Библиотека STL
(Standard Template Library)

Стандартная библиотека шаблонов (STL).

Если вам нужен какой-нибудь общий класс или алгоритм, то скорее всего в STL он уже есть. Вы можете использовать эти классы и алгоритмы без необходимости писать и заниматься их отладкой самостоятельно, а так же разбираться в том, как они реализованы.

Библиотека стандартных шаблонов (англ. Standard Template Library, STL) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Структура библиотеки.

- **контейнер** (*container*): управляет набором объектов в памяти.
- **алгоритм** (*algorithm*): определяет вычислительную процедуру
- **итератор** (*iterator*): обеспечивает для алгоритма средство доступа к содержимому контейнера.
- **функциональный объект** (*function object*): инкапсулирует функцию в объекте для использования другими компонентами.
- **адаптер** (*adaptor*): адаптирует компонент для обеспечения различного интерфейса.

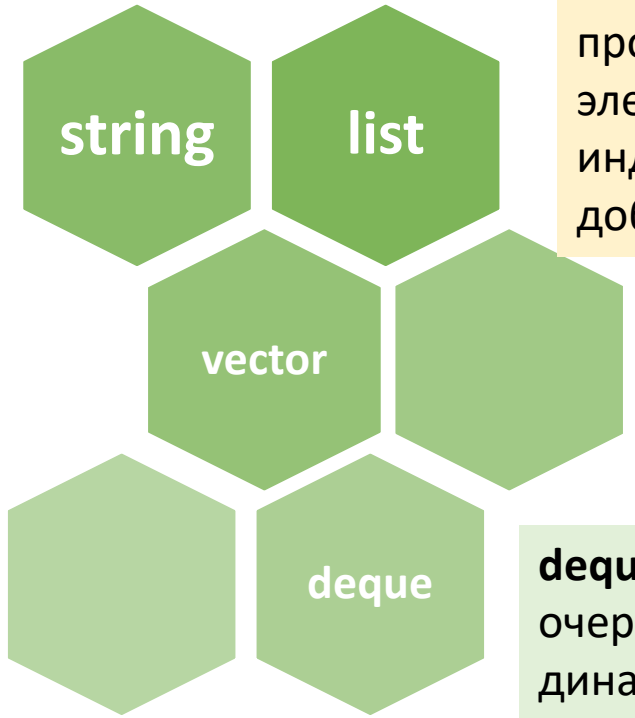
Контейнеры STL

Под **контейнером** понимают объект, содержащий другие (однотипные) объекты, называемые элементами контейнера. Стандартная библиотека C++ предоставляет типичные контейнеры, такие как: **list, vector, queue, map, set** и др. Доступ к элементам контейнера осуществляется через **итераторы**. К контейнерам выдвигается ряд общих требований. Это осуществляется для того, чтобы использование контейнеров было одинаковым, независимо от его реализации. Соответственно, часто контейнеры бывают взаимозаменяемы.

Последовательные контейнеры

Последовательные контейнеры хранят свои элементы в строго линейном порядке. К последовательным контейнерам относятся структуры данных вектор (vector), список (list), очередь (deque), а также строка символов (string).

vector – динамический массив, способный увеличиваться по мере необходимости для содержания всех элементов. Обеспечивает производительный доступ к своим элементам через оператор индексирования [], поддерживает добавление и удаление элементов.



string – фактически это вектор с элементами типа char.

list – «список» Двусвязный список, каждый элемент которого содержит 2 указателя. Контейнер предоставляет доступ только к началу и концу списка, запрещая произвольный доступ.

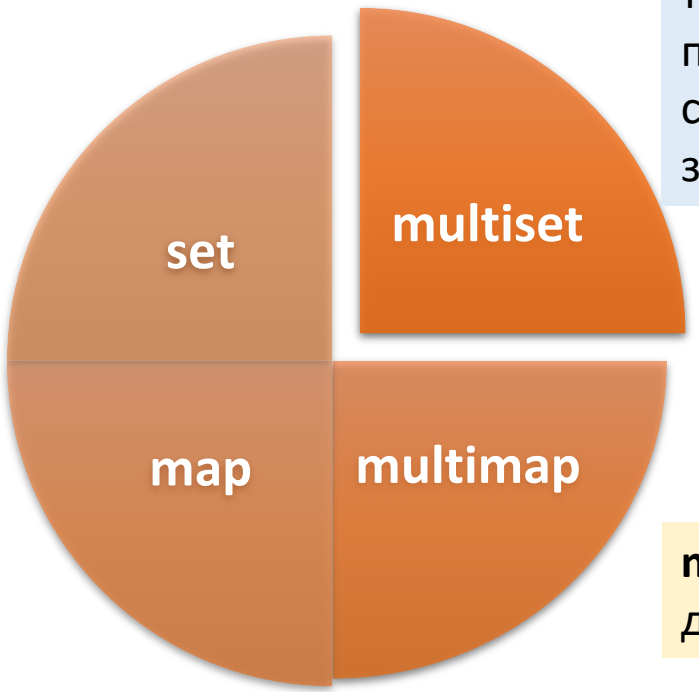
deque – (или «дек»). Двусторонняя очередь реализованная в виде динамического массива, который может расти с обоих концов.

