Подготовительная программа по программированию на C/C++

Занятие №9

Валентина Глазкова

Стандартная библиотека шаблонов STL

- Контейнеры и итераторы.
- Примеры шаблонных классов-контейнеров (vector, list, map, set)
- Сложность основных методов работы с контейнерами.
- Основные алгоритмы библиотеки STL.

Стандартная библиотека шаблонов (STL): история создания

Стандартная библиотека шаблонов (англ. Standard Templates Library, STL) была разработана в 1970-х – 1990-х гг. А. Степановым, Д. Мюссером (D. Musser) и др. как первая **универсальная библиотека обобщенных алгоритмов и структур данных** и составная часть стандартной библиотеки языка C++

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

Основные характеристики 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.

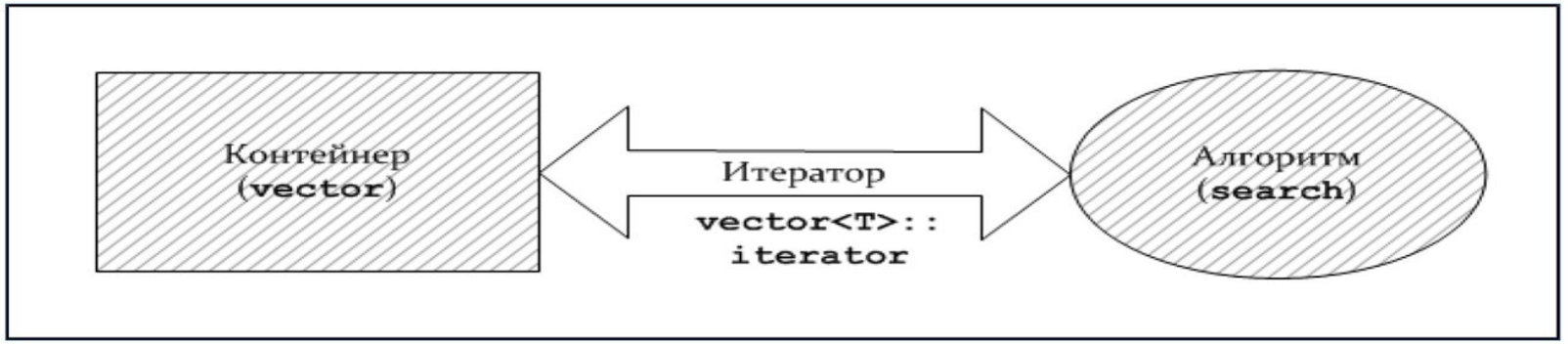
С технической точки зрения, STL представляет собой набор шаблонов **классов и алгоритмов (функций)**, предназначенных для совместного использования при решении широкого спектра задач

Состав STL (1/2)

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**.

Состав STL (2/2)




```
vector<int> vi =
      vector(10, 1); /* {1, 1, …, 1} */
vector<int>::iterator i = vi.begin();
vector<int>::iterator j = vi.end();
search(i, j, 1);
```

Гарантии производительности STL (1/3)

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.

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

Гарантии производительности STL (2/3)

Наибольшую значимость в STL имеют следующие оценки:

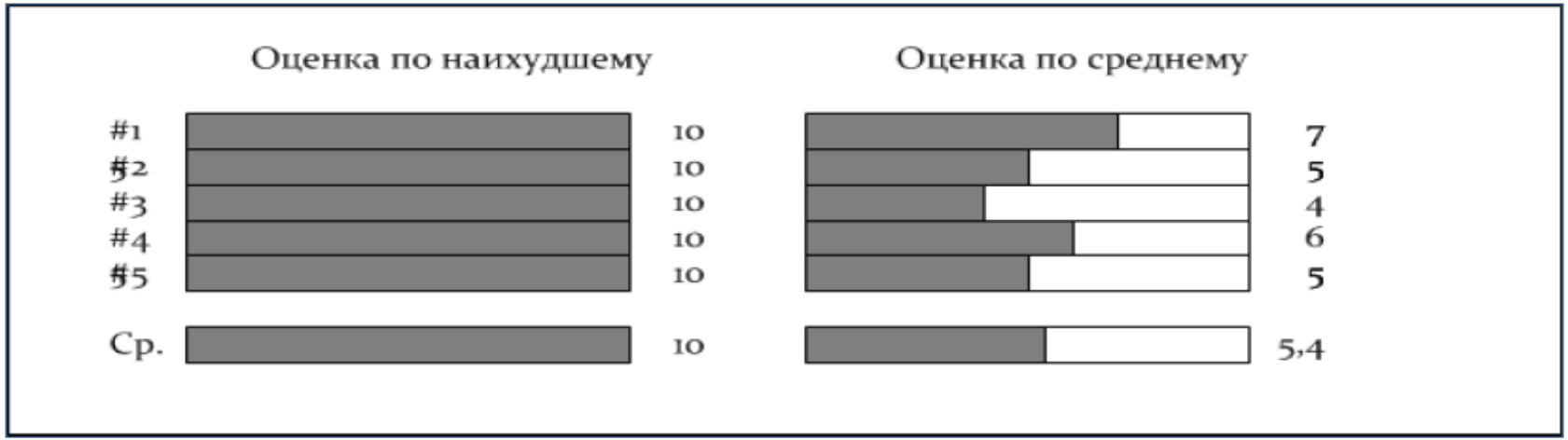
- **константное** время выполнения алгоритма: $T(N) = O(1)$;
- **линейное** время выполнения алгоритма: $T(N) = O(N)$;
- **квадратичное** время выполнения алгоритма: $T(N) = O(N^2)$;
- **логарифмическое** время выполнения алгоритма:
$$T(N) = O(\log N);$$
- время выполнения «***N* логарифмов *N***»:
$$T(N)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

Очевидно, что $O(1) \prec O(\log N) \prec O(N) \prec O(N \log N) \prec O(N^2)$.

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

Гарантии производительности STL (3/3)

Альтернативой оценке максимального времени является оценка среднего **амортизированного** времени выполнения алгоритма, под которым понимается совокупное время выполнения N операций, деленное на число N



Контейнеры: обзор

Контейнеры STL – классы, предназначенные для хранения других объектов классов (в том числе и контейнеров) и организации доступа к ним

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: С точки зрения STL, последовательными контейнерами в большинстве случаев могут считаться также канонический С-подобный массив T a[N] и класс String.

Последовательные контейнеры: сложность основных операций

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

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

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

Допускается создание векторов STL при помощи определений вида:

```
// 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);
```

Векторы: описание интерфейса (1/2)

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 (2/2)

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`)	$O(1)$

Векторы: пример (1/2)

```
template<class T> void print(const vector<T>& a)
{
  cout << "[s = " << a.size() << "]";
  if (!a.empty()) {
    cout << " {" << a.front();
    vector<T>::size_type i;
    for (i = 1; i < a.size(); i++) {
      vector<T>::value_type x = a[i];
      cout << ", " << x;
    }
    cout << "}";
  }
  cout << endl;
}
```

Векторы: пример (2/2)

```
void main() {
  vector<int> iv;
  print(iv);
  for (int i = 0; i < 5; i++) {
    iv.push_back(i);
    print(iv);
  }
  while (!iv.empty()) {
    iv.pop_back();
    print(iv);
  }
  print(iv);
}
```

```
[s = 0]
[s = 1] {0}
[s = 2] {0, 1}
[s = 3] {0, 1, 2}
[s = 4] {0, 1, 2, 3}
[s = 5] {0, 1, 2, 3, 4}
[s = 4] {0, 1, 2, 3}
[s = 3] {0, 1, 2}
[s = 2] {0, 1}
[s = 1] {0}
```

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>`.

