Углубленное программирование на языке C / C++

Лекция № 6

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.**

Слияние параметров шаблона по умолчанию: пример


```
// описание #1
template <class T, class U = long> class Sample;
// описание #2
template <class T = std::string, class U> class Sample;

// эквивалентно:
// template <class T = std::string, class U = long>
// class Sample;
```

Шаблоны переменных (C++14)

Шаблон переменной — элемент языка, определяющий, аналогично шаблонам функций и классов, семейство переменных или статических членов данных.

Шаблоны переменных **не могут использоваться** как шаблонные параметры других шаблонов классов, функций и переменных.

Введение шаблонов переменных **позволяет отказаться** от «обходных путей» при построении параметризованных переменных, каковыми до C++11 включительно были статические члены данных в составе шаблонов классов, а также шаблоны функций, помеченных как constexpr.

Спецификатор constexpr (C++11)

Спецификатор constexpr, как следует из названия, указывает на то, что значения, полученные при вычислении помеченных им объектов **во время компиляции кода**, могут использоваться в составе константных выражений **времени компиляции**.

Использование constexpr в определении объекта хранения подразумевает const, использование constexpr в определении функции подразумевает inline.

Использование результатов вычисления во время компиляции кода накладывает на constexpr-объекты существенные ограничения.

Спецификатор constexpr может использоваться с **переменными**, **функциями** и **конструкторами** классов.

Спецификатор constexpr: ограничения (C++11)

Переменные — объекты литеральных типов (скаляры, ссылки, массивы таковых, void (C++14) и некоторые классы с тривиальным деструктором) создаваемые или получающие значение в точке определения; при этом используемый конструктор должен отвечать требованиям к constexpr-конструктору, а его параметры должны содержать только литеральные значения либо constexpr-переменные (функции).

Функции — не виртуальные, с параметрами и результатом литеральных типов; исполняемая ветвь которых отвечает требованиям к константным выражениям, а тело содержит только пустые операторы, предложения static_assert / typedef / using и один оператор return.

- В C++14 требования смягчены до отсутствия ассемблерных вставок, операторов goto, try-блоков, определения переменных нелитеральных типов и не инициализируемых объектов статической и потоковой продолжительности хранения.

Спецификатор constexpr: пример (1 / 2, C++11)


```
class constStr {       // литеральный класс
    template <std::size_t SIZE>
    constexpr constStr(const char (&str)[SIZE]) :
        string{str}, size{SIZE - 1} {}
    constexpr char operator[] (std::size_t idx) const {
        return idx < size ? string[idx] :
                            throw std::out_of_range("");
    } // constexpr-функции сигнализируют об ошибках выбросом
      // исключений (в C++11 - из тернарного оператора ?:)
    constexpr std::size_t size() const { return size; }
private:
    const char *string;
    std::size_t size;
};
```

Спецификатор constexpr: пример (2 / 2, C++11)


```
// диалект C++11, в отл. от C++14, не допускает применения
// в constexpr-функциях локальных переменных и циклов
constexpr int fact(int n) {
    return n == 0 ? 1 : (n * fact(n - 1));
}

// использование литерального класса (см. выше)
constexpr std::size_t count(constStr s, std::size_t n = 0,
                            std::size_t c = 0) {
    return n == s.size() ?
        c : s[n] >= 'a' && s[n] <= 'z' ?
        count(s, n + 1, c + 1) : count (s, n + 1, c);
}
```

Шаблоны переменных: пример (C++14)


```
template <class T>
constexpr T PI = T(3.14159265358979323846264433);

template <class T> T area(T radius) {
    return PI<T> * radius * radius;
}
```

Пакеты параметров шаблонов. Развертывание пакетов (C++11)

Пакеты параметров шаблонов, подобно пакетам параметров функций, являются инструментом определения шаблонов с переменным числом параметров (англ. variadic templates).

Каждый пакет параметров (англ. parameter pack) вводит один параметр шаблона, способный в ходе конкретизации принимать один и более аргумент шаблона (типов, значений или шаблонов) либо не принимать ничего.

Развертывание пакета параметров в теле шаблона с переменным числом параметров носит название **образца** (англ. pattern). В ходе развертывания образец заменяется нулем или более экземплярами, разделенными запятыми и следующими в порядке их указания в качестве аргументов шаблона.

Если названия двух пакетов встречаются в одном образце, они должны быть одной длины, чтобы развертываться одновременно.

