**Доморощенов Александр Сергеевич**

**Группа: М8О-113Б**

**Реферат на тему: Игра на C++**

В данном реферате мы будем рассматривать создание игры на C++ , а конкретно - **“Крестики-нолики”**.

**Выбор редактора кода**

Начнем с того, что надо выбрать редактор кода. Редакторов на C++ очень много, все их используют в зависимости от того, насколько он для них удобен, но я выбрал Visual Studio из-за его многофункциональности. На Visual Studio существует такой шаблон проекта, как консольное приложение, оно нам и понадобится. Для того, чтобы этот шаблон был доступен, надо запустить Visual Studio с рабочей нагрузкой "Разработка классических приложений на C++", далее создать проект и при создании выбрать шаблон “Консольное приложение” с тегами C++, Windows и Консоль, либо выбрать “Создание пустого проекта CLR”

**Создание игровых форм**

После запуска нашего проекта мы уже потихоньку перейдем к созданию игры. Для начала нам надо определиться как будет выглядеть наша игра и из каких форм она будет состоять. Я создал две игровые формы: стартовая и основная. В стартовой форме пользователь сможет выбрать режим игры: PvP (с игроком) или PvE (с компьютером), размерность поля и длину комбинации для победы. Соответственно в основной форме у нас и будет происходить сами действия игры.

**Для того, чтобы создать формы надо**:

**Создание стартовый формы**

1. Создать стартовую форму

Перейти в “Обозреватель решений” -> нажать на проект правой кнопки мыши -> выбрать “Добавить” -> “Создать элемент” -> “Форма Windows Forms” -> указать название для формы (StartForm.h)

**Добавление элементов игры на форму**:

1. Текстовая часть

Переходим в панель элементов -> в поисковике вбиваем “Label” и создаем 3 таких элемента

1. Настройка элементов

Для настройки текстовой части можно перейти в окно свойств и далее настроить все необходимые параметры: имя, выводимый текст, размер и т.д.

**Создание компонентов для считывания выбранных полей**:

1. ListBox :

Все то же самое, что и предыдущими элементами:

Заходим в “Панель Элементов” -> вбиваем в поисковике “Listbox” -> выбираем “ListBox”

Проделываем все это 3 раза и корректируем параметры элемента в окне “Свойства”

**Создание кнопки для начла игры:**

1. Аналогично переходим в “Панель элементов” -> вбиваем в поиске “Button” -> выбираем “Button”

В данном случае нам понадобиться всего лишь одна кнопка

“Начать игру”

**Создание дополнительных компонентов:**

Далее я хочу создать 2 маленьких меню: “О программе” и “Выход”

Для этого нам необходимо перейти в “Панель элементов”. Далее выбрать MenuStrip и создать этот компонент на форме, введя название полей.

Со стартовой формой мы закончили, поэтому перейдем к игровой.

**Создание игрового поля**

1. Создать стартовую форму

Перейти в “Обозреватель решений” -> нажать на проект правой кнопки мыши -> выбрать “Добавить” -> “Создать элемент” -> “Форма Windows Forms” -> указать название для формы (StartForm.h)

1. Для начала нам понадобиться статус для того чтобы выводить, чей сейчас ход:

“Панель элементов” -> StatusStrip

1. Далее создаем маленькие меню вверху формы:

Для этого нам необходимо перейти в “Панель элементов”. Далее выбрать MenuStrip и создать этот компонент на форме, введя название полей: “Новая игра”, “Вернуться”, “О программе” и “Выход”

1. После всего этого нам необходимо создать какое-то пространство для игры:

“Панель элементов” -> DataGridView (Обычная таблица, в которой я буду выводить все необходимые элементы, и в качестве ячеек я буду туда выводить кнопочки, с которыми мы потом будем работать)

Вот мы и закончили с Frontend частью, теперь можно переходить к Backend-у.

