Практическая работа:

использование std::vector

Оглавление

[Цели и критерии 3](#_Toc65165020)

[Критерии полноты 3](#_Toc65165021)

[Введение 3](#_Toc65165022)

[Операции с вектором 8](#_Toc65165023)

[Создание объекта вектора 8](#_Toc65165024)

[Обращение к элементам вектора 9](#_Toc65165025)

[Вставка и удаление элементов вектора 11](#_Toc65165026)

[Принцип управления хранилищем вектора 14](#_Toc65165027)

[Пример выполнения задания 16](#_Toc65165028)

[Варианты 18](#_Toc65165029)

# Цели и критерии

**Цель**: освоить std::vector на базовом уровне.

# Критерии полноты

1. Реализовать требуемую операцию в виде *шаблона* функции, способной принимать vector с элементами произвольного типа и возвращающую vector.
2. Функция должна выполняться за линейное по размеру исходного vector время.
3. Реализовать по крайней мере два автоматических теста, размещённых в отдельной тестирующей функции, возвращающей код ошибки.

# Введение

Для организации хранения наборов объектов Стандартная библиотека C++ предоставляет ряд *шаблонов* классов, называемых **контейнерами** *containers*. Стандартные контейнеры обладают определённой общностью интерфейсов, позволяющей в некоторых случаях заменять один контейнер другим, не изменяя при этом кода, ими оперирующего. Существуют сторонние библиотеки, предоставляющие контейнеры, совместимые со стандартными (например, из состава пакета библиотек [Boost](http://www.boost.org/)).

Контейнер задаёт способ хранения помещённых в него объектов (**элементов контейнера**) и распоряжается их существованием. При уничтожении контейнера все хранимые им элементы также уничтожаются (деструктором контейнера).

Наиболее популярным контейнером является std::vector — “**вектор**”, которому посвящена данная работа. Название “вектор” сложилось исторически и означает “динамический массив, способный автоматически увеличивать размер хранилища по мере надобности”, а не вектор в математическом смысле.

Вектор является *шаблоном* класса и в качестве первого (и единственного обязательного) параметра принимает тип хранимых объектов: std::vector<int> — вектор целых чисел, std::vector<std::string> — вектор строк и т.п.

Для доступа к вектору следует подключить стандартный заголовочный файл vector:

#include <vector>

**Шаблоны**

**Шаблон** *template* в C++ — конструкция языка программирования, позволяющая задавать семейства функций (**шаблоны функций** *function templates*) и типов (в частности, **шаблоны классов** *class templates*), параметризованные типами или константами (целочисленных типов, указателей или ссылок). Подстановка параметров шаблона выполняется во время компиляции. Результат подстановки — конкретные функции (из шаблонов функций) или типы (из шаблонов классов).

Данное определение проще пояснить на примерах.

Представим, что мы пишем функцию возведения в квадрат (для простоты).

double sqr(double x)

{

**return** x \* x;

}

Выбор типа double в качестве параметра и результата функции достаточно произволен. Что, если мы возводим в квадрат целые числа и лишние преобразования типа и операции с плавающей точкой ни к чему?

int sqr(int x)

{

**return** x \* x;

}

Точно такую же по сути функцию можно написать для любого числового типа.

size\_t sqr(size\_t x)

{

**return** x \* x;

}

Очевидно, что клонирование одной и той же функции для разных типов — не слишком осмысленная операция. Обычным выходом из подобной ситуации в C++ является определение шаблона функции.

**template** <**class** Num>

Num sqr(Num x)

{

**return** x \* x;

}

Заголовок template <*параметры шаблона*> вводит шаблон (определение “заготовки” функции или типа находится под заголовком). Собственно шаблон не является функцией или типом до тех пор, пока не заданы фактические значения всех его параметров. Для подстановки параметров шаблона их следует перечислить после имени шаблона через запятую, взяв в “угловые скобки” (знаки “меньше” < и “больше” >), что напоминает вызов функции с параметрами.

Можно считать, что компилятор в этот момент подставляет в теле функции или класса вместо имён параметров шаблона их конкретные значения, порождая конкретную реализацию функции или класса.

