МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Углубленное программирование на языке C++

Алексей Петров

Лекция №6. Практическое введение в STL

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»);
- использование семантики **передачи объектов по значению**.

«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.

Состав STL

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**.

Гарантии производительности STL (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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.

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

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)

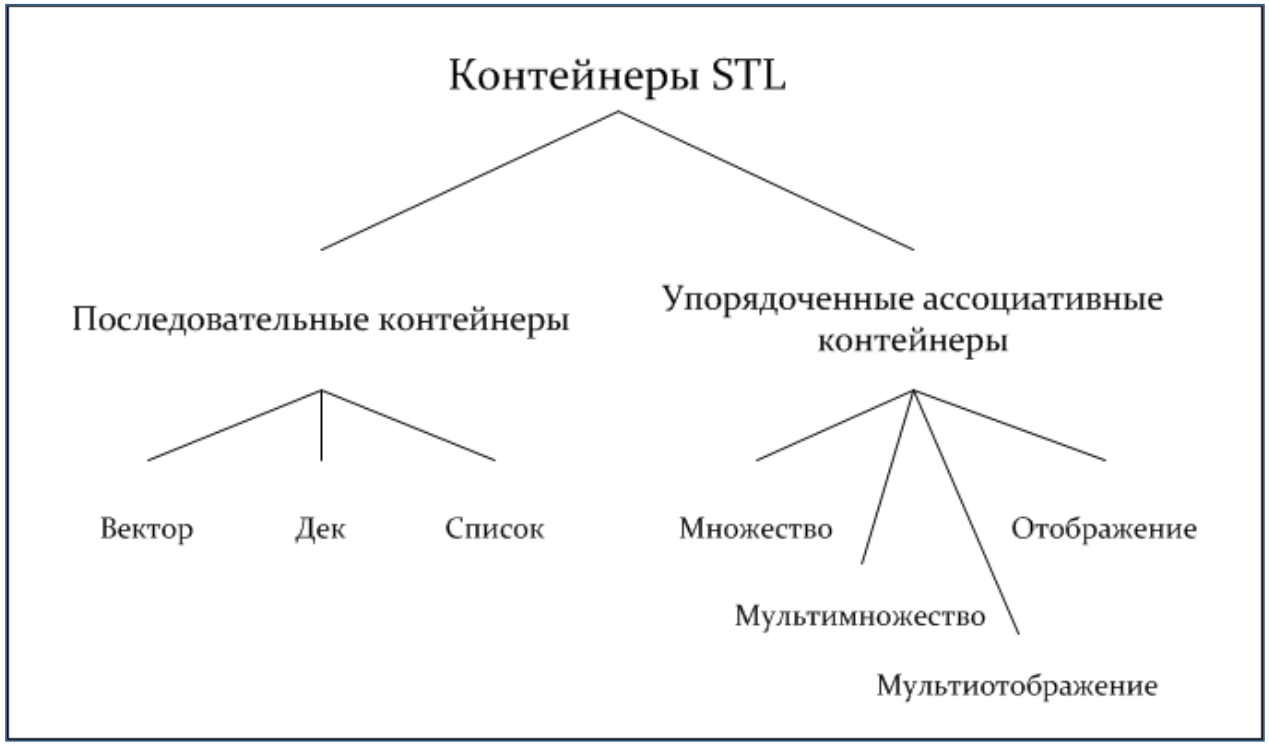
Недостатком оценки максимального времени является рассмотрение редко встречающихся на практике наихудших случаев (например, quicksoft в таком случае выполняется за время $O(N^2)$).

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 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 deque<T> с возможностью быстрой вставки и удаления элементов в начале и конце контейнера.

Список — контейнер типа 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

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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

Отображение — контейнер типа map<Key, T> с поддержкой уникальных ключей типа Key и быстрым доступом по ключам к значениям типа T. **Мультиотображение** — аналогичный отображению контейнер типа 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:

```
template<
        typename T,  // тип данных
        typename Allocator = allocator<T> >
```

Векторы: встроенные типы

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Итераторы:

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

Прочие встроенные типы:

- value_type — тип значения элемента (T);
- pointer — тип указателя на элемент (T*);
- const_pointer — тип константного указателя на элемент;
- reference — тип ссылки на элемент (T&);
- const_reference — тип константной ссылки на элемент;
- difference_type — целый знаковый тип результата вычитания итераторов;
- size_type — целый беззнаковый тип размера.

