

Углубленное программирование на языке C / C++

Лекция № 7



Санеев Илья

1. Предпосылки создания, назначение и гарантии производительности библиотеки Standard Templates Library (STL).
2. Итераторы STL: итераторы вставки и работа с потоками.
3. Контейнеры и адаптеры STL.
4. Обобщенные алгоритмы: основные характеристики и условия применения. Отношения сравнения.
5. STL в языке C++11.
6. **Постановка задач к практикуму №5.**

Стандартная библиотека шаблонов (STL): история создания



Стандартная библиотека шаблонов ([англ.](#) Standard Templates Library, STL) была задумана в 1970-х – 1990-х гг. А. Степановым, Д. Мюссером (D. Musser) и др. как первая **универсальная библиотека обобщенных алгоритмов и структур данных** и в качестве составной части стандартной библиотеки языка C++ является воплощением результатов изысканий в области теоретической информатики.

It so happened that C++ was the only language in which I could implement such a library to my personal satisfaction.

— Alexander Stepanov (2001)

Предпосылки создания STL



По словам А. Степанова, наибольшее значение при создании STL придавалось следующим фундаментальным идеям:

- **обобщенному программированию** как дисциплине, посвященной построению многократно используемых алгоритмов, структур данных, механизмов распределения памяти и др.;
- достижению **высокого уровня абстракции без потери производительности**;
- следованию **фон-неймановской модели** (в первую очередь — в работе с базовыми числовыми типами данных при эффективной реализации парадигмы процедурного программирования, а не программирования «в математических функциях»);
- использованию семантики **передачи объектов по значению**.

Концептуально в состав STL входят:

- **обобщенные контейнеры** (универсальные структуры данных) — векторы, списки, множества и т.д.;
- **обобщенные алгоритмы** решения типовых задач поиска, сортировки, вставки, удаления данных и т.д.;
- **итераторы** (абстрактные методы доступа к данным), являющиеся обобщением указателей и реализующие операции доступа алгоритмов к контейнерам;
- **функциональные объекты**, в объектно-ориентированном ключе обобщающие понятие функции;
- **адаптеры**, модифицирующие интерфейсы контейнеров, итераторов, функций;
- **распределители памяти.**

«Ключевые ценности» STL



Основное значение в STL придается таким архитектурным ценностям и характеристикам программных компонентов, как

- **многократное использование и эффективность** кода;
- **модульность**;
- **расширяемость**;
- **удобство применения**;
- **взаимозаменяемость** компонентов;
- **унификация** интерфейсов;
- **гарантии вычислительной сложности** операций.

С технической точки зрения, STL представляет собой набор **шаблонов классов и алгоритмов** (функций), предназначенных для совместного использования при решении широкого спектра задач.

Гарантии производительности STL (1 / 2)



Оценки вычислительной сложности обобщенных алгоритмов STL в отношении времени, как правило, **выражаются в терминах** традиционной ***O*-нотации** и призваны показать зависимость **максимального** времени выполнения $T(N)$ алгоритма применительно к обобщенному контейнеру из $N \gg 1$ элементов.

$$T(N) = O(f(N))$$

Наибольшую значимость в STL имеют:

- **константное** время выполнения алгоритма: $T(N) = O(1)$
- **линейное** время выполнения алгоритма: $T(N) = O(N)$
- **квадратичное** время выполнения алгоритма: $T(N) = O(N^2)$
- **логарифмическое** время выполнения алгоритма: $T(N) = O(\log N)$
- время выполнения « **N логарифмов N** »: $T(N) = O(N \log N)$

Гарантии производительности STL (2 / 2)



Недостатком оценки максимального времени является рассмотрение редко встречающихся на практике наихудших случаев (например, quicksort в таком случае выполняется за время $O(N^2)$).

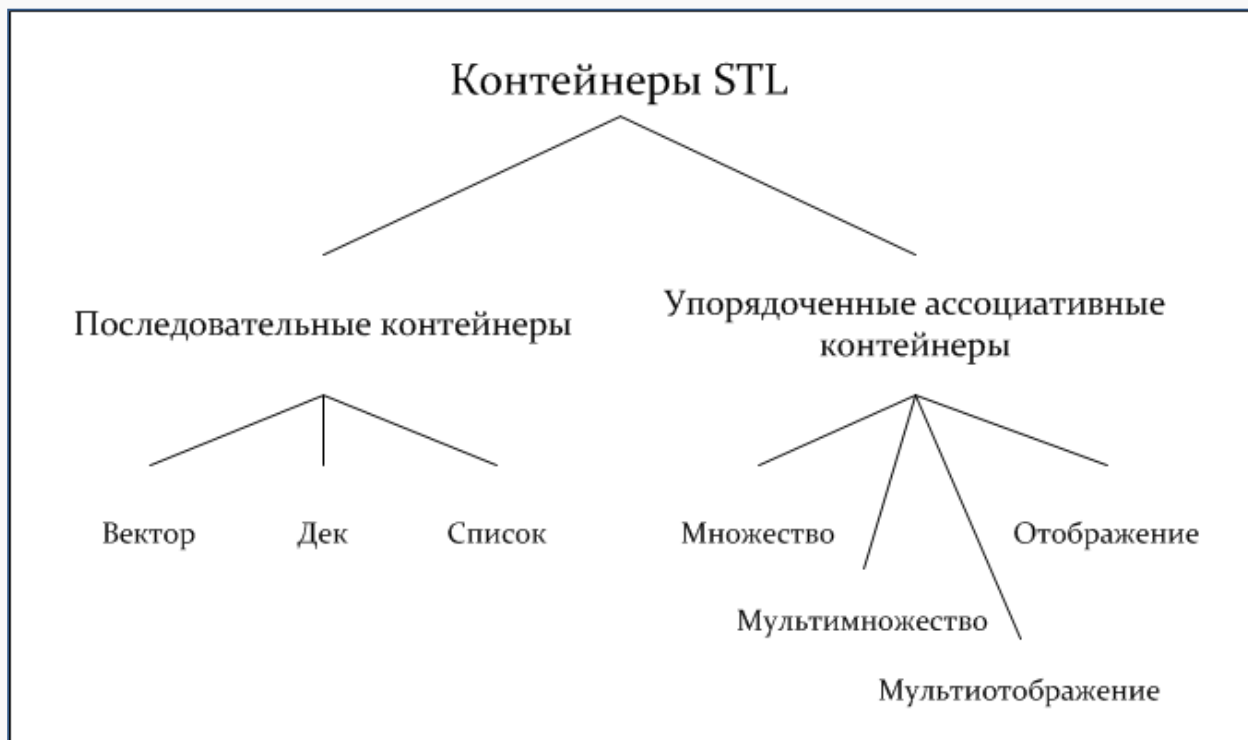
Альтернативными оценке максимального времени являются:

- оценка **среднего** времени (при равномерном распределении N);
- оценка **амортизированного** времени выполнения алгоритма, под которым понимается совокупное время выполнения N операций, деленное на число N .

Контейнеры: обзор



Контейнеры STL — объекты, предназначенные для хранения коллекций других объектов, в том числе и контейнеров.



Последовательные контейнеры



Последовательные контейнеры STL хранят коллекции объектов одного типа `T`, обеспечивая их строгое линейное упорядочение.

Вектор — динамический массив типа `std::vector<T>`, характеризуется произвольным доступом и автоматическим изменением размера при добавлении и удалении элементов.

Дек (двусторонняя очередь, от *англ.* deque — double-ended queue) — аналог вектора типа `std::deque<T>` с возможностью быстрой вставки и удаления элементов в начале и конце контейнера.

Список — контейнер типа `std::list<T>`, обеспечивающий константное время вставки и удаления в любой точке, но отличающийся линейным временем доступа.

Примечание: Последовательными контейнерами STL в большинстве случаев могут считаться массив `T a[N]` и класс `std::string`.

Последовательные контейнеры: сложность основных операций



Вид операции	Вектор	Дек	Список
Доступ к элементу	$O(1)$	$O(1)$	$O(N)$
Добавление / удаление в начале	$O(N)$	Амортизированное $O(1)$	$O(1)$
Добавление / удаление в середине	$O(N)$	$O(N)$	$O(1)$
Добавление / удаление в конце	Амортизированное $O(1)$	Амортизированное $O(1)$	$O(1)$
Поиск перебором	$O(N)$	$O(N)$	$O(N)$

Упорядоченные ассоциативные контейнеры



Упорядоченные ассоциативные контейнеры STL предоставляют возможность быстрого доступа к объектам коллекций переменной длины, основанных на работе с ключами.

Множество — контейнер типа `std::set<T>` с поддержкой уникальности ключей и быстрым доступом к ним. **Мультимножество** — аналогичный множеству контейнер типа `std::multiset<T>` с возможностью размещения в нем ключей кратности 2 и выше.

Отображение — контейнер типа `std::map<Key, T>` с поддержкой уникальных ключей типа `Key` и быстрым доступом по ключам к значениям типа `T`. **Мультиотображение** — аналогичный отображению контейнер типа `std::multimap<Key, T>` с возможностью размещения в нем пар значений с ключами кратности 2 и выше.

Векторы: общие сведения



Вектор — последовательный контейнер

- переменной длины;
- с произвольным доступом к элементам;
- с быстрой вставкой и удалением элементов в конце контейнера;
- с частичной гарантией сохранения корректности итераторов после вставки и удаления.

Технически вектор STL реализован как шаблон с параметрами вида:

```
// 1-й параметр - тип данных, 2-й - распределитель памяти  
template <  
    typename T,  
    typename Allocator = std::allocator<T> >
```

Векторы: встроенные типы



Имя типа	Семантика
<code>iterator</code>	Неконстантный итератор прямого обхода
<code>const_iterator</code>	Константный итератор прямого обхода
<code>reverse_iterator</code>	Неконстантный итератор обратного обхода
<code>const_reverse_iterator</code>	Константный итератор обратного обхода
<code>value_type</code>	Тип значения элемента (T)
<code>pointer</code>	Тип указателя на элемент (T*)
<code>const_pointer</code>	Тип константного указателя на элемент
<code>reference</code>	Тип ссылки на элемент (T&)
<code>const_reference</code>	Тип константной ссылки на элемент
<code>difference_type</code>	Целый знаковый тип результата вычитания итераторов
<code>size_type</code>	Целый беззнаковый тип размера



Векторы: варианты создания



```
// за время  $O(1)$ 
std::vector<T> vector1;

// за время  $O(N)$ , с вызовом  $T::T(T\&)$ 
std::vector<T> vector2(N, value);
// за время  $O(N)$ , с вызовом  $T::T()$ 
std::vector<T> vector3(N);

// за время  $O(N)$ 
std::vector<T> vector4(vector3);
std::vector<T> vector5(first, last);
```



Изменение размеров вектора



```
#include <vector>

std::vector<std::string> v = {"foo", "bar"};
v.reserve(42);
// перевыделение памяти при v.size == v.capacity
v.push_back("baz");
v.emplace_back("last");

void foo(std::string *, std::size_t);
foo(v.data(), v.size());

v.shrink_to_fit(); // C++11
std::vector<int>(v).swap(v); // shrink_to_fit C++03
std::vector<int>().swap(v); // clear + shrink_to_fit C++03
```




Массивы (C++11)



Класс-обёртка над статическим массивом

```
#include <array>
```

```
std::array<std::string, 3> a = {"foo", "bar", "baz"};
```

```
std::cout << a.size() << std::endl;
```

```
std::cout << a[1] << std::endl;
```

```
// operator[] != at
```

```
try {
```

```
std::cout << a.at(3) << std::endl;
```

```
} catch (std::out_of_range ex) {
```

```
}
```

Деки: общие сведения



Дек — последовательный контейнер

- переменной длины;
- с произвольным доступом к элементам;
- с быстрой вставкой и удалением элементов в начале и конце контейнера;
- без гарантии сохранения корректности итераторов после вставки и удаления.