**Backend Часть**

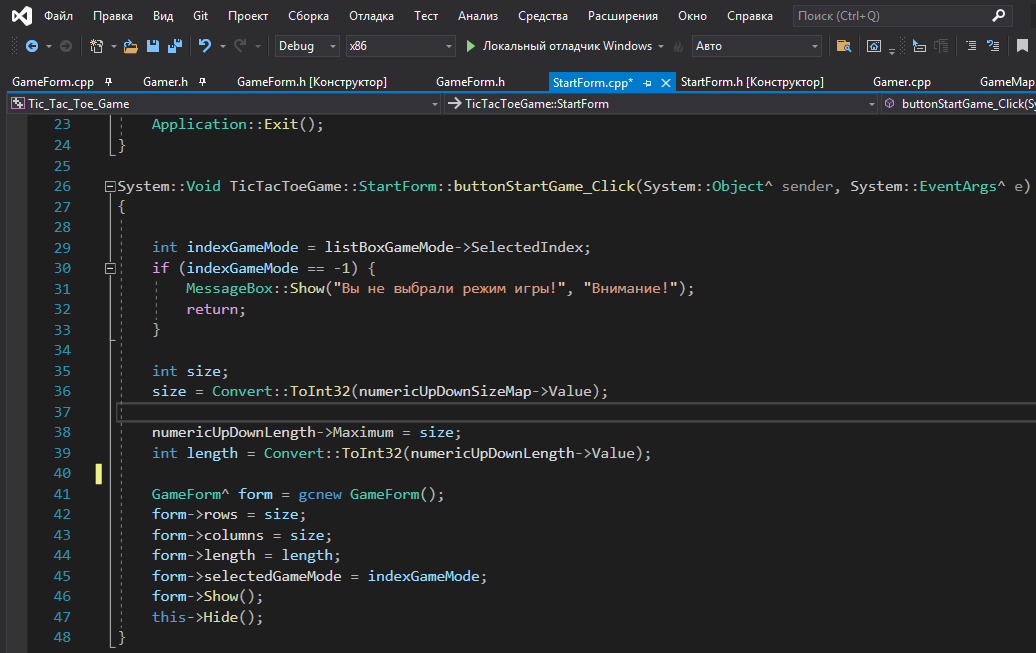
Т.к. в игре будет очень много кнопок и действий, которые нужно будет обрабатывать, то самое главное – в этом всем не запутаться.

В этой части реферата, я не буду представлять весь код, т.к. он слишком большой и в нем содержится много функций и процедур, поэтому я представлю лишь некоторую часть кода, чтобы вы поняли саму суть написания игры и обработки действий. А листинг всей программы будет выложен в репозитории github

**StartForm**

В стартовой форме в принципе не очень много надо делать. Нам нужно описать нашу программу в кнопке

“О программе”, обработать кнопку выхода и передать данные и размеры игрового поля в игровую форму. Это будет выглядеть вот так:

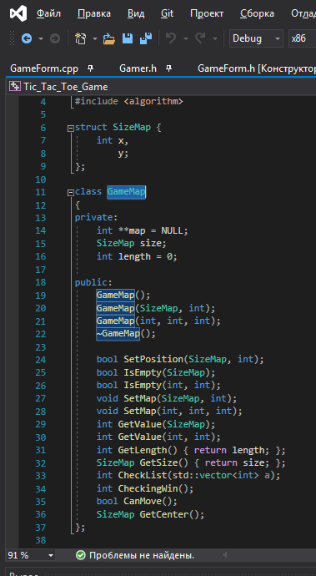


**GameForm**

Теперь перейдем к главной части GameForm. Именно тут я и опишу только несколько классов и процедур.

1. Начнем с первого класса GameMap, с помощью которого мы будем определять позиции игроков, выиграл ли игрок, пустая ли ячейка, может ли двигаться в какую-то ячейку игрок и т.д.