**auto** a = sqr<int>(10); *// Num = int*

*// a имеет тип int и значение 100*

**auto** f = sqr<float>(2.5f); *// Num = float*

*// f имеет тип float и значение 6.25f*

В том случае когда параметры шаблона функции могут быть выведены из параметров самой функции, их можно не указывать:

**auto** a = sqr(10); *// 10 имеет тип int, поэтому Num = int*

*// a имеет тип int и значение 100*

**auto** f = sqr(2.5f); *// 2.5f имеет тип float, поэтому Num = float*

*// f имеет тип float и значение 6.25f*

Аналогично функциям шаблоны могут принимать параметры по умолчанию.

**template** <**class** First, **class** Second = First>

**struct** Pair *// Пара значений типов First и Second.*

{

First first;

Second second;

*// Конструктор: First() и Second() -- вызовы конструкторов по умолчанию*

*// для типов First и Second соответственно (определены и для встроенных типов).*

Pair(const First &first = First(), const Second &second = Second())

: first(first), second(second) */\* список инициализации:*

*вызов конструкторов копирования для обоих полей \*/*

{}

};

*// Пара чисел с плавающей точкой.*

Pair<double> p2(1, 2); *// First = double, Second = First = double*

assert(p2.first == 1. && p2.second == 2.);

*// Пара "целое, строка".*

Pair<size\_t, string> ids; *// First = size\_t, Second = string;*

ids.first = 23;

ids.second = "23 is the new 42";

Кроме шаблонов функций и классов C++ (начиная с C++11) предоставляет возможность объявлять шаблоны синонимов типов, т.е. записать короткое имя для типа-шаблона с возможностью указать нужные параметры.

**template** <**class** T>

**using** Id\_block = Pair<size\_t, T>;

*// Теперь Id\_block<T> -- то же, что и Pair<size\_t, T> для любого T.*

Помимо возможности вводить сокращения для сложных шаблонных конструкций, шаблоны синонимов типов позволяют задавать нечто, напоминающее функции, но вычисляющее типы (“метафункции”). Строго говоря, для этого требуется использование других шаблонов, в то время как шаблон синонима типа вводит удобный интерфейс, но если использовать уже готовые конструкции (например, из Стандартной библиотеки уровня C++14), то может сложиться впечатление, что мы пишем что-то вроде функции, параметрами которой являются параметры шаблона, а значением — описание типа, стоящее справа от =.

#include <cstdint>

#include <type\_traits> *// C++14, conditional\_t, is\_same*

**using** **namespace** std;

*// Выбор встроенного целочисленного типа, имеющего ширину,*

*// не менее Bits бит, или void, если подходящего типа нет.*

**template** <unsigned Bits>

**using** Uint =

conditional\_t<(Bits <= 8), uint8\_t,

conditional\_t<(Bits <= 16), uint16\_t,

conditional\_t<(Bits <= 32), uint32\_t,

conditional\_t<(Bits <= 64), uint64\_t,

void>>>>;

*// Проверка условия, выполняемая во время компиляции.*

**static\_assert**(is\_same<Uint<18>, uint32\_t>::value, "Uint is flawed!");

# Операции с вектором

# Создание объекта вектора

Ниже приведены примеры использования некоторых конструкторов std::vector на примере вектора целых чисел.

vector<int> ve; *// пустой вектор*

assert(ve.empty());

vector<int> vn(10); *// вектор размера 10*

*// vn создаёт объекты со значением, возвращаемым конструктором без параметров,*

*// для встроенных типов чисел это 0*

assert(!vn.empty());

assert(vn.size() == 10);

assert(vn[0] == 0);

vector<int> vi(10, 42); *// вектор из 10 значений, равных 42*

*// в качестве второго параметра можно указать конкретное значение*

*// создаваемых объектов*

assert(vi.size() == 10);

assert(vi[0] == 42);

int arr[] { 1, 2, 3, 4, 5 };

vector<int> va(arr + 1, arr + 4); *// копия диапазона массива*

*// va содержит 3 элемента: 2, 3, 4*

assert(va.size() == 3);

assert(va[0] == arr[1] && va[1] == arr[2] && va[2] == arr[3]);

