Тема 6-3.

Стандартная библиотека STL

для АСУб и ЭВМб

Темы лекции

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

Библиотека стандартных шаблонов до включения в стандарт C++. была сторонней разработкой, вначале — фирмы HP, а затем SGI. Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода-вывода (iostream), подраздел Си и др.).

Проект под названием STLPort, основанный на SGI STL, осуществляет постоянное обновление STL, iostream и строковых классов. Некоторые другие проекты также занимаются разработкой частных применений стандартной библиотеки для различных конструкторских задач. Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется.

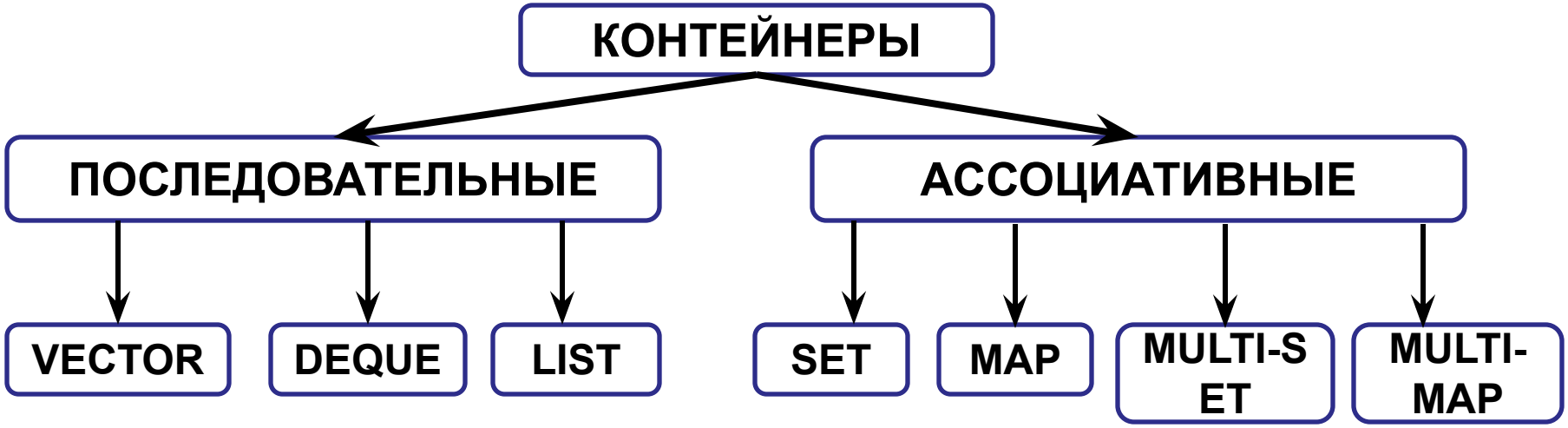
Архитектура STL была разработана Александром Степановым и Менг Ли.

Состав STL:

- контейнеры,
- итераторы,
- алгоритмы,
- аллокаторы,
- адаптеры.

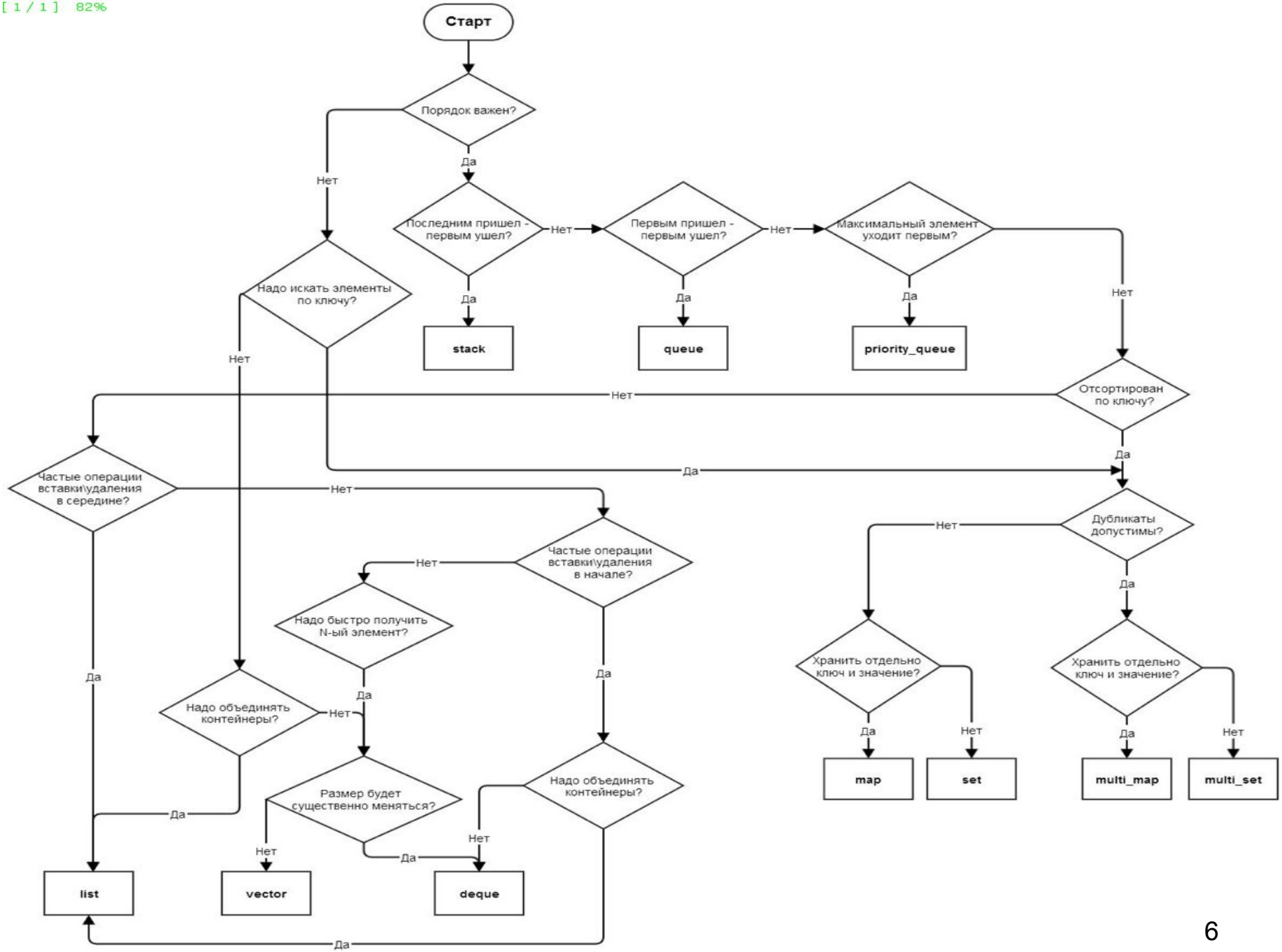
Контейнер **– хранилище, единицами хранения (элементами) являются другие объекты + набор методов, предназначенных для манипулирования элементами контейнера. В контейнере может быть только один тип данных.**