Пакеты параметров: пример (C++11)


```
// пакет параметров шаблона класса
template <class... Types> struct tuple {};
tuple<>                  t0;
tuple<void*>             t1;
tuple<double, double>    t2;

// пакет параметров шаблона функции
template <class... Types, int... N>
int foo(Types (&...arr)[N]) {}// параметр-троеточие в ()
int container[42];              // д.б. именован (CWG #1488)
int result = foo<const char, int>("42", container);
// Types (&...arr)[N] развертывается в
// const char (&)[3], int(&)[42], см. также примеры далее
```

Развертывание пакетов параметров: пример (1 / 2, C++11)


```
template <typename...> struct tuple {};
template <typename T, typename U> struct pair {};
template <class... Args1> struct outer {
    template <class... Args2> struct inner {
        // pair<Args1, Args2>... - развертывание шаблона
        // pair<Args1, Args2> - образец
        typedef tuple<pair<Args1, Args2>...> type;
    };
};
typedef outer<signed short, signed int>::
        inner<unsigned short, unsigned int>::type T;
// T есть tuple<pair<short, unsigned short>,
// pair<int, unsigned int>>
```

Развертывание пакетов параметров: пример (2 / 2, C++11)


```
// в спецификаторах базовых классов и списках инициализации
template <class... Mixins>
class Provider : public Mixins... {
    Provider(const Mixins&... mixins) : Mixins(mixins)... {}
};

// в операторе sizeof...
template <class... Types> struct count {
    static const std::size_t value = sizeof...(Types);
};

// в спецификаторе динамических исключений
template <class... Exceptions>
void bad(int) throw(Exceptions...) { }
```

Выражения-свертки (C++17)

Выражения-свертки выполняют свертку (редукцию) пакетов параметров по одной из 32 разрешенных бинарных операций языка C++17. При этом различают:

- правую унарную свертку:
$$(\mathfrak{P} \circ \cdots) \longrightarrow \mathfrak{p}_1 \circ \left(\cdots \circ (\mathfrak{p}_{N-1} \circ \mathfrak{p}_N)\right)$$
- левую унарную свертку:
$$(\cdots \circ \mathfrak{P}) \longrightarrow \left((\mathfrak{p}_1 \circ \mathfrak{p}_2) \circ \cdots \circ\right)\mathfrak{p}_N$$
- правую бинарную свертку:
$$(\mathfrak{P} \circ \cdots \circ I) \longrightarrow \mathfrak{p}_1 \circ \left(\cdots \circ (\mathfrak{p}_{N-1} \circ (\mathfrak{p}_N \circ I))\right)$$
- левую бинарную свертку:
$$(I \circ \cdots \circ \mathfrak{P}) \longrightarrow \left(\left((I \circ \mathfrak{p}_1) \circ \mathfrak{p}_2\right) \circ \cdots \circ\right) \mathfrak{p}_N$$

Операция, свертка по которой осуществляется, имеет наивысший приоритет. Свертка пакетов нулевой длины разрешена только по * (результат: 1), + (int()), & (-1), | (int()), && (true), || (false), , (void()).

Выражения-свертки: пример (C++17)


```
#include <iostream>
template <typename... Args> bool logProduct(Args... args) {
    return (... && args);     // левая унарная свертка
}
bool b = logProduct(true, true, true, false);
// ((true && true) && true) && false; b == false

template <typename... Args> void print(Args&&... args) {
    (std::cout << ... << args) << std::endl;
}
int main() {
    print(4, 2, " Forty-two");        // "42 Forty-two"
    return 0;
}
```

Шаблоны и автоматический вывод типов: пример (Concepts TS)


```
// описание укороченного шаблона (abbreviated template)
// с использованием неограниченного заместителя типа (auto)
void foo(auto a, auto *b);
// эквивалентно:
// template <typename T, typename U> foo(T a, U *b);
// каждый неограниченный заместитель ввод собственный
// параметр-тип
void bar(std::vector<auto*>...);
// эквивалентно:
// template <typename... T> void bar(std::vector<T*>...);
void foobar(auto (auto::*)(auto));
// эквивалентно:
// template <typename T, typename U, typename V>
// void foobar(T (U::*) (V));
```

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**.

