## **STL**

## **ТИПЫ КОНТЕЙНЕРОВ**

Для управления наборами объектов в стандартной библиотеке C++ определены контейнеры. Контейнер представляет коллекцию объектов определенного типа и позволяет и управлять доступом к этим элементам. В С++ есть два типа контейнеров: ассоциативные и последовательные.

**Последовательный контейнер** (sequential container) хранит элементы последовательно, элементы располагаются друг рядом с другом. Однако меняется их порядок в зависимости от конкретного контейнера.

Типы последовательных контейнеров:

* **array**: коллекция фиксированного размера.
* Поддерживает произвольный доступ к любому элементу в контейнере.
* Добавлять или удалять элементы из контейнера нельзя.
* **vector**: коллекция переменного размера.
  + Поддерживает произвольный доступ к любому элементу в контейнере.
  + Обеспечивает добавление и удаление элементов из любого места контейнера.
* **deque**: двусторонняя очередь.
  + Поддерживает произвольный доступ к любому элементу в контейнере.
  + Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
* **list**: двухсвязный список
  + Поддерживает только последовательный двухнаправленный доступ к элементам.
  + Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
* **forward\_list**: односвязный список.
  + Поддерживает только однонаправленный последовательный доступ к элементам.
  + Обеспечивает удаление и добавление элементов в начале и в конце контейнера.

Таким образом, стандартная библиотека C++ по умолчанию содержит ряд контейнеров, которые представляют определенные структуры данных.

Все они имеют как некоторые общие, так и специфические возможности. За исключением класса array все они поддерживают добавление и удаление элементов.

Основное различие между ними состоит в том, как они обеспечивают добавление и удаление элементов, а также доступ к элементам в контейнере. И в зависимости от ситуации и потребностей можно использовать тот или иной тип контейнеров.

Например, если надо иметь возможность получить произвольный элемент контейнера, то применяется array или vector (с list или forwarded\_list может потребоваться пробегаться по списку, чтобы найти нужный элемент).

Если же надо иметь возможность добавлять или удалить элементы в середине контейнера, то можно применять list или forwarded\_list, что с вектором сложнее сделать. Однако наиболее часто используется вектор, как более гибкий тип данных. Другие типы контейнеров применяются гораздо реже.

### **АДАПТЕРЫ КОНТЕЙНЕРОВ**

Кроме последовательных контейнеров есть так называемые **адаптеры контейнеров** (container adaptor). Технически они не являются контейнерами, а инкапсулируют один из вышеописанных контейнеров (например, вектор) и позвляют работать с этими контейнерами определенным образом. Это следующие типы

* **std::stack<>**: представляет структуру данных "стек"
* **std::queue<>**: представляет структуру данных "очередь"
* **std::priority\_queue<>**: также представляет очередь, но при этом ее элементы имеют приоритеты

### **АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ**

**Ассоциативные контейнеры** (associative containers) представляют такие контейнеры, где с каждым элементом ассоциирован некоторый ключ, и этот ключ применяется для доступа к элементу в контейнере.

### В С++ ассоциативные контейнеры представлены множествами (set) и картами/словарями (map).

### **ВЕКТОР**

Вектор представляет контейнер, который содержит коллекцию объектов одного типа. Для работы с векторами необходимо включить заголовок:

|  |  |
| --- | --- |
| 1 | #include <vector> |

Определим простейший вектор:

|  |  |
| --- | --- |
| 1 | std::vector<**int**> numbers; |

В угловых скобках указывается тип, объекты которого будут храниться в векторе. То есть вектор numbers хранит объекты типа int. Однако такой вектор пуст. Он не содержит никаких элементов.

Но мы можем инициализировать вектор одним из следующих способов:

|  |
| --- |
| std::vector<**int**> v1;              // пустой вектор  std::vector<**int**> v2(v1);          // вектор v2 - копия вектора v1  std::vector<**int**> v3 = v1;         // вектор v3 - копия вектора v1  std::vector<**int**> v4(5);  // вектор v4 состоит из 5 чисел, каждое число = 0  std::vector<**int**> v5(5, 2); // вектор v5 состоит из 5 чисел, каждое число = 2  std::vector<**int**> v6{1, 2, 4, 5};  // вектор v6 состоит из чисел 1, 2, 4, 5  std::vector<**int**> v7 = {1, 2, 3, 5}; // вектор v7 состоит из чисел 1, 2, 3, 5 |
|  |

Важно понимать отличие в данном случае круглых скобок от фигурных:

|  |
| --- |
| std::vector<**int**> v1(5);       // вектор состоит из 5 чисел, каждое число в векторе равно 0  std::vector<**int**> v2{5};       // вектор состоит из одного числа, которое = 5  std::vector<**int**> v3(5, 2);    // вектор состоит из 5 чисел, каждое число = 2  std::vector<**int**> v4{5, 2};    // вектор состоит из двух чисел 5 и 2 |

