**Контейнеры, итераторы.**

STL – Standart Template Library. Контейнер – класс, предназначенный для хранения объектов. Последовательные контейнеры (sequence containers) :list, vector, string, wstring, deque.

В STL заголовочные файлы не имеют расширения: #include <vector>. Принято подключать сишные заголовочные файлы по-другому в С++: #include <cstdio>. Нельзя путать #include <cstring> (раньше было <string.h>) – набор функций для работы со строками. #include <string> - это строки STL. Не путать с <cstring>.

Vector – это динамически изменяющийся массив, размер может расти. В стандарте С++ сказано, что вектор – это непрерывная область памяти.

std::vector<int> v; // т.к. класс шаблонный, указываем тип хранящихся объектов. Как работает вектор? Есть массив, наполняем его числами, если добавляем больше, чем размер, происходит перевыделение (на сколько больше выделится – зависит от реализации, с каким-то запасом обычно в коэффициент между 1 и 2). Операция добавления в конец v.push\_back(5); Из знания такой организации можно уже сказать о сложности операций.

1) добавление / удаление в конец : O(1)\* - амортизированное О от единицы. v.size(); - то, насколько массив сейчас заполнен, v.capacity(); - сколько выделено сейчас памяти. Пока size < capacity добавление элемента занимает одну операцию, иначе надо перевыделять памяти и добавление занимает больше операций. Удаление всегда O(1). Сам по себе массив меньше не становится. v.pop\_back() – удаляет элемент с конца и уменьшает size на 1.

2) вставка / удаление в середину/начало : O(n) т.к. приходится сдвигать все n элементов (если вставляем в начало массива). Удаляя элемент тоже приходится смещать всё влево.

3) поиск элемента : O(n) – справедливо для всех последовательных контейнеров.

4) произвольный доступ : да, возможен. Перегружен оператор [], можно писать v[1]=6 (нет проверок выхода за границы), также есть v.at(1)=6 (здесь есть проверки на ошибку, работает медленнее, будет сгенерировано исключение при выходе за границу).

Хочется, конечно, избегать лишних перевыделений у вектора. В конструкторе можно указать размер : std::vector<int> v(10); - capacity=10, size=0. Есть конструктор v(10,0); - у каждого из 10 элементов будет инициализация нулём (если контейнер на вашем классе, то вместо 0 можно передать конкретное значение объекта). v.resize(100); - увеличивает size до 100 и если size превысит capacity, то произойдёт перевыделение. Если написано std::vector<MyArray> v; v.resize(100); сработает, если есть конструктор по умолчанию. Перегруженный метод v.resize(100,MyArray(10)); указываем, как должны быть проинициализированы элементы. Есть также метод v.reserve(100); - изменяет только capacity. Метод v.clear(); - устанавливает size в ноль. Как уменьшить размерность вектора?

В книге Скотта Мейерса указан способ swap trick:

vector<int>(v).swap(v); создаётся временный объект на основе v неименованный и они с v меняются местами, это вовсе не гарантирует, что памяти будет выделено ровно столько, сколько нужно, но, скорее всего, будет лучше, чем было. В новых версиях стандарта появился метод v.shrink\_to\_fit(); - запрос к контейнеру для уменьшения capacity до size, то есть выглядеть должно как-то так:

std::vector<int> v(100); .... v.resize(10); v.shrink\_to\_fit(); по идее метод выполняет для вас swap trick.

Двусвязный список “подмассивов” deque (double ended queue): std::deque<int> d; d.push\_back(5); d.push\_front(1); Благодаря такой организации в виде выделенных подмассивов фиксированного размера, а не как у вектора непрерывный участок памяти:

1) добавление в начало и в конец занимает O(1)\*, тоже амортизированное О от одного. В конец, как и у вектора, как только достигли границы, выделяется ещё какой-то кусок памяти и записываем туда. То же самое и в начало. Вообще двусвязные списки полезны, во многих алгоритмах надо и добавлять, и удалять как с головы, так и с конца. По стандарту deque, кроме того, занимает в памяти меньше места, чем вектор. Лишняя память используется только на подмассивы в голове и в хвосте (незаполненные участки этих подмассивов). Грубо говоря deque это вектор, использующий меньше памяти, есть указатели на голову и хвост. Однако обращение по индексу (произвольный доступ) происходит гораздо дольше.