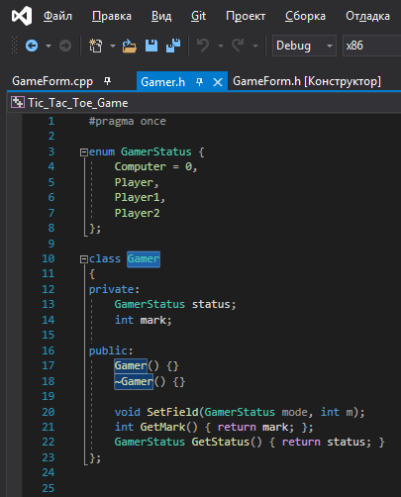
GameMap:



Напомню private – эта та часть данных, которые пользователю видеть не надо, а public – это та часть данных, которую пользователь будет видеть

1. И класс Gamer. Этот класс будет отвечать за статус игрока и его сделанную пометку.

Gamer:



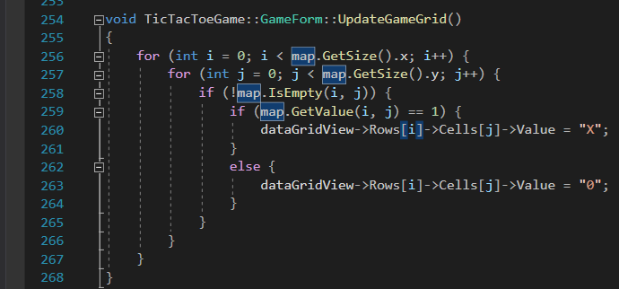
Далее я опишу несколько процедур и функций:

1. UpDate()

Эта процедура будет отвечать за обновление игры, а то есть выводить сообщение о победе, в случае конце игры, выводить в статус сообщение о том, чей сейчас ход, и соответственно обновлять игровое поле и обрабатывать игровую логику, если игра ведется с компьютером.

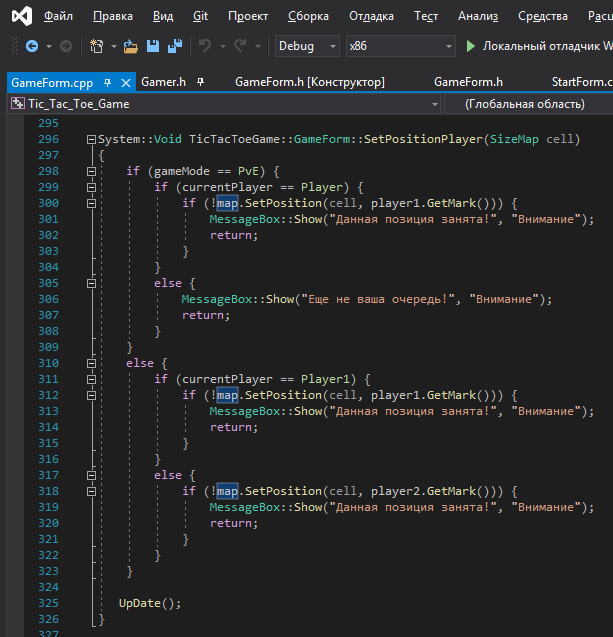
1. UpdateGameGrid()

Эта процедура будет заносить в dataGridView соответственно нолик или крестик работая с классом GameMap и переменной map:

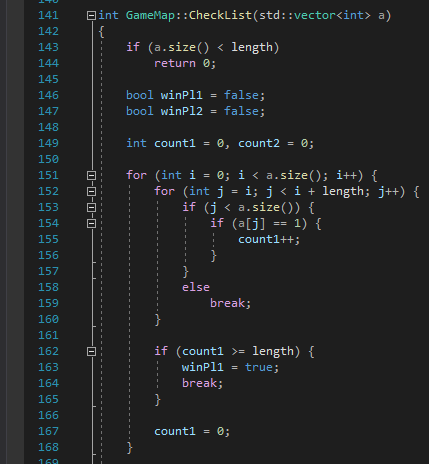


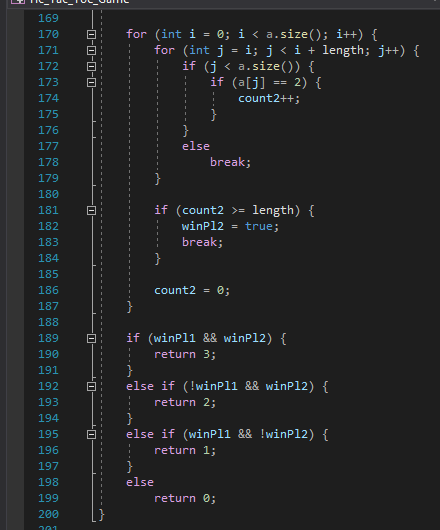
1. SetPositionPlayer()