*// Наконец, вектор можно создать из конкретного набора значений,*

*// используя фигурные скобки вместо круглых (C++11).*

vector<int> vl { 1, 2, 3, 4 }; *// можно также ставить "=": vl = { ...*

assert(vl.size() == 4);

assert(vl.front() == 1 && vl.back() == 4);

Как и другие типы в C++, векторы можно создавать как временные значения, вызывая конструктор явно. Например, может быть удобно проверить равенство содержимого некоторого вектора заданному набору значений следующим образом:

*// Используем последний конструктор из примера выше.*

vector<int> vl { 1, 2, 3, 4 };

*// Повторно используем тот же конструктор*

*// для создания временного объекта с заданным содержимым.*

*// Обратите внимание -- необходимо указывать тип элемента вектора (здесь int).*

assert(( vl == vector<int> { 1, 2, 3, 4 } ));

assert(( vl != vector<int> { 2, 3, 4 } ));

# Обращение к элементам вектора

Вектор представляет собой массив, размещённый в динамической памяти, все элементы которого идут подряд от младших адресов к старшим, точно так же как и в обычном массиве (требование стандарта C++11). Но в отличие от встроенных массивов, вектор “знает” свою длину. Кроме того, вектор может быть пустым (размер 0).

vector<int> vn(100); *// сто нулей*

*// Проверить вектор на пустоту -- функция empty.*

assert(!vn.empty());

*// Узнать размер вектора -- функция size.*

assert(vn.size() == 100);

*// Получить указатель на хранилище элементов -- функция data.*

assert(vn.data() == &vn[0]);

assert(&vn[99] - vn.data() == 99);

К элементам вектора можно обращаться по числовым индексам с помощью оператора []. Как и в обычном массиве, первый элемент имеет индекс 0, последний — (*размер вектора* – 1). Кроме того, для обращения к первому и последнему элементам определены специальные функции front и back. Это сделано для совместимости с контейнерами, не позволяющими обращаться к элементам по индексам, но позволяющими обращаться к первому и последнему элементам особо (например, таким является *двусвязный список* std::list).

*// Обратиться к первому элементу (индекс 0) -- функция front.*

assert(&vn.front() == &vn[0]);

*// Обратиться к последнему элементу (индекс size() - 1) -- функция back.*

assert(&vn.back() == &vn[vn.size() - 1]);

Оператор [] по умолчанию не выполняет проверку допустимости значения индекса. Попытка обратиться к несуществующему элементу ведёт к [неопределённому поведению](https://teccxx.neocities.org/mx1/ub.html). Вместо [] можно использовать функцию at, бросающую [*исключение*](https://teccxx.neocities.org/mx1/exceptions.html) в случае неверного индекса.

assert(&vn.at(10) == &vn[10]);

*// А здесь будет ошибка -- исключение, но не UB.*

vn.at(200) = 1000;

Если нет необходимости в прямом использовании индекса в процессе перебора всех элементов вектора, то для осуществления этого перебора можно воспользоваться циклом for(:):

vector<char> word { 'h', 'e', 'l', 'l', 'o', '!' };

**for** (char letter: word)

cout << letter;

cout << endl;

Следующий пример — функция, которая вычисляет сумму элементов произвольного вектора.

**template** <**class** T>

T sum(const std::vector<T> &xs)

{

T s = T(); *// ноль или, например, пустая строка (если T = string)*

**for** (**auto** &x: xs)

s += x;

**return** s;

}

# Вставка и удаление элементов вектора

Вектор предоставляет возможность вставки и удаления элементов в любой позиции. Впрочем, следует помнить, что вставка и удаление элементов не с конца вектора обязательно влечёт перемещение его “хвоста” в памяти (так как все элементы вектора идут в памяти подряд как в обычном массиве), что может быть времязатратной операцией.

vector<int> vi; *// пустой вектор*

assert(vi.empty());

*// Добавлять элементы в конец вектора можно функцией push\_back.*

vi.push\_back(1);

assert(vi.size() == 1 && vi[0] == 1);

vi.push\_back(2);

assert(vi.size() == 2 && vi[1] == 2);

vi.push\_back(3);

assert(( vi == vector<int> { 1, 2, 3 } ));