Деки: описание интерфейса (1/2)

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 (2/2)

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: общие сведения

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 (1/2)

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	$O(1)$

Списки: описание интерфейса (2/2)

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	$O(1)$

Строки: класс std::string

- Класс `string`, обеспечивает удобную работу со строками
- Для него перегружены операции копирования, присваивания, сравнения, сложения и вывода и реализовано автоматическое управление памятью
- Класс `string` объявлен в заголовочном файле `<string>`

Строки: пример 1

```
void main()
{
  char* str1 = "I'm a string literal!";
  char* str2 = str1;         // Not a copy!
  char* str3 = str1 + str2; // Error!

  string s1 = "I'm cool STR string!";
  string s2 = s1;       // Full copy
  string s3 = s1 + s2; // OK

  s3 = "std::string"; // Memory managed correctly

  cout << "Can write " << s3 << " to stream" << endl;
  cout << "Can mix " + s3 + " and literals" << endl;
}
```

Строки: пример 2

```
void main()
{
  string s1 = "abacb";
  string s2 = "abbca";

  cout << (s1 < s2); // 1

  const char* str = s1.c_str();
  cout << strlen(str); // 5

  reverse(s1.begin(), s1.end());
  cout << s1; // bcaba
}
```

Для класса string перегружены операторы лексикографического сравнения

Объект класса string можно преобразовать в обычную строку с терминирующим нулем, используя метод c_str

Класс string является контейнером и с ним можно работать соответствующим образом

Алгоритм reverse разворачивает последовательность элементов в контейнере

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.

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:

```
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`)

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

Допускается создание множеств STL при помощи следующих конструкторов:

```
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.

Множества и мультимножества: описание интерфейса (1/2)

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 (2/2)

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`	Поиск первого элемента, равного заданному значению	$O(\log N)$

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:

```
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>`.

Отображения и мультиотображения: описание интерфейса (1/2)

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 (2/2)

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`	Поиск первого элемента, равного заданному значению	$O(\log N)$

Обобщённые 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().

Категории алгоритмов (1/2)

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 параметра – функции (обобщенной функции) с одним или двумя параметрами.

Категории алгоритмов (2/2)

Немодифицирующие последовательные алгоритмы: обзор

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

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)

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: обзор

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

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 упорядочения: обзор

Алгоритмы упорядочения — все алгоритмы STL, работа которых опирается на наличие или установление порядка на элементах.

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: обзор

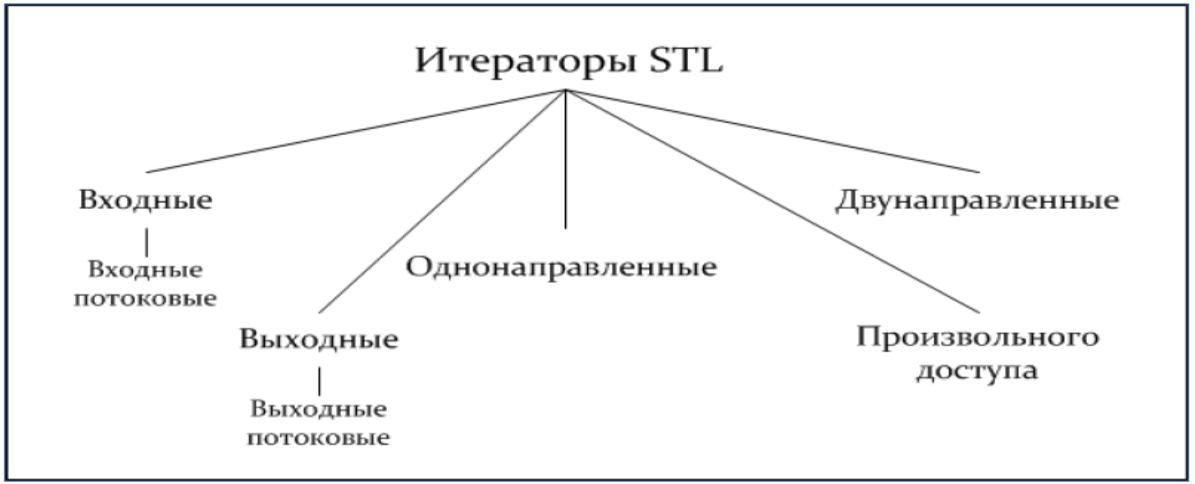
Реализуемые обобщенными алгоритмами STL операции над множествами имеют **традиционное теоретико-множественное значение** и выполняются над отсортированными диапазонами, находящимися **в любых контейнерах STL**.

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: $A \nabla B = A \setminus B \cup B \setminus A$	$O(N)$

Итераторы: обзор

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



Итераторы: реализация и использование

Итераторы (обобщенные указатели):

- реализованы в STL как типы данных, вложенные в контейнерные классы;
- позволяют реализовывать обобщенные алгоритмы без учета физических аспектов хранения данных;
- в зависимости от категории поддерживают минимально необходимый для эффективного использования набор операций.

Итераторы: операции и допустимые диапазоны

Категории итераторов различаются наборами операций, которые они гарантированно поддерживают.

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

$$[first; last)$$

Итераторы и 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. Например:

```
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 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

Итераторы: пример 1

```
void main() {
  vector<int> iv(5, 1);
  vector<int>::iterator i;
  vector<int>::size_type p;
  for (i = iv.begin(); i != iv.end(); i++) {
    cout << *i << " ";
  }
  for (p = 0; p < iv.size(); p++) {
    i--;
    cout << *i << " ";
  }
  for (p = 0; p < iv.size(); p++) {
    cout << i[p] << " ";
  }
}
```

Все контейнеры определяют тип `iterator` для описания итераторов, с которыми они работают

К виртуальному элементу нельзя обращаться, но с ним можно сравнивать

Контейнер `vector` предоставляет итераторы произвольного доступа

Итераторы: пример 2 (1/2)

```
template<class C> void print(const C& a)
{
  cout << "[s = " << a.size() << "]";
  if (!a.empty()) {
    cout << " {" << a.front();
    C::const_iterator i = ++a.begin();
    for ( ; i != a.end(); i++) {
      cout << ", " << *i;
    }
    cout << "}";
  }
  cout << endl;
}
```

Все контейнеры определяют тип const_iterator для доступа к элементам только на чтение

Эта функция будет работать для всех контейнеров STL (если для их элементов перегружена операция вывода в поток)

Итераторы: пример 2 (2/2)

```
void main() {
  list<int> lv(2, 1);
  for (int i = 0; i < 5; i++) {
    lv.push_back(lv.back() + *(++lv.rbegin()));
  }
  print(lv);
  list<int>::iterator i = lv.begin();
  while (i != lv.end()) {
    lv.insert(i, *i); i++;
  }
```

Метод insert вставляет элемент перед указанным

Если бы вставка происходила в контейнер vector, то все итераторы, адресующие элементы после вставленного, стали бы недействительны

```
[s = 7] {1, 1, 2, 3, 5, 8, 13}
[s = 14] {1, 1, 1, 1, 2, 2, 3, 3, 5, 5, 8, 8, 13, 13}
```

Итераторы и алгоритмы: пример 1

Алгоритм sort требует итераторы произвольного доступа!

```
void main()
{
  vector<int> iv(7);

  generate(iv.begin(), iv.end(), rand);
  print(iv);
  sort(iv.begin(), iv.end());
  print(iv);
  sort(iv.begin() + 1, iv.end() - 1, greater<int>());
  print(iv);
}
```

Алгоритм generate позволяет инициализировать элементы контейнера

Стандартный функтор, реализующий бинарный предикат сравнения

```
[s = 7] {41, 18467, 6334, 26500, 19169, 15724, 11478}
[s = 7] {41, 6334, 11478, 15724, 18467, 19169, 26500}
[s = 7] {41, 19169, 18467, 15724, 11478, 6334, 26500}
```

Итераторы и алгоритмы: пример 2

```
class rnd_mod {
  int mod;
public:
  rnd_mod(int mod) : mod(mod) {}
  int operator()() { return rand() % mod; }
};
void main()
{
  list<int> il(7);
  generate(il.begin(), il.end(), rnd_mod(4));
  print(il);
  size_t c = count(il.begin(), il.end(), 2);
  cout << c << endl;
}
```

Тип size_t тождественен unsigned int и в большинстве случаев безопасен для индексации элементов контейнера

Алгоритм count выполняет подсчет количества элементов контейнера, равных заданному

```
[s = 7] {1, 3, 2, 0, 1, 0, 2}
2
```

Функциональные объекты: обзор

Функциональные объекты (обобщенные функции) — программные артефакты, 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`.

Пример: функциональный объект
multiplies

```
template <typename T>
class multiplies :
        public binary_function<T, T, T>
{
public:
        T operator()(const T& x, const T& y)
        const {
                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> >`.

Валентина Глазкова

Спасибо за внимание!