Векторы: порядок конструкции

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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

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

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

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,  // тип данных
        typename Allocator = allocator<T> >
```

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

Списки: общие сведения

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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,  // тип данных
        typename Allocator = allocator<T> >
```

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

Списки: описание интерфейса (методы упорядочения)

Название метода	Назначение
`sort`	Аналогично алгоритму `sort`
`unique`	Аналогично алгоритму `unique`
`merge`	Аналогично алгоритму `merge`
`reverse`	Аналогично алгоритму `reverse`
`remove` `remove_if`	Аналогично алгоритму `remove`, но с одновременным сокращением размера контейнера

Множества и мультимножества: общие сведения

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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:

```
template<
        typename Key,                    // тип ключа
        typename Compare = less<Key>,// ф-я сравнения
        typename Allocator = 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: порядок конструкции

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
// сигнатуры конструктора 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

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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:

```
template<typename Key,    // тип ключа
         typename T,      // тип связанных данных
         typename Compare = less<Key>,// ф-я сравнения
         typename Allocator =
                allocator<pair<const Key, T> > >
```

Отображения и мультиотображения: встроенные типы, порядок конструкции

Итераторы:

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

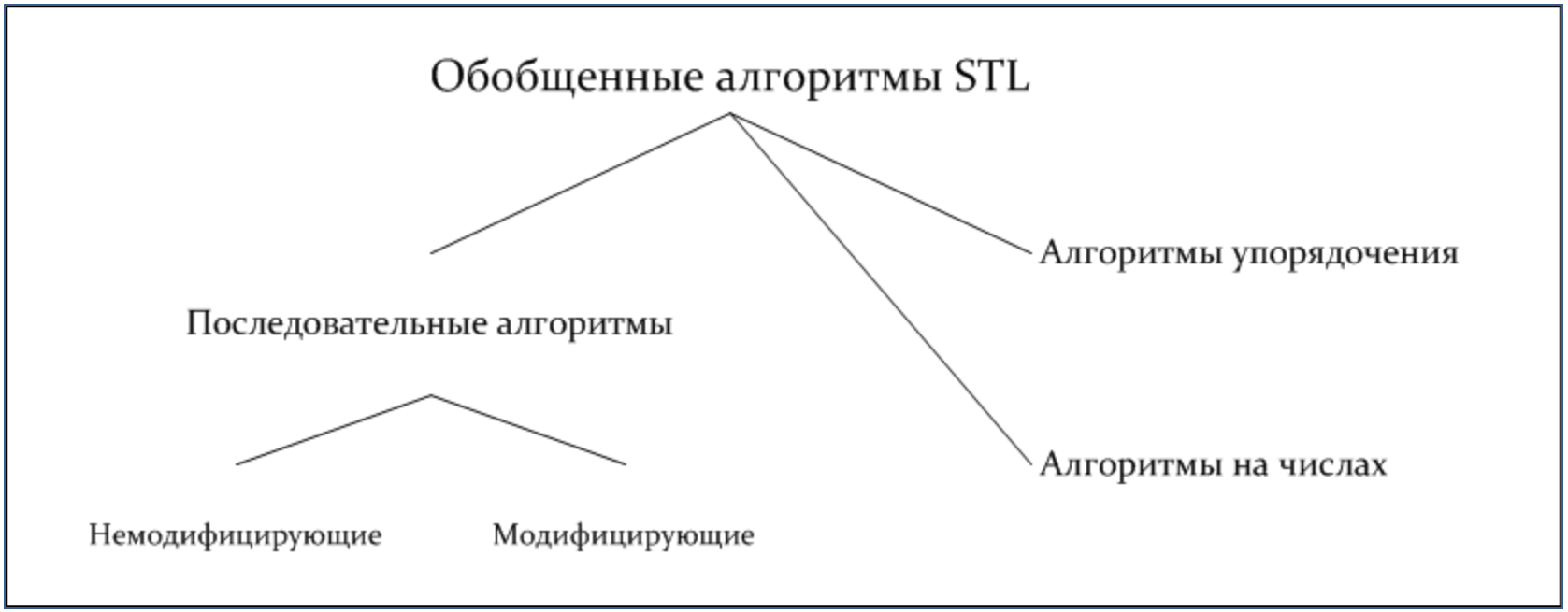
Прочие встроенные типы — аналогичны встроенным типам последовательных контейнеров (value_type — тип pair<const Key, T>) со следующими дополнениями:

- 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 set<T> и multiset<T>.

Обобщенные алгоритмы: обзор

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



Последовательные алгоритмы

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

- Например: find(), equal(), count().

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