*// Функция pop\_back, наоборот, удаляет последний элемент.*

vi.pop\_back();

assert(( vi == vector<int> { 1, 2 } ));

*// Функция clear удаляет все элементы.*

vi.clear();

assert(vi.empty());

Для вставки/удаления элементов в произвольных позициях предназначены функции insert и erase. *Позицию* можно получить с помощью функции begin (*начало*; можно добавить сдвиг ∈ [0, size()]) или end (*конец*; можно вычесть сдвиг ∈ [0, size()]).

vi = { 1, 2, 3, 4 }; *// теперь vi содержит 4 элемента: 1, 2, 3, 4.*

*// Вставить значение на место vi[2] (старый vi[2] переходит в vi[3] и т.д.).*

vi.insert(vi.begin() + 2, 5);

assert(( vi == vector<int> { 1, 2, 5, 3, 4 } ));

*// Вставить группу значений на vi[vi.size() - 2] == vi[3].*

vi.insert(vi.end() - 2, { 6, 7 });

assert(( vi == vector<int> { 1, 2, 5, 6, 7, 3, 4 } ));

*// Удалим один элемент (vi[1]).*

vi.erase(vi.begin() + 1);

*// И ещё один (vi[vi.size() - 2] == vi[4]).*

vi.erase(vi.end() - 2);

assert(( vi == vector<int> { 1, 5, 6, 7, 4 } ));

*// Можно удалить сразу диапазон элементов.*

*// Например, все, кроме первого и последнего.*

vi.erase(vi.begin() + 1, vi.end() - 1);

assert(( vi == vector<int> { 1, 4 } ));

Использование в качестве позиций смещений относительно результатов неких функцией begin и end вместо обычных индексов является элементом совместимости вектора с другими контейнерами (как правило, не позволяющими адресовать элементы по индексам). Значения, возвращаемые begin и end называются **итераторы** и ведут себя аналогично указателям (указатели — частный случай итераторов). Каждый контейнер определяет свои типы итераторов и набор определённых на них операций. Для вектора этот набор максимален и допускает арифметику, полностью аналогичную арифметике указателей.

Внешние стандартные функции begin и end, будучи применены к вектору, вернут то же самое, что и использовавшиеся выше функции-члены begin и end соответственно. А будучи применены к статическому массиву, данные функции вернут указатели на первый элемент и мнимый элемент, расположенный сразу за последним. Эти указатели суть итераторы статического массива.

Диапазоны в Стандартной библиотеке C++ всегда задаются как полуинтервалы [b, e). Элемент, на который указывает итератор e, в диапазон уже не включается (и может не существовать).

int arr[] = { 9, 8, 7 };

*// Вставить на vi[1] содержимое arr целиком.*

*// Вместо arr мог быть объект класса любого стандартного контейнера.*

vi.insert(begin(vi) + 1, begin(arr), end(arr));

assert(( vi == vector<int> { 1, 9, 8, 7, 4 } ));

С помощью функции resize можно изменять размер вектора.

*// В случае уменьшения лишние элементы удаляются с конца.*

vi.resize(3);

assert(( vi == vector<int> { 1, 9, 8 } ));

*// В случае увеличения новые элементы добавляются с конца.*

*// Если не указать значение, они инициализируются конструктором по умолчанию.*

vi.resize(4);

assert(( vi == vector<int> { 1, 9, 8, 0 } ));

vi.resize(6, 99);

assert(( vi == vector<int> { 1, 9, 8, 0, 99, 99 } ));

Если в приведённых выше примерах не всё ясно, то рекомендуется попробовать на основе этих примеров написать простую программу, которая:

1. Заполняет вектор числами от 0 до некоторого *n* и затем выводит результат (содержимое вектора) в консоль.
2. Возводит все элементы вектора в квадрат на месте.
3. Вставляет в середину вектора десять нулей.
4. Удаляет из начала вектора min(10, *размер вектора*/2) элементов.
5. Обращает порядок элементов вектора и выводит финальный результат в консоль.