Состав STL:



Последовательный контейнер **представляет множество объектов одного типа в строго линейной последовательности.**

Ассоциативный контейнер **обеспечивает быструю выборку данных, соответствующих указанному ключу. Для реализации ассоциативных контейнеров используются двоичные деревья.**


Старт
Порядок важен?
Да
Нет
Последним пришел - первым ушел?
Нет
Первым пришел - первым ушел?
Нет
Максимальный элемент уходит первым?
Да
Нет
stack
queue
priority_queue
Надо искать элементы по ключу?
Нет
Да
Отсортирован по ключу?
Нет
Да
Частые операции вставки\удаления в середине?
Нет
Да
Частые операции вставки\удаления в начале?
Нет
Да
Надо быстро получить N-ый элемент?
Да
Надо объединять контейнеры?
Нет
Да
Размер будет существенно меняться?
Нет
Да
Надо объединять контейнеры?
Нет
Да
Дубликаты допустимы?
Нет
Да
Хранить отдельно ключ и значение?
Да
Нет
Хранить отдельно ключ и значение?
Да
Нет
map
set
multi_map
multi_set
list
vector
deque

Шаблоны

STL основывается на понятии *шаблона.*

Предположим, что для некоторого числа *x > 0* нужно часто вычислять значение выражения:

2 * x + (x * x + 1) / (2 * x),

где *x* может быть типа *double* или *int.* Например, если *x* имеет тип *double* и равен 5.0, тогда значение выражения составляет 12.6, но если *x* имеет тип *int* и равен 5, то значение выражения будет 12.

Шаблонные функции

Вместо того чтобы писать две функции:

```
double  f(double  x)
{   double  x2  =  2  *  x;
     return x2  +  (x  *  x  +  1)/x2;
}
```

и

```
int f(int x)
{   int x2 = 2 * x;
    return x2 +  (x * x + 1)/x2;
}
```

нам достаточно создать один *шаблон:*

Шаблонные функции

```
// ftempl.cpp: Шаблонная функция.
#include <iostream>
using namespace std;
 template <typename T>
T f (T x)
{    T x2 = 2 * x;
      return x2 + (x * x + 1)/x2;
}
int main()
{    cout << f(5.0) << endl << f(5) << endl;
     return 0;
}
```

Программа выведет: 12.6 12

В этом шаблоне T – тип, задаваемый аргументом при вызове f.

Шаблонные структуры

Пусть нам нужна структура Pair, чтобы хранить пары значений. Иногда оба значения принадлежат к типу *double*, иногда к типу *int*. Тогда вместо двух новых структур:

```
struct PairDouble
{
void showQ();
double x, y;
};
void
PairDouble::showQ()
{
    cout << x/y << endl;
}
```

```
struct PairInt
{
void showQ();
int x, y;
};
void PairInt::showQ()
{
    cout << x/y << endl;
}
```

Шаблонные структуры

Вместо этого напишем одну шаблонную структуру:

```
// cltempl.cpp:
#include <iostream.h>

template <typename T>
struct  Pair
{
 void showQ();
 T x, y;
};
```

Шаблонные структуры

```
template < typename T>
void Pair<T>::showQ()
{
    cout << x/y << endl;
}
int main()
{
 Pair<double> a(37.0,  5.0);
 Pair<int> u(37,  5);
 a.showQ();
 u.showQ();
 return  0;
}
```

Замечания

Как пользователи STL мы можем не беспокоиться об определениях, так как шаблонные функции, структуры и классы STL доступны в виде *файлов заголовков,* которые можно использовать, не вдаваясь в подробности их программирования.

Единственный аспект применения шаблонов, который мы увидим в наших программах, это обозначение фактического типа с помощью конструкции наподобие *Pair<double>.*

Процесс создания конкретной версии шаблонной функции называется *инстанцированием* шаблона или *созданием экземпляра.*

Идея итераторов:

Итератор **– аналог указателя. Получив итератор какого-то элемента контейнера при помощи операторов инкремента ++ и -- можно перейти к следующему и предыдущему элементу контейнера**

Итераторы **можно назвать посредниками между алгоритмами и контейнерами, потому что многие алгоритмы используют итераторы для того, чтобы перебрать элементы в контейнере.**

Разыменованный итератор – это данные элемента контейнера, их можно получить или изменить.

Для каждого класса контейнеров определяется свой класс итераторов, однако интерфейсы итераторов разных контейнеров полностью совпадают.

Итераторы

Какие функции должен выполнять итератор?

1. **Возможность разыменования того объекта, на который указывает итератор.**
2. **Возможность изменить объект, на который указывает итератор.**
3. **Возможность сравнения итераторов (==), например, определить, дошли ли до конечного элемента контейнера.**

Итераторы, обладающие такими свойствами называются базовыми (*Trivial Iterator*) и практического применения такие итераторы не имеют. К итераторам добавляются новые свойства и получаем новые виды контейнеров.

Типы Итераторы

Input Iterator – **получаем доступ только для чтения данных (необходим инкремент).**

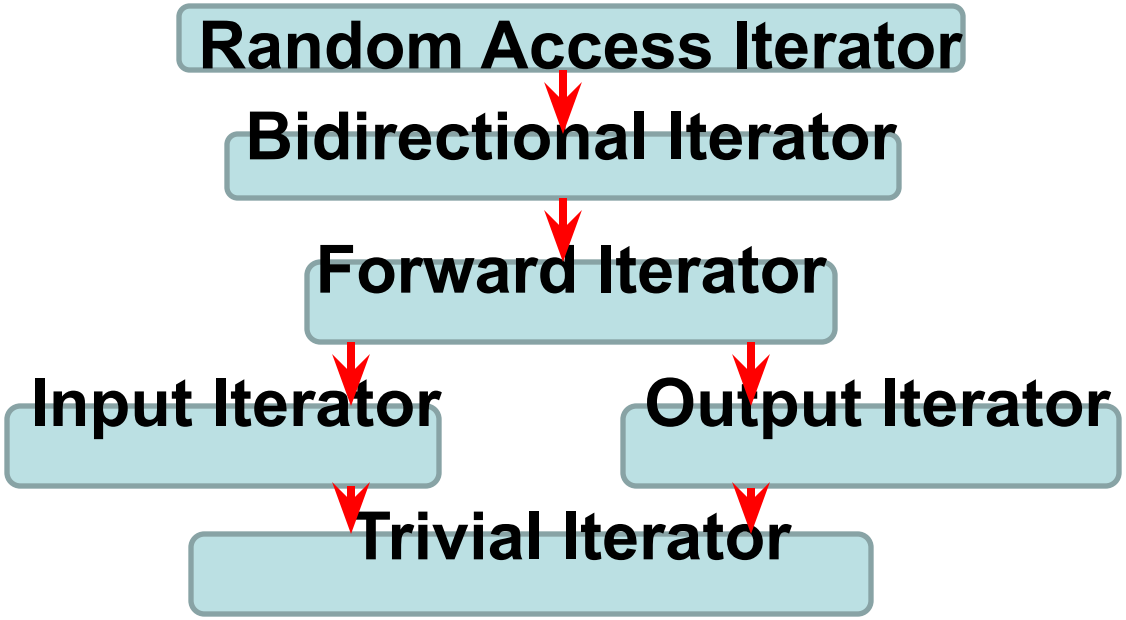
Output Iterator – **нужно уметь записывать данные в контейнер.**