2) вставка / удаление в середине – O(n) т.к. надо подвинуть элементы внутри подмассивов

3) поиск O(n)

4) Произвольный доступ есть d[9], d.at(9) – но эффективность хуже чем у вектора. Надо помнить, что с непрерывной областью памяти можно много штук интересных делать, копировать, кусок подставить и т.д.

String и wstring. В принципе очень похоже на вектор. На самом деле в STL есть шаблонный класс basic\_string. String – это typedef basic\_string<char> string; typedef basic\_string<wchar\_t> wstring; Вообще wchar\_t – это длинные символы, которые могут не по байту занимать. В Linux это 32 бита (можно юникод представлять в таких wchar\_t). Кроме похожести на вектор с символами, есть ещё методы специальные, облегчающие работу со строками. #include <string>

std::string s(“Hello world”); s+=”!”; Конкатенация строк (сложение). Можно складывать строки со строками, можно складывать строки C++ со строками Си, со строковыми литералами. Как внутри устроен string? Внутри возможно используется принцип COW copy on write (зависит от реализации). std::string s1=s; Нечто похожее на работу с умными указателями. s1 и s в памяти будут указывать на одну и ту же строку. Копирование будет производиться только в том случае, когда происходит позже модификация и при этом есть несколько хозяев (указывающих на эту строку). Это может быть реализовано для экономии памяти и эффективности операций.

std::string s1=s; Если бы это были вектора, то производилось бы копирование, со строками не так – просто два хозяина для строки, пока они не изменяют строку, не нужно создавать реальную копию. Если теперь s1[0]=’!’; то уже нужно делать реальную копию. Также для string есть операторы == , < , >, есть методы аналогичные сишным строкам: find(), substr(). Если нужно работать одновременно и с Си строками и с С++ есть полезная функция:

const char \* c\_str() // s1.c\_str(); // Т.к. это указатель на константу то менять строку по нему нельзя. Возвращается указатель на Си-строку, заканчивающуюся нулём, можно передать это printf(“%s”,s1.c\_str()); Ещё нюанс про string: пусть есть функция void f(std::string s) { printf(“%s”,s.c\_str()); } и её вызов f(“Hello”); В этом случае произойдёт создание временного безымянного объекта с помощью вызова конструктора string, потом по значению в функцию передастся такой объект. Как этого избежать? Если написать void f(std::string& s) то не скомпилируется, т.к. временный объект менять бессмысленно, надо тогда делать void f (const std::string& s);

Внутри basic\_string реализовано много алгоритмов в привычном виде индексов (в STL общие алгоритмы в виде итераторов все же представлены): find там и прочее.

Как преобразовывать все эти строки между собой? Здесь помогут конструкторы перегруженные, а также c\_str.

Двусвязный список List. Рисунок. 1) вставка в конец/начало O(1) 2) вставка / удаление в середину O(1) 3) Поиск O(n) 4) Произвольного доступа нет. #include <list>

std::list<int> l; l.push\_back(5); l.resize(10); всё очень похоже на предыдущие контейнеры. Как вставлять / удалять из середины? По вектору и деке можно бегать указателями, в списке такого принципиально нету. Нужно пользоваться итераторами.