Эта процедура будет проверять правильность ход игрока, а то есть не нажал ли он на занятую ячейку, сходил ли он тогда, когда его очередь, а не игрока, при этом проверяя все это через вектор cell, в котором хранятся все координаты нажатых ячеек:

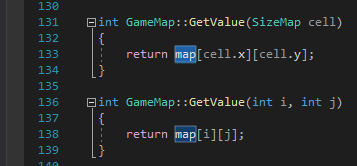


1. Одна из самых важных функций (int CheckList()) – это проверка того, победил ли кто-то из игроков. Тут используются методы обхода матриц, которые мы использовали в 14 и 15 лабораторных работах. Тут мы просто проходим матрицы и проверяем не победил ли кто-то из игроков, и с помощью счетчиков замеряем количество одинаковых меток, далее сравниваем счетчики с длинной комбинации для победы.





1. Ну и последнее - это функция которая заносит в класс map все координаты занятых или нажатых ячеек:



В данных функциях я использую 2 метода, а то есть либо передача данных по координатам напрямую, либо через вектор, который также содержит координаты нажатых ячеек

Теперь, когда про создание игры в кратце рассказано, я бы хотел уделить отдельное внимание контейнерам и рассказать о том, какие контейнеры бывают, какие сложности по времени/памяти для разных операций и про вставку/поиск/удаление элементов в контейнере.

**Контейнеры в С/С++**

Стандартная библиотека предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. Контейнеры являются шаблонами классов. При объявлении переменной контейнера указывается тип элементов, которые будут храниться в контейнере. Контейнеры могут создаваться с использованием списков инициализаторов. Они имеют функции элементов для добавления и удаления элементов и выполнения других операций.

Итерация элементов в контейнере и доступ к отдельным элементам осуществляются с помощью [итераторов](https://docs.microsoft.com/ru-ru/cpp/standard-library/iterators?view=msvc-160)(Структур данных, которая указывает на некоторый элемент контейнера). Итераторы можно использовать явно, используя функции и операторы их членов и глобальные функции. Вы можете также использовать их неявно, например с помощью цикла range-for. Итераторы для всех контейнеров стандартной библиотеки C++ имеют общий интерфейс, но каждый контейнер определяет собственные специализированные итераторы.

Контейнеры можно разделить на три категории: последовательные контейнеры, ассоциативные контейнеры и контейнеры-адаптеры.

**Последовательные контейнеры**

Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов.

**1)Контейнер vector** ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер vector является наиболее предпочтительным последовательным контейнером для большинства областей применения. Если вы сомневаетесь в выборе вида последовательного контейнера, начните с использования вектора. Дополнительные сведения см. в разделе класс.

**2)Контейнер deque** (двусторонняя очередь) обеспечивает быструю вставку и удаление в начале и в конце контейнера. Он предоставляет преимущества для произвольного доступа и гибкой длины vector , но не непрерывно.

**3)List Контейнер** — это двунаправленный связанный список, который обеспечивает двунаправленный доступ, быструю вставку и быстрое удаление в любом месте в контейнере, но не может получить случайный доступ к элементу в контейнере. Дополнительные сведения см. в разделе класс.

**4)Контейнер forward\_list** — однонаправленный список. Это версия контейнера list только с доступом в прямом направлении. Дополнительные сведения см. в разделе класс.

## Ассоциативные контейнеры

В ассоциативных контейнерах элементы вставляются в предварительно определенном порядке — например, с сортировкой по возрастанию. Также доступны неупорядоченные ассоциативные контейнеры. Ассоциативные контейнеры можно объединить в два подмножества: сопоставления (set) и наборы (map).