Forward Iterator – **чтение и запись данных (перемещение в одном направлении).**

Bidirectional Iterator – **перемещение не только в прямом, но и в обратном направлениях.**

Random Access Iterator – **произвольный доступ к любому элементу контейнера таким образом, что время доступа к любому элементу не зависит от размера контейнера.**

Иерархия итераторов



Все итераторы, которые находятся выше, включают в себя функциональность операторов, находящихся ниже.

Итераторы обеспечивают доступ к элементам контейнера. С помощью итераторов очень удобно перебирать элементы. Итератор описывается типом **iterator**. Но для каждого контейнера конкретный тип итератора будет отличаться.

Для получения итераторов контейнеры в C++ обладают такими функциями, как **begin()** и **end().**

- Функция **begin()** возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов).
- Функция **end()** возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера.

Если контейнер пуст, то итераторы, возвращаемые обоими методами **begin и end** совпадают.

Если итератор **begin** не равен итератору **end**, то между ними есть как минимум один элемент.

Обе этих функции возвращают итератор для конкретного типа контейнера

Операции с итераторами

С итераторами можно проводить следующие операции:

***iter**: получение элемента, на который указывает итератор

++iter: перемещение итератора вперед для обращения к следующему элементу

--iter: перемещение итератора назад для обращения к предыдущему элементу. Итераторы контейнера forward_list не поддерживают операцию декремента.

iter1 == iter2: два итератора равны, если они указывают на один и тот же элемент

iter1 != iter2: два итератора не равны, если они указывают на разные элементы

Алгоритмы

Алгориты **делятся на несколько категорий:**

- ***немодифицирующие*** **алгоритмы (не изменяющие порядок следования элементов в контейнере) – count, count_if, find, find_if и др.;**
- ***модифицирующие*** **алгоритмы (изменяющие порядок следования элементов в контейнере) - copy, replace, reverse, swap (обмен местами двух элементов) и др.;**
- ***алгоритмы сортировки и поиска*** **(упорядочивание, поиск, слияние и т.д.) – merge, sort, stable_sort (сохраняет порядок для одинаковых элементов), partial_sort (частичная сортировка), max_element и др.;**
- ***алгоритмы работы с множествами*** **– accumulate, includes set_intersection set_difference и др.**
- ***Численные алгоритмы*** **(подключаем заголовочный файл)**

Аллокаторы

Аллокаторы **предназначены для выделения и освобождения памяти, – низкоуровневый интерфейс. Если контейнер выделяет память при помощи аллокатора, то при удалении контейнера можно не заботиться об освобождении памяти, всё делается автоматически.**

Адаптеры **– это классы, которые упрощают интерфейс для доступа к объектам другого класса.**

Одним из недостатков STL является не очень удобная работа с итераторами, т.е. после добавления или удаления элемента в/из контейнера итератор может стать недействительным.

Пример обобщенного алгоритма Алгоритм find.

Обобщенный алгоритм find используется для поиска заданного значения в контейнере. Может использоваться с любыми контейнерами STL.

Синтаксис вызова обобщенного алгоритма find

```
where = find(first, last, value);
```

Параметры:

- first - итератор, указывающий на позицию начала поиска;
- last - итератор, указывающий на позицию завершения поиска.
- value - искомое значение.

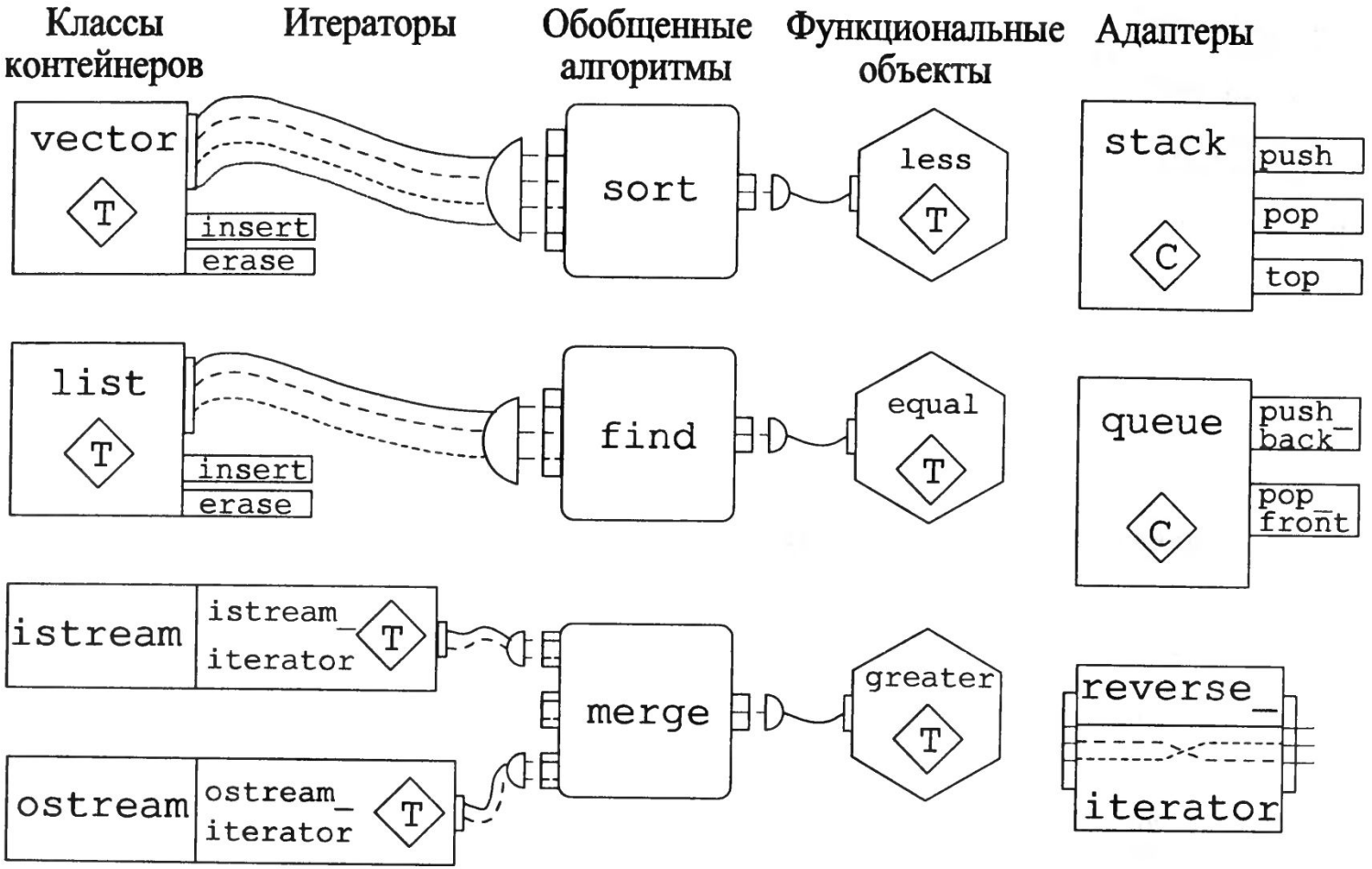
Возвращаемое значение:

итератор where, указывающий на позицию, в которой был обнаружен элемент со значением value. Возвращается 1-й по порядку элемент. Если ни одного элемента со значением value не найдено, find возвращает итератор, который указывает на значение "за концом" контейнера (last).

```
#include <iostream>
#include <cassert>
#include<cstring>
#include <algorithm> // Алгоритм find
using namespace std;
int main()
{
   cout << "Демонстрация работы обобщенного алгоритма "
        << "find с массивом." << endl;
   char s[] = "C++ is a better C";

   int len = strlen(s);
     // Вызываем find - ищем первое вхождение символа 'e':
   const char* where = find(&s[0], &s[len], 'e');
   assert (*where == 'e' && *(where+1) == 't');
   cout << " ======= Ok!" << endl;
   return 0;
}
```

Понятие итератора является ключевым в STL. Алгоритмы STL написаны с применением итераторов в качестве параметров, а контейнеры STL предоставляют итераторы, которые затем могут "включаться" в алгоритмы.



Vector (Вектор)

Vector — это замена стандартному динамическому массиву, память для которого выделяется вручную, с помощью оператора **new.**

Разработчики языка рекомендуют в использовать именно vector вместо ручного выделения памяти для массива. Это позволяет избежать утечек памяти и облегчает работу программисту.

Пример:

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    // Вектор из 10 элементов типа int
    vector<int> v1(10);

    // Вектор из элементов типа float
    // С неопределенным размером
    vector<float> v2;

    // Вектор, состоящий из 10 элементов типа int
    // По умолчанию все элементы заполняются
нулями
    vector<int> v3(10, 0);
}
```

Методы Vector:

push_back() — добавить последний элемент

pop_back() — удалить последний элемент

clear() — удалить все элементы вектора

empty() — проверить вектор на пустоту

size() — возврат размера

```
std::vector<int> v = { 1,2,3,4 };
std::vector<int>::iterator iter = v.begin();  // получаем итератор
```

Ниже - пример работы алгоритма find с вектором.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm> // Алгоритм find
using namespace std;
<Функция make (создание контейнера символов) >

int main()
{
   cout << "Работа алгоритма find с вектором." << endl;
   vector<char> vector1 =
      make< vector<char> >("C++ is a better C");

   vector<char>::iterator where =
      find(vector1.begin(), vector1.end(), 'e');
   assert (*where == 'e' && *(where + 1) == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

List

Контейнер **list** представляет двухсвязный список. Для его использования необходимо подключить заголовочный файл **list**:

```
#include <list>
```

Создание списка:

```
std::list<int> list1;      // пустой список
std::list<int> list2(5);  // список list2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию

std::list<int> list3(5, 2);          // список list3 состоит из 5 чисел, каждое число равно 2
std::list<int> list4{ 1, 2, 4, 5 };   // список list4 состоит из чисел 1, 2, 4, 5
std::list<int> list5 = { 1, 2, 3, 5 }; // список list5 состоит из чисел 1, 2, 4, 5
std::list<int> list6(list4);         // список list6 - копия списка list4
std::list<int> list7 = list4;         // список list7 - копия списка list4
```

Получение элементов

В отличие от других контейнеров для типа list не определена операция обращения по индексу или функция at(), которая выполняет похожую задачу.

Тем не менее для контейнера list можно использовать функции **front()** и **back()**, которые возвращают соответственно первый и последний элементы.

Чтобы обратиться к элементам, которые находятся в середине (после первого и до последнего элементов), придется выполнять перебор элементов с помощью циклов или итераторов:

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

int main()
{
    std::list<int> numbers = { 1, 2, 3, 4, 5 };

    int first = numbers.front();  // 1
    int last = numbers.back();  // 5

    // перебор в цикле
    for (int n : numbers)
        std::cout << n << "\t";
    std::cout << std::endl;

    // перебор с помощью итераторов
    for (auto iter = numbers.begin(); iter != numbers.end(); iter++)
    {
        std::cout << *iter << "\t";
    }
    std::cout << std::endl;
    return 0;
}
```

Размер списка

Для получения размера списка можно использовать функцию **size()**:

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
int size = numbers.size();   // 5
```

Функция **empty()** позволяет узнать, пуст ли список. Если он пуст, то функция возвращает значение true, иначе возвращается значение false:

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
if (numbers.empty())
    std::cout << "The list is empty" << std::endl;
else
    std::cout << "The list is not empty" << std::endl;
```

С помощью функции resize() можно изменить размер списка. Эта функция имеет две формы:

- resize(n): оставляет в списке n первых элементов. Если список содержит больше элементов, то он усекается до первых n элементов. Если размер списка меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию

- resize(n, value): также оставляет в списке n первых элементов. Если размер списка меньше n, то добавляются недостающие элементы со значением value

```
std::list<int> numbers = { 1, 2, 3, 4, 5, 6 };
numbers.resize(4);  // оставляем первые четыре элемента -
numbers = {1, 2, 3, 4}

numbers.resize(6, 8);    // numbers = {1, 2, 3, 4, 8, 8}
```

Изменение элементов списка

Функция **assign()** позволяет заменить все элементы списка определенным набором. Она имеет следующие формы:

- **assign(il)**: заменяет содержимое контейнера элементами из списка инициализации il
- **assign(n, value)**: заменяет содержимое контейнера n элементами, которые имеют значение value
- **assign(begin, end)**: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы begin и end