При этом можно хранить в векторе элементы только одного типа, который указан в угловых скобках. Значения других типов в вектор сохранить нельзя, как например, в следующем случае:

|  |  |
| --- | --- |
| 1 | std::vector<**int**> v{5, "hi"}; |

### **ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ И ИХ ПЕРЕБОР**

Для обращения к элементам вектора можно использовать разные способы:

* **[index]**: получение элемента по индексу (также, как и в массивах), индексация начинается с нуля
* **at(index)**: функция возращает элемент по индексу
* **front()**: возвращает первый элемент
* **back()**: возвращает последний элемент

Выполним перебор вектора и получим некоторые его элементы:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers {1, 2, 3, 4, 5};    **int** first = numbers.front();    // 1  **int** last = numbers.back();      // 5  **int** second = numbers[1];        // 2      std::cout << "first: " << first << std::endl;      std::cout << "second: " << second << std::endl;      std::cout << "last: " << last << std::endl;        numbers[0] = 6;  // изменяем значение  **for**(**int** n : numbers)          std::cout << n << "\t"; // 6  2  3  4  5        std::cout << std::endl;  } |

При этом следует учитывать, что индексация не добавляет элементов. Например, если вектор содержит 5 элементов, то мы не можем обратиться к шестому элементу:

|  |
| --- |
| std::vector<**int**> numbers {1, 2, 3, 4, 5};  numbers[5] = 9; |
|  |

При таком обращении результат неопределен. Некоторые комиляторы могут генерировать ошибку, некоторые продолжат работать, но даже в этом случае такое обращение будет ошибочно, и оно в любом случае не добавит в вектор шестой элемент.

Чтобы избежать подобных ситуаций, можно использовать функцию **at()**, которая хотя также возвращает элемент по индексу, но при попытке обращения по недопустимому индексу будет генерировать исключение **out\_of\_range**:

|  |
| --- |
| #include <iostream>  #include <vector>  #include <stdexcept>    **int** main()  {      std::vector<**int**> numbers { 1, 2, 3, 4, 5};  **try**      {  **int** n = numbers.at(8);      }  **catch** (std::out\_of\_range e)      {          std::cout << "Incorrect index" << std::endl;      }  } |

**Итераторы** обеспечивают доступ к элементам контейнера и представляют реализацию распространенного паттерна объектно-ориентированного программирования "Iterator". С помощью итераторов очень удобно перебирать элементы. В C++ итераторы реализуют общий интерфейс для различных типов контейнеров, что позволяет использовать единой подход для обращения к элементам разных типов контейнеров.

Стоит отметить, что итераторы имеют только контейнеры, адаптеры контейнеров — типы **std::stack**, **std::queue** и **std::priority\_queue** итераторов не имеют.

Итератор описывается типом **iterator**. Для каждого контейнера конкретный тип итератора будет отличаться. Так, итератор для контейнера list<int> представляет тип **list<int>::iterator**, а итератор контейнера vector<int> представляет тип **vector<int>::iterator** и так далее. Однако общий функционал, который применяется для доступа к элементам, будет аналогичен.

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:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers{ 1,2,3,4 };      std::vector<**int**>::iterator iter = numbers.begin();  // получаем итератор  } |

В данном случае создается вектор - контейнер типа vector, который содержит значения типа int. И этот контейнер инициализируется набором {1, 2, 3, 4}. И через метод begin() можно получить итератор для этого контейнера. Причем этот итератор будет указывать на первый элемент контейнера.

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
* **iter + n**: возвращает итератор, который смещен от итератора iter на n позиций вперед
* **iter - n**: возвращает итератор, который смещен от итератора iter на n позиций назад
* **iter += n**: перемещает итератор на n позиций вперед
* **iter -= n**: перемещает итератор на n позиций назад
* **iter1 - iter2**: возвращает количество позиций между итераторами iter1 и iter2
* **>, >=, <, <=**: операции сравнения. Один итератор больше другого, если указывает на элемент, который ближе к концу

Стоит отметить, что итераторы не всех контейнеров поддерживают все эти операции.

Итераторы для типов **std::forward\_list**, **std::unordered\_set** и  **std::unordered\_map** не поддерживают операции --, -= и -. (поскольку std::forward\_list - однонаправленный список, где каждый элемент хранит указатель только на следующий элемент)

Итераторы для типа **std::list** поддерживают операции инкремента и декремента, но не поддерживаются операции +=, -=, + и -. Те же ограничения имеют итераторы контейнеров **std::map** и std::set.

