**Конспект: Контейнеры STL**

В современных проектах, написанных на с++, очень часто появляются такие понятия как “контейнер”, “STL”, “Алгоритмическая сложность”, “Итератор” и многие другие страшные для новичков слова. Попробуем описать эти понятия на примерах:  
  
Начнем с того, **что такое STL?**

**S**tandard **T**emplate **L**ibrary - это набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Если сравнивать это с реальной жизнью можно привести в пример огромный гараж какого-то очень богатого человека, у которого есть велосипеды, машины, вертолеты, грузовики, лыжи и вообще все виды транспорта :)



Когда человеку нужно доставить **много груза и не важно сколько времени** это займет, он берет **грузовик** (если умеет им управлять)

Когда человеку нужно отправить **небольшую посылку в соседний дом** он может доехать на **машине** или **велосипеде**

Когда человеку нужно **доставить срочно, много, далеко и не важно сколько это отнимет ресурсов** он берет **самолет**

Библиотека STL это огромный гараж, только транспортом здесь служат контейнеры, которые были написаны для максимальной эффективности в конкретных задачах.

И тут встает важный вопрос: Разумно ли посылать самолет с маленькой посылкой в соседний дом? Или ехать на велосипеде, волоча за собой 100 кг груза, в другую страну?

Таким образом, контейнер - это тип (грубо говоря), который реализует в себе некоторую структуру хранения и обработки данных.

От программиста зависит правильность применения контейнеров в проектах. Конечно, в маленьких проектах будет незаметное падение скорости или засорение памяти, но как только проект станет больше – уже будет поздно что-то менять.

Итак, наиболее часто используются следующие контейнеры:

* **vector — коллекция элементов, сохраненных в массиве, изменяющегося по мере необходимости размера (обычно, увеличивающегося).** Все любят векторы, все их постоянно используют в проектах, и это действительно удобный контейнер. Им можно заменить обычный или динамический массив, и он даже лучше в плане добавления новых элементов в любое место. Но как дело доходит до скорости на форумах начинаются скандалы, интриги, расследования. Связано это с постоянным перевыделением памяти и копировании блоков с элементами в новый динамический массив (мы напрямую это увидеть не можем, так как это происходит автоматически, но мы знаем как это работает)



1. Создание вектора (vector)

#include <iostream>

#include <vector> //подключаем библиотеку для работы с векторами

using namespace std;

int main()

{

//vector <тип данных который хранит> имя вектора;

vector<int> v1;

v1.push\_back(5); //добавить в конец вектора

cout << "vector size: " << v1.size() << endl; //кол-во элементов

cout << "last element: " << v1.back() << endl; //последний элемент

//v1.pop\_back() //удаляет последний элемент вектора

v1.clear(); //удаление всех элементов из вектора

cout << "vector size: "<< v1.size() << endl;

system("pause");

return 0;

}

1. Вектор insert(), begin(), end()

#include <iostream>

#include <vector> //подключаем библиотеку для работы с векторами

using namespace std;

int main()

{

vector<int> v1{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

//begin() итератор, который указывает на начало вектора

//end() итератор, который указывает на конец вектора

//v1.insert работает только с итераторами

//первый параметр “куда вставляем”, второй параметр “что вставляем”

v1.insert(v1.begin(), 15); //вставка в 0-й элемент числа 15

v1.insert(v1.begin() + 3, 16); //вставка в 3-й элемент

v1.insert(v1.end(), 17); //вставка в последний элемент

v1.insert(v1.end() - 1, 18); //вставка в предпоследний элемент

//v1.insert(v1.begin() - 1, 15); //ошибка! мы не можем уходить в отрицательные индексы

//v1.insert(v1.end() + 1, 19); //ошибка! в таком случае лучше использовать push\_back()

for (auto elem : v1) { //range based for

cout << elem << endl;

}

for (auto p = v1.begin(); p != v1.end(); p++) { //как раскрывается range based for

auto elem = \*p;

cout << elem << endl;

}

for (int i=0; i < v1.size(); i++) { //как мы привыкли делать

cout << v1[i] << endl;

}

system("pause");

return 0;

}

1. Обращение к элементам вектора

#include <iostream>

#include <vector> //подключаем библиотеку для работы с векторами

using namespace std;

int main()

{

vector<int> v1{ 15, 2, -4, 56, 9};

cout << v1.at(0) << endl;

cout << v1[0] << endl;

//cout << v1.at(5) << endl; //выбрасывает исключение при выходе за границы

//cout << v1[5] << endl; //при выходе за границы может вызвать неопределенное поведение