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };

numbers.assign({ 21, 22, 23, 24, 25 }); // numbers = { 21, 22, 23, 24,
25 }

numbers.assign(4, 3);       // numbers = {3, 3, 3, 3}

std::list<int> values = { 6, 7, 8, 9, 10, 11 };
auto start = ++values.begin();  // итератор указывает на второй
элемент из values
auto end = values.end();
numbers.assign(start, end); //  numbers = { 7, 8, 9, 10, 11 }
```

Функция **swap()** обменивает значениями два списка:

```
std::list<int> list1 = { 1, 2, 3, 4, 5 };
std::list<int> list2 = { 6, 7, 8, 9};
list1.swap(list2);
// list1 = { 6, 7, 8, 9};
// list2 = { 1, 2, 3, 4, 5 };
```

Добавление элементов

Для добавления элементов в контейнер list применяется ряд функций.

- **push_back(val)**: добавляет значение val в конец списка
- **push_front(val)**: добавляет значение val в начало списка
- **emplace_back(val)**: добавляет значение val в конец списка
- **emplace_front(val)**: добавляет значение val в начало списка
- **emplace(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент
- **insert(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент
- **insert(pos, n, val)**: вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.
- **insert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.
- **insert(pos, values)**: вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

Функции push_back(), push_front(), emplace_back() и emplace_front():

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
numbers.push_back(23);  // { 1, 2, 3, 4, 5, 23 }
numbers.push_front(15); // { 15, 1, 2, 3, 4, 5, 23 }
numbers.emplace_back(24);   // { 15, 1, 2, 3, 4, 5, 23, 24 }
numbers.emplace_front(14);  // { 14, 15, 1, 2, 3, 4, 5, 23, 24 }
```

Добавление в середину списка с помощью функции **emplace()**:

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
auto iter = ++numbers.cbegin(); // итератор указывает на второй
элемент
numbers.emplace(iter, 8); // добавляем после первого элемента
numbers = { 1, 8, 2, 3, 4, 5};
```

Добавление в середину списка с помощью функции **insert()**:

```
std::list<int> numbers1 = { 1, 2, 3, 4, 5 };
auto iter1 = numbers1.cbegin(); // итератор указывает на первый элемент
numbers1.insert(iter1, 0); // добавляем начало списка
//numbers1 = { 0, 1, 2, 3, 4, 5};

std::list<int> numbers2 = { 1, 2, 3, 4, 5 };
auto iter2 = numbers2.cbegin(); // итератор указывает на первый элемент
numbers2.insert(++iter2, 3, 4); // добавляем после первого элемента три
четверки
//numbers2 = { 1, 4, 4, 4, 2, 3, 4, 5};

std::list<int> values = { 10, 20, 30, 40, 50 };
std::list<int> numbers3 = { 1, 2, 3, 4, 5 };
auto iter3 = numbers3.cbegin(); // итератор указывает на первый элемент
// добавляем в начало все элементы из values
numbers3.insert(iter3, values.begin(), values.end());
//numbers3 = { 10, 20, 30, 40, 50, 1, 2, 3, 4, 5};

std::list<int> numbers4 = { 1, 2, 3, 4, 5 };
auto iter4 = numbers4.cend();   // итератор указывает на позицию за
последним элементом
// добавляем в конец список из трех элементов
numbers4.insert(iter4, { 21, 22, 23 });
//numbers4 = { 1, 2, 3, 4, 5, 21, 22, 23};
```

Удаление элементов

Для удаления элементов из контейнера list могут применяться следующие функции:

clear(p): удаляет все элементы

pop_back(): удаляет последний элемент

pop_front(): удаляет первый элемент

erase(p): удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент

erase(begin, end): удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
numbers.pop_front();    // numbers = { 2, 3, 4, 5 }
numbers.pop_back();     // numbers = { 2, 3, 4 }
numbers.clear();    // numbers ={}

numbers = { 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // указатель на первый элемент
numbers.erase(iter);    // удаляем первый элемент
// numbers = { 2, 4, 5, 6 }

numbers = { 1, 2, 3, 4, 5 };
auto begin = numbers.begin(); // указатель на первый элемент
auto end = numbers.end();       // указатель на последний элемент
numbers.erase(++begin, --end);  // удаляем со второго элемента до последнего
//numbers = {1, 5}
```

Обобщенный алгоритм find и связанный список. Отличие от примера с вектором - для списка list не определена операция +, но определена ++.

```
#include <iostream>
#include <cassert>
#include <list>
#include <algorithm>
using namespace std;

<Функция make (создание контейнера символов)>

int main()
{
   cout << "Работа алгоритма find со списком." << endl;
   list<char> list1 = make<list<char>>("C++ is a better C");

   list<char>::iterator where =
      find(list1.begin(), list1.end(), 'e');
   list<char>::iterator next = where;
   ++next;
   assert(*where == 'e' && *next == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Использование алгоритма find с деком.

```
#include <iostream>
#include <cassert>
#include <deque>
#include <algorithm>
using namespace std;
<Функция make (создание контейнера символов)>

int main()
{
   cout << "Работа алгоритма find с деком." << endl;
   deque<char> dequel =
      make<deque<char>>("C++ is a better C");

   deque<char>::iterator where =
       find(dequel.begin(), dequel.end(), 'e');
   assert(*where == 'e' && *(where + 1) == 't');
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Map

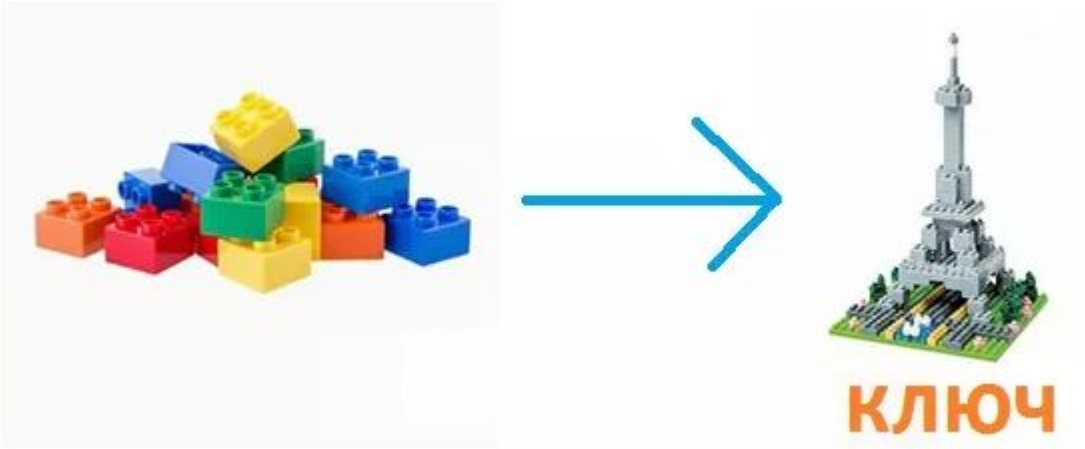
Что такое map

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 угодна. От обычной переменной до класса

```
1  mp1[0] = 7; // ключ - число
2
3  mp2["zero"] = 4;  // ключ - строка
4
5  pair <int, int> p = make_pair(1, 3);
6  mp3[p] = 3;  // ключ - пара
```