Операции +=, -=, +, -, <, <=, >, >= и <=> поддерживаются только итераторами произвольного доступа (итераторы контейнеров **std::vector**, **array** и **deque**)

### **ПОЛУЧЕНИЕ И ИЗМЕНЕНИЕ ЭЛЕМЕНТА КОНТЕЙНЕРА**

Поскольку итератор по сути представляет указатель на определенный элемент, то через этот указатель мы можем получить текущий элемент итератора и изменить его значение:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers{ 1,2,3,4 };      auto iter { numbers.begin() };  // получаем итератор        // получаем элемент, на который указывает итератор      std::cout << \*iter << std::endl;    // 1      // изменяем элемент      \*iter = 125;      // проверяем изменение элемента      std::cout << numbers[0] << std::endl;    // 125  } |

После получения итератора он будет указывать на первый элемент контейнера. То есть при выражении \*iter возвратит первый элемент вектора.

Прибавляя или отнимая определенное число, можно переместить итератор вперед или назад на определенное количество элементов:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers{ 10, 20, 30, 40 };      auto iter { numbers.begin() };  // получаем итератор        // переходим на 1 элемент вперед ко 2-му элементу      ++iter;      std::cout << \*iter << std::endl;    // 20        // переходим на 2 элемента вперед к 4-му элементу      iter +=2;      std::cout << \*iter << std::endl;    // 40        // переходим назад на 3 элемента к 1-му элементу      iter = iter - 3;      std::cout << \*iter << std::endl;    // 10  } |

Опять же повторю, что стоит учитывать, что не все операции поддерживаются итераторами всех контейнеров.

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ВЕКТОР**

Для добавления элементов в вектор применяется функция **push\_back()**, в которую передается добавляемый элемент:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers;     // пустой вектор      numbers.push\_back(5);      numbers.push\_back(3);      numbers.push\_back(10);  **for**(**int** n : numbers)          cout << n << "\t";      // 5    3   10        std::cout << std::endl;  } |

Векторы являются динамическими структурами в отличие от массивов, где мы скованы его заданым размером. Поэтому мы можем динамически добавлять в вектор новые данные.

Функция **emplace\_back()** выполняет аналогичную задачу - добавляет элемент в конец контейнера:

|  |
| --- |
| std::vector<**int**> numbers{ 1, 2, 3, 4, 5 };  numbers.emplace\_back(8);    // numbers = { 1, 2, 3, 4, 5, 8 }; |
|  |

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ НА ОПРЕДЕЛЕННУЮ ПОЗИЦИЮ**

Ряд функций позволяет добавлять элементы на определенную позицию.

* **emplace(pos, value)**: вставляет элемент value на позицию, на которую указывает итератор pos
* **insert(pos, value)**: вставляет элемент value на позицию, на которую указывает итератор pos, аналогично функции emplace
* **insert(pos, n, value)**: вставляет n элементов value начиная с позиции, на которую указывает 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
* **insert(pos, values)**: вставляет список значений начиная с позиции, на которую указывает итератор pos

Функция emplace:

|  |
| --- |
| std::vector<**int**> numbers{ 1, 2, 3, 4, 5 };  auto iter = numbers.cbegin();   // константный итератор указывает на первый элемент  numbers.emplace(iter + 2, 8); // добавляем после второго элемента  numbers = { 1, 2, 8, 3, 4, 5}; |

Функция insert:

|  |
| --- |
| std::vector<**int**> numbers1{ 1, 2, 3, 4, 5 };  auto iter1 = numbers1.cbegin(); // константный итератор указывает на первый элемент  numbers1.insert(iter1 + 2, 8); // добавляем после второго элемента  //numbers1 = { 1, 2, 8, 3, 4, 5};    std::vector<**int**> numbers2 { 1, 2, 3, 4, 5 };  auto iter2 = numbers2.cbegin(); // константный итератор указывает на первый элемент  numbers2.insert(iter2 + 1, 3, 4); // добавляем после первого элемента три четверки  //numbers2 = { 1, 4, 4, 4, 2, 3, 4, 5};    std::vector<**int**> values { 10, 20, 30, 40, 50 };  std::vector<**int**> numbers3 { 1, 2, 3, 4, 5 };  auto iter3 = numbers3.cbegin(); // константный итератор указывает на первый элемент  // добавляем после первого элемента три первых элемента из вектора values  numbers3.insert(iter3 + 1, values.begin(), values.begin() + 3);  //numbers3 = { 1, 10, 20, 30, 2, 3, 4, 5};    std::vector<**int**> numbers4 { 1, 2, 3, 4, 5 };  auto iter4 = numbers4.cend();   // константный итератор указывает на позицию за последним элементом  // добавляем в конец вектора numbers4 элементы из списка { 21, 22, 23 }  numbers4.insert(iter4, { 21, 22, 23 });  //numbers4 = { 1, 2, 3, 4, 5, 21, 22, 23}; |

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Если необходимо удалить все элементы вектора, то можно использовать функцию **clear**:

|  |
| --- |
| std::vector<**int**> v { 1,2,3,4 };  v.clear(); |

Функция **pop\_back()** удаляет последний элемент вектора:

|  |
| --- |
| std::vector<**int**> v { 1,2,3,4 };  v.pop\_back();       // v = { 1,2,3 } |