- Например: copy(), random_shuffle(), 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: sort(), binary_search(), set_union().

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

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

Копирующие, предикатные и алгоритмы, работающие на месте

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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 функции (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 того же типа; может являться производной от binary_function<T, T, T>;
- обобщенная логическая функция (предикат) одного аргумента; может являться производной от unary_function<T, bool>;
- обобщенная логическая функция (предикат) двух аргументов; :может являться производной от 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:

$$xRy \quad \text{или} \quad yRx \quad \text{или} \quad x = y.$$

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

- операцией < над базовыми типами языка C++;
- операцией-функцией operator<() класса-строки;
- входящим в 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 greater<T>().

Алгоритмы сортировки

Название алгоритма	Назначение	Наибольшее время
`sort`	Нестабильная сортировка на месте (вариант quicksort) в среднем за $O(N \log N)$	$O(N^2)$
`partial_sort`	Нестабильная сортировка на месте (вариант heapsort; допускает получение отсортированного поддиапазона длины k)	$O(N \log N)$ или $O(N \log k)$
`stable_sort`	Стабильная сортировка на месте (вариант mergesort; 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.

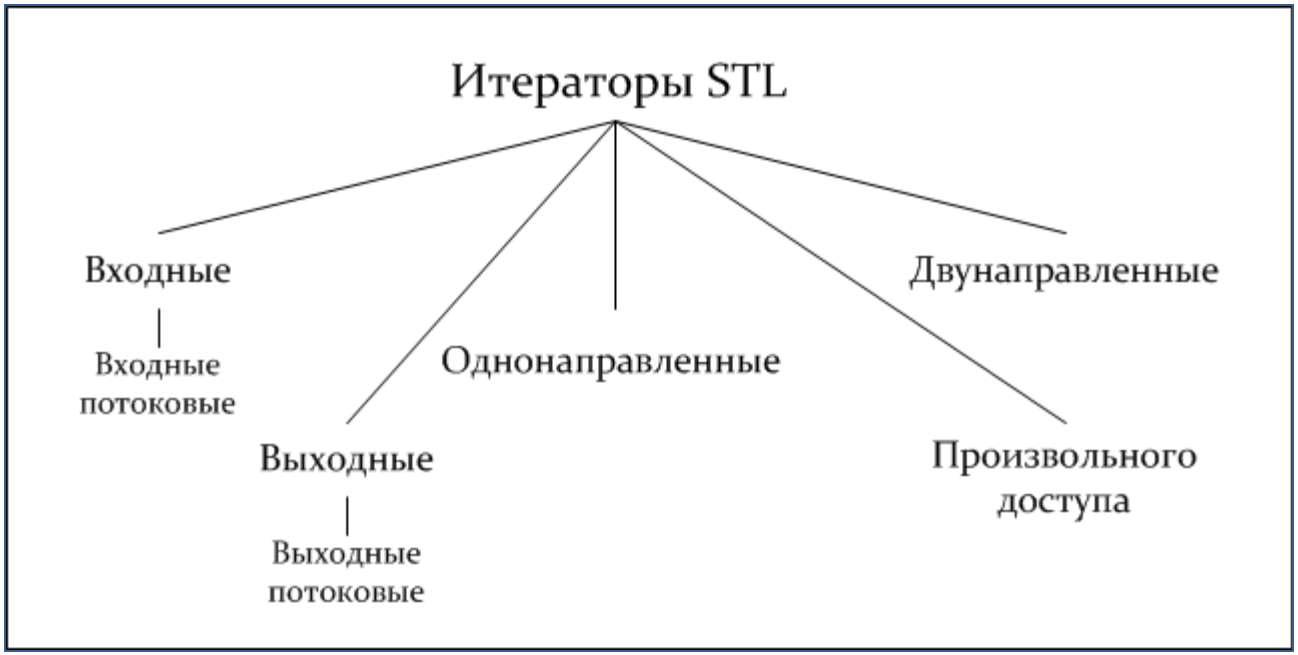
Алгоритмы на числах: обзор

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

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

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.

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];

// ...

copy(&a[0], &a[N], &b[0]);
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, единственными параметром которых является контейнерный тип `Container`:

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

Итераторы вставки (2 / 2)

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

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

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

Потоковые итераторы

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

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

Конструкторы:

- istream_iterator<T>(std::istream&) — входной итератор для чтения значений типа T из заданного входного потока;
- istream_iterator<T>() — входной итератор – маркер «конец потока» (англ. EOS, end-of-stream);
- 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