**1)Контейнер map**, который иногда называют словарем, состоит из пар "ключ-значение". Ключ используется для упорядочивания последовательности, а значение связано с ключом. Например, map может содержать ключи, представляющие каждое уникальное ключевое слово в тексте, и соответствующие значения, которые обозначают количество повторений каждого слова в тексте.

**2)set** — это контейнер уникальных элементов, упорядоченных по возрастанию. Каждое его значение также является и ключом. set — это неупорядоченная версия unordered\_set.

Контейнеры map и set разрешают вставку только одного экземпляра ключа или элемента. Если необходимо включить несколько экземпляров элемента, следует использовать контейнер multimap или multiset.

## Контейнеры-адаптеры

Контейнер-адаптер — это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Адаптеры контейнеров не поддерживают итераторы.

**1)Контейнер queue** соответствует семантике FIFO (первым поступил — первым обслужен). Первый элемент, который отправляется, то есть вставляется, в очередь, должен быть первым элементом, извлекаемым из очереди.

Контейнер priority\_queue упорядочен таким образом, что первым в очереди всегда оказывается элемент с наибольшим значением.

**2)Контейнер stack** соответствует семантике LIFO (последним поступил — первым обслужен). Последний элемент, отправленный в стек, становится первым извлекаемым элементом. Так как адаптеры контейнеров не поддерживают итераторы, их нельзя использовать с алгоритмами стандартной библиотеки C++.

Теперь я расскажу про поиск, вставку и удаление элемента и сравню все эти методы по времени и занимаемой памяти.

Более подробно я расскажу про контейнер vector, т.к. он более актуален для 1 семестра 1 курса. А про остальные я расскажу в кратце про их суть, чтобы вы понимали что это такое в принципе. Ну и приведу таблицу сравнения сложностей разных команд для контейнеров.

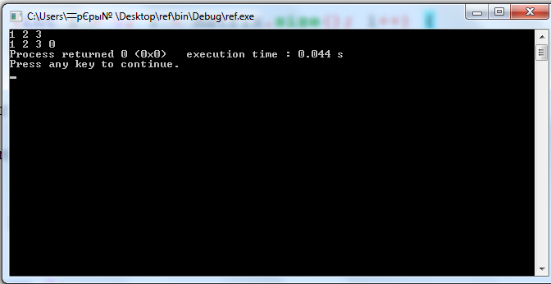
**Поиск/вставка/удаление элемента в последовательном контейнере**

**1)Vector**

**Вставка элемента**:

Вставка элемента осуществляется командами push\_back(), insert(), которые добавляют элементы в вектор соответственно в конец и на любую выбранную позицию.

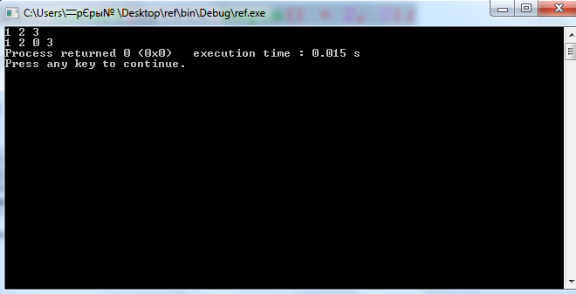
Пример push\_back() с добавлением 0:



Данная операция имеет сложность О(1)

Пример с insert() с добавлением 0 на 2 позицию:

matrix.insert(matrix.begin() + 2, 0);



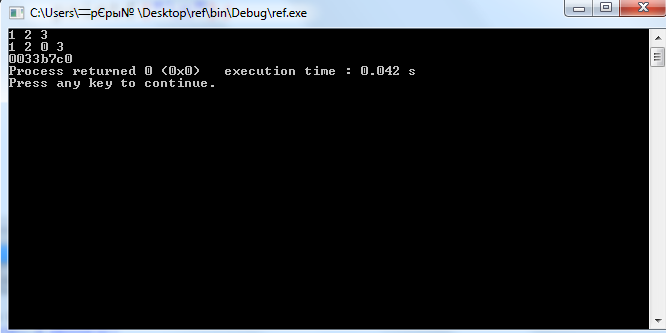
Эта операция имеет сложность О(n)

**Поиск элемента:**

Найти элемент в векторе можно несколькими способами и это зависит от того отсортирован ли вектор или нет.