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



Как создать map

Сперва понадобится подключить соответствующую библиотеку:

```
#include <map>
```

Чтобы создать map нужно воспользоваться данной конструкцией:

```
map < <L>, <R> > <имя>;
```

<L> — этот тип данных будет относиться к значению ключа.
<R> — этот тип данных соответственно относится к значению.

```
map <string, int> mp; // пример
```

Также имеется возможность добавить значения при инициализации (C++ 11 и выше):

```
map <string, string> book = {{"Hi", "Привет"},
                             {"Student", "Студент"},
                             {"!", "!"}};

cout << book["Hi"];
```

Множества и словари

map (multimap) **– ассоциативный контейнер, который отсортированные список пар в соответствии с уникальным (неуникальным) ключом. Сортировка производится согласно функции Compare. Операции поиска, удаления и включения имеют логарифмическую сложность.**

std::multimap

```
template<
    class Key,
    class T,
    class Compare =std::less<Key>,
class Allocator = std::allocator<std::pair<const Key, T> >
> class multimap;
```

Итераторы для map

Итераторы для map

Использование итераторов одна из главных тем, если вам понадобится оперировать с этим контейнером. Создание итератора, как обычно происходит так:

```
map <тип данных> :: iterator <имя>;
```

С помощью его можно использовать две операции (**it — итератор**):

Чтобы обратится к ключу нужно сделать так: **it->first**.

Чтобы обратится к значению ячейки нужно сделать так: **it->second**.

Нельзя обращением к ключу **(...->first)** изменять его значение, а вот изменять таким образом значение ячейки **(...->second)** легко.

Нельзя использовать никакие арифметические операции над итератором.

Чтобы не писать циклы для увеличения итератора на большие значения, можно воспользоваться функцией **advance()**:

```
advance(it, 7);
advance(it, -5);
```

Она сдвигает указанный итератор вниз или вверх на указанное количество ячеек. В нашем случае он сначала увеличит на 7, а потом уменьшит на 5, в итоге получится сдвиг на две вверх.

```
#include <iostream>
#include <map>

using namespace std;
int main() {
   setlocale(0, "");
   map <int, int> mp;

   cout << "Введите количество элементов: "; int n; cin >> n;

   for (int i = 0; i < n; i++) {
      cout << i << ") "; int a; cin >> a;
      mp[a] = i;  // добавляем новые элементы
   }

   map <int, int> :: iterator it = mp.begin();
   cout << "А вот все отсортированно: " << endl;
   for (int i = 0; it != mp.end(); it++, i++) {  // выводим их
      cout << i << ") Ключ " << it->first << ", значение " << it->second << endl;
   }

   system("pause");
   return 0;
}
```

Методы map

- **insert**

Это функция вставки нового элемента.

```
mp.insert(make_pair(num_1, num_2));
```

num_1 — ключ.

num_2 — значение.

Мы можем сделать то же самое вот так:

```
mp[num_1] = num_2;
```

- **count**

Возвращает количество элементов с данным ключом. В нашем случае будет возвращать — 1 или 0.

Эта функция больше подходит для multimap, у которого таких значений может быть много.

```
mp[0] = 0;

mp.count(0); // 1
mp.count(3); // 0
```

Нужно помнить, что для строк нужно добавлять кавычки — count("Good")

- **find**

У этой функции основная цель узнать, есть ли *определенный ключ в контейнере.*

- Если он есть, то передать итератор на его местоположение.
- Если его нет, то передать итератор на конец контейнера.

```
#include <iostream>
#include <map>  // подключили библиотеку

using namespace std;

int main() {
  setlocale(0, "");
  map <string, string> book;
  book["book"] = "книга";

  map <string, string> :: iterator it, it_2;

  it = book.find("book");
  cout << it->second << endl;

  it_2 = book.find("books");

  if (it_2 == book.end()) {
      cout << "Ключа со значением 'books' нет";
  }

  system("pause");
  return 0;
}num_1, num_2));
```

- **erase**

Иногда приходится удалять элементы. Для этого у нас есть функция — **erase().**
Давайте посмотрим как она работает на примере:

```
map <string, string> passport;

passport["maxim"] = "Denisov";     // добавляем
passport["andrey"] = "Puzerevsky"; // новые
passport["dima"] = "Tilyupo";      // значения

cout << "Size: " << passport.size() << endl;

map <string, string> :: iterator full_name; // создали
итератор на passport

full_name = passport.find("andrey"); // находим ячейку
passport.erase(full_name);           // удаляем

cout << "Size: " << passport.size();
```

Множества и словари

set (multiset) **– ассоциативный контейнер, который содержит элементы, отсортированные в соответствии с уникальным (неуникальным) ключом. Сортировка производится с использованием функции Compare. Операции поиска, удаления и включения имеют логарифмическую сложность.**

std::set

```
template<
    class Key,
    class Compare = std::less <Key>,
    class Allocator = std::allocftor <Key>
> class set;
```

Обобщенный алгоритм merge.

Алгоритм merge используется для объединения двух отсортированных последовательностей в единую (отсортированную) последовательность.

Синтаксис вызова

```
merge(first1, last1, first2, last2, result);
```

Параметры

- first1 и last1 - итераторы начала и конца первой входной последовательности элементов некоторого типа T.
- first2 и last2 - итераторы начала и конца второй входной последовательности элементов того же типа T.
- result - итератор начала последовательности, в которой будет сохранен результат слияния входных последовательностей.

Результат

Предполагается, что две входные последовательности упорядочены в возрастающем порядке в соответствии с оператором < для типа T. Результат работы алгоритма будет содержать все элементы входных последовательностей, причем они также будут отсортированы в порядке возрастания.