Технически дек реализован как шаблон с параметрами вида:

```
template <  
    typename T,  
    typename Allocator = std::allocator<T> >
```

Предоставляемые встроенные типы и порядок конструкции аналогичны таковым для контейнера `std::vector<T>`.

Деки: пример



```
#include <deque>
```

```
std::deque<std::string> d = {"foo", "bar"};
```

```
d.emplace_back("baz");
```

```
d.emplace_front("first");
```

```
std::cout << d[1] << std::endl;
```

Список — последовательный контейнер

- переменной длины;
- с двунаправленными итераторами для доступа к элементам;
- с быстрой вставкой и удалением элементов в любой позиции;
- со строгой гарантией сохранения корректности итераторов после вставки и удаления.

Технически список реализован как шаблон с параметрами вида:

```
template <  
    typename T,  
    typename Allocator = std::allocator<T> >
```

Предоставляемые встроенные типы и порядок конструкции аналогичны таковым для контейнера `std::vector<T>`.

Списки: пример



```
#include <list>
```

```
std::list<std::string> l = {"foo", "bar"};
```

```
l.emplace_back("baz");
```

```
l.emplace_front("first");
```

```
std::cout << l.front() << std::endl;
```

Односвязные списки (C++11)



```
#include <forward_list>
```

```
std::forward_list<std::string> fl = {"foo", "bar"};
```

```
fl.emplace_front("baz");
```

```
fl.push_front("first");
```

```
// *_after методы
```

```
// before_begin – итератор перед первым элементом
```

Списки: описание интерфейса (методы упорядочения)



Название метода	Назначение
sort	Аналогично алгоритму <code>std::sort()</code>
unique	Аналогично алгоритму <code>std::unique()</code>
merge	Аналогично алгоритму <code>std::merge()</code>
reverse	Аналогично алгоритму <code>std::reverse()</code>
remove remove_if	Аналогично алгоритму <code>std::remove()</code> , но с одновременным сокращением размера контейнера

Множества и мультимножества: общие сведения



Множества, мультимножества — упорядоченные ассоциативные контейнеры

- переменной длины;
- с двунаправленными итераторами для доступа к элементам;
- с логарифмическим временем доступа.

Технически множества и мультимножества STL реализованы как шаблоны с параметрами вида:

```
// 1-й пар. - тип ключа, 2-й - функция сравнения
```

```
template <
```

```
    typename Key,
```

```
    typename Compare    = std::less<Key>,
```

```
    typename Allocator = std::allocator<Key> >
```


Множества и мультимножества: встроенные типы



Итераторы:

- `iterator, const_iterator;`
- `reverse_iterator, const_reverse_iterator.`

Прочие встроенные типы — аналогичны встроенным типам последовательных контейнеров (`value_type` — тип значения элемента (`Key`)) со следующими дополнениями:

- `key_type` — тип значения элемента (`Key`);
- `key_compare` — тип функции сравнения (`Compare`);
- `value_compare` — тип функции сравнения (`Compare`).

Примечание: функция сравнения определяет отношение порядка на множестве ключей и позволяет установить их эквивалентность (ключи `K1` и `K2` эквивалентны, когда `key_compare(K1, K2)` и `key_compare(K2, K1)` одновременно ложны).



Множества и мультимножества: варианты создания



```
// сигнатуры конструктора std::set::set()  
set(const Compare& comp = Compare());  
  
template <typename InputIterator>  
set(InputIterator first, InputIterator last,  
    const Compare& comp = Compare());  
  
set(const set<Key, Compare, Allocator>& rhs);  
  
// мультимножества создаются аналогично
```

Отображения и мультиотображения: общие сведения



Отображения, мультиотображения — упорядоченные ассоциативные контейнеры переменной длины:

- моделирующие структуры данных типа «ассоциативный массив с (не)числовой индексацией»;
- с двунаправленными итераторами для доступа к элементам;
- с логарифмическим временем доступа.

Технически отображения и мультиотображения STL реализованы как шаблоны с параметрами вида:

```
// 1-й, 2-й пар. - тип ключа и связанных данных,  
// 3-й - функция сравнения  
  
template <typename Key, typename T,  
          typename Compare = std::less<Key>,  
          typename Allocator =  
          std::allocator<std::pair<const Key, T> > >
```

Отображения и мультиотображения: встроенные типы, варианты создания



Итераторы:

- `iterator;`
- `const_iterator;`
- `reverse_iterator;`
- `const_reverse_iterator.`

Прочие встроенные типы — аналогичны встроенным типам последовательных контейнеров (`value_type` — тип `std::pair<const Key, T>`) со следующими дополнениями:

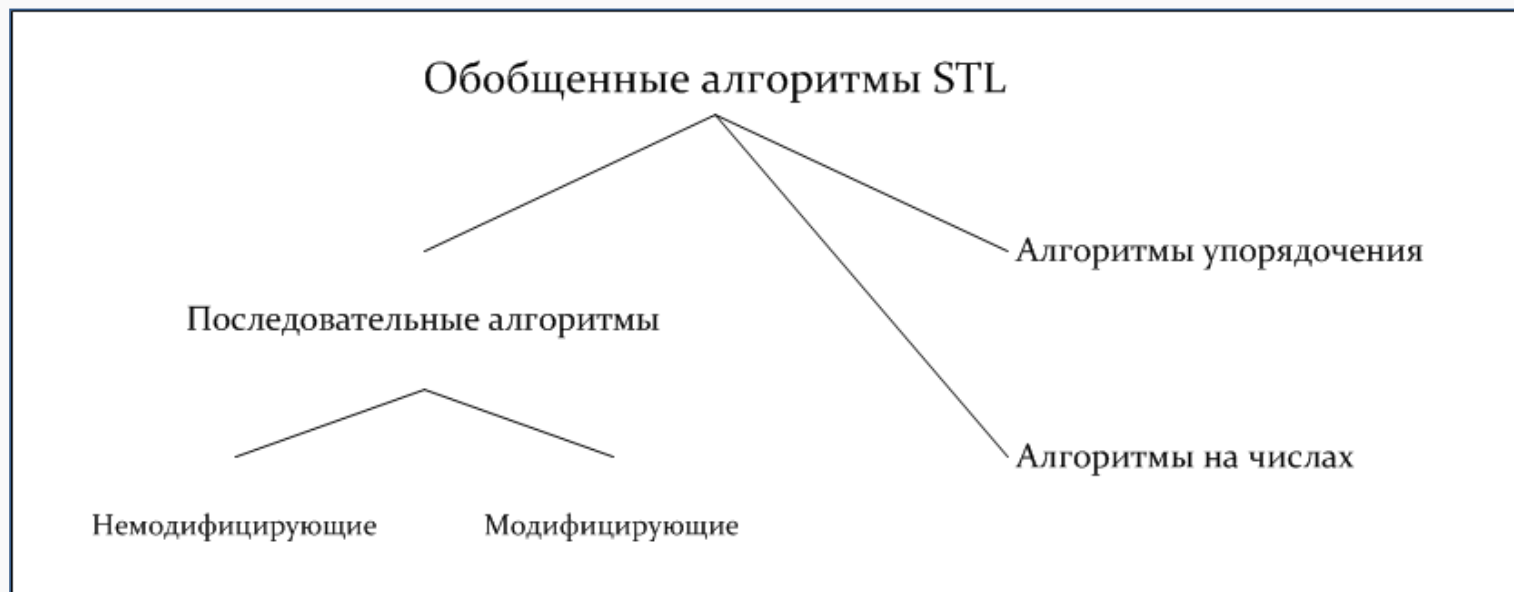
- `key_type` — тип значения элемента (`Key`);
- `key_compare` — тип функции сравнения (`Compare`);
- `value_compare` — тип функции сравнения двух объектов типа `value_type` только на основе ключей.

Порядок конструкции аналогичен таковому для контейнеров `std::set<T>` и `std::multiset<T>`.

Обобщенные алгоритмы: обзор



Обобщенные алгоритмы STL предназначены для эффективной обработки обобщенных контейнеров и делятся на четыре основных группы.



Немодифицирующие последовательные алгоритмы — не изменяют содержимое контейнера-параметра и решают задачи поиска перебором, подсчета элементов и установления равенства двух контейнеров.

- Например: `std::find()`, `std::equal()`, `std::count()`.

Модифицирующие последовательные алгоритмы — изменяют содержимое контейнера-параметра, решая задачи копирования, замены, удаления, размешивания, перестановки значений и пр.

- Например: `std::copy()`, `std::random_shuffle()`, `std::replace()`.

Алгоритмы упорядочения.

Алгоритмы на числах



Алгоритмы упорядочения — все алгоритмы STL, работа которых опирается на наличие или установление отношения порядка на элементах. К данной категории относятся алгоритмы сортировки и слияния последовательностей, бинарного поиска, а также теоретико-множественные операции на упорядоченных структурах.

- Например: `std::sort()`, `std::binary_search()`, `std::set_union()`.

Алгоритмы на числах — алгоритмы обобщенного накопления, вычисления нарастающего итога, попарных разностей и скалярных произведений.

- Например: `std::accumulate()`, `std::partial_sum()`, `std::inner_product()`.

Копирующие, предикатные и алгоритмы, работающие на месте



Среди обобщенных алгоритмов STL выделяют:

- **работающие на месте** — размещают результат поверх исходных значений, которые при этом безвозвратно теряются;
- **копирующие** — размещают результат в другом контейнере или не перекрывающей входные значения области того же контейнера;
- **принимающие функциональный параметр** — допускают передачу на вход (обобщенной) функции с одним или двумя параметрами.

Наибольшее значение среди функций, принимаемых на вход обобщенными алгоритмами, имеют следующие:

- обобщенная функция двух аргументов типа `T`, возвращающая значение типа `T`; может наследоваться от `std::binary_function<T, T, T>`;
- обобщенная логическая функция (предикат) одного аргумента; может наследоваться от `std::unary_function<T, bool>`;
- обобщенная логическая функция (предикат) двух аргументов; может наследоваться от `std::binary_function<T, T, bool>`.

Отношения сравнения (1 / 2)



Используемые в обобщенных алгоритмах STL **отношения сравнения** формально являются **бинарными предикатами**, к которым — для получения от алгоритмов предсказуемых результатов — предъявляется ряд требований. Так, если отношение сравнения R определяется на множестве S , достаточно (но более, чем необходимо!), чтобы:

- для всех $x, y, z \in S$ имело быть утверждение: $xRy \wedge yRz \Rightarrow xRz$;
- для всех $x, y \in S$ имело быть только одно из следующих утверждений: xRy или yRx или $x = y$.

Отвечающее указанным требованиям отношение сравнения является **строгим полным порядком** и реализуется, например:

- операцией `<` над базовыми типами языка C++;
- операцией-функцией `operator<()` класса `std::string`;
- входящим в состав STL предикатным функциональным объектом `less<T>`.

Отношения сравнения (2 / 2)



Необходимым условием применимости бинарного предиката R как отношения сравнения в алгоритмах STL является допущение о том, что элементы $x, y \in S$, для которых одновременно неверны утверждения xRy , yRx , $x = y$, тем не менее признаются эквивалентными (по отношению R — **строгий слабый порядок**).

В этом случае **любые два элемента, взаимное расположение которых по отношению R не определено, объявляются эквивалентными.**

Примечание: такая трактовка эквивалентности не предполагает никаких суждений относительно равенства элементов, устанавливаемого операцией сравнения `==`.