Если вектор не отсортирован то нужный элемент можно найти командой find (iterator, iterator, element). Эта команда возвращает адрес или итератор, который указывает на найденный элемент, а если он его не нашла, то возвращает значение vector.end(). Если разыменовать итератор, то он нам вернет значение которое мы искали

Пример с нахождением 1 в vector:



Как видите эта команда вернула нам адрес элемента записанный в 16-ой системе счисления. У этой команды сложность O(n)

А если вектор отсортирован, то найти элемент можно быстрее и с помощью команды binary\_search. В нее также подается на вход 2 итератора (начало и конец диапазона поиска) и сам элемент который нужно найти. Результат у нее будет аналогичный, но с меньшей сложность O(log(n)).

Бинарный поиск отличается от обычного только способом поиска. При бинарном поиске наш вектор постоянно делится на 2 части и программа ищет элемент в одной из этих 2 частей и т.д.

**Удаление элемента:**

Аналогично со вставкой есть 2 метода: pop\_back() и erase(). Они действуют абсолютно аналогично только удаляют. Сложность у них соответствующая.

**Итог:**

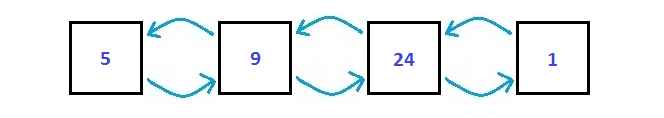
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Вставка | | Поиск | | Удаление | |
|  | push\_back() | insert() | find() | binary\_search() | pop\_back() | erase() |
| vector | 1 | n | n | log(n) | 1 | n |

**2)Deque**

Этот контейнер обычно называют дек и представляет он из себя список элементов, в которой добавление новых элементов и удаление существующих производится с обоих концов. Deqeu очень похож с vector, но главное отличие заключается в том, что элементы можно добавлять и удалять с обоих концов и что данные deqeu могут располагаться разорванными кусками, а данные vector размещаются в памяти последовательно.

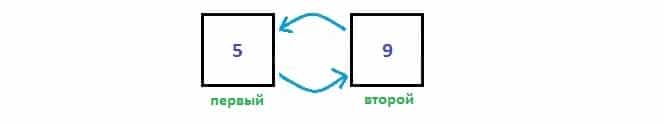
**3)List**

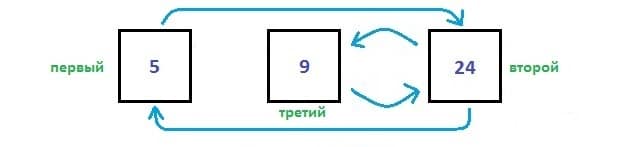
Это структура данных, которая построена на двусвязных списках. Это значит, что любой элемент знает только о предыдущем и о следующем элементах.

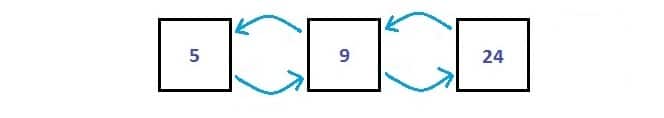
На картинке ниже показана, как это устроено:[](https://codelessons.ru/wp-content/uploads/%D0%91%D0%B5%D0%B7%D1%8B%D0%BC%D1%8F%D0%BD%D0%BD%D1%8B%D0%B9-4-1.png)

У двусвязного списка нет индексов, но вместо их в C++ есть итераторы.

Программисты используют этот контейнер из-за быстрого добавления и удаление значений. Это происходит так быстро, потому что не приходиться перемещать элементы между собой, нужно лишь правильно манипулировать указателями.





На примере выше в начале было два элемента, потом мы решили добавить один элемент между ними.



А так совершается удаление.