Примеры использования:

Итераторы – это класс для доступа к контейнерам, синтаксически похожий на указатели. Для него определены операции ++, --, ->, \*. Цикл прохода по списку:

std::list<int> l; std::list<int>::iterator it; // мы уже это с вами видели, это вложенный класс.

for(it=l.begin(); it!=l.end(); ++it)

printf(“%d ”,\*it);

Рисунок. Begin() возвращает итератор на начало списка, на нулевой элемент, end() возвращает итератор на место за последним элементом списка. Внутри операции ++ реализован переход по ссылкам типа next. Результат перегруженной операции разыменования будет значение элемента, в нашем случае int. Почему именно ++it? В реализации it++ на одно копирование больше, кто его знает насколько эффективно там класс реализован. Принятая практика использовать префиксный инкремент. Постфиксный как правило реализован через префиксный.

it++ () {

tmp=curr;

++curr;

return tmp;

}

Альтернативные способы пробегаться по контейнерам:

Range-based for (цикл фор на основе диапазона значений):

List <string> l = {“One”, “Two”}; // std initializer list – список инициализации

for(string& s : l) { cout <<s; }

Точно также по другим контейнерам

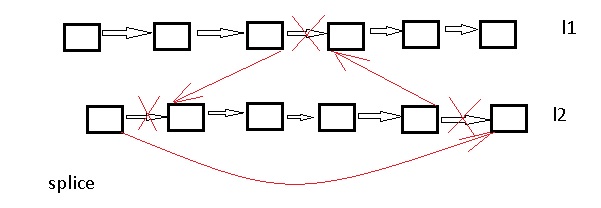
vector <int> v = {0,1,2,3};

for(auto& i : v) { cout <<i; } // либо int& i

Для более сложных вещей все же нужны итераторы !

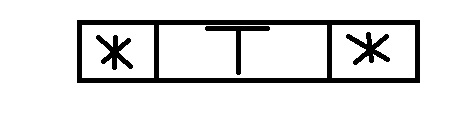
Бывают ещё другие итераторы. std::list<int>::const\_iterator it; При попытке (\*it)=...; будет ошибка компиляции. Как вставлять и удалять элементы? l.insert(it, 5);

!!! Смаль: в стандарте про List сказано либо size() за O(1) либо splice за O(1):



Почему так? Есть разные реализации, возможно в каких-то задачах очень часто нужен splice(), но чаще нужен size() всё же и скорее всего вы будете попадать на реализацию size() за O(1).

Ещё забавный факт: в памяти нарисовать добавление элемента типа T в list:



Про перегрузку оператора new для типа T: допустим вы его перегрузили, а на самом деле он может не вызываться вообще, т.к. там будет некая новая структура, которая внутри себя будет содержать T и как минимум два указателя.

Итератор помечает место для вставки/удаления. l.erase(it); После выполнения некоторых операций итератор указывает туда, куда не должен. Такой итератор называется инвалидированным (неправильным). Когда это происходит? Когда происходит инвалидация итераторов?

Был вектор, были итераторы указывающие куда-то. После этого можно сделать push\_back или shrink\_to\_fit после resize. По идее некоторые итераторы могут инвалидироваться. Либо при перевыделении памяти (при добавлении элементов и изменении capacity). Т.к. выделяется новый непрерывный кусок памяти где-то, то итераторы уже указывают совсем не туда, куда нужно.

Был list. Инвалидации итераторов происходит при удалении, например. Именно поэтому erase возвращает новый правильный итератор. Пример:

auto seed = chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

mt19937 mt\_rand(seed);

std::list<int> l;

int count = 10;

while (count)

{

l.push\_back(mt\_rand() % 10);

count--;

}

for (auto& i : l) { cout << i << " "; } cout << endl;

std::list<int>::iterator it = l.begin();

while (it != l.end()) {

if ((\*it) % 2 == 0)

l.erase(it); //it=

else ++it;

}

for (auto& i : l) { cout << i << " "; } cout << endl;

Такая программа скорее всего упадёт. После erase итератор не понятно на что указывает, и упадёт после ++it в следующем else. Если сделать it=l.erase(it); то все будет ок, запишется итератор на следующий элемент. Insert возвращает итератор на вставленный элемент.

Примеры использования list.

**Адаптеры и псевдо-контейнеры.**

Адаптеры – это такие обертки над контейнерами, сами по себе не имеют внутри себя хранилища. Они параметризуются тем хранилищем, который будут использовать. В них внутри логики хранения нету, они используют внутри себя тот тип (контейнер), который вы передаёте.

