STL

Standard Template Library

Шаблоны C++

- Шаблонные функции и классы позволяют писать обобщенный код без привязки к типам данных

```
int sum(int x, int y){
  return x + y;
}

double sum(double x, double y){
  return x + y;
}
```

```
template<typename T> //Объявление шаблона
T sum(T x, T y){
  return x + y;
}

int main(){
  std::cout << sum(2,5) << '\n';
  std::cout << sum(2,5.0) << '\n'; //Ошибка
  std::cout << sum<double>(2,5.0) << '\n';
  return 0;
}
```

Шаблонные классы/структуры

Каждая комбинация параметров шаблона создает свой отдельный тип данных

```
template<typename T, typename R>
struct MyPair
{
  T first;
  R second;
};
int main(){
  MyPair<char, int> pair1 = {'A',1};
  MyPair<std::string, int> pair2 = {"Hello",2};
  //Выведет 0 (false), т.к. разные тип данных
  std::cout << (typeid(pair1) == typeid(pair2)) << '\n';
  return 0;
}
```

STL

- STL - Стандартная библиотека шаблонов. Содержит набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.
- Состоит из компонентов:
  - Контейнеры
  - Итераторы
  - Адаптеры
  - Алгоритмы
  - Функциональный объект

Контейнеры

- Позволяют хранить данные в определенном виде
- Автоматически выделяют и удаляют память
- Предоставляют операции по добавлению, удалению элементов
- Можно работать через *итераторы*
- Имеются следующие контейнеры находящиеся в соответствующих модулях:
  - **std::array<T, N>** - массив фиксированный длины N типа T
  - **std::vector<T>** - массив переменной длины типа T
  - **std::list<T>** - список типа T. От std::vector отличается видом доступа к данным
  - **std::map<Key, T>** - словарь с ключом тип Key и значением типа T
  - **std::set<T>** - множество типа T. Все элементы имеют уникальные значения

std::array

std::array<T, N> - сочетает в себе производительность массивов в С-стиле и преимущества контейнеров STL.

```
//массив символов размером 4
std::array<char,4> arr = {'A','b','C','!'}; //Можно сразу инициализировать
arr[1] = 'B'; //Доступ к элементу по индексу, проверки границ нет
arr.at(3) = 'D'; //Доступ к элементу по индексу с проверкой границы
//методы front и back позволяют обратиться к первому и последнему элементу
std::cout << arr.front() << ' ' << arr.back() << '\n';
std::array<char,4> arr_2 = {'Z','B','C','D'};
//Для массивов определены операции сравнения
//В лексикографическом порядке
std::cout << (arr < arr_2) << '\n';
//С С++17 есть дедукция параметров шаблона
std::array arr_deducted = {0.5, 2.5, 5.1};
//Метод size возвращает размер массива
std::cout << arr_deducted.size() << '\n';
```

std::vector

std::vector<T> - массив, размер которого может изменяться.

Доступ к элементам O(1), вставка и удаление из конца O(1), вставка и удаление O(n).

```
//Тип хранимого элемента можно вывести
std::vector<int> vec = {1,2,3,4,5};
//Доступ к элементам аналогичен std::array
std::cout << vec.front() << vec.back() << vec[1] << vec.at(2) << '\n';
//push_back - добавить в конец
vec.push_back(42);
//size - длина массива, capacity - под сколько элементов выделена память
std::cout << vec.size() << ' ' << vec.capacity() << '\n';
//pop_back - удалить из конца
vec.pop_back();
vec.clear(); //очистка вектора, выделенная память остается
vec.reserve(20); //изменить количество выделенное памяти
vec.resize(10); //изменить размер массива
```