# Принцип управления хранилищем вектора

Вектор обеспечивает вставку элементов в конец в среднем за *O*(1)-время, несмотря на тот факт, что периодически приходится выделять новый достаточно большой блок памяти и переносить туда всё содержимое вектора, после чего старый блок памяти освобождается. Это достигается за счёт того, что при выделении нового блока вектор обычно выделяет памяти больше, чем требуется именно в момент увеличения, и часть хранилища остаётся до поры незадействованной. Узнать полный объём текущего хранилища (в элементах) можно с помощью функции capacity (из них реально занято size() позиций).

vector<int> vi;

*// Посмотрим, как полный объём выделенного хранилища*

*// соотносится с занятым объёмом (размером вектора).*

**for** (size\_t i = 1; i < 35; ++i)

{

vi.push\_back(i);

cout << vi.size() << '\t' << vi.capacity() << endl;

}

При реальном увеличении хранилища реализации вектора обычно используют простую схему: *новый объём* = *g*\**старый объём*, где *g* ∈ (1, 2] — константа роста (*grow factor*). Обычно *g* = 1.5 или *g* = 2. Впрочем, значения *g* ≥ 2 могут привести к нежелательному эффекту: невозможности повторно задействовать ранее выделенную и затем освобождённую память. Каждая строчка ниже соответствует одному выделению нового хранилища. Предположим, что доступная память простирается на непрерывном диапазоне адресов, достаточном для размещения 16 элементов.

Пример для *g* = 2

Легко показать, что в этом случае освобождённое начало диапазона никогда не может быть повторно задействовано в качестве хранилища вектора, поскольку всегда будет не хватать ровно одного элемента: 20 + 21 + … + 2*n* = 2*n*+1 – 1. В случае *g* < 2 иногда будет происходить “возврат”, потому что накопленные освобождённые блоки рано или поздно достигнут требуемого для хранилища объёма. Например, *g* = 1.5 позволяет в том же примере выделить хранилище объёмом на 1 элемент больше:

Пример для *g* = 1.5

Вообще, чем больше *g*, тем реже будет происходить выделение памяти и копирование элементов. Чем меньше *g*, тем эффективнее будет использоваться доступное хранилище (будет меньше незанятых элементов).

*O*(1)-время на вставку элемента в среднем достигается в пределе. Пусть было *n* перераспределений памяти и, начиная с некоторого *i*, на перераспределении *i* выделяется блок размера ⌊*gi*⌋, в него копируется полный предыдущий блок, т.е. ⌊*gi*–1⌋ элементов. Итак, после *n* перераспределений было 1 + max(⌊*g*⌋, 2) + … + ⌊*gn*–1⌋ ≤ (*gn* – 1)/(*g* – 1) + *O*(*n*) копирований. Так как всего было добавлено ⌊*gn*–1⌋ + 1 элементов, то в среднем на элемент было осуществлено ((*gn* – 1)/(*g* – 1) + *O*(*n*)) / (⌊*gn*–1⌋ + 1) копирований, что при *g* > 1 и *n* → ∞ даёт 1 в пределе.

Вектор позволяет заранее выделить хранилище нужного объёма, не создавая дополнительных элементов, с помощью функции reserve:

vector<int> vi;

vi.reserve(100);

assert(vi.size() == 0 && vi.capacity() == 100);

Функция reserve не может уменьшить размер выделенного хранилища. Передавая ей некоторое число, мы просим гарантировать наличие хранилища объёмом не менее указанного числа. Впрочем, если мы экономим память, то можно “подогнать” размер хранилища под реально занятый объём с помощью функции shrink\_to\_fit:

vi.push\_back(100);

vi.shrink\_to\_fit();

assert(vi.size() == 1 && vi.capacity() == 1);

# Пример выполнения задания

Функцией clamp(*x*, *p*, *q*) назовём функцию, возвращающую *x*, если *x* ∈ [*p*, *q*]; *p*, если *x* < *p*; *q*, если *x* > *q*. Требуется написать функцию, принимающую вектор некоторых значений и границы *p* и *q* и возвращающую (новый) вектор результатов clamp, применённой к каждому элементу исходного вектора.