- Например: сравнение строк без учета регистра символов.

Обратные отношения



При необходимости отношение C , обратное R на множестве S , такое, что $xCy \Leftrightarrow yRx$, может быть смоделировано средствами STL.

Так, при наличии `operator<()` для произвольного типа T обратное отношение определяется реализованным в STL шаблоном обобщенной функции сравнения вида:

```
template <typename T>
inline bool operator > (const T& x, const T& y) {
    return y < x;
}
```

Для удобства использования данная функция инкапсулирована в предикатный функциональный объект `std::greater<T>()`.

Алгоритмы сортировки



Название алгоритма	Назначение	Наибольшее время
<code>std::sort()</code>	Нестабильная сортировка на месте (вариант <code>quicksort</code>) в среднем за $O(N \log N)$	$O(N^2)$
<code>std::partial_sort()</code>	Нестабильная сортировка на месте (вариант <code>heapsort</code> ; допускает получение отсортированного поддиапазона длины k)	$O(N \log N)$ или $O(N \log k)$
<code>std::stable_sort()</code>	Стабильная сортировка на месте (вариант <code>mergesort</code> ; адаптируется к ограничениям памяти, оптимально — наличие памяти под $N/2$ элементов)	От $O(N \log N)$ до $O(N(\log N)^2)$ (при отсутствии памяти)

Операции над множествами и хипами: обзор



Реализуемые обобщенными алгоритмами STL операции над множествами имеют **традиционное теоретико-множественное значение** и выполняются над отсортированными диапазонами, находящимися **в любых контейнерах STL**.

В дополнение к прочим STL вводит в рассмотрение такую структуру данных, как хип. **Хип** ([англ.](#) max heap) — порядок организации данных с произвольным доступом к элементам в диапазоне итераторов $[a; b)$, при котором:

- значение, на которое указывает итератор a , является наибольшим в диапазоне и может быть удалено из хипа операцией извлечения ([pop](#)), а новое значение — добавлено в хип за время $O(\log N)$ операцией размещения ([push](#));
- результатами операций [push](#) и [pop](#) являются корректные хипы.

Алгоритмы на числах: обзор



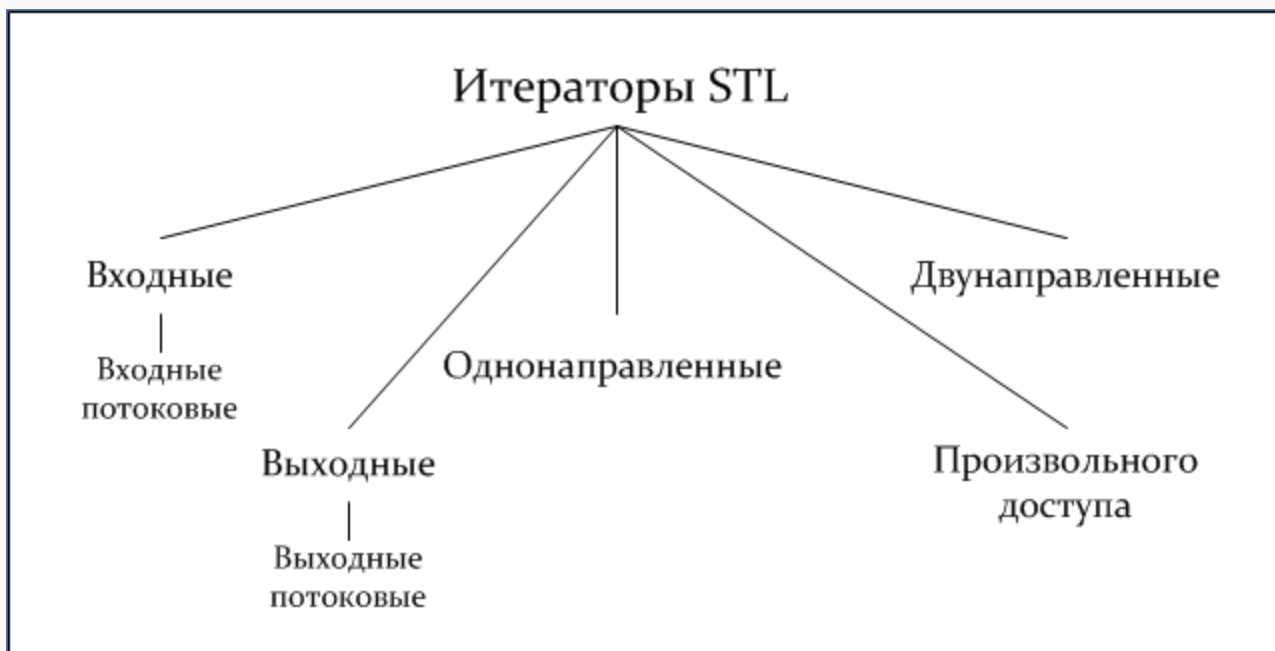
Алгоритмы на числах — алгоритмы обобщенного накопления, вычисления нарастающего итога, попарных разностей и скалярных произведений.

Название алгоритма	Вход	Выход
<code>std::accumulate()</code>	$x_0, x_1, x_2, \dots, x_{N-1}$	$a + \sum_{i=0}^{N-1} x_i$ или $a \circ x_0 \circ x_1 \circ \dots \circ x_{N-1}$
<code>std::partial_sum()</code>	$x_0, x_1, x_2, \dots, x_{N-1}$	$x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots, \sum_{i=0}^{N-1} x_i$
<code>std::adjacent_difference()</code>	$x_0, x_1, x_2, \dots, x_{N-1}$	$x_1 - x_0, x_2 - x_1, \dots, x_{N-1} - x_{N-2}$
<code>std::inner_product()</code>	$x_0, x_1, x_2, \dots, x_{N-1}$ $y_0, y_1, y_2, \dots, y_{N-1}$	$\sum_{i=0}^{N-1} x_i \times y_i$ или $(x_0 * y_0) \circ \dots \circ (x_{N-1} * y_{N-1})$

Итераторы: обзор



Итераторы (**обобщенные указатели**) — объекты, предназначенные для обхода последовательности объектов в обобщенном контейнере. В контейнерных классах являются вложенными типами данных.