Главным отличием list от vector заключается во времени доступа к элементам. В vector доступ к элементам является произвольны, а в list последовательным. Вот пример программы, которая вычисляет время вставки элемента в вектор и в список:

#include <algorithm>

#include <iostream>

#include <list>

#include <random>

#include <sys/time.h>

using std::cout; using std::endl;

using std::list; using std::vector;

float time\_diff(struct timeval \*start, struct timeval \*end)

{

return (end->tv\_sec - start->tv\_sec) + 1e-6\*(end->tv\_usec - start->tv\_usec);

}

const int MIN = 1;

const int MAX = 100;

const int CAPASITY = 100000;

int main() {

struct timeval start{};

struct timeval end{};

std::random\_device rd;

std::mt19937 eng(rd());

std::uniform\_int\_distribution<int> distr(MIN, MAX);

vector<int> vec1;

list<int> list1;

vec1.reserve(CAPASITY);

for (int i = 0; i < CAPASITY; ++i) {

if (i % 39999 == 0) {

vec1.push\_back(111);

continue;

}

vec1.push\_back(distr(eng));

}

for (int i = 0; i < CAPASITY; ++i) {

if (i % 39999 == 0) {

list1.push\_back(111);

continue;

}

list1.push\_back(distr(eng));

}

auto iter = std::find(vec1.begin(), vec1.end(), 111);

gettimeofday(&start, nullptr);

vec1.insert(iter, 1111);

gettimeofday(&end, nullptr);

printf("insert vector: %0.8f sec\n", time\_diff(&start, &end));

auto iter2 = std::find(list1.begin(), list1.end(), 111);

gettimeofday(&start, nullptr);

list1.insert(iter2, 1111);

gettimeofday(&end, nullptr);

printf("insert list : %0.8f sec\n", time\_diff(&start, &end));

return EXIT\_SUCCESS;

}

Результат:

insert vector: 0.00053000 sec

insert list : 0.00000100 sec

**4)Forward\_list**

Класс **forward**\_**list** реализует односвязный список, элементы которого хранят указатель только на следующий элемент (в списках класса list элементы хранят указатели и на предыдущий, и на последующий элементы). Вставка и удаление элементов происходит быстро, достаточно изменить ссылку, но получить доступ к элементу списка по индексу нельзя, т. к. элементы могут быть расположены в разных местах памяти.

## Ассоциативные контейнеры

**1)Map**

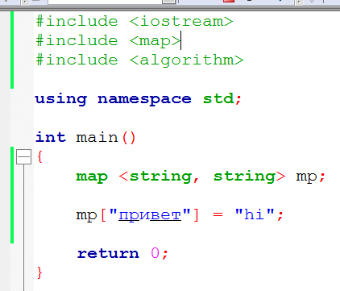
Это ассоциативный контейнер, который работает по принципу — [ключ — значение]. Он схож по своему применению с вектором и массивом, но есть некоторые различия:

1) Ключом может быть все что угодно. От обычной переменной до класса.

2) При добавлении нового элемента контейнер будет отсортирован по возрастанию.

Поэтому можно с легкостью сделать словарь:

* Ключом в нашем случае будет — русское слова.
* А значением — английское.



**2)Set**

Set – это множество, которое содержит несколько отсортированных элементов. При добавлении нового элемента в множество он сразу становится на свое место так, чтобы не нарушать порядка сортировки. Множества содержат только уникальные элементы, а мультимножества могут содержать дубликаты. Операции поиска, удаления и вставки имеют логарифмическую сложность.

С множествами мы работали уже в 13 лабораторной работе, так что я не думаю, что это стоит подробно пояснять.

## Контейнеры-адаптеры

## 1)Контейнер queue

**Очередь** — это структура данных (как было сказано выше), которая построена по принципу **LILO** (last in — last out: последним пришел — последним вышел). В C++ уже есть готовый STL контейнер — queue.

В очереди, если вы добавите элемент, который вошел самый первый, то он выйдет тоже самым первым. Получается, если вы добавите 4 элемента, то первый добавленный элемент выйдет первым.

Чтобы понять принцип работы очереди вы можете представить себе магазинную очередь. И вы стоите посреди нее, чтобы вы оказались напротив кассы, сначала понадобится всех впереди стоящих людей обслужить. А вот для последнего человека в очереди нужно, чтобы кассир обслужил всех людей кроме него самого.