Ассоциативные контейнеры

Ассоциативные контейнеры автоматически **сортируют** свои элементы. К ассоциативным контейнерам относятся набор уникальных элементов (set), набор с повторениями(multiset), ассоциативный массив(map), словарь(multimap).

map – это set, в котором каждый элемент является парой «ключ-значение». «Ключ» используется для сортировки и индексации данных и должен быть уникальным, а «значение» - это фактические данные.

multimap – это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания. Значение можно получить по ключу.

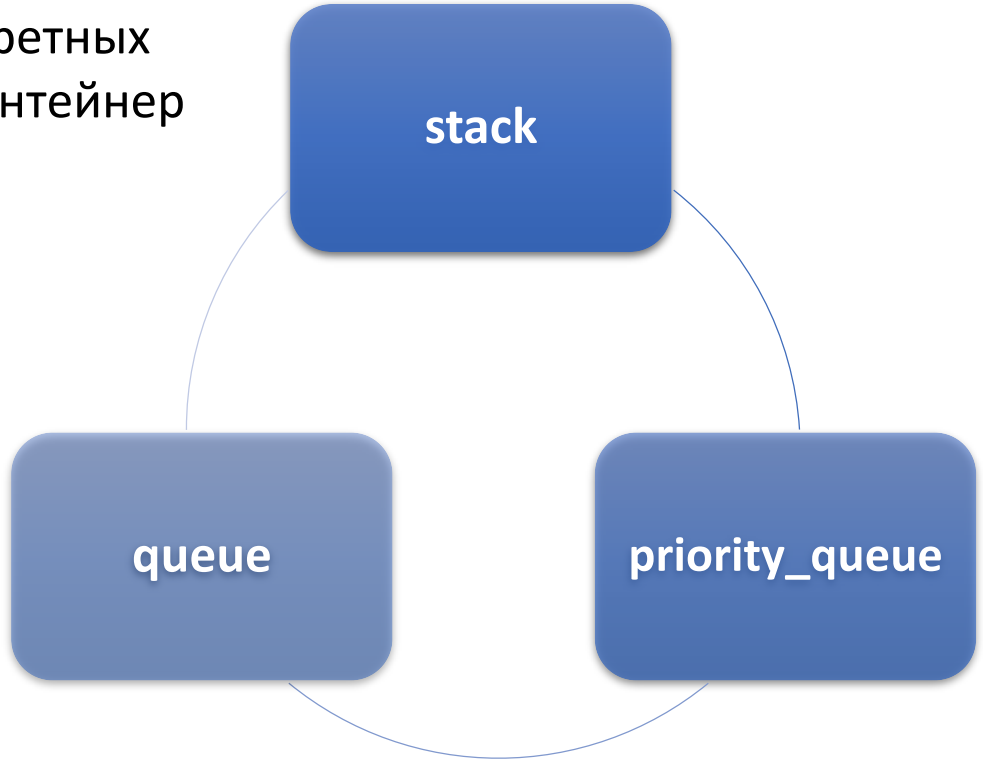


set – контейнер в котором хранятся только уникальные элементы, повторения запрещены. Элементы сортируются в соответствии с их значениями.

multiset – это set в котором допускаются повторения.

Контейнеры адаптеры

Контейнеры адаптеры – специальные последовательные контейнеры, адаптированные для выполнения конкретных задач. Можно выбирать какой последовательный контейнер будет использовать адаптер.



stack – контейнер элементы которого работают по принципу **LIFO** .т.е. элементы добавляются в конец контейнера и удаляются так же с конца. Внутри может использоваться **vector** или **list**. Но по умолчанию используется **deque.**

priority_queue – разновидность очереди, в которой все элементы отсортированы по приоритету. При добавлении элемента он автоматически сортируется. Элемент с наивысшим приоритетом находится в самом начале очереди с приоритетом и будет удален первым из очереди.

queue – контейнер элементы которого работают по принципу **FIFO**. Внутри по умолчанию используется **deque**.

Итераторы STL

Итератор – объект, который способен перебирать элементы контейнера без необходимости сообщать пользователю реализацию контейнерного класса. Во многих типов контейнеров (ассоциативных) итераторы являются основным способом доступа к элементам этих контейнеров.