1.stack 2.queue 3.priority\_queue

По умолчанию stack и queue используют контейнер deque, а priority\_queue использует vector. Но никто не запрещает вам создать stack на списке, если из каких-то соображений вы считаете, что это будет эффективно. Ну или очередь построить на list, а не на vector.

stack<int,vector<int>> numbers;

int value = 99;

while (value)

{

numbers.push(value % 2);

value /= 2;

}

while (numbers.empty()==false)

{

cout << numbers.top();

numbers.pop();

}

cout << endl;

Чем вообще deque лучше, чем список? Возможность обращения по индексу – да, но допустим мы не хотим обращаться по индексу, но в каких случаях все равно deque лучше list? Если храним небольшие объекты, например инты, то меньше памяти тратится при хранении в deque, и соотношение (кпд) контейнера дека получится больше (ну или накладных расходов меньше), чем контейнера list (ведь там два указателя ещё рядышком).

Адаптеры ограничивают методы контейнера, который внутри них вшит. Например, в стеке, построенном на deque уже нельзя пробежаться по всем элементам, стек выдаёт только то, что на вершине. А зачем вообще это нужно? ОЧЕНЬ Много алгоритмов естественным образом реализуются в терминах стеков, очередей, приоритетных очередей. Это некое самоограничение программистов от самих себя, чтобы не сделать лишних неправильных вещей.

Можно передать свой контейнер для создания стека.

A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided Compare can be supplied to change the ordering, e.g. using std::greater<T> would cause the smallest element to appear as the top().

Working with a priority\_queue is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

**Псевдоконтейнеры** 1.vector<bool> // ненастоящий контейнер, то есть не хранит реально байты

для обращения здесь [] просто не подойдёт, используются proxy – объекты

Вы можете на один бит (bool 1/0) тратить либо 8 битов (unsigned char например использовать внутри контейнера) либо реально 1 бит (запаковывать как-то) и тут уж сами думайте, насколько это важно (КПД 100% или КПД 12.5%). Реальный пример из жизни. Перебор графов в памяти (на HDD долго сбрасывать и прочее) в качестве курсовой работы, один граф занимает порядка 4 гигабайт. В результате правильной упаковки – уже 500 мегабайт. Это позволит уже проводить модификации графа, ещё лишний раз проверить его на нужное нам свойство (переборная задача) и не свопая ничего на диск, продолжить дальше. В общем не стоит думать что память сейчас резиновая, она никогда не была резиновая, и забывать (или не знать) механизмы эффективной работы с памятью – слишком дорогое удовольствие.

2.std::bitset <size\_t s> служит для хранения битовых масок. Похож на vector<bool> фиксированного размера. Это не контейнер, не хранит никакие типы, шаблон, который параметризуется количеством битов. Удобно можно доставать биты, записывать биты. Есть операторы с этими масками, and or и т.д. очень удобно. Пример перевода числа из 10 в 2:

int j = -1;

cout << j << endl;

bitset<32> bits(j);

cout << bits << endl;

bitset<32> bits2(bits);

bits2[0] = 0;

bits2[31] = 0;

cout << bits2 << endl;

bits ^= bits2;

cout << bits << endl;

3.valarray – шаблон для хранения числовых массивов, оптимизированный для достижения повышенной вычислительной производительности. Например на компиляторе intel этот шаблон может быть специализирован так, что используются SSE, многопоточность и прочее. Если перемножать два valarray то возможно будет эффективнее.

Можно сказать, что в текущее время не особо полезная штука, т.к. есть более специализированные библиотеки и подходы.

Про vector ещё: как на место десятого элемента вставить новый? erase(v.begin()+10), insert(v.begin()+10,5), итераторы ведут себя как указатели.