```
merge(
        vector1.begin(), vector1.end(),
        istream_iterator<int>(cin), // рабочий итератор
        istream_iterator<int>(),    // итератор EOS
        back_inserter(list1));

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

1 2 3 4 5 6

Пример: обобщенный алгоритм copy

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

Пример: обобщенный алгоритм replace

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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

1 2 3 4

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 unary_function и binary_function.

Функциональные объекты: базовые классы

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
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)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
// 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)

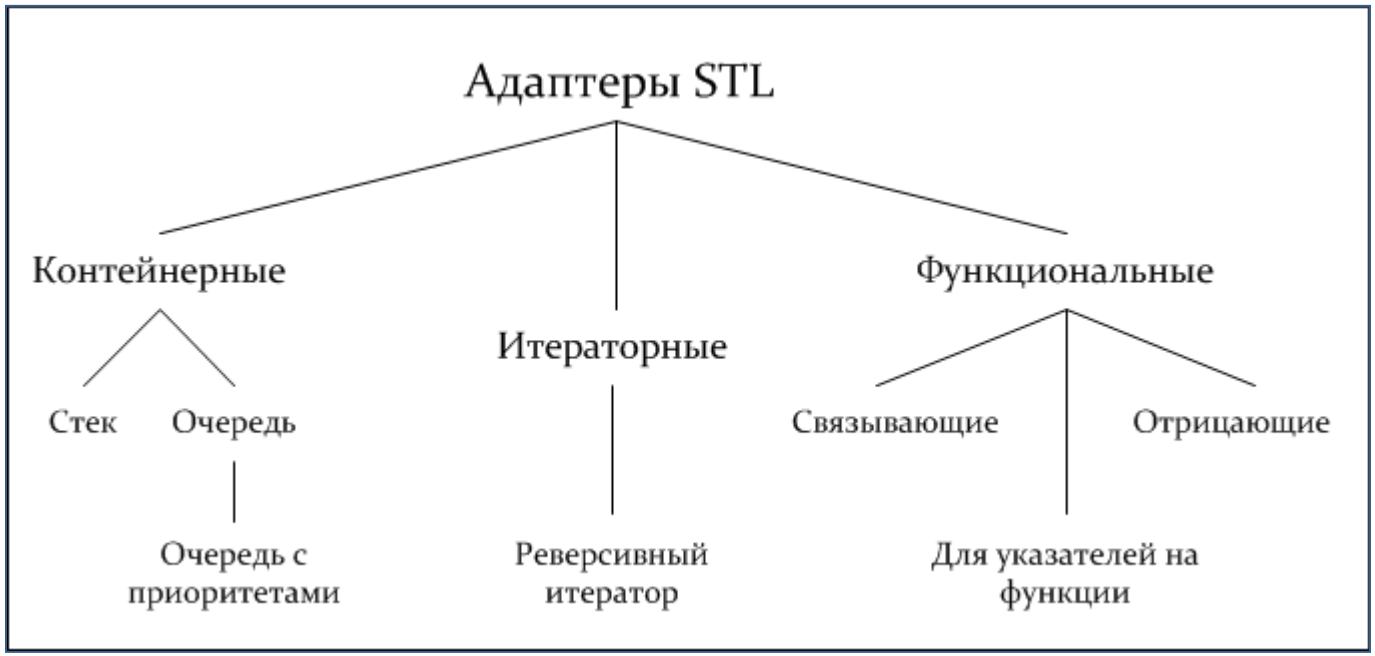

```
// для операций сравнения
template<typename T> struct equal_to;      // равно
template<typename T> struct not_equal_to; // не равно
template<typename T> struct greater;       // больше
template<typename T> struct less;          // меньше
// больше или равно
template<typename T> struct greater_equal;
// меньше или равно
template<typename T> struct less_equal;

// 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>
class multiplies :                                  (1)
        public binary_function<T, T, T>
{
public:
        T operator()(const T& x, const T& y) const {   (2)
                return x * y;
        }
};
```

