**Лабораторная работа по теме**

**«Библиотека алгоритмов С/С++»**

**Разработали: Лустгартен Марк**

**Сергеев Никита**

В этой лабораторной работе вы познакомитесь с некоторыми полезными функция библиотеки <algorithm>. Для выполнения работы укажите в настройках проекта версию языка С++14.

**Функция qsort()**

qsort – функция языка С, предназначенная для сортировки элементов любого типа.

Сортирует в порядке возрастания заданный массив, на который указывает ptr. Этот массив должен содержать count элементов, размером в size байт. Функция, заданная по указателю comp, используется для сравнения объектов.

Если функция comp определила, что два элемента одинаковые, то их порядок будет не определён.

void qsort( void \**ptr*, std::size\_t *count*, std::size\_t *size*,*/\*compare-pred\*/*\* comp );

void qsort( void \**ptr*, std::size\_t *count*, std::size\_t *size*,*/\*c-compare-pred\*/*\* comp );

**Параметры**

|  |  |  |
| --- | --- | --- |
| ptr | — | Указатель на массив для сортировки |
| count | — | Количество элементов в массиве |
| size | — | Размер каждого элемента в массиве |
| comp | — | Функция, которая будет сравнивать объекты. Она возвращает отрицательное целое число, если первый аргумент этой функции меньше второго, положительное число, если первый аргумент больше второго, и, наконец, если они равны, то она возвращает 0.  Определение сравнивающей функции должно быть следующим:**int cmp(const void \*a, const void \*b);**  Эта функция не должна модифицировать переданные ей объекты. |

**Возвращаемое значение**

(нет)

**Примечание**

Несмотря на имя, стандарт С++, Си или POSIX не требует, чтобы эта функция реализовывала алгоритм "быстрой сортировки" или давала каких-то гарантий по стабильности и степени сложности алгоритма.

Тип элементов массива должен быть [TrivialType](https://ru.cppreference.com/w/cpp/concept/TrivialType" \o "cpp/concept/TrivialType) (тривиальным типом), иначе поведение этой функции не определено.

**Использование**

При использовании данной функции нам сильно помогут лямбда функции. У вас уже возник вопрос, зачем здесь они здесь? Лямбды нужны, когда функция необходима «здесь и сейчас», нет смысла описывать её как обычную функцию с возможностью использовать потом.

Именно такой чаще всего и должна быть функция comp();

Пусть нам необходимо упорядочить по убыванию элементы вектора. Решим эту задачу с помощью функции qsort().

    std::vector<int> vec = {1,2,3,4,5,6,7,8,9};

std::qsort(vec.begin().base(), size\_t(vec.size()), sizeof(vec.at(0)),

               [](const void \**a*, const void \**b*)

    {

        auto arg1 = \*static\_cast<const int \*>(a);

        auto arg2 = \*static\_cast<const int \*>(b);

        if (arg1 > arg2) return  -1;

        if (arg1 < arg2) return  1;

        return 0;

    } );

Результат



В зависимости от того, как мы опишем правило сравнения элементов, результат будет отсортирован либо по возрастанию, либо по убыванию. Но понятие «возрастание» и «убывание» в данном случае несколько условны, т.к. правила сравнения задаем мы. Например, можно возвращать 1, если число кратно 2 (и не является двойкой), -1, если не кратно и 0, если число равно 2.

**Функция iota()**

iota() заполняет диапазон [first, last) последовательно возрастающими значениями, начиная с value и вычисляя каждое следующее значение как ++value.

template< class ForwardIterator, class T >

void iota( ForwardIterator *first*, ForwardIterator *last*, T *value* );

(начиная с C++11)

(до C++20)

template< class ForwardIt, class T >

constexpr void iota( ForwardIt *first*, ForwardIt *last*, T *value* );

(начиная с C++20)

Эквивалентная операция:

\*(d\_first)   = value;

\*(d\_first+1) = ++value;

\*(d\_first+2) = ++value;

\*(d\_first+3) = ++value;

### Параметры

**first, last** - диапазон элементов для заполнения последовательно увеличивающимися значениями начиная с value.

**Value** - начальное значение для записи в диапазон; выражение ++value должно быть корректно определено (well-formed).

**Возвращаемое значение**

(Нет)

**Сложность**

Ровно last - first инкрементов и присваиваний.