**Ассоциативные контейнеры**: set, multiset, map, multimap.

1.std::set<int> s; s.insert(10); s.insert(20); s.insert(10); s.size()==2;

Элементы в множестве могут встречаться только один раз. Второй десятки не будет, set будет состоять из {10,20}. Как устроен внутри set? Это бинарное дерево поиска. Рисунок бинарного дерева. Если тебе нужно найти элемент, в каждой вершине начиная с корня дерева надо понять, куда тебе идти – влево или вправо. Слева допустим меньше , справа больше значения в вершине дерева. Это даёт быстрый поиск O(logn). Если делать insert бездумно, то оно может расти вниз неправильно и поиск будет O(n), вставлять нужно правильно. Сбалансированные деревья получаются таким образом. Set хранит внутри себя элементы в отсортированном виде. Нам понадобится вспомогательный класс STL:

Примеры операций взять с C:\\_main\\_univer\Documents\Chelgu\PP\Prog\Cpp\Get\Александр Смаль\часть 2 - 2017\Ассоциативные контейнеры и итераторы.pdf

std::pair<T,V> T first V second // Внутри два параметра разных типов, ключ+значение. Insert возвращает пару:

std::pair<iterator, bool> если элемент был вставлен (такого не было ещё в множестве), тогда bool==true а итератор указывает на вставленный элемент, если элемент уже имелся, то bool==false, а итератор указывает на тот элемент, который уже есть в множестве. Какие есть полезные методы?

iterator find(const T& obj) // например:

std::set::iterator it=s.find(10);

if(it==s.end())

printf(“not found”);

Предположим, элемент нашелся и я хочу сделать \*it=15; Так делать нельзя, т.к. структура множества сложная, это дерево. Если просто в узле поменять значение, нарушится структура и нельзя будет эффективно искать. Поэтому возвращаемый итератор константный у метода find. Можно только удалять + вставлять новые элементы. iterator lower\_bound (const T& obj); // возвращает итератор на элемент с условием >=obj, iterator upper\_bound (const T& obj); // возвращает итератор на первый элемент, который >obj. Find работает и для multiset (там повторяющиеся элементы, там тоже дерево). Если для multiset сделать lower\_bound (10) и upper\_bound (10) я получу два итератора на начало и конец участка, где лежат 10. То есть если в multiset 10 10 10 20, то lower вернёт итератор указывающей перед первым, upper вернёт итератор указывающий на 20. Вся прелесть этих операций в том, что они выполняются за логарифм. С помощью двух этих функций можно реализовать поиск диапазона / поиск объектов, находящихся в диапазоне. Очень эффективно. Чем мы заплатили за такое счастье? Добавление элементов теперь тоже занимает логарифм, у последовательных контейнеров это было O(1).

lower\_bound

Returns an iterator pointing to the first element in the range [first, last) that is *not less* than (i.e. greater or equal to) value, or last if no such element is found.

The range [first, last) must be partitioned with respect to the expression element < value or comp(element, value), i.e., all elements for which the expression is true must precede all elements for which the expression is false. A fully-sorted range meets this criterion.

The first version uses operator< to compare the elements, the second version uses the given comparison function comp.

upper\_bound

Returns an iterator pointing to the first element in the range [first, last) that is *greater* than value, or last if no such element is found.

Примеры использования map multimap set multiset: C:\\_main\\_univer\Projects\C++\КОД ДЛЯ ЛЕКЦИЙ\map examples

https://en.cppreference.com/w/cpp/iterator/iterator\_traits

Функторы, предикаты.

Если хочется свои объекты класть в Set, использовать в качестве ключа в Map для них нужно перегрузить оператор<. Реализация:

bool Person::operator<(const Person& p) {

return age<p.age;

}

Таким образом внутри Set объекты Person будут отсортированы автоматически по возрасту, внутри map<Person,int> тоже по возрасту. Вдруг в другом месте программы понадобится сортировать не по возрасту, а по имени.

Способ № 2. Функторы – это классы, с перегруженным operator().

