

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ



**Дніпровський національний університет  
залізничного транспорту імені академіка В. Лазаряна**

Кафедра «Комп'ютерні інформаційні технології»

**Лабораторна робота № 11**

**з дисципліни «Об'єктно-орієнтоване програмування»**

**на тему: «Послідовні контейнери list, vector»**

Виконав:  
студент гр.ПЗ1911  
Сафонов Д. Є.  
Прийняла:  
Демидович І.М.

Дніпро, 2021

## Тема. Послідовні контейнери list, vector

**Завдання.** Написати програму яка демонструвала б роботу зі списком і вектором. Розглянути 2 випадки: елементом контейнера є стандартний тип, елементом контейнера є призначений для користувача тип. Для демонстрації роботи використовувати операції додавання, видалення, перегляду і не менше трьох інших алгоритмів (методів) бібліотеки STL. При роботі з контейнерами використовувати ітератори.

### Текст програми.

“main.cpp”

```
#include "oop11v1.h"
#include "oop11v2.h"
#include <iostream>

auto main() -> int {
    const double MAX = 10.0;
    std::function<double(double)> f = [] (double x) {return x;};
    std::function<double(double)> g = [MAX] (double x) {return cos(x) * MAX;};
    auto range_ = std::make_pair(0.0, MAX);

    std::cout << "f(x) = x" << std::endl;
    std::cout << "g(x) = 10 * cos(x)" << std::endl;
    std::cout << "f = g on range [0, 10] for xs: ";
    auto xs = v1::intersections(f, g, range_);
    for (auto x : xs) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
}
```

## “oop11v1.h”

```
/// @file oop11v1.h
#ifndef OOP11V1_H
#define OOP11V1_H

#include <algorithm>
#include <cmath>
#include <functional>
#include <ranges>
#include <vector>

namespace v1 {
/**
 * @brief Returns intersections of f and g on range_.
 * @param[in] f y(x) function on 2d plane which exists for all xs in range_
 * @param[in] g y(x) function on 2d plane which exists for all xs in range_
 * @param[in] range_ Range on which intersections between f and g will be searched
 * @tparam N Floating point Numeric Type
 * @return vector<N> of xs in which f and g intersect
 */
template<class N>
auto intersections(std::function<N(N)> f, std::function<N(N)> g, std::pair<N, N> range_) ->
std::vector<N>;

/**
 * @brief Returns num evenly spaced samples, calculate over the interval [start, stop].
 * @param[in] start The starting value of the sequence
 * @param[in] stop The end value of the sequence
 * @param[in] num Number of samples to generate
 * @tparam N Floating point Numeric Type
 * @return num equally spaced samples in the closed interval [start, stop]
 */
template<class N>
auto linspace(N start, N stop, size_t num) -> std::vector<N>;

/**
 * @brief The sign function returns -1 if x < 0, 0 if x==0, 1 if x > 0.
 * @param[in] x input value
 * @tparam N Numeric Type
 * @return The sign of x
 */
template<class N>
auto sign(N x) -> int;
```

```

template<class N>
auto intersections(std::function<N(N)> f, std::function<N(N)> g, std::pair<N, N> range_) ->
std::vector<N> {
    const size_t NUMS = 1000;
    auto xs = linspace(range_.first, range_.second, NUMS);
    auto signs = xs | std::views::transform([f, g] (auto x) {return sign(f(x) - g(x));});
    auto signs_vec = std::vector(signs.begin(), signs.end());
    std::transform(signs_vec.begin(), signs_vec.end(), ++signs_vec.begin(), signs_vec.begin(), []
(auto a, auto b) {return a - b;});
    std::transform(xs.begin(), xs.end(), signs_vec.begin(), xs.begin(), [] (auto x, auto sign_) {return
sign_ == 0 ? INFINITY : x;});
    auto res = xs | std::views::filter([] (auto x) {return x != INFINITY;});
    return std::vector(res.begin(), --res.end()); //last num wasnt checked
}

```

```

template<class N>
auto linspace(N start, N stop, size_t num) -> std::vector<N> {
    std::vector<N> vec(num);
    N step = (stop - start) / (num - 1);
    std::generate(vec.begin(), vec.end(), [n = start - step, step] () mutable {n += step; return n;});
    return vec;
}

```

```

template<class N>
auto sign(N x) -> int {
    return x > 0 ? 1
        : x < 0 ? -1
        : 0;
}