Если нужно удалить элемент из середины или начала контейнера, применяется функция **std::erase()**, которая имеет следующие формы:

* **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

Применение функции:

|  |
| --- |
| std::vector<**int**> numbers1 { 1, 2, 3, 4, 5, 6 };  auto iter = numbers1.cbegin(); // указатель на первый элемент  numbers1.erase(iter + 2);   // удаляем третий элемент  // numbers1 = { 1, 2, 4, 5, 6 }    std::vector<**int**> numbers2 = { 1, 2, 3, 4, 5, 6 };  auto begin = numbers2.cbegin(); // указатель на первый элемент  auto end = numbers2.cend();     // указатель на последний элемент  numbers2.erase(begin + 2, end - 1); // удаляем с третьего элемента до последнего  // numbers2 = {1, 2, 6} |

Также начиная со стандарта С++20 в язык была добавлена функция **std::erase()**. Она не является частью типа vector. В качестве первого параметра она принимает вектор, а в качестве второго - элемент, который надо удалить:

|  |
| --- |
| std::vector<**int**> numbers3 { 1, 2, 3, 1, 5, 6 };  std::erase(numbers3, 1); // numbers3 = { 2, 3, 4, 5, 6 } |

В данном случае удаляем из вектора numbers3 все вхождения числа 1.

### **РАЗМЕР ВЕКТОРА**

С помощью функции **size()** можно узнать размер вектора, а с помощью функции **empty()** проверить, путой ли вектор:

|  |
| --- |
| #include <iostream>  #include <vector>    **int** main()  {      std::vector<**int**> numbers{1, 2, 3};  **if**(numbers.empty())          std::cout << "Vector is empty" << std::endl;  **else**          std::cout << "Vector has size " << numbers.size() << std::endl;  } |

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

* **resize(n)**: оставляет в векторе n первых элементов. Если вектор содержит больше элементов, то его размер усекается до n элементов. Если размер вектора меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию
* **resize(n, value)**: также оставляет в векторе n первых элементов. Если размер вектора меньше n, то добавляются недостающие элементы со значением value

Применение функции:

|  |
| --- |
| std::vector<**int**> numbers1 { 1, 2, 3, 4, 5, 6 };  numbers1.resize(4); // оставляем первые четыре элемента - numbers1 = {1, 2, 3, 4}    numbers1.resize(6, 8);   // numbers1 = {1, 2, 3, 4, 8, 8} |

Важно учитывать, что применение функции resize может сделать некорректными все итераторы, указатели и ссылки на элементы.

### **ИЗМЕНЕНИЕ ЭЛЕМЕНТОВ ВЕКТОРА**

Функция **assign()** позволяет заменить все элементы вектора определенным набором:

|  |  |
| --- | --- |
| 1  2 | std::vector<std::string> langs = { "Java", "JavaScript", "C"};  langs.assign(4, "C++"); // langs = {"C++", "C++", "C++", "C++"} |

В данном случае элементы вектора заменяются набором из четырех строк "C++".

Также можно передать непосредственно набор значений, который заменит значения вектора:

|  |
| --- |
| std::vector<std::string> langs{ "Java", "JavaScript", "C"};  langs.assign({ "C++", "C#", "C"}); // langs = { "C++", "C#", "C"} |

Еще одна функция - **swap()** обменивает значения двух контейнеров:

|  |
| --- |
| std::vector<std::string> clangs { "C++", "C#", "Java" };  std::vector<std::string> ilangs { "JavaScript", "Python", "PHP"};  clangs.swap(ilangs);    // clangs = { "JavaScript", "Python", "PHP"};  **for**(std::string lang : clangs)  {      std::cout << lang <<  std::endl;  } |

### **СРАВНЕНИЕ ВЕКТОРОВ**

Векторы можно сравнивать - они поддерживают все операции сравнения: <, >, <=, >=, ==, !=. Сравнение контейнеров осуществляется на основании сравнения пар элементов на тех же позициях. Векторы равны, если они содержат одинаковые элементы на тех же позициях. Иначе они не равны:

|  |
| --- |
| std::vector<**int**> v1 {1, 2, 3};  std::vector<**int**> v2 {1, 2, 3};  std::vector<**int**> v3 {3, 2, 1};    **bool** v1v2 = v1 == v2;   // true  **bool** v1v3 = v1 != v3;   // true  **bool** v2v3 = v2 == v3;   // false |

## **ARRAY**

Контейнер **array** из одноименного модуля <array> представляет аналог массива. Он также имеет фиксированный размер.

### **ОПРЕДЕЛЕНИЕ И ИНИЦИАЛИЗАЦИЯ**

Для создания объекта **array** в угловых скобках после названия типа необходимо передать его тип и размер:

|  |
| --- |
| #include <array>    **int** main()  {      std::array<**int**, 5> numbers;       // состоит из 5 чисел int  } |

В данном случае определен объект array из 5 чисел типа int. По умолчанию все элементы контейнера имеют неопределенные значения.