Интерфейс шаблона функции merge достаточно гибок. Входная и выходная последовательности могут находиться в контейнерах разного типа.

```
#include <...>
using namespace std;

int main()
{
  cout << "merge с массивом, списком и деком" << endl;
  char s[] = "aeiou";
  int len = strlen(s);
  list<char> list1 = make<list<char>>("bcdfghjklmnpqrstvwxyz");
    // Инициализация deque1 26 символами x
  deque<char> deque1(26, 'x');
    // слияние s и list1, размещение результата в deque1
  merge(&s[0],&s[len],list1.begin(),list1.end(),
       deque1.begin());
  assert(deque1 ==
     make<deque<char>>("abcdefghijklmnopqrstuvwxyz"));
  cout << " ===== Ok!" << endl;
  return 0;
}
```

Алгоритм merge может использоваться для объединения отдельных частей последовательностей.

```
#include <...>
using namespace std;
int main()
{
  cout << "Объединение частей массива и дека"
       << " с размещением результата в списке" << endl;
  char s[] = "acegikm";
  deque<char> deque1 =
      make<deque<char>>("bdfhjlnopqrstuvwxyz");
    // Инициализация списка 26 символами x:
  list<char> list1(26, 'x');
    // Слияние первых 5 символов массива s с первыми
    // 10 символами из deque1, с размещением результата
    // в списке list1:
  merge(&s[0], &s[5], deque1.begin(), deque1.begin()+10,
      list1.begin());
  assert(list1 ==
      make<list<char>>("abcdefghijlnopqxxxxxxxxxxx"));
  cout << " ===== Ok!" << endl;
  return 0;
}
```

Алгоритмы стандартной библиотеки

Немодифицирующие операции над последовательностями:

all_of
any_of
none_of // Проверяют, является ли предикат верным (true) для всех (all_of), хотя бы одного из (any_of) или ни одного (none_of) из элементов в диапазоне
for_each // Применяет функцию к диапазону элементов
count
count_if // Возвращает количество элементов, удовлетворяющих определенным критериям
find
find_if
find_if_not // Находят первый элемент, удовлетворяющий определенным критериям
find_end // Ищет последнее вхождение подпоследовательности элементов в диапазон
find_first_of // Ищет в множестве элементов первое вхождение любого элемента другого множества

adjacent_find // Ищет в диапазоне два одинаковых смежных элемента

mismatch // Находит первую позицию, в которой два диапазона отличаются

equal // Определяет, одинаковы ли два множества элементов

lexicographical_compare // Возвращает истину, если один диапазон
// лексикографически меньше, чем другой

is_permutation // определяет, является ли последовательность
// перестановкой другой последовательности

search // Ищет первое вхождение последовательности элементов в диапазон

search_n // Ищет в диапазоне первую последовательность n
// одинаковых элементов, каждый из которых равен
// заданному значению

Модифицирующие операции над последовательностями:

copy
copy_if // Копирует ряд элементов
copy_n // Копирует ряд элементов в новое место
copy_backward // Копирует диапазон элементов в обратном порядке
move // перемещает диапазон элементов в новое место
move_backward // перемещает диапазон элементов в новое место в
// обратном порядке
fill // присваивает определенное значение набору элементов
fill_n // присваивает значение заданному числу элементов
transform // применяет функцию к различным элементам
generate // сохраняет результат функции в диапазоне
generate_n // сохраняет результат N раз примененной функции

remove
remove_if // удаляет элементы, удовлетворяющие определенным критериям

remove_copy
remove_copy_if // Копирует диапазон элементов опуская те, которые
// удовлетворяют определенным критериям

replace
replace_if // заменяет все значения, удовлетворяющие определенным
// критериям с другим значением
swap // обмен значения двух объектов
swap_ranges // обмен элементов в двух диапазонах
iter_swap // обмен элементов, на которые указывают итераторы
reverse // изменяет порядок элементов в диапазоне на обратный
reverse_copy // создает копию диапазон, который меняется на
// противоположную
rotate // вращает (сдвигает) последовательность элементов
// циклически до заданного элемента
rotate_copy // копирует и сдвигает в элементы диапазона
random_shuffle // перемешивает элементы на заданном диапазоне
// случайным образом
unique // удаляет все последовательные эквивалентные элементы
// кроме первого
unique_copy // создает копию некоторого диапазона элементов,
// который не содержит последовательных дубликатов

Операции разделения:

is_partitioned // определяет, разделен ли диапазон данным предикатом

partition // делит диапазон элементов на две группы
// т.е. образует два раздела в заданном диапазоне, размещая
// элементы, удовлетворяющие заданному условию, перед
// теми, которые этому условию не соответствуют.

partition_copy // копирует диапазон, разделяющий элементы на две группы

stable_partition // делит диапазон на две группы, сохраняя относительный
// порядок элементов

partition_point // находит точку разделения разделенного диапазона

Операции, относящиеся к упорядочиванию:

is_sorted // проверяет, является ли диапазон отсортированным в порядке возрастания
is_sorted_until // находит первый несортированный элемент в диапазоне
sort // сортирует диапазон в порядке возрастания
partial_sort // сортирует первые N элементов в диапазоне
partial_sort_copy // копирует и частично сортирует диапазон элементов
stable_sort // сортирует диапазон элементов при сохранении порядка между равными элементами
nth_element // помещает n-й элемент в позицию, которую он занимал бы после сортировки всего диапазона

Операции двоичного поиска (на отсортированных диапазонах) :

lower_bound // находит первый элемент диапазона больший чем заданное
// число или равный ему

upper_bound // находит первый элемент диапазона больший, чем
// заданное число

binary_search // определяет, находится ли элемент в некотором диапазоне

equal_range // возвращает набор элементов для конкретного ключа

Операции над множествами (на отсортированных диапазонах) :

merge // слияние двух отсортированных диапазонов

inplace_merge // слияние двух отсортированных диапазонов на месте

includes // возвращает истину, если один набор является
// подмножеством другого

set_difference // вычисляет разницу между двумя наборами

set_intersection // вычисляет пересечение двух множеств

set_symmetric_difference // вычисляет симметрическая разность между
// двумя наборами

set_union // объединяет два множества

Операции над пирамидой (кучей) :

is_heap // проверяет является ли данный диапазон пирамидой

is_heap_until // находит наибольший поддиапазон, который является кучей

make_heap // создает пирамиду из ряда элементов

push_heap // добавляет элемент в пирамиду

pop_heap // удаляет наибольший элемент из пирамиды

sort_heap // Сортировка элементов пирамиды

Некоторые алгоритмы (работа с числами) находятся в <numeric>:

Этот заголовочный файл содержит набор алгоритмов для выполнения определенных операций над последовательностями числовых значений.

Благодаря своей гибкости они также могут быть адаптированы для других видов последовательностей.

```
iota                  //  заполняет диапазон, последовательностью значений,
                      //     начиная с заданного стартового значения
accumulate            //     суммирует диапазон элементов
inner_product         //   вычисляет скалярное произведение двух диапазонов
adjacent_difference     //  вычисляет разницу между соседними элементами в
                          // диапазоне
partial_sum             //    вычисляет частичную сумму ряда элементов
```

Операции минимума/максимума:

max // Возвращает наибольший из двух аргументов
max_element // Возвращает наибольший элемент в диапазоне
min // Возвращает меньший из двух элементов
min_element // Возвращает наименьший элемент в диапазоне
minmax // Возвращает большее и меньшее из двух элементов
minmax_element // возвращает наименьший и наибольший элемент в диапазоне

Алгоритмы из библиотеки C :

<cstdlib>

qsort // Сортирует диапазон элементов любого типа

bsearch // Ищет в массиве элемент любого типа

Предикат. Функция-предикат и функтор

Предикат, это нечто функциональное, возвращающее тип bool. Есть две возможности организации такой функции: собственно функция-предикат и функтор (объект-функция).