Об итераторе можно думать как об указателе на определенный элемент контейнера с дополнительным набором перегруженных операторов для выполнения четко определенных функций.

Оператор = присваивает итератору новую позицию (обычно начальный или конечный элемент контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор =.

Оператор * возвращает элемент, на который в данный момент указывает итератор.

Операторы == и != используются для определения того, указывают ли оба итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают оба итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор == или оператор !=.

Оператор ++ перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор −− для перехода к предыдущему элементу.

Итераторы STL

Каждый контейнерный класс имеет 4 основных метода для работы с оператором =:

- **метод begin()** возвращает итератор, представляющий начальный элемент контейнера;
- **метод end()** возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере;
- **метод cbegin()** возвращает константный (только для чтения) итератор, представляющий начальный элемент контейнера;
- **метод cend()** возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Все контейнеры предоставляют (как минимум) два типа итераторов:

container::iterator — итератор для чтения/записи;

container::const_iterator — итератор только для чтения.

Итераторы должны быть реализованы для каждого контейнера отдельно, поскольку итератор должен знать реализацию контейнерного класса. Таким образом, итераторы всегда привязаны к конкретным контейнерным классам.

П. Итераторы STL (std::vector)

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> myVector;
    for (int count=0; count < 5; ++count)
        myVector.push_back(count);
    std::vector<int>::const_iterator it; // объявляем итератор только для чтения
    it = myVector.begin();                  // присваиваем ему начальный элемент вектора
    while (it != myVector.end())        // пока итератор не достигнет последнего элемента
        {
        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
        ++it; // и переходим к следующему элементу
        }
    std::cout << '\n';
}
```

П. Итераторы STL (std::list)

```
#include <iostream>
#include <list>

int main()
{
    std::list<int> myList;
    for (int count=0; count < 5; ++count)
        myList.push_back(count);

    std::list<int>::const_iterator it; // объявляем итератор
    it = myList.begin(); // присваиваем ему начальный элемент списка
    while (it != myList.end()) // пока итератор не достигнет последнего элемента
    {
        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
        ++it; // и переходим к следующему элементу
    }
    std::cout << '\n';
}
```

П. Итераторы STL (std::set)

```
#include <iostream>
#include <set>
int main(){
    std::set<int> mySet;
    mySet.insert(8);
    mySet.insert(3);
    mySet.insert(-4);
    mySet.insert(9);
    mySet.insert(2);

    std::set<int>::const_iterator it; // объявляем итератор
    it = mySet.begin();                  // присваиваем ему начальный элемент set-а
    while (it != mySet.end())          // пока итератор не достигнет последнего элемента
    {
        std::cout << *it << " "; // выводим значение элемента, на который указывает итератор
        ++it;                            // и переходим к следующему элементу
    }
    std::cout << '\n';
}
```

П. Итераторы STL (std::map)

```
#include <iostream>
#include <map>
#include <string>
int main(){
        std::map<int, std::string> myMap;
        myMap.insert(std::make_pair(3, "cat"));
        myMap.insert(std::make_pair(2, "dog"));
        myMap.insert(std::make_pair(5, "chicken"));
        myMap.insert(std::make_pair(4, "lion"));
        myMap.insert(std::make_pair(1, "spider"));

        std::map<int, std::string>::const_iterator it; // объявляем итератор
        it = myMap.begin();                            // присваиваем ему начальный элемент вектора
        while (it != myMap.end()) // пока итератор не достигнет последнего элемента
        {
          std::cout << it->first << "=" << it->second << " "; // выводим значение элемента, на который указывает итератор
                ++it;                                         // и переходим к следующему элементу
        }
        std::cout << '\n';
}
```

Функторы

Функтор — это сокращение от функциональный объект, представляющий собой конструкцию, позволяющую использовать объект класса как функцию. В C++ для определения функтора достаточно описать класс, в котором переопределена операция ().

Операция () в классе может быть переопределена (точнее определена, поскольку она не имеет реализации по умолчанию) с произвольным числом, типом параметров и типом возвращаемого значения (или даже вовсе без возвращаемого значения).

Но особо широкое применение функторы приобрели в алгоритмах STL, рассмотренных ранее, когда они передаются в вызов в качестве параметра, вместо функции, определяющей **действие** или **предикат** алгоритма.

Выгода функтора состоит в том, что
а). его **можно параметризовать** при создании объекта (перед вызовом) используя конструктор объекта с параметрами и
б). может **создаваться временный объект** исключительно на время выполнения функционального вызова.

П. Функтор. Простой пример

```
#include <iostream>

class Sum
{
public:
    int operator()(int x, int y) const  {
        return x + y;
    }
};
int main()
{
    Sum sum;                    // определяем объект функции
    int result {sum(2, 3)};     // вызываем объект функции
    std::cout << result << std::endl;           // 5
    std::cout << sum(5, 3) << std::endl;        // 8
    std::cout << sum(12, 13) << std::endl;  // 25
}
```

П. Функтор. Пример посложнее