Адаптеры: обзор

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 priority_queue требует также функции сравнения, по умолчанию — less<T>).

Адаптер stack допускает конкретизацию вида:

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

Контейнерные адаптеры (2 / 2)

Адаптер queue допускает конкретизацию вида:

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

Адаптер priority_queue допускает конкретизацию вида:

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

Функциональные адаптеры

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 (bind1st()) или второго (bind2nd()) параметра заданной бинарной функции;
- **отрицающие** — инвертируют результат унарного (not1()) или бинарного (not2()) предиката.

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

STL в C++11: контейнеры

Последовательные контейнеры:

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

Неупорядоченные ассоциативные контейнеры:

- unordered_set< Key, Hash, KeyEqual, Allocator> — набор неповторяющихся объектов типа Key с амортизированным константным временем поиска, вставки и удаления (контейнер для хранения повторяющихся объектов — unordered_multiset);
- 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 — unordered_multimap).

STL в C++11: алгоритмы

Набор алгоритмов STL расширен такими новыми элементами, как

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

STL в C++11: прочие элементы

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

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

Практикум №5

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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.

Спасибо за внимание

Алексей Петров

Приложение

Итераторы в последовательных контейнерах

Тип контейнера	Тип итератора	Категория итератора
`T a[N]`	`T*`	Изменяемый, произвольного доступа
`T a[N]`	`const T*`	Константный, произвольного доступа
`vector<T>`	`vector<T>::iterator`	Изменяемый, произвольного доступа
`vector<T>`	`vector<T>::const_iterator`	Константный, произвольного доступа
`deque<T>`	`deque<T>::iterator`	Изменяемый, произвольного доступа
`deque<T>`	`deque<T>::const_iterator`	Константный, произвольного доступа
`list<T>`	`list<T>::iterator`	Изменяемый, двунаправленный
`list<T>`	`list<T>::const_iterator`	Константный, двунаправленный

Итераторы в упорядоченных ассоциативных контейнерах

Тип контейнера	Тип итератора	Категория итератора
`set<T>`	`set<T>::iterator`	Константный, двунаправленный
`set<T>`	`set<T>::const_iterator`	Константный, двунаправленный
`multiset<T>`	`multiset<T>::iterator`	Константный, двунаправленный
`multiset<T>`	`multiset<T>::const_iterator`	Константный, двунаправленный
`map<Key, T>`	`map<Key, T>::iterator`	Изменяемый, двунаправленный
`map<Key, T>`	`map<Key, T>::const_iterator`	Константный, двунаправленный
`multimap<Key, T>`	`multimap<Key, T>::iterator`	Изменяемый, двунаправленный
`multimap<Key, T>`	`multimap<Key, T>::const_iterator`	Константный, двунаправленный

Немодифицирующие последовательные алгоритмы (1 / 2)

Название алгоритма	Назначение	Сложность
`find` `find_if`	Поиск первого элемента, равного заданному значению или обращающего в истину заданный унарный предикат (выполняется перебором)	$O(N)$
`adjacent_find`	Поиск первой пары смежных значений, равных друг другу или обращающих в истину заданный бинарный предикат	$O(N)$
`count` `count_if`	Подсчет элементов, равных заданному значению или обращающих в истину заданный унарный предикат	$O(N)$