Однако, объекты-функции (функторы) гораздо предпочтительнее. Причина проста - объекты функций обеспечивают **более эффективный** код.

В книге Скотта Мейерса на этот случай приводится пример с функцией std::sort - использование объектов функций всегда работает быстрее, выигрыш в скорости может составлять от 50% до 160%.

Объясняется все тривиально - при использовании объекта функции компилятор способен **встроить** передаваемую функцию **в тело алгоритма (inline)**, а при использовании обычной функции встраивания не производится.

```
bool f (const int& x,const int& y){ return x<y;}          // функция-предикат

// создаем синоним типа - указатель на функцию сравнения
typedef     bool (*pf) (const int&  ,const int&  ) ;

struct pred{                                          // функтор
          bool operator () (const int& x, const int& y){ return x>y; }
};
```

Функтор

Есть большое множество вариантов их практического применения:

- Необходим вызов обычной функции или функции-члена класса.
- Алгоритм может требовать бинарную или унарную функцию.
- Требуется передача в функцию дополнительных параметров.
- Объект, которому принадлежит функция, может быть константным или неконстантным.
- Аргументы в функцию могут передаваться по значению, указателем или по ссылке.
- Функция может быть реализована в классе.
- Реализация может потребовать использования нескольких функций.

any_of

template <class InputIterator, class **UnaryPredicate**>
bool **any_of** (InputIterator first, InputIterator last, **UnaryPredicate** pred);

Проверяет, соответствует ли какой-либо элемент в диапазоне *условию*

Возвращает true, если **предикат** pred возвращает true **для любого** из элементов в диапазоне [first,last), и false в противном случае.

Если [first, last) это пустой диапазон, функция возвращает false.
Поведение этого шаблона функции эквивалентно:

```
template<class InputIterator, class UnaryPredicate>
bool any_of (InputIterator first, InputIterator last, UnaryPredicate pred) {
  while ( first!=last ) {
    if (pred (*first) ) return true;
    ++first;
  }
  return false;
}
```

```
#include <iostream>     // std::cout
#include <algorithm>    // std::any_of
#include <array>        // std::array

int main () {
  std::array <int,7>  foo  =  {0,1,-1,3,-3,5,-5};

  if ( std::any_of ( foo.begin(), foo.end(),
                       [ ](int i) {return i<0;}
                     )
     )
    std::cout << "There are negative elements in the range.\n";

  return 0;
}
```

Output:
There are negative elements in the range.

Алгоритм accumulate

При вызове `accumulate(first, last, init)`, где first и last - итераторы, а init - некоторое значение, алгоритм accumulate добавляет к init значения в позициях от first до last (не включая значение в последней позиции) и возвращает получившуюся сумму.

Пример: программа расчета суммы элементов вектора.

```
#include <...>
#include <numeric> // Алгоритм accumulate
using namespace std;

int main()
{
   cout << "Демонстрация функции accumulate." << endl;
   int x[5] = {2, 3, 5, 7, 11};
      // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int sum = accumulate(v1.begin(), v1.end(), 0);
   assert (sum == 28);
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Рассмотрим, как функция accumulate использует итераторы.

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init)
{
  while(first != last)
  {
    init = init + *first;
    ++first;
  }
  return init;
}
```

Типы итераторов STL и поддерживаемые ими операции

Функциональные объекты

Из определения шаблона accumulate в предыдущем примере видно, что для элементов контейнера задан **оператор +** (в выражении init = init + *first). Однако абстрактное понятие **накопления (accumulation)** применимо не только к суммированию; можно так же накапливать, к примеру, произведение значений элементов. Потому в STL определена еще один шаблон для функции accumulate

```
template <typename Inputlterator, typename T,
          typename BinaryOperation>
T accumulate(InputIterator first, Inputlterator last,
          T init, BinaryOperation binary_op)
{
   while(first != last)
   {
     init = binary_op(init, *first);
     ++first;
   }
   return init;
}
```

Пример. Использование accumulate для расчета произведения элементов

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>
using namespace std;

int mult(int x, int y) { return x*y; };

int main()
{
   cout << "Расчет произведения элементов вектора" << endl;
   int x[5] = {2, 3, 5, 7, 11};
     // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int product = accumulate(v1.begin(), v1.end(), 1, mult);
   assert(product == 2310);
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Пример. Расчет произведения с применением функционального объекта

```
class multiply
{
  public:
    int operator()(int x, int y) const { return x*y; }
};

int main()
{
   cout << "Использование алгоритма accumulate и "
        << "функционального объекта multiply для "
        << " вычисления произведения." << endl;
   int x[5] = {2, 3, 5, 7, 11};
     // Инициализация вектора элементами от x[0] до x[4]:
   vector<int> v1(&x[0], &x[5]);
   int p = accumulate(v1.begin(), v1.end(), 1, multiply());
   assert(p == 2310);
   cout << " ===== Ok!" << endl;
   return 0;
}
```

Пример использования словаря

Составить программу учета заявок на авиабилеты.

Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок, упорядоченных по пунктам назначения;
- вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «пункт назначения».

Создаём структуру *TicketInfo,* который будет содержать все необходимые поля для заявки, а также этот класс будет использоваться при описании словаря *multimap.*

Описание класса

```
struct TicketInfo
{
    long ticket_id;  // идентификатор билета
    long n_reis;     // номер рейса
    string dest;     // пункт назначения
    string fio;      // ФИО пассажира
    string date;     // дата вылета
};
#include <map>
multimap<long, TicketInfo> m_Tickets;
```

int add_ticket()

```
int add_ticket()
{   setlocale(LC_ALL,"Russian");
    long reis_n, id_ticket;
    char ch[30]; string str;
    TicketInfo tickets;
    multimap<long, TicketInfo>::iterator it;
    bool yes(false), no(false), item_not_found(false);
    char user_respond[2], respond_no[ ] = "n",
respond_yes[ ] = "y";
    while (strcmp(respond_no, user_respond) != 0)
    {   system("cls");
    cout << "Добавление заявки: (en words only)\n";
        cout << "Введите номер заявки: ";
        cin >> id_ticket;
        it = m_Tickets.find(id_ticket);
        if (it != m_Tickets.end())  и т.д.
```