```

```

} //namespace v1
#endif

```

## “oop12v2.h”

```
/// @file oop11v2.h
#ifndef OOP11V2_H
#define OOP11V2_H

#include <functional>
#include <list>

namespace v2 {
/**
 * @brief Returns intersections of f and g on range_.
 * @param[in] f y(x) function on 2d plane which exists for all xs in range_
 * @param[in] g y(x) function on 2d plane which exists for all xs in range_
 * @param[in] range_ Range on which intersections between f and g will be searched
 * @tparam N Floating point Numeric Type
 * @return list<N> of xs in which f and g intersect
 */
template<class N>
auto intersections(std::function<N(N)> f, std::function<N(N)> g, std::pair<N, N> range_) ->
std::list<N>;

/**
 * @brief Returns num evenly spaced samples, calculate over the interval [start, stop].
 * @param[in] start The starting value of the sequence
 * @param[in] stop The end value of the sequence
 * @param[in] num Number of samples to generate
 * @tparam N Floating point Numeric Type
 * @return num equally spaced samples in the closed interval [start, stop]
 */
template<class N>
auto linspace(N start, N stop, size_t num) -> std::list<N>;

/**
 * @brief The sign function returns -1 if x < 0, 0 if x==0, 1 if x > 0.
 * @param[in] x input value
 * @tparam N Numeric Type
 * @return The sign of x
 */
template<class N>
auto sign(N x) -> int;

template<class N>
```

```

auto intersections(std::function<N(N)> f, std::function<N(N)> g, std::pair<N, N> range_) ->
std::list<N> {
    const size_t NUMS = 1000;
    auto xs = linspace(range_.first, range_.second, NUMS);
    std::list<int> signs(NUMS);

    for (auto x = xs.begin(), s = signs.begin(); x != xs.end(); x++, s++) {
        *s = sign(f(*x) - g(*x));
    }

    auto res = std::list<N>();

    for (auto x = xs.begin(), s1 = signs.begin(), s2 = ++signs.begin(); s2 != signs.end(); x++, s1++,
s2++) {
        if (*s1 - *s2 != 0) {
            res.push_back(*x);
        }
    }

    return res;
}

```

```

template<class N>
auto linspace(N start, N stop, size_t num) -> std::list<N> {
    std::list<N> l(num);
    N step = (stop - start) / (num - 1);
    N val = start;
    for (auto& i : l) {
        i = val;
        val += step;
    }
    return l;
}

```

```

template<class N>
auto sign(N x) -> int {
    return x > 0 ? 1
        : x < 0 ? -1
        : 0;
}

```

```

} // namespace v2
#endif

```

## Приклад роботи програми.

```
f(x) = x  
g(x) = 10 * cos(x)  
f = g on range [0, 10] for xs: 1.42142 5.26527 7.06707
```

Рисунок 1(Результат роботи програми)



Solutions

$x \approx 1.42755$

$x \approx 5.26712$

$x \approx 7.06889$

Рисунок 2(Результат наданий Wolfram Alpha(для порівняння))

## Висновки.

`std::list` та `std::vector` — послідовні контейнери з стандартної бібліотеки C++. Головна різниця між ними у тому що `std::vector` зберігає елементи послідовно, а `std::list` є реалізацією двозв'язного списку. З цих відмінностей витікають усі інші:

- у `std::list` стала складність додавання та видалення елементів, але лінійна складність доступу до елементів(через те, що доступ до кожного елементу(окрім першого) можливий тільки через доступ до попереднього елементу(саме через це у цього контейнера немає оператора доступу за індексом)),
- у `std::vector` стала складність доступу до елементів та додавання або видалення їх у кінець. `std::vector` зберігає додаткову пам'ять щоб можна було додавати елементи у сталий час, але якщо оперативна пам'ять дуже сильно фрагментована можуть бути випадки коли виділити пам'ять завчасно неможливо — тоді додавання нового елементу буде коштовним(якщо кількість елементів відома завчасно можна зарезервувати пам'ять під них, так само можна повернути зарезервовану пам'ять системі, якщо відомо що додавати елементи більше не потрібно). Але `std::vector` як і `std::list` має мінус, а саме — додавання та видалення елементів де завгодно окрім кінця(усі елементи від кінця до потрібного елементу переносяться на кількість вставлених або видалених елементів).

Виходячи з цієї інформації можна визначити найкращі випадки використання обох контейнерів. А саме:

- `std::vector` — коли доступ до усіх елементів важливіший ніж їх видалення або додавання, або додавання та видалення елементів не у кінці не потребується.
- `std::list` — коли швидке модифікація контейнеру важливіша ніж доступ до елементів.

Але це не зовсім правильний висновок. Насправді `std::list` має набагато вужчу спеціалізацію, через те, що перенесення невеликої кількості невеликих елементів не дуже коштовне, для цих випадків можна використати `std::vector`, але навіть якщо елементи занадто великі можна зберігати вказівники на них, тож виходячи з цього `std::list` потрібен тільки тоді коли є проблеми з наявністю оперативної пам'яті або її фрагментованістю. Але подібне можна сказати й про інші спеціалізовані контейнери з стандартної бібліотеки C++, тож `std::vector` підходить для використання у більшості простих не спеціальних випадках.