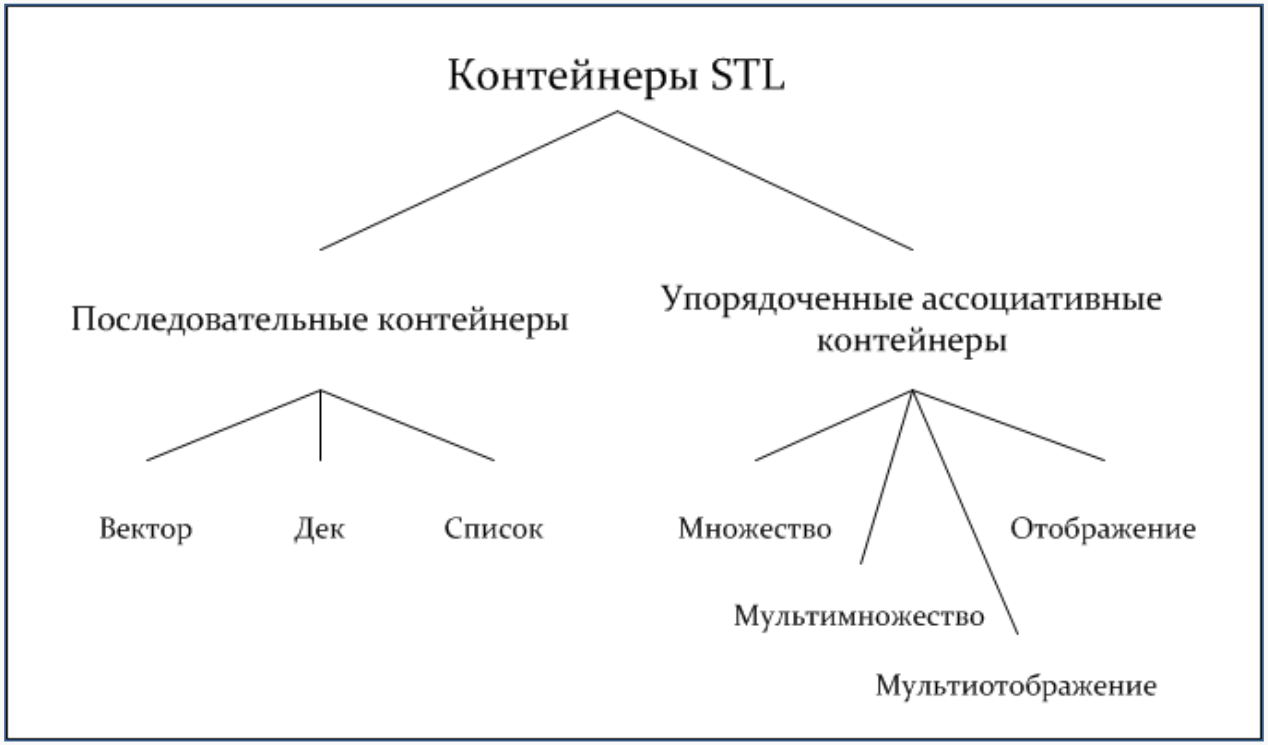
Состав STL

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**.