Для произвольного типа значений Val, на которых задан порядок, функцию clamp можно определить в виде следующего шаблона:

**template** <**class** Val>

**inline** Val clamp(Val value, Val min\_bound, Val max\_bound)

{

**return** value < min\_bound ? min\_bound

: max\_bound < value ? max\_bound

: value;

}

Применить clamp к вектору достаточно просто: создаём вектор нужного размера и записываем в него результаты clamp поэлементно. В конце возвращаем из функции новый вектор.

**template** <**class** Val>

vector<Val> clamped(const vector<Val> &values, Val min\_bound, Val max\_bound)

{

vector<Val> clv(values.size());

**for** (size\_t i = 0, sz = values.size(); i < sz; ++i)

clv[i] = clamp(values[i], min\_bound, max\_bound);

**return** clv;

}

Тест в данном примере написан на основе чтения чисел в вектор из потока. В заданиях так делать необязательно. Цель примера — показать возможную реализацию чтения вектора заранее неизвестного размера из потока ввода.

**template** <**class** Val>

vector<Val> read\_vector(istream &in)

{

vector<Val> values;

**for** (Val val; in >> val;)

values.push\_back(val);

in.clear(ios::failbit);

values.shrink\_to\_fit();

**return** values;

}

Чтобы использовать функцию read\_vector, необходимо явно указывать тип Val (например read\_vector<int>(cin)), так как компилятору неоткуда узнать, значения какого типа мы планируем прочитать из потока ввода. Собственно тест:

int test\_clamped()

{

stringstream test;

test << "10 4 -5. 2 51 2222.022 4 1.2 25 7 8.5 -";

**auto** values = read\_vector<float>(test);

**if** (clamped(values, 0.f, 100.f) != vector<float>

{ 10.f, 4.f, 0.f, 2.f, 51.f, 100.f, 4.f, 1.2f, 25.f, 7.f, 8.5f })

**return** 1;

**if** (clamped(values, 3.f, 8.f) != vector<float>

{ 8.f, 4.f, 3.f, 3.f, 8.f, 8.f, 4.f, 3.f, 8.f, 7.f, 8.f })

**return** 2;

**return** 0;

}

[C++](https://teccxx.neocities.org/mx1/cpp2/1000-vector_clamp.cpp), [HTML](https://teccxx.neocities.org/mx1/cpp_listing_2.html#vector_clamp.cpp)

# Варианты

Обозначим исходный вектор буквой *a*, результирующий вектор — буквой *b*.

**1**

Вычислить бегущий размах (разность max и min всех встретившихся элементов): *bi* = max { *a*0, *a*1, …, *ai* } – min { *a*0, *a*1, …, *ai* }.

**2**

Вычислить вектор арифметических средних пройденного участка: *b*0 = *a*0, *bi* = (*a*0 + *a*1 + … + *ai*) / (*i* + 1).

**3**

Вычислить вектор частичных разностей (т.е. разности соседних элементов исходного вектора): *b*0 = *a*0, *b*1 = *a*1 – *a*0, …, *bi* = *ai* – *ai*–1.

**4**

Вычислить вектор геометрических средних пройденного участка: *b*0 = *a*0, *bi* = (*a*0 \* *a*1 \* … \* *ai*)1/(*i*+1).

**5**

Вычислить накопленные длины фрагментов ломаной на плоскости. Ломаная задана парами координат вершин (можно вместо *a* использовать два вектора *x* и *y*).

*b*0 = 0, *b*1 = ||*a*1 – *a*0||, *b*2 = *b*1 + ||*a*2 – *a*1||, …, *bi* = *bi*–1 + ||*ai* – *ai*–1||.

**6**

Вычислить вектор арифметических средних каждой группы соседних *k* элементов в исходном векторе: *bi* = (*ai* + *ai*+1 + … + *ai*+*k*–1)/*k*.

**7**

Вычислить вектор “обратных” накопленных сумм: (пусть *n* = размер вектора *a* – 1) *b*0 = *a*0 + *a*1 + … + *an*, *b*1 = *a*1 + *a*2 + … + *an*, …, *bi* = *ai* + *ai*+1 + … + *an*, …, *bn* = *an*.