Немодифицирующие последовательные алгоритмы (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Название алгоритма	Назначение	Сложность
`for_each`	Обход контейнера с применением к каждому элементу заданной функции (результат функции игнорируется)	$O(N)$
`mismatch`, `equal`	Сравнение диапазонов на равенство или эквивалентность элементов (по отношению, заданному бинарным предикатом-параметром)	$O(N)$
`search`	Поиск в диапазоне #1 (длины m) подпоследовательности, элементы которой равны или эквивалентны (по отношению, заданному бинарным параметром-предикатом) элементам диапазона #2 (длины n)	$O(N^2)$

Модифицирующие последовательные алгоритмы (1 / 4)

Название алгоритма	Назначение	Сложность
`copy` `copy_backward`	Копирование элементов между диапазонами (диапазоны могут перекрываться, что обеспечивает линейный сдвиг)	$O(N)$
`fill` `fill_n`	Заполнение диапазона копией заданного значения	$O(N)$
`generate`	Заполнение диапазона значениями, возвращаемыми передаваемой на вход функцией без параметров	$O(N)$
`partition` `stable_partition`	Разбиение диапазона в соответствии с заданным унарным предикатом-параметром	$O(N)$

Модифицирующие последовательные алгоритмы (2 / 4)

Название алгоритма	Назначение	Сложность
`random_shuffle`	Случайное перемешивание элементов диапазона с использованием стандартного или заданного параметром генератора псевдослучайных чисел	$O(N)$
`remove` `remove_copy`	Удаление элементов, равных заданному значению или обращающих в истину заданный предикат, без изменения размера контейнера (стабильный алгоритм)	$O(N)$

Модифицирующие последовательные алгоритмы (3 / 4)

Название алгоритма	Назначение	Сложность
`replace` `replace_copy`	Замена элементов, равных заданному значению или обращающих в истину заданный предикат	$O(N)$
`rotate`	Циклический сдвиг элементов контейнера влево	$O(N)$
`swap`	Взаимный обмен двух значений	$O(1)$
`swap_ranges`	Взаимный обмен элементов двух неперекрывающихся диапазонов	$O(N)$

Модифицирующие последовательные алгоритмы (4 / 4)

Название алгоритма	Назначение	Сложность
`transform`	Поэлементное преобразование значений диапазона (диапазонов) при помощи заданной унарной (бинарной) функции (результат — в отдельном диапазоне)	$O(N)$
`unique`	Устранение последовательных дубликатов без изменения размера контейнера	$O(N)$

Алгоритмы сортировки

Название алгоритма	Назначение	Наибольшее время
`sort`	Нестабильная сортировка на месте (вариант quicksort) в среднем за $O(N \log N)$	$O(N^2)$
`partial_sort`	Нестабильная сортировка на месте (вариант heapsort; допускает получение отсортированного поддиапазона длины k)	$O(N \log N)$ или $O(N \log k)$
`stable_sort`	Стабильная сортировка на месте (вариант mergesort; 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)

Прочие алгоритмы упорядочения (1 / 3)

Название алгоритма	Назначение	Сложность
`nth_element`	Размещение на N-й позиции элемента диапазона, который должен занимать ее в отсортированном контейнере, с одновременным размещением левее N «меньших», а правее — «больших» значений	$O(N^2)$ (средняя — $O(N)$)
`binary_search`	Бинарный поиск в отсортированном диапазоне	$O(\log N)$
`lower_bound` `upper_bound`	Поиск первой (последней) позиции, в которую без нарушения порядка возможна вставка заданного значения	$O(\log N)$

Прочие алгоритмы упорядочения (2 / 3)

Название алгоритма	Назначение	Сложность
`equal_range`	Поиск первой и последней позиции, в которую без нарушения порядка возможна вставка заданного значения	$O(\log N)$
`merge`	Слияние двух отсортированных диапазонов (с размещением в отдельном диапазоне)	$O(N)$
`inplace_merge`	Слияние двух смежных отсортированных диапазонов (с размещением на месте)	$O(N \log N)$ (при наличии памяти — $O(N)$)

Прочие алгоритмы упорядочения (3 / 3)

Название алгоритма	Назначение	Сложность
`min` `max`	Определение меньшего (большего) из пары значений	$O(1)$
`min_element` `max_element`	Определение наименьшего (наибольшего) значения в диапазоне	$O(N)$
`lexicographical_compare`	Лексикографическое сравнение диапазонов (аналогично строкам) из элементов, поддерживающих < как строгий слабый (полный) порядок	$O(N)$
`prev_permutation` `next_permutation`	Получение лексикографически предыдущей (следующей) перестановки элементов, поддерживающих < как строгий слабый (полный) порядок	$O(N)$