Допустимые диапазоны и операции



Категории итераторов различаются наборами операций, которые они гарантированно поддерживают.

*i (чтение)	== !=	++i i++	*i (запись)	--i i--	+ - += -= < > <= >=
Входные (find)			Запрещено		
Запрещено		Выходные (copy)			
Однонаправленные (replace)					
Двунаправленные (reverse)					
Произвольного доступа (binary_search)					

Обход контейнера итератором осуществляется в пределах диапазона, определяемого парой итераторов (обычно с именами **first** и **last**, соответственно). При этом итератор **last** никогда не разыменовывается: [**first**; **last**).

Встроенные указатели C++



Встроенные типизированные указатели C++ по своим возможностям эквивалентны итераторам произвольного доступа и могут использоваться как таковые в любом из обобщенных алгоритмов STL.

```
const int N = 100;
int a[N], b[N];

// ...

std::copy(&a[0], &a[N], &b[0]);
std::replace(&a[0], &a[N / 2], 0, 42);
```

Итераторы в стандартных контейнерах: общие сведения



Шаблоны классов контейнеров STL содержат определения следующих типов итераторов:

- изменяемый итератор прямого обхода (допускает преобразование к константному итератору (см. ниже); *i — ссылка):

`Container<T>::iterator`

- константный итератор прямого обхода (*i — константная ссылка):

`Container<T>::const_iterator`

- изменяемый итератор обратного обхода:

`Container<T>::reverse_iterator`

- константный итератор обратного обхода:

`Container<T>::const_reverse_iterator`

Итераторы вставки (1 / 2)



Итераторы вставки «переводят» обобщенные алгоритмы из «режима замены» в «режим вставки», при котором **разыменование итератора `*i`** влечет за собой **добавление элемента** при помощи одного из предоставляемых контейнером методов вставки.

С технической точки зрения, реализованные в STL итераторы вставки являются шаблонами классов, единственным параметром которых является контейнерный тип `Container`:

- `std::back_insert_iterator<Container>` — использует метод класса `Container::push_back()`;
- `std::front_insert_iterator<Container>` — использует метод класса `Container::push_front()`;
- `std::insert_iterator<Container>` — использует метод класса `Container::insert()`.

Итераторы вставки (2 / 2)



Практическое использование итераторов вставки, формируемых «на лету», упрощает применение шаблонов обобщенных функций `std::back_inserter()`, `std::front_inserter()` и `std::inserter()` вида:

```
template <typename Container>
inline std::back_inserter_iterator<Container>
back_inserter(Container &c) {
    return std::back_inserter_iterator<Container>(c);
}

std::copy(list1.begin(), list1.end(),
          back_inserter(vector1));
// back_inserter_iterator<std::vector<int> >(vector1));
```

Потоковые итераторы STL предназначены **для обеспечения работы обобщенных алгоритмов со стандартными потоками ввода-вывода**. Технически представляют собой шаблоны классов:

- `std::istream_iterator<T>` — входной потоковый итератор;
- `std::ostream_iterator<T>` — выходной потоковый итератор.

Конструкторы:

- `std::istream_iterator<T>(std::istream&)` — входной итератор для чтения значений типа `T` из заданного входного потока;
- `std::istream_iterator<T>()` — входной итератор – маркер «конец потока» (англ. EOS, end-of-stream);
- `std::ostream_iterator<T>(std::ostream&, char*)` — выходной итератор для записи значений типа `T` в заданный выходной поток через указанный разделитель.



Пример: потоковый итератор; обобщенный алгоритм find



```
// 3-й и 4-й пар. - рабочий итератор и end-of-stream (EOS)
std::merge(vector1.begin(), vector1.end(),
           std::istream_iterator<int>(std::cin),
           std::istream_iterator<int>(),
           std::back_inserter(list1));

template <typename InputIterator, typename T>
InputIterator find(
    InputIterator first, // 1 // поиск перебором
    InputIterator last, // 2 // конец диапазона
    const T& value) { // 3 // значение
    while(first != last && *first != value) // 4
        ++first; // 5
    return first; // 6
}
```



Пример: обобщенный алгоритм copy



```
template <typename InputIterator, 1  
          typename OutputIterator> 2  
OutputIterator copy(InputIterator first,  
                    InputIterator last,  
                    OutputIterator result) {  
    while(first != last) {  
        3 *result = *first;  
        ++first;  
        4 ++result;  
    }  
    return first;  
}
```



Пример: обобщенный алгоритм replace



```
template <typename ForwardIterator1, typename T>
void replace(ForwardIterator first,
             ForwardIterator last,
             const T& x, const T& y) {
    while(first != last) {
        if(*first == x) 2
            *first = y; 3
            ++first; 4
    }
    return first;
}
```


Функциональные объекты: обзор



Функциональные объекты (**обобщенные функции**) — программные компоненты, применимые к известному количеству фактических параметров (числом 0 и более) для получения значения или изменения состояния вычислительной системы.

STL-расширением функции является пользовательский **объект типа класса** (**class**) или **структуры** (**struct**) с перегруженной **операцией-функцией** **operator()**.

Базовыми классами стандартных функциональных объектов STL выступают шаблоны структур **std::unary_function** и **std::binary_function**.