//v1.begin() это итератор, который работает похожим образом на указатель массива

cout << \*v1.begin() << endl;

cout << \*(v1.begin() + 1) << endl;

cout << \*(v1.begin() + 2) << endl;

cout << \*(v1.begin() + 3) << endl;

cout << \*(v1.begin() + 4) << endl;

system("pause");

return 0;

}

1. Обращение к элементам вектора

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> v1(7); //выделили область в памяти на 7 целых чисел при создании вектора

for(auto elem : v1){

cout<<elem;

}

//на экране 7 нулей, так как по умолчанию такие элементы заполняются нулями

return 0;

}

О всех плюсах и минусах можно написать очень много “интересного текста”, но я напишу одно:

Используйте вектор когда не знаете сколько элементов в нём будет храниться и когда вам может понадобиться быстрый доступ по индексу. С этой задачей он справляется идеально и не нужно придумывать “велосипед” с постоянным выделением памяти под новые динамические массивы, ведь вектор в своей реализации уже все сделал за вас.

Если элементов действительно очень много, то вам как программисту достаточно указать приблизительный начальный размер вектора, чтобы не так сильно терять в скорости по сравнению с массивами.

* **stack, queue, deque — контейнеры, для хранения элементов без индексов и с разными видами доступа.** На картинках из жизни разберем как работает каждый из них:

stack представляет из себя детскую пирамидку, представим что каждое цветное колечко это элемент в нашем контейнере, но как либо взаимодействовать мы можем только с самым верхним (с тем кто пришел последний).   
Под взаимодействием имеется в виду, что мы можем изменять, убирать, добавлять элемент только сверху, элемент ниже для нас недоступен пока наверху есть другой.



Где применяются стеки, примеры:  
**История действий:** В текстовом редакторе, браузере, фотошопе можно использовать контейнер stack для хранения истории действий пользователя. Каждое действие (например, ввод текста или переход на новую веб-страницу) может быть сохранено в стеке. При необходимости можно отменить последнее действие, извлекая его из стека. Таким образом действия будут в обратном порядке возвращаться к исходному и мы избежим ошибок.

**Обработка вызовов функций:** Во время выполнения программы вызовы функций могут быть сохранены в стеке. Каждый раз, когда функция вызывается, информация о вызове помещается в стек. При завершении функции она извлекается из стека, и управление возвращается к предыдущей функции. В таком случае программа задает приоритет действиям, что также позволяет избежать ошибок.

В общем, контейнер stack полезен там, где необходимо сохранять и извлекать элементы в порядке "последний пришел - первый обслужен", подобно стеку объектов в реальной жизни. Он широко применяется в программировании и решает множество задач, связанных с управлением и организацией данных.

1. Создание и работа со стеком (stack)

#include<iostream>

#include<stack> //библиотека для стеков

using namespace std;

int main()

{

stack<int> s; //стек, который хранит целые числа

s.push(1); //первый элемент в стеке

s.push(2); //второй элемент в стеке

cout << s.top() << endl; //у нас есть доступ только к последнему элементу

s.pop(); //удаляем последний добавленный

cout << s.top() << endl;