Чтобы инициализировать контейнер определенными значениями, можно использовать инициализатор - в фигурных скобках передать значения элементам контейнера:

|  |
| --- |
| std::array<**int**, 5> numbers {};    // состоит из 5 нулей |

В данном случае пустой инициализатор инициализирует все элементы контейнера numbers нулями. Также можно указать конкретные значения для элементов:

|  |
| --- |
| std::array<**int**, 5> numbers {2, 3, 4, 5, 6}; |

Фиксированный размер накладывает ограничение на инициализацию: количество передаваемых контейнеру элементов не должно превышать его размер. Можно передать меньше значений, которые будут переданы первым элементам контейнера, а остальные элементы получат значения по умолчанию (например, для целочисленных типов это число 0):

|  |  |
| --- | --- |
|  | std::array<**int**, 5> numbers {2, 3, 4};     // {2, 3, 4, 0, 0} |

Однако если при инициализации мы предадим большее количество элементов, нежели размер контейнера, то мы столкнемся с ошибкой.

Стоит отметить, что начиная со стандарта C++17 при инициализации можно не указывать тип и количество элементов - компилятор выводит это автоматически исходя из списка инициализации:

|  |  |
| --- | --- |
|  | std::array numbers {2, 3, 4, 5, 6}; |

Однако в этом случае в списке инициализации в фигурных скобках должно быть, как минимум одно значение.

### **ДОСТУП К ЭЛЕМЕНТАМ**

Для доступа к элементам контейнера array можно применять тот же синтаксис, что при работе с массивами - в квадратных скобках указывать индекс элемента, к которому идет обращение:

|  |
| --- |
| #include <array>  #include <iostream>    **int** main()  {      std::array<**int**, 5> numbers {2, 3, 4, 5, 6};      // получаем значение элемента  **int** n = numbers[2];      std::cout << "n = " << n << std::endl; // n = 4      // меняем значение элемента      numbers[2] = 12;      std::cout << "numbers[2] = " << numbers[2] << std::endl; // numbers[2] = 12  } |

### **ПЕРЕБОР КОНТЕЙНЕРА**

С помощью стандартных циклов можно перебрать контейнер array:

|  |
| --- |
| #include <iostream>  #include <array>  #include <string>    **int** main()  {  **const** unsigned n = 5;      std::array<std::string, n> people { "Tom", "Alice", "Kate", "Bob", "Sam" };        // обращение через индексы  **for**(**int** i{}; i < n; i++)      {          std::cout << people[i] << std::endl;      }      std::cout << std::endl;      // перебор последовательности  **for** (auto person : people)      {          std::cout << person << std::endl;      }  } |

### **ОСНОВНЫЕ ФУНКЦИИ ARRAY**

В контейнер array нельзя добавлять новые элементы, так же, как и удалять уже имеющиеся. Основные функции типа array, которые мы можем использовать:

* **size()**: возвращает размер контейнера
* **at(index)**: возвращает элемент по индексу index
* **front()**: возвращает первый элемент
* **back()**: возвращает последний элемент
* **fill(n)**: присваивает всем элементам контейнера значение n

Применение методов:

|  |
| --- |
| #include <iostream>  #include <array>  #include <string>    **int** main()  {      std::array<std::string, 3> people { "Tom", "Bob", "Sam" };      std::string second = people.at(1);    // Bob      std::string first = people.front();  // Tom      std::string last = people.back();    // Sam        std::cout << second << std::endl;   // Bob      std::cout << first << std::endl;    // Tom      std::cout << last << std::endl;     // Sam        // присваиваем всем элементам "Undefined"      people.fill("Undefined");  // people = { "Undefined", "Undefined", "Undefined" }      // проверяем  **for** (**int** i{}; i< people.size(); i++)      {          std::cout << people[i] << std::endl;      }  } |

Несмотря на то, что объекты array похожи на обычные массивы, тип array более гибок. Например, мы не можем присваивать одному массиву напрямую значения второго массива. В то же время объекту array мы можем передавать данные другого объекта array:

|  |
| --- |
| std::array<**int**, 5> numbers1 { 1, 2, 3, 4, 5 };  std::array<**int**, 5> numbers2 = numbers1;       // так можно сделать    **int** nums1[] = { 1,2,3,4,5 };  //int nums2[] = nums1;      // так нельзя следать |

Также мы можем сравнивать два контейнера array:

|  |
| --- |
| std::array<**int**, 5> numbers1 { 1, 2, 3, 4, 5 };  std::array<**int**, 5> numbers2 { 1, 2, 3, 4, 5 };    std::cout << std::boolalpha << (numbers1 == numbers2) << std::endl;       // true  std::cout << std::boolalpha << (numbers1 != numbers2) << std::endl;       // false  std::cout << std::boolalpha << (numbers1 > numbers2) << std::endl;         // false  std::cout << std::boolalpha << (numbers1 < numbers2) << std::endl;     // false |

Два контейнера сравниваются поэлементно. Так, в примере выше очевидно, что контейнеры numbers1 и numbers2 равны. Тогда как сравнение массивов начиная со стандарта C++20 объявлено устаревшим.