**Использован**ие

Самое просто применение – начальная инициализация элементов вектора.

    std::cout << "iota test";

    std::vector<double> iota\_vec(10);

    std::iota(iota\_vec.begin(), iota\_vec.end(), 0);

    for(auto el : iota\_vec)

    {

        std::cout << el << " ";

    }

    std::cout << std::endl;



**Функция random\_shuffle()**

Изменяет порядок элементов в заданном диапазоне [first, last) таким образом, что все возможные перестановки этих элементов могут появиться с одинаковой вероятностью.

Есть 3 варианта:

1) Генератор случайных чисел определяется реализацией, но часто используется функция [std::rand](https://ru.cppreference.com/w/cpp/numeric/random/rand" \o "cpp/numeric/random/rand).

2) Генератором случайных чисел является функциональный объект r.

3) Генератором случайных чисел является функциональный объект g.

template< class RandomIt >

void random\_shuffle( RandomIt *first*, RandomIt *last* );

(1) (deprecated in C++14)

(убрано в C++17)

(2)

template< class RandomIt, class RandomFunc >

void random\_shuffle( RandomIt *first*, RandomIt *last*, RandomFunc& *r* );

(до C++11)

template< class RandomIt, class RandomFunc >

void random\_shuffle( RandomIt *first*, RandomIt *last*, RandomFunc&& *r* );

(начиная с C++11)

(deprecated in C++14)

(убрано в C++17)

template< class RandomIt, class URBG >

void shuffle( RandomIt *first*, RandomIt *last*, URBG&& *g* );

(3) (начиная с C++11)

**Параметры**

**first, last** - диапазон элементов для перемешивания

**r** - функциональный объект, возвращающий случайное значение из интервала [0, n) преобразуемого в iterator\_traits<RandomIt>::difference\_type типа, будучи вызванным как r(n)

**g** - функциональный объект, возвращающий случайное значение из интервала [g.min(), g.max()] типа URNG::result\_type, будучи вызванным как g() (н-р, любой равномерный генератор случайных чисел из заголовочного файла <random>)

|  |
| --- |
| **Требования к типам** |
| - RandomIt RandomIt должен соответствовать требованиям ValueSwappable и RandomAccessIterator. |
| - URNG должен соответствовать требованиям UniformRandomNumberGenerator. |

**Возвращаемое значение**

(Нет)

**Сложность**

Линейная по расстоянию между first и last.

**Примечание**

Стандарт не навязывает какую-либо реализацию random\_shuffle, поэтому даже использование тех же RandomFunc и URBG в разных реализациях стандартной библиотеки может привести к разным результатам.

**Использование**

Нужно перемещать элементы вектора. Решим задачу с помощью функции random\_shuffle().

    std::random\_shuffle(vec.begin(), vec.end());

    for(auto el : vec)

    {

        cout << el << " ";

    }



**Функция partial\_sum().**

#include <numeric>

     template <class InputIterator, class OutputIterator>

     OutputIterator partial\_sum (InputIterator *first*,

                                InputIterator *last*,

                                OutputIterator *result*);

     template <class InputIterator,

              class OutputIterator,

              class BinaryOperation>

     OutputIterator partial\_sum (InputIterator *first*,

                                InputIterator *last*,

                                OutputIterator *result*,

                                BinaryOperation *binary\_op*);

partial\_sum – алгоритм из библиотеки <numeric>, создающий новую последовательность, в которой каждый элемент формируется путем добавления всех значений предыдущих элементов или, во второй форме алгоритма, путем последовательного применения операции binary\_op к каждому предыдущему элементу.

Как не сложно догадаться, при использовании второй формы данного алгоритма удобнее всего использовать лямбда функцию. Во-первых, потому, что нет особого смысла обрамлять одно действие с двумя переменными в отдельную функцию, а во-вторых, так код становится более понятным.