Функциональные объекты: базовые классы



```
template <typename Arg, typename Result>
struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
};

template <typename Arg1,
          typename Arg2, typename Result>
struct binary_function {
    typedef Arg1    first_argument_type;
    typedef Arg2    second_argument_type;
    typedef Result  result_type;
};
```



Стандартные функциональные объекты STL (1 / 2)



```
// для арифметических операций  
  
template<typename T> struct plus;           // сложение  
template<typename T> struct minus;         // вычитание  
  
template<typename T> struct multiplies;    // умножение  
template<typename T> struct divides;       // деление  
  
template<typename T> struct modulus;        // остаток  
template<typename T> struct negate;        // инверсия знака
```



Стандартные функциональные объекты STL (2 / 2)



```
// для операций сравнения

template<typename T> struct equal_to;           // равно
template<typename T> struct not_equal_to;       // не равно
template<typename T> struct greater;           // больше
template<typename T> struct less;               // меньше

// больше или равно

template<typename T> struct greater_equal;
template<typename T> struct less_equal; // меньше или равно


// для логических операций

template<typename T> struct logical_and;        // конъюнкция
template<typename T> struct logical_or;         // дизъюнкция
template<typename T> struct logical_not;        // отрицание
```



Пример: функциональный объект multiplies



```
template <typename T> 1  
class multiplies : public binary function<T, T, T> {  
public: 2  
    T operator() (const T& x, const T& y) const {  
        return x * y;  
    }  
};
```

Адаптеры модифицируют интерфейс других компонентов STL и технически представляют собой шаблоны классов, конкретизируемые шаблонами контейнеров, итераторов и др.



Контейнерные адаптеры (1 / 2)



С технической точки зрения, **контейнерные адаптеры STL** являются **шаблонами классов**, конкретизируемыми **типами** хранимых в них **элементов** и **несущих** последовательных **контейнеров** (адаптер `std::priority_queue` требует также функции сравнения, по умолчанию — `std::less<T>`).

Адаптер `std::stack` допускает конкретизацию вида:

- `std::stack< T >` (эквивалентно `std::stack< T, std::deque<T> >`)
- `std::stack< T, std::vector<T> >`
- `std::stack< T, std::list<T> >`

Контейнерные адаптеры (2 / 2)



Адаптер `std::queue` допускает конкретизацию вида:

- `std::queue< T >` (эквивалентно `std::queue< T, std::deque<T> >`);
- `std::queue< T, std::deque< T > >`.

Адаптер `priority_queue` допускает конкретизацию вида:

- `std::priority_queue< T >` (эквивалентно `std::priority_queue< T, std::vector<T>, std::less<T> >`);
- `std::priority_queue< T, std::deque<T>, std::greater<T> >`.

Функциональные адаптеры решают задачу конструирования новых функций из существующих и технически представляют собой шаблоны функций и классов.

Наибольшее практическое значение имеют следующие адаптеры:

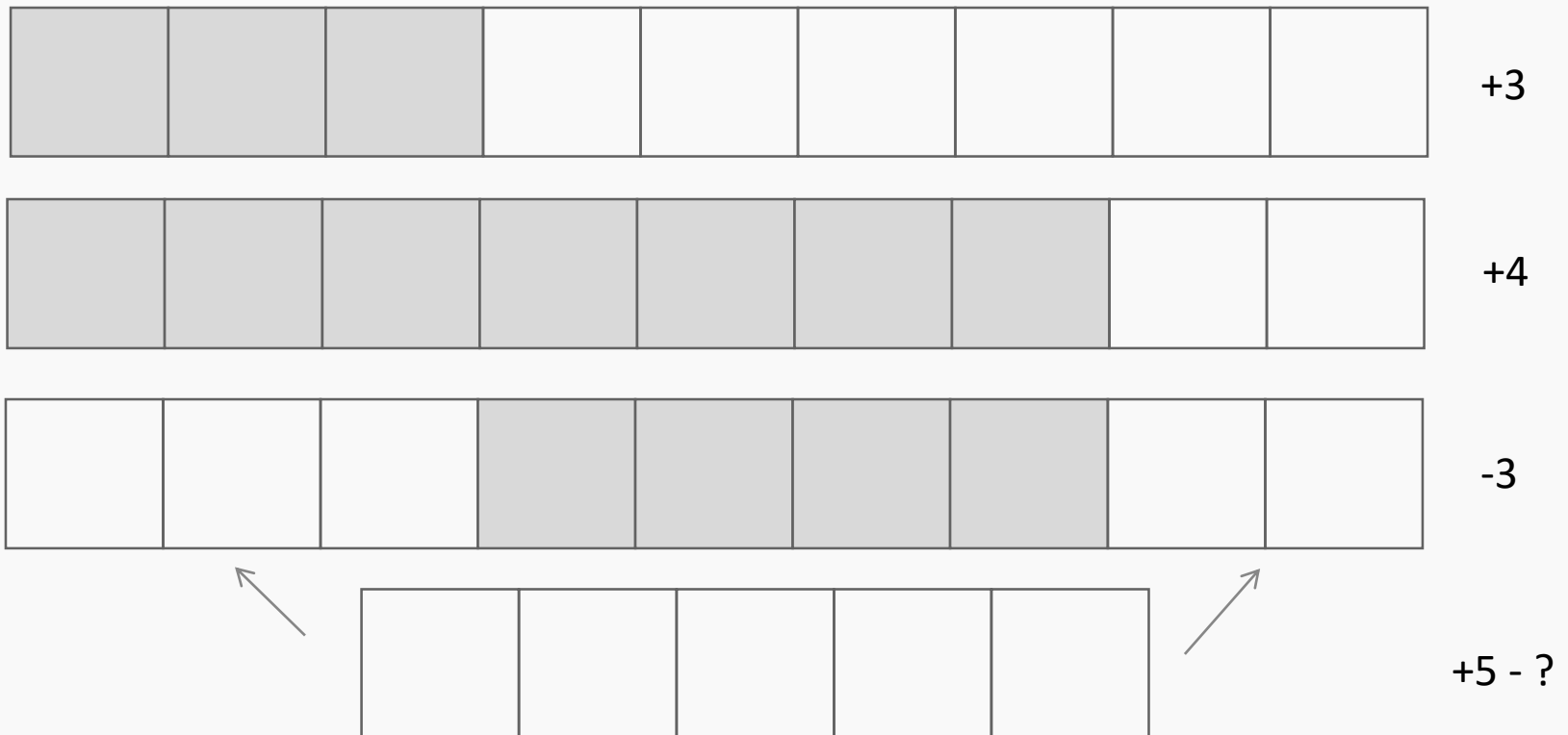
- **связывающие** — устанавливают в константу значение первого (`std::bind1st()`) или второго (`std::bind2nd()`) параметра заданной бинарной функции;
- **отрицающие** — инвертируют результат унарного (`std::not1()`) или бинарного (`std::not2()`) предиката.

```
std::vector<int>::iterator *where =  
    std::find_if(vector1.begin(), vector1.end(),  
                std::bind2nd(std::greater<int>(), 100));  
// std::not1(std::bind2nd(std::greater<int>(), 100));
```