## **LIST**

Контейнер **list** представляет двухсвязный список, то есть такой список, где каждый элемент имеет указатели на предыдущий и последовательный элемент. Благодаря чему мы можем перемещаться по списку как вперед, так и назад. Для использования списка необходимо подключить заголовочный файл **list**.

Создание списка:

|  |
| --- |
| #include <list>    **int** main()  {      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        std::cout << "First: " << first << std::endl;      std::cout << "Last: " << last << std::endl;        // 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;  } |

### **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

Применение функции:

|  |
| --- |
| 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

Применение функции:

|  |
| --- |
| 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

Применение функций:

|  |
| --- |
| 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, 3, 4, 5 }    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} |

## **FORWARD\_LIST**

Контейнер **forward\_list** представляет односвязный список, то есть такой список, где каждый элемент хранит указатель на следующий элемент. Для использования данного типа списка необходимо подключить заголовочный файл **forward\_list**.

Создание односвязного списка:

|  |
| --- |
| std::forward\_list<**int**> list1;                     // пустой список  std::forward\_list<**int**> list2(5);                  // list2 имеет 5 элементов, каждый из которых имеет значение по умолчанию  std::forward\_list<**int**> list3(5, 2);               // list3 состоит из 5 чисел, каждое число равно 2  std::forward\_list<**int**> list4{ 1, 2, 4, 5 };       // list4 состоит из чисел 1, 2, 4, 5  std::forward\_list<**int**> list5 = { 1, 2, 3, 4, 5 }; // list5 состоит из чисел 1, 2, 3, 4, 5  std::forward\_list<**int**> list6(list4);              // list6 - копия списка list4  std::forward\_list<**int**> list7 = list4;             // list7 - копия списка list4  std::forward\_list<**int**> list8({ 1, 2, 3, 4, 5, 6 }); // list8  состоит из чисел 1, 2, 3, 4, 5, 6 |

### **ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ**

Напрямую в списке forward\_list можно получить только первый элемент. Для этого применяется функция **front()**. Для перебора элементов также можно использовать цикл:

|  |
| --- |
| #include <iostream>  #include <forward\_list>    **int** main()  {      std::forward\_list<**int**> numbers { 1, 2, 3, 4, 5 };    **int** first {numbers.front()};    // 1      std::cout << "First: " << first << std::endl;    **for** (**int** n : numbers)          std::cout << n << "\t";      std::cout << std::endl;  } |

Также для перебора и получения элементов можно использовать итераторы

|  |
| --- |
| #include <iostream>  #include <forward\_list>    **int** main()  {      std::forward\_list<**int**> numbers{ 1, 2, 3, 4, 5 };        auto current = numbers.begin(); // итератор на начало списка      auto end = numbers.end();       // указатель на конец списка  **while** (current != end)      {          std::cout << \*current << "\t";          current++;      }      std::cout << std::endl;  } |

Причем класс forward\_list добавляет ряд дополнительных функций для получения итераторов: **before\_begin()** и **cbefore\_begin()**. Обе функции возвращают итератор (вторая функция возвращает константный итератор const\_iterator) на несуществующий элемент непосредственно перед началом списка. К значению по этому итератору обратиться нельзя.

|  |
| --- |
| #include <iostream>  #include <forward\_list>    **int** main()  {      std::forward\_list<**int**> numbers{ 1, 2, 3, 4, 5 };        auto prev = numbers.before\_begin();      auto end = numbers.end();  **while** (++prev != end)      {          std::cout << \*prev << "\t";      }      std::cout << std::endl;  } |

### **РАЗМЕР СПИСКА**

По умолчанию класс forward\_list не определяет никаких функций, которые позволяют получить размер контейнера. В этом классе только функция **max\_size()**, которая позволяет получить масимальный размер контейнера.

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

|  |
| --- |
| std::forward\_list<**int**> numbers{ 1, 2, 3, 4, 5 };  **if** (numbers.empty())      std::cout << "The forward\_list is empty" << std::endl;  **else**      std::cout << "The forward\_list is not empty" << std::endl; |

Для изменения размера контейнера можно использовать функцию **resize()**, которая 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

Использование функции:

|  |
| --- |
| std::forward\_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

Применение функции:

|  |
| --- |
| std::forward\_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::forward\_list<**int**> list1{ 1, 2, 3, 4, 5 };  std::forward\_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 forward\_list применяются следующие функции:

* **push\_front(val)**: добавляет объект val в начало списка
* **emplace\_front(val)**: добавляет объект val в начало списка
* **emplace\_after(p, val)**: вставляет объект val после элемента, на который указывает итератор p. Возвращает итератор на вставленный элемент. Если p представляет итератор на позицию после конца списка, то результат неопределен.
* **insert\_after(p, val)**: вставляет объект val после элемента, на который указывает итератор p. Возвращает итератор на вставленный элемент.
* **insert\_after(p, n, val)**: вставляет n объектов val после элемента, на который указывает итератор p. Возвращает итератор на последний вставленный элемент.
* **insert\_after(p, begin, end)**: вставляет после элемента, на который указывает итератор p, набор объектов из другого контейнера, начало и конец которого определяется итераторами begin и end. Возвращает итератор на последний вставленный элемент.
* **insert\_after(p, il)**: вставляет после элемента, на который указывает итератор p, список инициализации il. Возвращает итератор на последний вставленный элемент.