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
`iterator`	Неконстантный итератор прямого обхода
`const_iterator`	Константный итератор прямого обхода
`reverse_iterator`	Неконстантный итератор обратного обхода
`const_reverse_iterator`	Константный итератор обратного обхода
`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: 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);
```

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>.

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

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

Списки: описание интерфейса (методы упорядочения)

Название метода	Назначение
`sort`	Аналогично алгоритму `std::sort()`
`unique`	Аналогично алгоритму `std::unique()`
`merge`	Аналогично алгоритму `std::merge()`
`reverse`	Аналогично алгоритму `std::reverse()`
`remove` `remove_if`	Аналогично алгоритму `std::remove()`, 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.

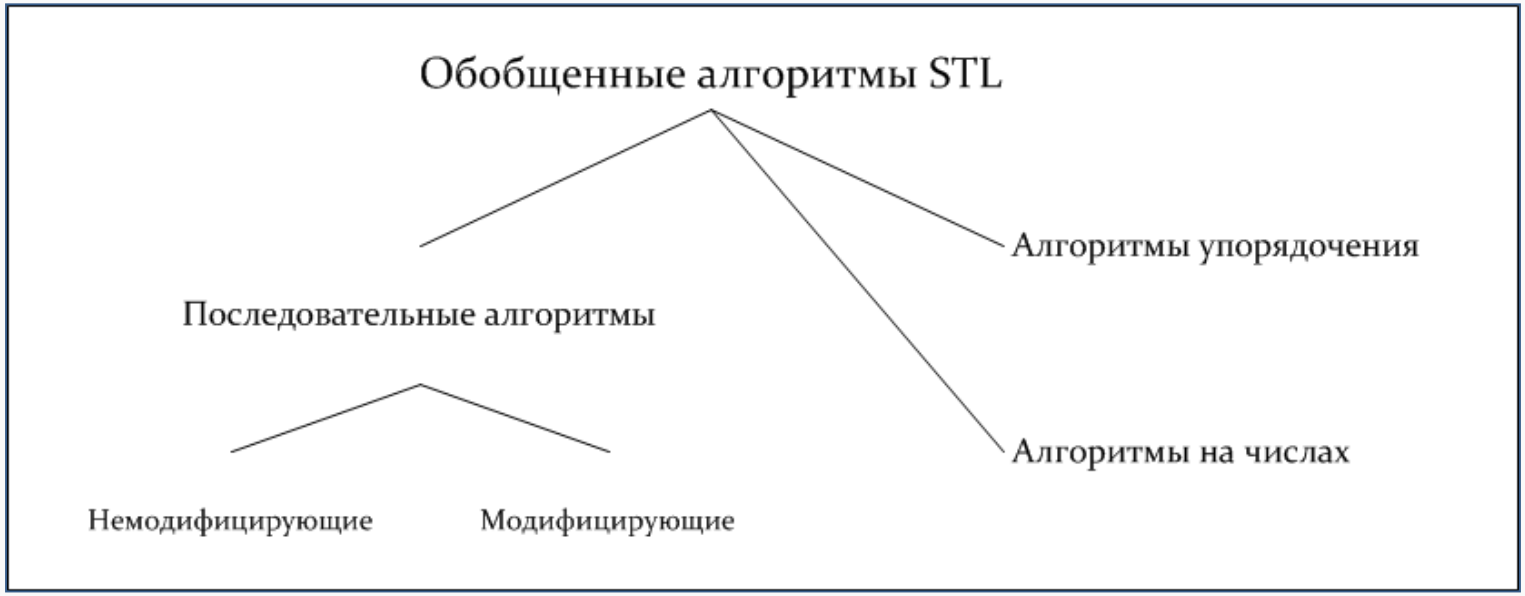
Прочие встроенные типы — аналогичны встроенным типам последовательных контейнеров (value_type — тип std::pair<const Key, T>) со следующими дополнениями:

- 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.



Последовательные алгоритмы

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 STL предикатным функциональным объектом std::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
`std::sort()`	Нестабильная сортировка на месте (вариант quicksort) в среднем за $O(N \log N)$	$O(N^2)$
`std::partial_sort()`	Нестабильная сортировка на месте (вариант heapsort; допускает получение отсортированного поддиапазона длины k)	$O(N \log N)$ или $O(N \log k)$
`std::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.

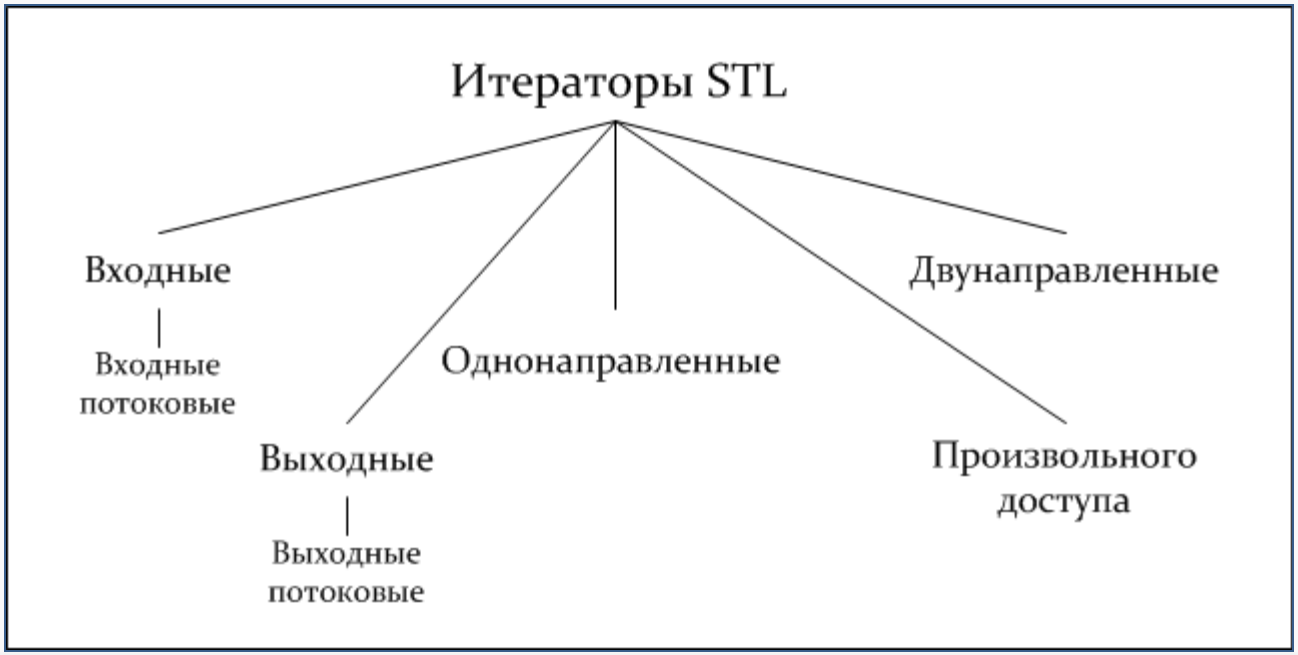
Алгоритмы на числах: обзор

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

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

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];

// ...

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, единственными 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_insert_iterator<Container>
back_inserter(Container &c) {
    return std::back_insert_iterator<Container>(c);
}

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

Потоковые итераторы

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(                          // 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) {
     (3) *result = *first;
         ++first;
     (4) ++result;
    }
    return first;
}
```