Итераторы

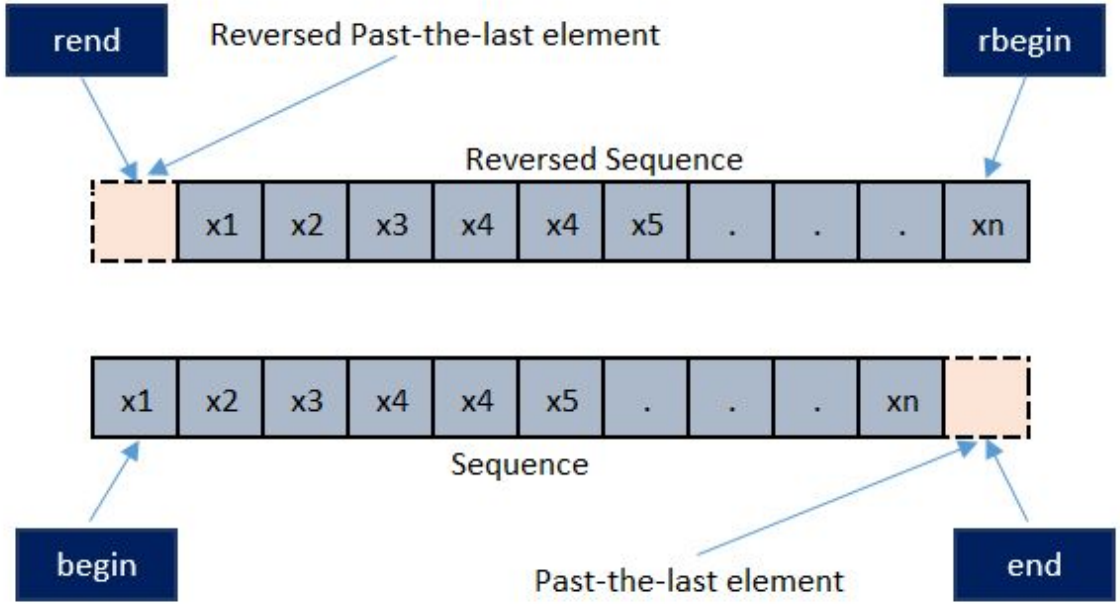
Итератор - поведенческий паттерн GOF. Предоставляет унифицированный подход к обходу коллекций, не раскрывая их внутренней структуры.

В C++ почти у всех контейнеров определены методы для работы с итераторами:

- **begin**() - возвращает итератор на начало
- **end**() - возвращает итератор на элемент после последнего
- **rbegin**() - возвращает итератор на конец
- **rend**() - возвращает итератор на элемент перед первым
- **cbegin**() и **cend**(), **crbegin**(), **crend**() - константные аналоги итераторов (нельзя менять элементы на которые они указывают)

Итераторы (2)

Итераторы позволяют перемещаться от начала (begin) к концу (end) при помощи оператора инкремента ++, и в некоторых случаях оператора декремента --



Категории итераторов в C++

- **input_iterator_tag** – разыменуется как rvalue
- **output_iterator_tag** – разыменуется как lvalue
- **forward_iterator_tag** – проход только в одну сторону
- **bidirectional_iterator_tag** – проход в оба направления
- **random_access_iterator_tag** – можно переходить на произвольное кол-во элементов вперед и назад
- **contiguous_iterator_tag** (C++20) – тоже самое, что и **random_access**, но элементы в памяти расположены последовательно

Свойства категорий итераторов

<table>
<tr><th colspan="4">Категория</th><th>Характеристика</th><th>Выражения</th></tr>
<tr><td colspan="4" rowspan="3">Все категории</td><td>Construct/Copy/Move/Delete</td><td>X b(a), b = a</td></tr>
<tr><td>Увеличение на 1</td><td>++a, a++</td></tr>
<tr><td>Разыменовывание</td><td>*a</td></tr>
<tr><td rowspan="8">Random</td><td rowspan="5">Bidirectional</td><td rowspan="3">Forward</td><td rowspan="2">Input</td><td>Сравнение на равенство</td><td>a == b, a != b</td></tr>
<tr><td>Разыменование как rvalue</td><td>*a, a->m</td></tr>
<tr><td>Output</td><td>Разыменование как lvalue</td><td>*a = o</td></tr>
<tr><td colspan="2" rowspan="4"></td><td>Множественный доступ</td><td>X a(b); ++a == ++b</td></tr>
<tr><td>Уменьшение на 1</td><td>--a, a--</td></tr>
<tr><td rowspan="3">Можно использовать + и -</td><td rowspan="3">a + n, a -= n</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2"></td><td>Разыменование по индексу</td><td>a[n] == *(a + n)</td></tr>
<tr><td colspan="2"></td><td>Сравнение больше/меньше</td><td>a < b, a > b, a <= b, a >= b</td></tr>
</table>