Операции над множествами

Название алгоритма	Назначение	Сложность
`includes`	Проверка вхождения элементов диапазона A в диапазон B: $A \subset B$	$O(N)$
`set_union`	Объединение диапазонов: $A \cup B$	$O(N)$
`set_intersection`	Пересечение диапазонов: $A \cap B$	$O(N)$
`set_difference`	Разность диапазонов: $A \setminus B$	$O(N)$
`set_symmetric_difference`	Симметрическая разность диапазонов: $A \nabla B = A \setminus B \cup B \setminus A$	$O(N)$

Операции над хипами

Название алгоритма	Назначение	Сложность
`make_heap`	Преобразование диапазона итераторов $[a; b)$ в хип	$O(N)$
`sort_heap`	Сортировка хипа в диапазоне итераторов $[a; b)$	$O(N \log N)$
`push_heap`	Расширение границ хипа от $[a; b-1)$ до $[a; b)$ с включением в хип нового элемента (справа)	$O(\log N)$
`pop_heap`	Сжатие границ хипа с $[a; b)$ до $[a; b-1)$ путем выталкивания наибольшего элемента (направо)	$O(\log N)$

Алгоритмы на числах

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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

Векторы: описание интерфейса (мутаторы)

Название метода	Назначение	Сложность
`push_back`	Вставка конечного элемента	Аморт. $O(1)$
`insert`	Вставка элемента в произвольную позицию	$O(N)$
`reserve`	Обеспечение min необходимой емкости контейнера (с возможным перераспределением памяти)	Не выше $O(N)$
`pop_back`	Удаление конечного элемента	$O(1)$
`erase`	Удаление элемента в произвольной позиции	$O(N)$
`operator= assign`	Присваивание значений из другого контейнера или диапазона	$O(N)$
`swap`	Обмен содержимым с другим контейнером	$O(1)$

Векторы: описание интерфейса (аксессоры)

Название метода	Назначение	Сложность
`begin` `rbegin`	Получение итератора на элемент в начале контейнера	$O(1)$
`end` `rend`	Получение итератора «за концом» контейнера	$O(1)$
`size`	Количество элементов	$O(1)$
`capacity`	Емкость контейнера	$O(1)$
`empty`	Признак пустоты контейнера	$O(1)$
`front` `back`	Получение ссылки на элемент в начале (конце) контейнера	$O(1)$
`operator[N]` `at`	Получение ссылки на N-й элемент (`at` возбуждает `out_of_range`)	$O(1)$

Деки: описание интерфейса (мутаторы)

Название метода	Назначение	Сложность
`push_back`	Вставка конечного элемента	$O(1)$
`push_front`	Вставка начального элемента	$O(1)$
`insert`	Вставка элемента в произвольную позицию	Не выше $O(N)$
`pop_back`	Удаление конечного элемента	$O(1)$
`pop_front`	Удаление начального элемента	$O(1)$
`erase`	Удаление элемента в произвольной позиции	$O(N)$
`operator= assign`	Присваивание значений из другого контейнера или диапазона	$O(N)$
`swap`	Обмен содержимым с другим контейнером	$O(1)$

Деки: описание интерфейса (аксессоры)

Название метода	Назначение	Сложность
`begin` `rbegin`	Получение итератора на элемент в начале контейнера	$O(1)$
`end` `rend`	Получение итератора «за концом» контейнера	$O(1)$
`size`	Количество элементов	$O(1)$
`empty`	Признак пустоты контейнера	$O(1)$
`front` `back`	Получение ссылки на элемент в начале (конце) контейнера	$O(1)$
`operator[N]` `at`	Получение ссылки на N-й элемент (`at` возбуждает `out_of_range`)	$O(1)$

Списки: описание интерфейса (мутаторы)

