добавлении-удалении (кроме итератора на сам удаляемый элемент)
- поддерживает ряд оптимизированных методов (аналогов алгоритмов)

CQG