Как показывает практика, если большая программа состоит из большого количества маленьких и максимально независимых друг от друга кусочков — тем легче ее развивать, чинить и менять. Современные объектно-ориентированные языки программирования в качестве кусочка предлагают нам объекты — собственно, отсюда и название. Объектом как правило является экземпляр класса, который делает что-то нужно и полезное. Можно всю программу сделать из одного ба-а-а-альшого класса — 'god object antipattern' — и через пару лет внесение любого изменения превратится в проклятье шестого уровня. А можно разбить программу на мелкие, максимально независящие друг от друга классы, и тогда всем будет сухо и комфортно. Но разбив программу таким образом возникает второй вопрос — а как эти классы будут взаимодействовать между собой? Тут на помощь программисту приходят средства декомпозиции, предоставляемые языком программирования. Самое простое средство декомпозиции в C++ — это сделать классы глобальными и непосредственно вызывать их методы. Подход не лишен недостатков — понять кто кого вызывает по прошествии времени становится все труднее, и через пару лет такой подход приведет к тем же последствиям, что и использование единственного класса. Одобренный святой инквизицией вариант — это передача классам указателей на те классы, с которыми им надобно общаться. Причем желательно, чтобы указатели были не просты, а на интерфейсы — тогда менять и развивать программу по прошествии времени станет намного проще.

Тем не менее, интерфейсы не являются серебряной пулей (некоторые говорят, что такой пули, как и ложки — вообще нет). Если объекту нужно всего несколько взаимодействий, например кнопке уведомить о том, что на нее кликнули — то реализация для этих целей отдельного интерфейса займет ощутимое количество строк кода. Также интерфейсы не решают задачу когда одному объекту нужно уведомить о чем-то несколько других — создание и поддержание листа подписки на основании интерфейсов тоже не самое малое число строк кода.

В динамических языках программирования конкуренцию интерфейсам составляют делегаты. Как правило, делегат очень похож на «указатель на функцию» в C++, с той основной разницей, что делегат может указывать на метод произвольного объекта.

Если operator() возвращает bool, то такой функтор называется предикатом. Это такой жесткий способ добавить в С++ указатели на функции. Пример, напишем функтор, который позволит нам сортировать по имени объекты Person:

struct by\_name {

bool operator()(const Person& a, const Person& b) {

return (a.name < b.name); }

};

Параметрами, которые берёт такой перегруженный оператор будут являться объекты класса Person. Фактически записали одну функцию внутри объекта. Будем передавать туда, где нужна функция сравнения ссылку на этот объект. Создаем объект: by\_name cmp; Теперь в

Person p1(“Ivanov”,20); Person p2(“Petrov”,40);

printf( “%d”, cmp(p1,p2) ); // вызываем у объекта перегруженный оператор(), то есть объектом пользуемся типа как функцией, синтаксис похож. На экран выведется 1, т.к. ‘I’ < ‘P’.

У шаблонов могут быть аргументы по умолчанию, у map на самом деле три аргумента.

В заголовочном файле написано что-то типа std::map<K,V,std::less<K> >; где less – это стандартный функтор в STL, который написан примерно следующим образом, он сравнивает по ключу:

template <class K> struct less {

bool operator() (const K& a, const K& b) {

return a<b;

}

};

На самом деле функтор по умолчанию в STL всегда использует оператор< для класса, которые сравниваются. Не самая очевидная вещь. Таким образом, когда мы пишем map<Person,int> компилятор видит:

std::map<Person,int,std::less<Person>> m; Можно использовать наш функтор для сравнения по умолчанию!

map<Person, int, by\_name> m1;

Вообще идеологически вещь простая (использование указателей на функции). Но т.к. в шаблонах надо использовать типы объектов, пришлось создавать именно объект, так и появились именно классы функторов. Остались ещё объекты, построенные поверх контейнеров : стек, ...

**АЛГОРИТМЫ.**

STL. Алгоритмы. Итератор – класс имитирующий правила работы с указателем. Обычно есть операторы \*, ->. Ещё у итератора может быть [], ++, -- тогда он называется RA (random access) итератор. Есть BiDi (BiDirectional) ++, --. Есть однонаправленные итераторы (только для чтения или только для записи потока). RA итераторы у vector, deque контейнеров. Алгоритмы зависят от итераторов. Пример того, почему важно понимать, с каким итератором вы работаете:

size\_t std::distance(It I, It j); // обычно итератор указывает на положение элемента внутри контейнера, функция distance вычисляет расстояние. Для итератора RA сложность такого алгоритма будет O(1). Для итератора BiDi сложность будет O(n). Ещё необходимо помнить, при работе с итераторами, что begin() указывает на начало контейнера, а end() указывает на место после последнего элемента контейнера. Если бы end() указывал на последний, то сложно было бы различить ситуацию, когда вообще нет элементов и есть один элемент в контейнере, надо бы было заводить спец. переменную и прочее.