Название метода	Назначение	Сложность
`push_back`	Вставка конечного элемента	$O(1)$
`push_front`	Вставка начального элемента	$O(1)$
`insert`	Вставка в произвольную позицию	$O(1)$
`pop_back`	Удаление конечного элемента	$O(1)$
`pop_front`	Удаление начального элемента	$O(1)$
`erase`	Удаление элемента в произвольной позиции	$O(1)$
`operator=` `assign`	Присваивание значений из другого контейнера или диапазона	$O(N)$
`swap`	Обмен содержимым с другим контейнером	$O(1)$
`splice`	Перенос элементов	$O(1)$ или $O(N)$

Списки: описание интерфейса (методы упорядочения)

Название метода	Назначение
`sort`	Аналогично алгоритму `sort`
`unique`	Аналогично алгоритму `unique`
`merge`	Аналогично алгоритму `merge`
`reverse`	Аналогично алгоритму `reverse`
`remove` `remove_if`	Аналогично алгоритму `remove`, но с одновременным сокращением размера контейнера

Списки: описание интерфейса (аксессоры)

Название метода	Назначение	Сложность
`begin` `rbegin`	Получение итератора на элемент в начале контейнера	$O(1)$
`end` `rend`	Получение итератора «за концом» контейнера	$O(1)$
`size`	Количество элементов	$O(1)$
`empty`	Признак пустоты контейнера	$O(1)$
`front` `back`	Получение ссылки на элемент в начале (конце) контейнера	$O(1)$

Множества и мультимножества: описание интерфейса (мутаторы)

Название метода	Назначение	Сложность
`insert`	Вставка в контейнер	От аморт. $O(1)$ до $O(\log N)$
`erase`	Удаление элементов по позиции или ключу (E — количество удаляемых)	$O(\log N + E)$
`operator=`	Присваивание значений из другого контейнера	$O(N)$
`swap`	Обмен содержимым с другим контейнером	$O(1)$

Множества и мультимножества: описание интерфейса (аксессоры)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Название метода	Назначение	Сложность
`begin` `rbegin`	Получение итератора на элемент в начале контейнера	$O(1)$
`end` `rend`	Получение итератора «за концом» контейнера	$O(1)$
`size`	Количество элементов	$O(1)$
`empty`	Признак пустоты контейнера	$O(1)$
`find`	Аналогично алгоритму `find`	$O(\log N)$
`lower_bound` `upper_bound`	Аналогично алгоритмам `lower_bound` и `upper_bound`	$O(\log N)$
`equal_range`	Аналогично алгоритму `equal_range`	$O(\log N)$
`count`	Расстояние E между позициями `lower_bound` и `upper_bound`	$O(\log N + E)$

Отображения и мультиотображения: описание интерфейса (мутаторы)

Название метода	Назначение	Сложность
`insert` `operator[]`	Вставка в контейнер (`operator[]` определен только для контейнера `map`)	От аморт. $O(1)$ до $O(\log N)$
`erase`	Удаление элементов по позиции или ключу (E — количество удаляемых)	$O(\log N + E)$
`operator=`	Присваивание значений из другого контейнера	$O(N)$
`swap`	Обмен содержимым с другим контейнером	$O(1)$

Отображения и мультиотображения: описание интерфейса (аксессоры)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Название метода	Назначение	Сложность
`begin` `rbegin`	Получение итератора на элемент в начале контейнера	$O(1)$
`end` `rend`	Получение итератора «за концом» контейнера	$O(1)$
`size`	Количество элементов	$O(1)$
`empty`	Признак пустоты контейнера	$O(1)$
`find` `operator[]`	Аналогично алгоритму `find` (`operator[]` — только для `map`)	$O(\log N)$
`lower_bound` `upper_bound`	Аналогично алгоритмам `lower_bound` и `upper_bound`	$O(\log N)$
`equal_range`	Аналогично алгоритму `equal_range`	$O(\log N)$
`count`	Расстояние E между позициями `lower_bound` и `upper_bound`	$O(\log N + E)$

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)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
// 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)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
// для операций сравнения
template<typename T>struct equal_to;     // равенство
template<typename T>struct not_equal_to;// нерав-во
template<typename T>struct greater;      // больше
template<typename T>struct less;         // меньше
// больше или равно
template<typename T>struct greater_equal;
// меньше или равно
template<typename T>struct less_equal;

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