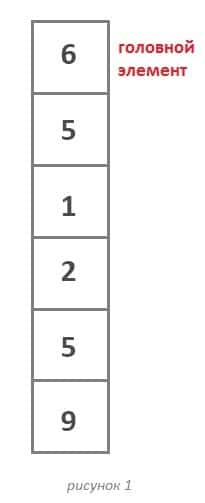
[](https://codelessons.ru/wp-content/uploads/%D0%91%D0%B5%D0%B7%D1%8B%D0%BC%D1%8F%D0%BD%D0%BD%D1%8B%D0%B9-2.jpg)На рисунке слева находятся 7 чисел: 2, 4, 7, 1, 4, 9, 10. Если нам понадобится их извлечь, то извлекать мы будем **в таком же порядке** как они находятся на рисунке!

Так например чтобы извлечь число 4 нам понадобится сначала обслужить число 2, а потом уже и само число 4.

Хотя в стеке присутствует функция peek(), шаблоне очереди невозможно обратится к **определенному элементу**.

**2)Контейнер stack**

Stack очень похож с **queue**, но принцип работы с ним немного другой. Стек — это структура данных, которая работает по принципу **FILO**(first in — last out; первый пришел — последний ушел). В C++ уже есть готовый шаблон — stack.

В стеке элемент, который вошел самый **первый** — выйдет самым **последним**. Получается, если вы добавили три элемента в стек первым будет удален последний добавленный элемент.  
[](https://codelessons.ru/wp-content/uploads/%D0%91%D0%B5%D0%B7%D1%8B%D0%BC%D1%8F%D0%BD%D0%BD%D1%8B%D0%B9-1.jpg)На рисунке 1 вы можете увидеть 6 чисел: 6, 5, 1, 2, 5, 9. Кстати извлекать их будем **в таком же порядке**. Например чтобы извлечь число 1 нам придется сначала извлечь числа 6 и 5, а потом уже 1. Так как это стек, эти числа мы добавляли в обратном порядке. Если быть точным вот так: 9, 5, 2, 1, 5, 6.

В стеке нет индексов как в массиве, а значит вы не можете обратиться к определенному элементу. Все потому что, стек построен на связных списках.

Это значит что каждый элемент (кроме последнего — он показывает на NULL, если простыми словами, то на ничего) имеет указатель на следующий элемент. Но есть элемент, на который нет указателя — первый (или как его еще называют *головной*).

Вы наверно спросите зачем использовать связные списки, если с таким же успехом можно было использовать простой массив. Тем более даже новичку не понадобится много времени чтобы разобраться в нем.

Но все достоинство шаблонного стека заключается в добавлении и удалении элементов. Эти операции происходят за **константное время** (это хороший плюс).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Container | insert in begin | insert in end | insert in | remove in begin | remove in end | remove | Index search | find |
| vector | n/a | O(1) | O(n) | O(1) | O(1) | O(n) | O(1) | O(log(n)) |
| list | O(1) | O(1) | O(1) | O(1) | O(1) | O(1) | n/a | O(n) |
| deqeu | O(1) | O(1) | n/a | O(1) | O(1) | O(1) | n/a | n/a |
| queue | n/a | O(1) | n/a | O(1) | n/a | n/a | O(1) | O(log(n)) |
| stack | O(1) | n/a | n/a | O(1) | n/a | n/a | n/a | n/a |
| map | n/a | n/a | O(log(n)) | n/a | n/a | O(log(n)) | O(1) | O(log(n)) |
| multimap | n/a | n/a | O(log(n)) | n/a | n/a | O(log(n)) | O(1) | O(log(n)) |
| set | n/a | n/a | O(log(n)) | n/a | n/a | O(log(n)) | O(1) | O(log(n)) |
| multiset | n/a | n/a | O(log(n)) | n/a | n/a | O(log(n)) | O(1) | O(log(n)) |

Теперь я приведу таблицу, которая сравнивает вставку удаление и поиск всех этих контейнеров:

## n/a – Not Available(недоступно)