Цикл for при работе с контейнерами

C++ позволяет в цикле **for** итерироваться по всему контейнеру

```
std::vector<int> vec = {1,2,3,-4,5,-6};
//"Классический" подход
for(size_t i = 0; i < vec.size(); ++i)
  std::cout << vec[i] * vec[i] << ' ';

//Использование итераторов
for(auto it = vec.begin(); it != vec.end(); ++it)
  std::cout << (*it) * (*it) << ' ';

//Автоматические итерирование
for(int elem: vec)
  std::cout << elem * elem << ' ';
```

std::vector - продолжение

Для вставки и удаления элементов из массива необходимо использовать итераторы

```
std::vector vec = {1,-1,10,5,-6,15};
auto it = vec.begin(); //итератор на начало
it += 2; //перемещение итератора на 2 позиции вперед
vec.insert(it, 42); //вставит 42 ПЕРЕД 10, итератор становится невалидным
it = vec.begin();
++it; //смещение итератора на 1 позицию вперед
vec.insert(it, 5, 100); //вставить пять 100 после первого элемента
vec.erase(vec.end() - 2); //удалить предпоследний элемент
```

std::list

std::list<T> - хранит последовательность элементов в виде двусвязного списка. Работа с **std::list** аналогично вектору, только отсутствует обращение к элементу по индексу. То есть итерироваться можно используя операторы ++ и -- .
Итератор становится невалидным только после удаления элемента.

std::forward_list<T> - односвязный список, поэтому двигаться можно в одну сторону используя оператор ++.

У **std::list** для вставки и удаления есть методы: **insert** и **erase** (перед итератором), **push_back** и **pop_back** (в конце), **push_front** и **pop_front** (в начале).

У **std::forward_list** для вставки и удаления есть методы: **insert_after** и **erase_after** (ПОСЛЕ итератора), **push_front** и **pop_front** (в начале)

std::list и std::forward_list пример

```
std::list lst = {1,2,3,4,5};
//Копировать в контейнер другого типа можно передав итераторы
std::forward_list<int> flst(lst.begin(), lst.end());
//Получение итераторов на элемент с индексом 1
auto lst_it = ++lst.begin();
auto flst_it = ++flst.begin();
lst.insert(lst_it, 100); //Получим список 1 100 2 3 4 5
flst.insert_after(flst_it, 100); //Получим список 1 2 100 3 4 5
```

std::deque

std::deque<T> - хранит элементы в памяти в виде нескольких непрерывных участков. Методы вставки и удаления аналогичны **std::list**, но также предоставляет свободный доступ к элементы по индексу**.**

Вставка и удаление элементов из начала и конца O(1). В середину O(n). Но в отличие от **std::vector** эффективнее выделяет память под новые элементы.

std::map

std::map<Key, T> - ассоциативный массив, который хранит элементы типа **T** по уникальным ключам типа **Key**. Хранение реализовано через красно-черное дерево, поэтому операции вставки, удаления и поиска имеются сложность O(log(n)). Итерирование происходит по возрастанию ключей.
std::unordered_map - хранит элементы в хеш таблице.

```cpp
//Создания словаря
std::map<char, int> ci_map = {{'E', 42}, {'R', 12}};
ci_map.at('E') *= 10; //Получение доступа с проверкой ключа
ci_map['R'] = 10; //Получение доступа без проверки;
ci_map['A'] = -10; //Если ключа нет, будет создан новый элемент
//Всё равно будет создан элемент с значением по умолчанию
std::cout << ci_map['X'] << '\n';
//Хранит элементы в виде пар, .first - ключ, .second - значение
for(auto elem: ci_map)
  std::cout << elem.first << ' ' << elem.second << '\n';
```