Применение функций:

|  |
| --- |
| #include <iostream>  #include <list>  #include <forward\_list>    **int** main()  {      std::forward\_list<**int**> numbers{ 7, 8 };      numbers.push\_front(6);  // добавляем в начало число 6      // numbers =  { 6, 7, 8 }        numbers.emplace\_front(-3);  // добавляем в начало число -3      // numbers =  { -3, 6, 7, 8 }        auto iter = numbers.begin();      iter = numbers.emplace\_after(iter, -2); // добавляем после итератора число -2      // numbers =  { -3, -2, 6, 7, 8 }        iter = numbers.insert\_after(iter, -1);      // numbers =  { -3, -2, -1, 6, 7, 8 }        iter = numbers.insert\_after(iter, 3, 0);    // добавляем три нуля      // numbers =  { -3, -2, -1, 0, 0, 0, 6, 7, 8 }        std::list<**int**> values{ 1, 2, 3 };      iter = numbers.insert\_after(iter, values.begin(), values.end()); // добавляем все элементы из values      // numbers =  { -3, -2, -1, 0, 0, 0, 1, 2, 3, 6, 7, 8 }        numbers.insert\_after(iter, { 4, 5 }); // добавляем список { 4, 5 }      // numbers =  { -3, -2, -1, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8 }    **for**(**int** n : numbers)          std::cout << n << "\t";      std::cout << std::endl;  } |

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Чтобы удалить элемент из контейнера forward\_list, можно использовать следующие функции:

* **clear()**: удаляет все элементы
* **pop\_front()**: удаляет первый элемент
* **erase\_after(p)**: удаляет элемент после элемента, на который указывает итератор p. Возвращает итератор на элемент после удаленного
* **erase\_after(begin, end)**: удаляет диапазон элементов, на начало и конец которого указывают соответственно итераторы begin и end. Возвращает итератор на элемент после последнего удаленного

Использование функций:

|  |
| --- |
| std::forward\_list<**int**> numbers{ 1, 2, 3, 4, 5, 6, 7};    numbers.pop\_front();  // numbers = { 2, 3, 4, 5, 6, 7};    auto iter = numbers.erase\_after(numbers.begin());  // numbers = { 2, 4, 5, 6, 7 };  // iter указывает на элемент 4    numbers.erase\_after(iter, numbers.end());  // numbers = { 2, 4 }; |

## **DEQUE**

**Deque** представляет двухстороннюю очередь. Для использования данного контейнера нужно подключить заголовочный файл **deque**.

Способы создания двухсторонней очереди:

|  |
| --- |
| std::deque<**int**> deque1;                   // пустая очередь  std::deque<**int**> deque2(5);                // deque2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию  std::deque<**int**> deque(5, 2);              // deque3 состоит из 5 чисел, каждое число равно 2  std::deque<**int**> deque4{ 1, 2, 4, 5 };     // deque4 состоит из чисел 1, 2, 4, 5  std::deque<**int**> deque5 = { 1, 2, 3, 5 };  // deque5 состоит из чисел 1, 2, 3, 5  std::deque<**int**> deque6({ 1, 2, 3, 4, 5 }); // deque6  состоит из чисел 1, 2, 3, 4, 5  std::deque<**int**> deque7(deque4);           // deque7 - копия очереди deque4  std::deque<**int**> deque8 = deque7;          // deque8 - копия очереди deque7 |

### **ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ ОЧЕРЕДИ**

Для получения элементов очереди можно использовать операцию [] и ряд функций:

* **[index]**: получение элемента по индексу
* **at(index)**: возращает элемент по индексу
* **front()**: возвращает первый элемент
* **back()**: возвращает последний элемент

|  |
| --- |
| #include <iostream>  #include <deque>    **int** main()  {      std::deque<**int**> numbers { 1, 2, 3, 4, 5 };    **int** first = numbers.front();    // 1  **int** last = numbers.back();      // 5  **int** second = numbers[1];        // 2  **int** third = numbers.at(2);      // 3      std::cout << first << second << third << last << std::endl; // 1235  } |

Стоит отметить, что если мы будем обращаться с помощью операции индексирования по некорректному индексу, который выходит за границы контейнера, то результат будет неопредленным:

|  |
| --- |
| std::deque<**int**> numbers { 1, 2, 3, 4, 5 };  **int** eighth = numbers[7]; |