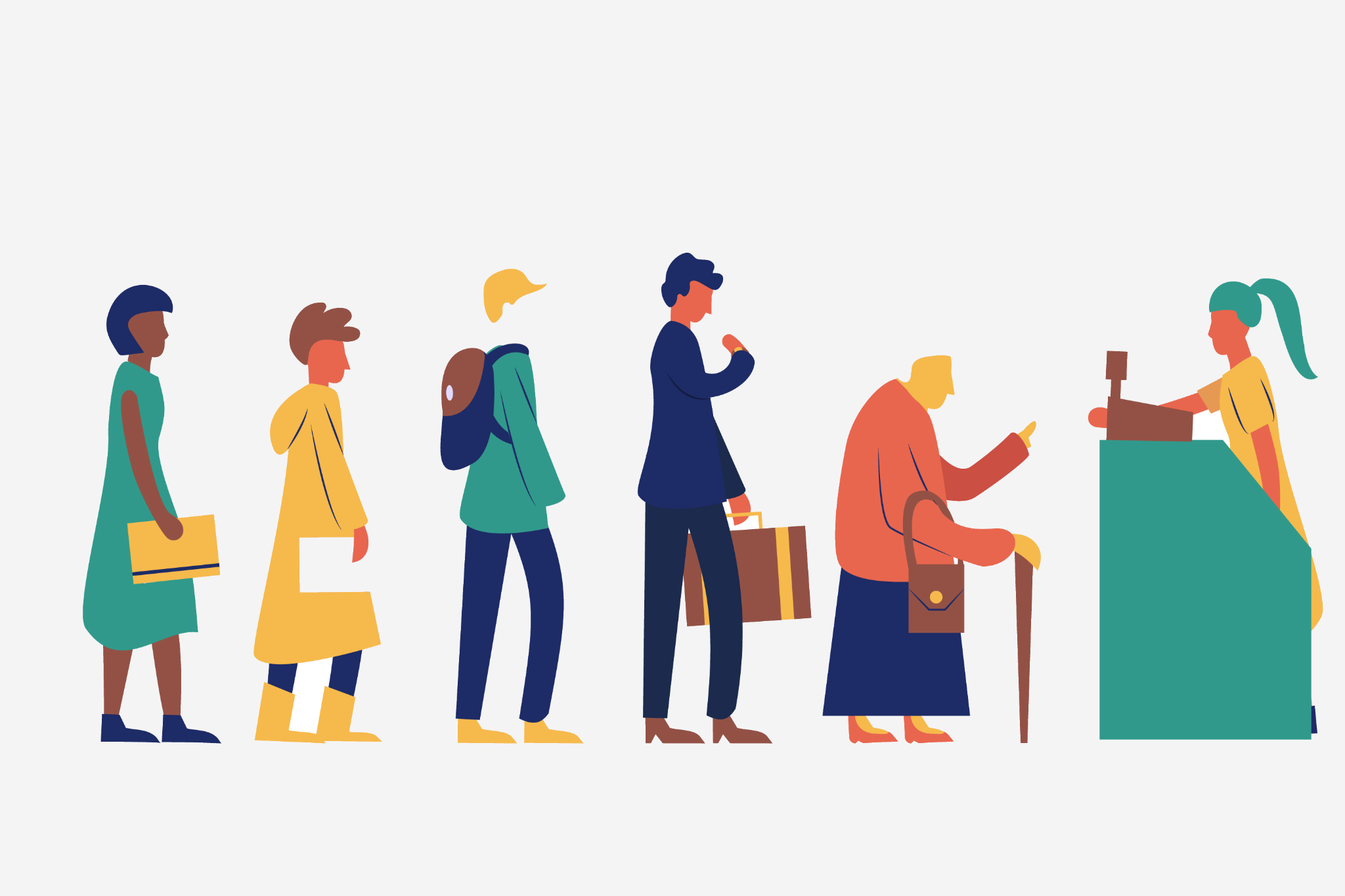
s.pop();

cout << boolalpha << s.empty() << endl; //пустой ли стек?

return 0;

}

**queue** или по другому очередь применяется в ситуациях обратных стеку. Нам теперь нужно обслужить элементы именно в том порядке в котором они пришли в контейнер. Как очередь в больнице: первым заходит к доктору тот, кто первый пришел и встал в очередь.



Очереди широко применяются в **системах заявок** например в службах поддержки, банках, игровых серверах. Пользователь занял свое место в очереди, но должен дождаться пока обслужат всех кто занял место раньше.

1. Создание очереди (queue)

#include<iostream>

#include<queue> //библиотека для очередей

using namespace std;

int main()