Аллокаторы



Основная задача **аллокатора** – **выделение, освобождение и эффективное перераспределение динамической памяти.**



Аллокаторы в STL



Аллокатор `std::allocator<T>` не хранит состояния. Это означает, что аллокаторы одного типа являются **взаимозаменяемыми**.

```
template<class T, class Allocator = std::allocator<T>> class vector;
```

```
std::allocator<int> a1;
```

```
int *a = a1.allocate(1); // выделение памяти
```

```
a1.construct(a, 42); // вызов конструктора в выделенной области
```

```
a1.deallocate(a, 1); // освобождение памяти
```

```
std::allocator<std::string> a2;
```

```
std::string *s = a2.allocate(2);
```

```
a2.construct(s, "foo"); a2.construct(s + 1, "bar");
```

```
a2.destroy(s); a2.destroy(s + 1); a2.deallocate(s, 2);
```

- Все пользовательские аллокаторы также не должны хранить состояние (до C++11)
- Начиная с C++11 пользовательские аллокаторы могут хранить состояние. Каждый контейнер или объект, на который распространяет своё действие аллокатор, хранит объект аллокатора и контролирует его замену при помощи `std::allocator_traits`
- С C++17 стандартный аллокатор удовлетворяет требованиям полноты аллокатора:
 - Аллокатор типа `X` для типа `T` дополнительно удовлетворяет требованиям полноты, если выполняются оба условия:
 - `X` является законченным типом
 - Все типы `std::allocator_traits<X>` кроме, возможно, `value_type`, являются законченными типами

Пример пользовательского аллокатора



```
#include <cstdlib>
#include <new>
template <class T> struct Mallocator {
    typedef T value_type; // using value_type = T;
    Mallocator() = default;
    template <class U> constexpr Mallocator(const Mallocator<U>&) noexcept {}
    [[nodiscard]] T *allocate(std::size_t n) {
        if (n > std::size_t(-1) / sizeof(T)) throw std::bad_alloc();
        if (auto p = static_cast<T*>(std::malloc(n*sizeof(T)))) return p;
        throw std::bad_alloc();
    }
    void deallocate(T *p, std::size_t) noexcept { std::free(p); }
};
template <class T, class U> bool operator==(const Mallocator<T>&, const Mallocator<U>&) { return true; }
template <class T, class U> bool operator!=(const Mallocator<T>&, const Mallocator<U>&) { return false; }
```

Последовательные контейнеры:

- `std::array< T, N >` — массив значений типа `T` из `N` элементов;
- `std::forward_list< T, Allocator >` — однонаправленный (в отличие от `std::list`) список элементов с «полезной нагрузкой» типа `T` и дисциплиной распределения памяти, заданной распределителем `Allocator`.

Неупорядоченные ассоциативные контейнеры:

- `std::unordered_set< Key, Hash, KeyEqual, Allocator>` — набор неповторяющихся объектов типа `Key` с амортизированным константным временем поиска, вставки и удаления (контейнер для хранения повторяющихся объектов — `std::unordered_multiset`);
- `std::unordered_map< Key, T, Hash, KeyEqual, Allocator>` — набор пар «ключ – значение» с уникальными ключами типа `Key` с амортизированным константным временем поиска, вставки и удаления (контейнер для хранения пар с неуникальными ключами — `std::unordered_multimap`).

Набор алгоритмов STL расширен такими новыми элементами, как

- немодифицирующие последовательные алгоритмы: `std::all_of()`, `std::any_of()`, `std::none_of()`, `std::find_if_not()`;
- модифицирующие последовательные алгоритмы: `std::copy_if()`, `std::copy_n()`, `std::move()`, `std::move_backward()`, `std::shuffle()`;
- алгоритмы разбиения: `std::is_partitioned()`, `std::partition_copy()`, `std::partition_point()`;
- алгоритмы сортировки: `std::is_sorted()`, `std::is_sorted_until()`;
- алгоритмы на хипах: `std::is_heap()`, `std::is_heap_until()`;
- алгоритмы поиска наибольших и наименьших: `std::minmax()`, `std::minmax_element()`, `std::is_permutation()`;
- алгоритмы на числах: `std::iota()`.

STL в C++11: прочие элементы



Наконец, новыми элементами STL в C++11 являются:

- `std::move_iterator< Iterator >` — итератор переноса, формируемый перегруженной функцией `std::move_iterator< Iterator >()`;
- `std::next< ForwardIterator >()`, `std::prev< BidirectionalIterator >()` — функции инкремента и декремента итераторов;
- `std::begin< Container >()`, `std::end< Container >()` — функции возврата итераторов в начало или конец контейнера или массива.

Постановка задачи

- Дополнить учебный проект с использованием возможностей стандартной библиотеки шаблонов (STL) и иных промышленных библиотек для разработки на языке C++.
- **Цель** — спланировать и осуществить системную оптимизацию проекта с применением STL и прочих известных участникам и необходимых для нужд проекта промышленных библиотек: Qt Framework, Google Protocol Buffers и др.