std::set

std::set<T> - контейнер, который хранит только уникальные элементы в виде красно-черного дерева.
std::unordered_set - хранит элементы в хеш таблице.

```
std::vector vec = {5,1,3,2,1,1,3,2,5,2};
std::set int_set(vec.begin(), vec.end());
for(auto elem: int_set){
  std::cout << elem << ' '; //выведет 1 2 3 5
}
int_set.insert(4); //Будет добавлено, еще нет в set
int_set.insert(2); //Уже есть в set, изменений не будет
int_set.erase(1); //Будет удалено
int_set.erase(0); //Всё ок, элемента итак нет
for(auto elem: int_set){
  std::cout << elem << ' '; //выведет 2 3 4 5
}
```

Адаптеры контейнеров

Адаптеры контейнеров - используют существующий контейнер для хранения данных, но предоставляют другой интерфейс по работе с контейнером.

std::stack<T> - LIFO контейнер. По умолчанию использует **std::deque**.
Методы: **top** - просмотреть верхний элемент, **push** - поместить в стек, **pop** - убрать из стека.

std::queue<T> - FIFO контейнер. По умолчанию использует **std::deque**.
Методы: **front** - просмотреть начало очереди, **back** - просмотреть конец очереди, **push** - добавить в конец очереди, **pop** - убрать из начала очереди.

std::priority_queue<T> - очередь с приоритетом. По умолчанию использует **std::vector**.
Методы: **top** - просмотреть начало очереди, **push** - добавить в конец очереди, **pop** - убрать из начала очереди.

Адаптеры итераторов

- **back_insert_iterator** - при присваивании ему, вызывает метод **push_back** у контейнера, создается функцией **back_inserter**
- **front_insert_iterator -** при присваивании ему, вызывает метод **push_front** у контейнера, создается функцией **front_inserter**
- **insert_iterator -** при присваивании ему, вызывает метод **insert** у контейнера, создается функцией **inserter**

```
std::vector vec = {1,2,3,4,5};
std::insert_iterator it(vec, vec.end()-1);
//можно создать как
//auto it = std::inserter(vec, vec.end()-1);
for(int i = 10; i < 16; ++i)
  it = i;
//выведет 1 2 3 4 10 11 12 13 14 15 5
for(auto elem: vec)
  std::cout << elem << ' ';
```

Потоковые итераторы

- **istream_iterator<T> -** позволяет итерироваться по входному потоку. Параметр T определяет, какой тип данных считывается. В качестве аргумента конструктора передается поток для чтения. без аргумента, то итератор на конец.

- **std::ostream_iterator<T> -** позволяет итерировать по выходному потоку. Параметр T определяет, какой тип данных выводится. В качестве аргументов конструктора передается потом для записи и разделительную строку.

Алгоритмы

- Стандартная библиотека предоставляет набор алгоритмов для работы с контейнерами следующих видов:
  - for_each
  - Поиск
  - Копирование
  - Обмен
  - Преобразование
  - Генерация значений
  - Удаление
  - Изменение порядка
  - Случайный выбор значения
  - Сортировка
- Алгоритмы находятся в модуле <algorithm>

for_each

- Функция **for_each** применяет унарную функцию к диапазону итераторов
- **for_each_n** - применяет к первым n элементам. *_n есть у многих алгоритмов*
- Унарная функция может быть самой функцией, лямбда-выражением или функтором

```
void square(int& x){
  x *= x;
}
int main(){
  std::vector vec = {1,2,3,3,4,5};
  //Применит функцию возведения в квадрат к каждому элементу вектора
  //так как функция принимает ссылку, то изменения отобразятся в векторе
  std::for_each(vec.begin(), vec.end(), square);
  //Передаем лямбда-выражение, которое выводит каждый элемент в потом
  std::for_each(vec.begin(), vec.end(), [](int x){std::cout << x << ' ';});
}
```

Поиск (1)