{

queue<int> q; //очередь, которая хранит целые числа

q.push(1); //первый элемент в очереди

q.push(2); //второй элемент в очереди

q.push(3); //третий элемент в очереди

cout << q.front() << endl; //у нас есть к первому элементу

cout << q.back() << endl; //и к последнему

//элементы в середине нам не получить пока не удалим эти элементы

q.pop(); // важно! С помощью pop() мы удаляем ПЕРВЫЙ

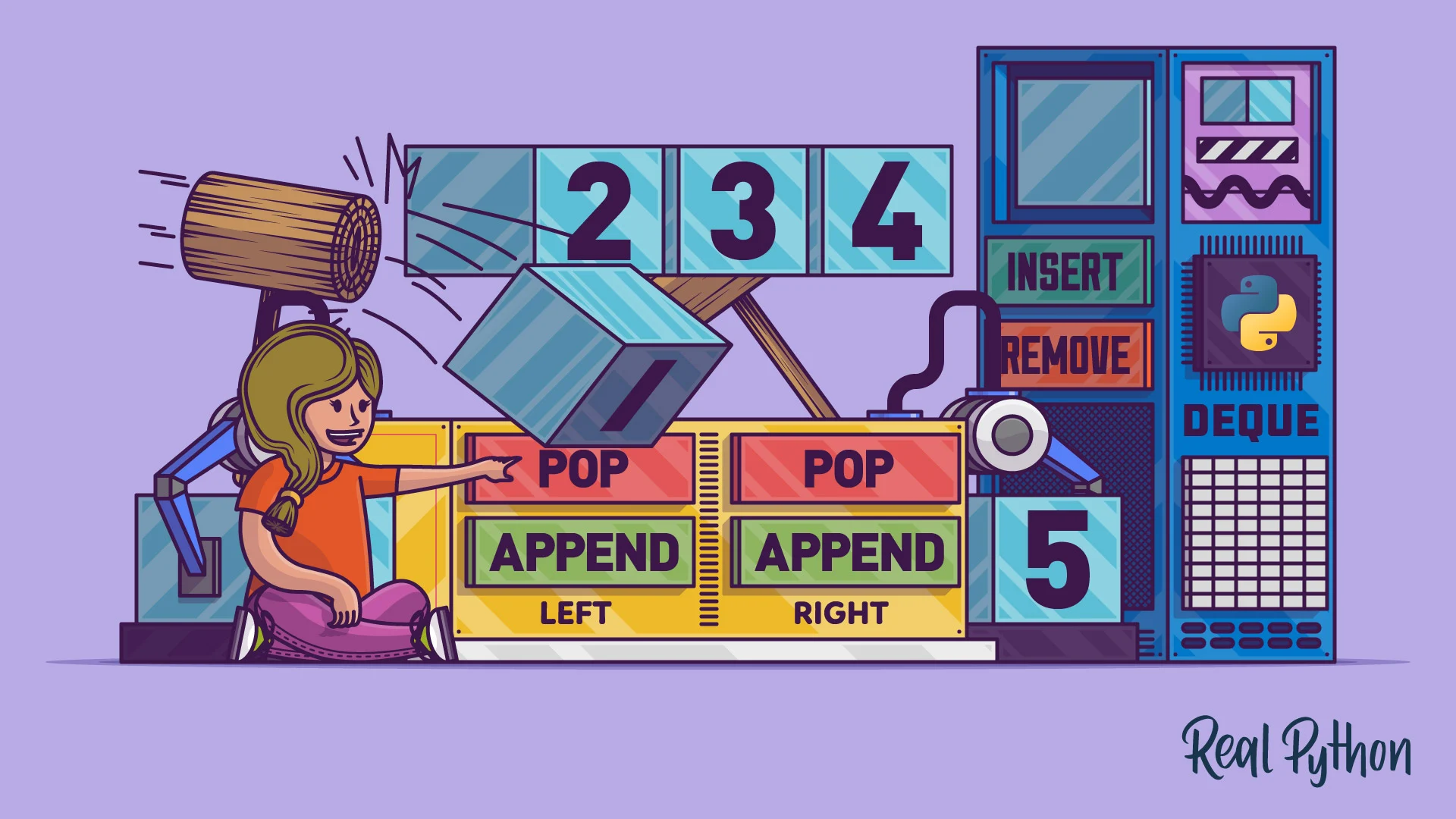
cout << q.front() << endl; //первый вошедший - вышел раньше из очереди

cout << boolalpha << q.empty() << endl; //пустая ли очередь?

return 0;

}

**deque** это и стек и очередь одновременно, но чаще все таки говорят что дек это двусторонняя очередь.



Наиболее яркие примеры:

**Кэширование**: deque может быть использован для реализации кэша данных. Новые или обновленные данные могут быть добавлены в deque, а при достижении предела размера кэша, старые или реже используемые данные могут быть автоматически удалены с одного или обоих концов deque.

**Очередь задач**: deque может использоваться для организации очереди задач в многопоточных системах. Новые задачи могут быть добавлены в конец deque, а рабочие потоки могут извлекать задачи из начала deque для выполнения.

1. Создание дека (deque)

#include<iostream>

#include<deque> //библиотека для деков

using namespace std;

int main()

{

deque<int> d; //дек, который хранит целые числа

d.push\_back(1); //первый элемент

d.push\_front(2); //второй элемент вставляем в начало

d.push\_back(3); //третий элемент вставляем в конец

cout << d.front() << endl; //у нас есть к первому элементу

cout << d.back() << endl; //и к последнему

//элементы в середине нам не получить пока не удалим эти элементы

d.pop\_back(); //удаляем последний

d.pop\_front(); //удаляем первый

cout << d.front() << endl; //остается то что было в середине

cout << boolalpha << d.empty() << endl; //пустой ли дек?

return 0;

}

*В зависимости от масштабов и потребностей вы можете использовать стеки, очереди и деки в своих проектах, но учитывайте то, что их* ***ограничения являются их главными плюсами****, во многих задачах важно оптимизировать память и эти контейнеры оптимизируют её как нельзя лучше.*