Пример использования:

int main(int *argc*, char \**argv*[])

{

    vector<int> vec = { 1,2,3,4,5,6,7,8,9 };

    vector<int> vec\_sum(vec.size());

    auto res = partial\_sum(vec.begin(), vec.end(),

        vec\_sum.begin(),

        [](int *a*, int *b*) {return a \* b; });

    cout << " ";

    cout << typeid (res).name() << endl;

    cout << "Calculating multiplexing neightbors\n";

    for (auto num : vec\_sum)

    {

        cout << num << endl;

    }

    return 0;

}

**Функция search().**

#include <algorithm>

template< class ForwardIt1, class ForwardIt2 >

ForwardIt1 search( ForwardIt1 *first*,

                ForwardIt1 *last*,

                ForwardIt2 *s\_first*,

                ForwardIt2 *s\_last* );

template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >

ForwardIt1 search( ForwardIt1 *first*,

                ForwardIt1 *last*,

                ForwardIt2 *s\_first*,

                ForwardIt2 *s\_last*,

                BinaryPredicate *p* );

Алгоритм search ищет первое вхождение последовательности элементов [s\_first, s\_last) в диапазон [first, last - (s\_last - s\_first)). Первый вариант использует operator == для сравнения элементов, второй вариант использует заданный бинарный предикат p.

Пример:

int main(int *argc*, char \**argv*[])

{

    cout << "Search testing";

    vector<int> testVec = {1,2,3,4,5,6,7};

    vector<int> templ = {3,4,5};

    auto templ\_pos = search(testVec.begin(), testVec.end(), templ.begin(), templ.end());

    cout << "\nTemplate position is " << distance(testVec.begin(), templ\_pos) << endl;

    return 0;

}

**Функция search\_n().**

#include <algorithm>

template< class ForwardIt, class Size, class T >

ForwardIt1 search\_n( ForwardIt *first*,

                    ForwardIt *last*,

                    Size *count*,

                    const T& *value* );

template< class ForwardIt, class Size, class T, class BinaryPredicate >

ForwardIt1 search\_n( ForwardIt *first*,

                    ForwardIt *last*,

                    Size *count*,

                    const T& *value*,

                    BinaryPredicate *p* );

Алгоритм search\_n можно считать частным случаем предыдущего алгоритма. Search\_n ищет в диапазоне [first, last) первую последовательность count одинаковых элементов, каждый из которых равен заданному значению value. Первый вариант использует operator == для сравнения элементов, второй вариант использует заданный бинарный предикат p.

Пример:

int main(int *argc*, char \**argv*[])

{

    vector<int> vec = { 1,2,2,4,5,6,7,8,9 };

    vector<int> vec\_sum(vec.size());

    cout << "Searching template {2,2}\n";

    auto pos = search\_n(begin(vec), end(vec), 2, 2);

    if(pos == vec.end())

    {

        cout << "Template wasn't found\n" ;

    }

    else {

        cout << "Template position is " << distance(vec.begin(),pos);

    }

    return 0;

}

**Задание 1.**

1. С помощью функции iota() заполните вектор соответствующим вашему варианту образом.
2. Перемешайте содержимое вектора случайным образом.
3. Отсортируйте вектор по указанному для варианта принципу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер варианта | Тип данных | Количество элементов | Значение первого элемента | Правило сортировки |
| 1 | Int | 23 | 12 | По кратности 3 |
| 2 | Double | 15 | 5.113548 | По четности разности пар (если разность 0, сравнение вернет 0) |
| 3 | Char | 13 | a | По убыванию |
| 4 | Uint16\_t | 20 | 0x3145 | По возрастанию |
| 5 | Uint64\_t | 20 | 0x465467f | По убыванию |

**Задание 2.**

|  |  |
| --- | --- |
| Номер варианта | Задание |
| 1 | При помощи функций библиотеки <algorithm> найти наибольшую последовательность из одинаковых символов в случайно сгенерированном массиве char размером 1000 элементов. |
| 2 | Дан массив из целочисленных значений ускорения автомобиля в каждый момент времени. При помощи функций библиотеки <algorithm> получите аналогичные массивы со скоростью и положением автомобиля. |
| 3 | При помощи функций библиотеки <algorithm> найти наибольшую последовательность из одинаковых чисел в случайно сгенерированном массиве int размером 1000 элементов. |
| 4 | Даны 3 целочисленных массива, описывающий перемещение объекта в пространстве (Например: Z = {-1; 10; -15; 5 …}). При помощи функций библиотеки <algorithm> найти абсолютные значения координат объекта, на каждом из шагов. Начальное положение объекта задается пользователем. |