- Функции **all_of** / **any_of** / **none_of** принимают два итератора и предикат, и проверяют что **все** / **хотя бы один** / **ни один** элемент вернул true для для предиката
- Предикат - функция от 1 аргумента, которая возвращает булево значение

```
bool greaterThanZero(int x){
  return x > 0;
}

int main(){
  std::list lst = {1,-2,3,4,5,6,11};
  bool is_all_greater = std::all_of(lst.begin(), lst.end(), greaterThanZero);
  bool is_any_greater = std::any_of(lst.begin(), lst.end(), greaterThanZero);
  bool is_none_greater = std::none_of(lst.begin(), lst.end(), greaterThanZero);
  std::cout << is_all_greater << is_any_greater << is_none_greater;
}
```

Поиск (2)

- Функция **find** ищет элемент совпадающий с переданным значением
- Функция **find_if** ищет элемент удовлетворяющий предикату
- Функции возвращают итератор на первый найденный элемент. Если не найден - то итератор на конец

```
std::vector vec = {1,-2,3,3,1,4,3,4,2,2,43,12,2,-4,2};
auto start = std::find(vec.begin(), vec.end(), 0);
//Метод base преобразует reverse итератор в forward
auto finish = std::find(vec.rbegin(), vec.rend(), 0).base();
//Выводим значения от первого нуля до последнего нуля в вектое
if(start != vec.end())
  std::for_each(start, finish, [](int x){std::cout << x << ' ';});
```

Копирование

- Функция **copy** копирует диапазон значений в новое место используя итератор
- Функция **copy_if** работает также как **copy**, но копирование только если предикат true

```
std::vector vec = {1,2,3,4,5,6,7,8,9,10};
//Копируем в поток вывода. Вывод всех чисел
std::copy(vec.begin(), vec.end(),
          std::ostream_iterator<int>(std::cout, " "));
std::list<int> lst;
//Копируем из вектора в список только четные числа, вставля в начало
std::copy_if(vec.begin(), vec.end(),
             std::front_inserter(lst),
             isEven); //isEven написанная функция проверяющая четность
std::vector<int> vec2;
vec2.resize(5);
//Копируем первые пять элементов в обратном порядке
std::copy_n(vec.begin(), 5, vec2.rbegin());
```

Преобразование

- Функция **transform** применяет функцию в каждому элементу и записывает результат по итератору

```
//Преобразует число в текст в зависимости от четности
std::string numToText(int value){
  return value % 2 == 0? "Even" : "Odd";
}
int main(){
  std::vector num_vec = {1,2,-4,3,15,2,0,23};
  std::vector<std::string> str_vec;
  std::transform(num_vec.begin(), num_vec.end(),
                 std::back_inserter(str_vec),
                 numToText);
  std::for_each(str_vec.begin(),str_vec.end(),[](auto str){std::cout << str << '\n';});
}
```

Удаление

- Функция **remove** удаляет все элементы равные заданному значению
- Функция **remove_if** удаляет все элементы для которых предикат true
- Удаление происходит перемещением элементов в конец. Фактически они не удаляются из памяти.
- Функции возвращают итератор на новый конец.

```
bool isNegative(int x){
  return x < 0;
}

int main(){
  std::vector vec = {-1,-2,5,3,-8,2,8,-9,2};
  auto end_after_2 = std::remove(vec.begin(), vec.end(), 2);
  auto end_after_neg = std::remove_if(vec.begin(), end_after_2, isNegative);
  vec.erase(end_after_neg, vec.end()); //Удаление элементов в конце
  std::for_each(vec.begin(), vec.end(), [](int x){std::cout << x << ' ';});
}
```

Генерация значений

- Функция **fill** заполняет диапазон заданным значением
- Функция **generate** заполняет диапазон согласно заданной функции

```
//Создали функциональный объект перегрузив оператор ()
class FibonacciMaker{
  int a = 0; int b = 1;
public:
  int operator () (){
    int new_number = a + b;
    a = b; b = new_number;
    return a;
  }
};
int main(){
  std::vector<int> vec(15); //создаем вектор размера 15
  FibonacciMaker func;
  //Заполнили вектор числами фибоначи
  std::generate(vec.begin(), vec.end(), func);
  std::for_each(vec.begin(),vec.end(), [](int x){std::cout << x << ' ';});
}
```