Пример: обобщенный алгоритм replace

```
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 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

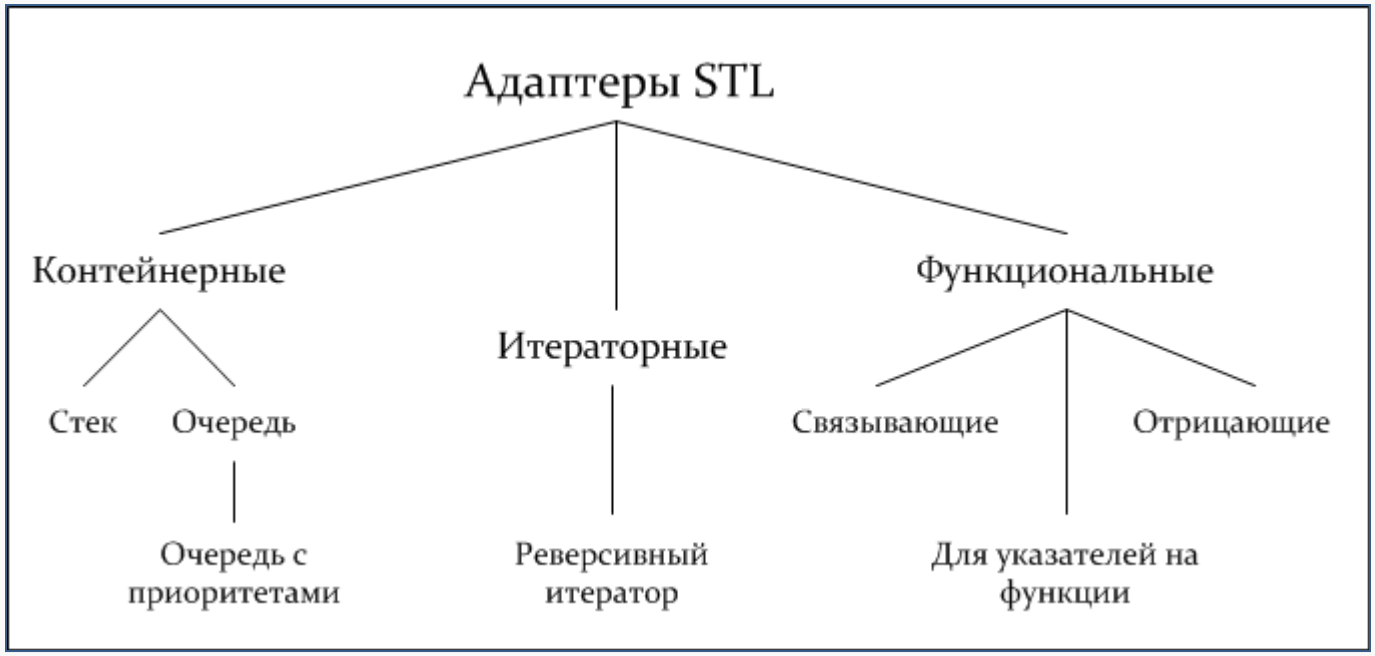

```
template <typename T>
class multiplies : public binary_function<T, T, T> {
public:
    T operator()(const T& x, const T& y) const {
        return x * y;
    }
};
```

1

2

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 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

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)));
```

STL в C++11: контейнеры

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).

STL в C++11: алгоритмы

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.

Практикум №5

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.

Алексей Петров

Спасибо за внимание!

Приложение

«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.

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

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.