```
#include <iostream>
#include <vector>

using namespace std;

class summator : private vector<int> {
public:
    summator(const vector<int> & ini) {
        for (auto x : ini)
            this->push_back(x);
    }
};
```

```
int operator()(bool even) {
        int sum = 0;
        auto i = begin();
        if (even) i++;
        while (i < end()) {
        sum += *i++;
        if (i == end()) break;
          i++;
       }
   return sum;
}
```

```
int main(void) {
  summator sums(vector<int>({ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }));
  cout << "сумма чётных = " << sums(true) << endl << "сумма нечётных = " << sums(false) << endl;
}
```

Алгоритмы STL

#include <algorithm>

Алгоритмы STL реализованы в виде глобальных функций, которые работают с использованием итераторов со всеми контейнерами, которые предоставляют набор итераторов (включая и ваши собственные (пользовательские) контейнерные классы).

П. Алгоритмы STL

Алгоритм find(), чтобы найти определенное значение в списке, а затем используем функцию list::insert() для добавления нового значения в список

```
#include <iostream>
#include <list>
#include <algorithm>

int main(){
    std::list<int> li;
    for (int nCount=0; nCount < 5; ++nCount)
        li.push_back(nCount);
    std::list<int>::iterator it; // объявляем итератор
    it = find(li.begin(), li.end(), 2); // ищем в списке число 2
    li.insert(it, 7); // используем алгоритм list::insert() для добавления числа 7 перед числом 2
    for (it = li.begin(); it != li.end(); ++it) // выводим с помощью цикла и итератора элементы списка
        std::cout << *it << ' ';

    std::cout << '\n';
}
```

Алгоритмы min_element() и max_element() находят минимальный и максимальный элементы в контейнере:

```
#include <iostream>
#include <list>
#include <algorithm>

int main(){
    std::list<int> li;
    for (int nCount=0; nCount < 5; ++nCount)
        li.push_back(nCount);

    std::list<int>::const_iterator it; // объявляем итератор
    it = min_element(li.begin(), li.end());
        std::cout << *it << ' ';
    it = max_element(li.begin(), li.end());
        std::cout << *it << ' ';

    std::cout << '\n';
}
```

П. Использование стандартной библиотеки

Считать из файла input.txt массив целых чисел, разделенных пробельными символами. Отсортировать их и записать в файл output.txt.

```
#include <vector>
#include <algorithm>
#include <fstream>

int main(){
      std::ifstream fin("input.txt");
      std::ofstream fout("output.txt");
      std::vector<int> v;
      std::copy(std::istream_iterator<int>(fin),
                  std::istream_iterator<int>(),
                  std::inserter(v, v.end()));
     std::sort(v.begin(), v.end());
     std::copy(v.begin(),
                 v.end(),
                std::ostream_iterator<int>(fout, " "));
return 0;
}
```

П. Использование стандартной библиотеки

```
#include <string>
#include <vector>
#include <fstream>
#include <algorithm>
struct Man{
    std::string firstname, secondname;
    size_t age;
};
std::ostream& operator << (std::ostream& out, const Man& p){
    out << p.firstname << " " << p.secondname << " " << p.age;
    return out;
}
std::istream& operator >> (std::istream& in, Man& p){
    in >> p.firstname >> p.secondname >> p.age;
    return in;
}
struct comparator{
    comparator(){}
    bool operator ()(const Man& p1, const Man& p2){
        return p1.age < p2.age;
    }
};
struct Predicat{
    size_t begin, end;
    Predicat(int p1, int p2): begin(p1), end(p2) {}
    bool operator ()(const Man& p){
        return (p.age > begin) && (p.age < end);
    }
};
```

В файле input.txt хранится список, содержащий информацию о людях: фамилия, имя, возраст. Считать эти данные в массив, отсортировать их по возрасту и записать в файл output.txt. Вывести на экран информацию о человеке, чей возраст более 20, но менее 25 лет.

```
int main(){

    std::ifstream fin("input.txt");
    std::ofstream fout("output.txt");
    std::vector<Man> v;
    std::vector<Man>::iterator i;
    std::copy(std::istream_iterator<Man>(fin),
        std::istream_iterator<Man>(),
        std::inserter(v, v.end()));
    std::sort(v.begin(), v.end(), comparator());
    i = std::find_if(v.begin(), v.end(), Predicat(20, 25));
    std::cout << (*i) << std::endl;
    std::copy(v.begin(),  v.end(),
        std::ostream_iterator<Man>(fout, "\n"));
    return 0;
}
```