**8**

Вычислить накопленные длины пути в (трёхмерном) пространстве, состоящего из прямолинейных участков, соединяющими точки с заданными тройками координатами (можно вместо *a* использовать три вектора *x*, *y* и *z*).

*b*0 = 0, *b*1 = ||*a*1 – *a*0||, *b*2 = *b*1 + ||*a*2 – *a*1||, …, *bi* = *bi*–1 + ||*ai* – *ai*–1||.

**9**

Вычислить вектор геометрических средних каждой группы соседних *k* элементов в исходном векторе: *bi* = (*ai* \* *ai*+1 \* … \* *ai*+*k*–1)1/*k*.

**10**

Даны два вектора: *a* и *p* одинаковой длины *N*, причём элементы *p* — целые числа из множества { 0, 1, …, *N*–1 }.

Считая *p* записью перестановки, получить вектор *b* как результат применения перестановки *p* к вектору *a*. “Формульно”: *bi* = *api* (*pi* здесь читать как *pi*).

**Пример**: *a* = { 1, –2.5, 0.3, 8, –3, 5.7 }, *p* = { 5, 1, 3, 2, 0, 4 }. Тогда *b* = { 5.7, –2.5, 8, 0.3, 1, –3 }.

**11**

Дан вектор *a*, содержащий перестановку размера *N* (набор целых чисел { 0, 1, …, *N*–1 } в произвольном порядке). Вычислить вектор *b*, являющийся обратной перестановкой к вектору *a*. “Формульно”: *bai* = *i* (*ai* здесь читать как *ai*).

**Пример**: *a* = { 0, 5, 1, 4, 2, 3 }, тогда *b* = { 0, 2, 4, 5, 3, 1 }.

**12**

Дан вектор целых чисел *a*. Построить вектор *b* такой, что *bi* = сумме цифр десятичного представления *ai*.

**Пример**: *a* = { 5, 10, 15, 20, 25 }, тогда *b* = { 5, 1, 6, 2, 7 }.

**13**

Дан вектор целых чисел *a*. Построить вектор *b* равный вектору *a*, в котором обращён порядок всех нечётных чисел (чётные числа остаются на старых позициях).

**Пример**: *a* = { 0, 1, 2, 3, 5, 6, 4, 8, 7 }, тогда *b* = { 0, 7, 2, 5, 3, 6, 4, 8, 1 }.

**Подсказка**. Для решения данной задачи хорошо подходит следующий алгоритм:

* Завести два индекса: *i* = 0, *j* = размер вектора – 1.
* Повторять:
  + Увеличивать *i*, пока *i* < *j* и *i*-й элемент — чётное число.
  + Уменьшать *j*, пока *i* < *j* и *j*-й элемент — чётное число.
  + Если *i* >= *j*, закончить.
  + Обменять *i*-й и *j*-й элементы местами.
  + *i* = *i* + 1, *j* = *j* – 1.

**14**

**Медианой** последовательности с нечётным числом членов будем называть значение, которое встаёт в середину, если последовательность отсортировать. Т. е. половина значений последовательности не меньше медианного элемента и половина значений не больше медианы.

Для заданного вектора *a* построить вектор *b* из медиан подряд идущих троек элементов. Для неполных троек брать арифметическое среднее.

**Пример**. Дано *a* = { 1, 5, 1, 4, 5, 6, 2, 1, 3, 4, 4, 4, 5, 7 }. Разбиваем на тройки: { 1, 5, 1 }, { 4, 5, 6 }, { 2, 1, 3 }, { 4, 4, 4 }, { 5, 7 }, последняя “тройка” неполная — два элемента. Получаем набор медиан (последнее значение — арифметическое среднее последних двух элементов): *b* = { 1, 5, 2, 4, 6 }.

**15**

Для заданного *k* > 0 и вектора *a* вычислить вектор *b* разностей максимума и минимума элементов всех групп соседних элементов размера *k*. *bi* = max { *ai*, *ai*+1, …, *ai*+*k* } – min { *ai*, *ai*+1, …, *ai*+*k* }.