Бывает iterator, бывает const\_iterator, бывает reverse\_iterator:

l.rbegin() --- l.rend() итератор, который при ++(увеличении) идёт с конца в начало контейнера. В принципе можно обычным итератором встать в конец и делать --.

Вспомним ещё про функторы и предикаты. Функтор – класс у которого перегружен оператор(). Если возвращает bool, то это предикат.

class compare {

int operator()(int a, int b) { return (a-b); } }; Раньше мы использовали предикат bool less(a,b), если a<b то true. Некоторые контейнеры хранят упорядоченно значения. Поэтому им нужно уметь сравнивать. У map например есть три параметра, третий может быть функтором по умолчанию: map<K,V,F> m; Можно сделать свой функтор для сравнения элементов в map, F f; f(3,5); Это чем-то похоже на указатель на функцию, но поскольку в шаблонах нужен тип, сделали функторы. Такой класс – типа функция (больше ничего в нём полезного нет).

#include <algorithms> // всего там порядка 100 штук алгоритмов, посмотреть подробно можно на cplusplus.com например

1)Микроалгоритмы: swap(T &a, T &b), iter\_swap(It p, It q) – меняет местами два элемента, на которые указывают два итератора, T& max(const T& a, const T& b), T& min(const T& a, const T& b).

2)Немодифицирующие алгоритмы: find(It p, It q, const T& value) – между итераторами p и q ищется элемент value, возвращаемое значение сравнивается с end(), если не равно end() то нашёлся. find\_if(It p, It q, Pred pr) – если в find мы искали совпадающий элемент с value, то в find\_if мы ищем согласно предикату, то есть о равенстве/неравенстве говорит предикат. Пример предиката:

class A {

bool operator()(int a) {

return a%2; } };

find\_if(i, j, A()); // используем неименованный объект класса A – предикат.

Ещё алгоритм count(It p, It q, const T& value) и аналогичный count\_if – подсчет количества вхождений.

bool equal(It p, It q, Itr i) // вернёт true / false, если равны / не равны элементы в контейнерах

pair<It, Itr> mismatch(It p, It q, Itr i) // вернёт итераторы на первое различие в 1 и 2 контейнере

Пусть есть два контейнера или две пары итераторов, нужно посмотреть одинаковые ли внутри них элементы. Например, есть вектор и список, их подмножества надо проверить на равенство. Параметра три, а не четыре, т.к. длина должна быть одинаковая.

F& for\_each (It p, It q, F func); // к каждому элементу контейнера применяется функтор, можно в функторе завести переменную и в неё накапливать.

bool binary\_search(It p, It q, const T& value); // предполагается, что контейнер отсортирован, тогда в нём будет сделан двоичный поиск, который вернёт true, если такой элемент в контейнере есть.

3)Модифицирующие контейнер алгоритмы: fill(It p, It q, const T& value); // заполнить контейнер между p и q значениями элементов value, в Си был аналог memset().

copy(It p, It q, It out); // Есть один контейнер между его p и q итераторами всё копируется в другой/этот же контейнер начиная с out.

generate(It p, It q, F gen); // похож на fill, но здесь функтор будет задавать значение. Например функтор может генерировать случайные числа.

random\_shuffle(It p, It q); // случайным образом переставляются элементы между p и q.

reverse(It p, It q); // изменить порядок элементов на обратный.

sort(It p, It q); // сортировка quicksort, есть версии с предикатом.

Есть стандартные шаблонные функторы: less, greater шаблонные. Отсортировать вектор по убыванию:

sort(v.degin(), v.end(), greater<int>());

**ПОКАЗАТЬ ТАБЛИЧКУ ВЫБОР АЛГОРИТМА STl**