В этом случае использование функции **at()** является более предпочтительным, так как при обращении по некорректному индексу она генерирует исключение **out\_of\_range**:

|  |
| --- |
| #include <iostream>  #include <deque>    **int** main()  {      std::deque<**int**> numbers { 1, 2, 3, 4, 5};  **try**      {  **int** n { numbers.at(7) };          std::cout << n << std::endl;      }  **catch** (**const** std::out\_of\_range&)      {          std::cout << "Incorrect index" << std::endl;      }  } |

Также в цикле или с помощью итераторов можно перебрать элементы контейнера:

|  |
| --- |
| #include <iostream>  #include <deque>    **int** main()  {      std::deque<**int**> numbers { 1, 2, 3, 4, 5 };    **for** (**int** n : numbers)          std::cout << n << "\t";      std::cout << std::endl;    **for** (unsigned i {}; i < numbers.size(); i++)          std::cout << numbers[i] << "\t";      std::cout << std::endl;    **for** (auto iter = numbers.begin(); iter != numbers.end(); iter++)          std::cout << \*iter << "\t";      std::cout << std::endl;    **return** 0;  } |

### **РАЗМЕР ОЧЕРЕДИ**

Чтобы узнать размер очереди, можно использовать функцию **size()**. А функция **empty()** позволяет узнать, содержит ли очередь элементы. Она возвращает значение true, если в очереди есть элементы:

|  |
| --- |
| std::deque<**int**> numbers { 1, 2, 3, 4, 5 };  **if** (numbers.empty())  {      std::cout << "Deque is empty" << std::endl;  }  **else**  {  **int** n {numbers.size()};      std::cout << "Deque has " << n << " elements" << std::endl;  } |

Функция **resize()** позволяет изменить размер очереди. Эта функция имеет две формы:

* **resize(n)**: оставляет в очереди n первых элементов. Если deque содержит больше элементов, то размер контейнера усекается до первых n элементов. Если размер очереди меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию
* **resize(n, value)**: также оставляет в очереди n первых элементов. Если размер очереди меньше n, то добавляются недостающие элементы со значением value

Применение функции:

|  |
| --- |
| std::deque<**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} |

Важно учитывать, что применение функции resize может сделать некорректными все итераторы, указатели и ссылки на элементы.

### **ИЗМЕНЕНИЕ ЭЛЕМЕНТОВ ОЧЕРЕДИ**

Функция **assign()** позволяет заменить все элементы очереди 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

Применение функции:

|  |
| --- |
| std::deque<**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::deque<**int**> values { 6, 7, 8, 9, 10, 11 };  auto start = values.begin() + 2;    // итератор указывает на третий элемент  auto end = values.end();            // итератор указывает на последний элемент  numbers.assign(start, end);         //  numbers = { 8, 9, 10, 11 } |

Функция **swap()** обменивает значениями две очереди:

|  |  |
| --- | --- |
| 1  2  3 | std::deque<**int**> deque1 { 1, 2, 3, 4, 5 };  std::deque<**int**> deque2 { 6, 7, 8, 9};  deque1.swap(deque2);    // deque1 = { 6, 7, 8, 9}; |

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ**

Чтобы добавить элементы в очередь deque, можно применять ряд функций:

* **push\_back(val)**: добавляет значение val в конец очереди
* **push\_front(val)**: добавляет значение val в начало очереди
* **emplace\_back(val)**: добавляет значение val в конец очереди
* **emplace\_front(val)**: добавляет значение val в начало очереди
* **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::deque<**int**> numbers { 1, 2, 3, 4, 5 };  numbers.push\_back(6);   // { 1, 2, 3, 4, 5, 6 }  numbers.push\_front(0);  // { 0, 1, 2, 3, 4, 5, 6 }  numbers.emplace\_back(7);    // { 0, 1, 2, 3, 4, 5, 6, 7 }  numbers.emplace\_front(-1);  // { -1, 0, 1, 2, 3, 4, 5, 6, 7 } |

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

|  |
| --- |
| std::deque<**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::deque<**int**> numbers1 { 1, 2, 3, 4, 5 };  auto iter1 = numbers1.cbegin(); // итератор указывает на второй элемент  numbers1.insert(iter1 + 2, 8); // добавляем после второго элемента  //numbers1 = { 1, 2, 8, 3, 4, 5};    std::deque<**int**> numbers2 { 1, 2, 3, 4, 5 };  auto iter2 = numbers2.cbegin(); // итератор указывает на первый элемент  numbers2.insert(iter2, 3, 4); // добавляем вначало три четверки  //numbers2 = { 4, 4, 4, 1, 2, 3, 4, 5};    std::deque<**int**> values { 10, 20, 30, 40, 50 };  std::deque<**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::deque<**int**> numbers4 { 1, 2, 3, 4, 5 };  auto iter4 = numbers4.cend();   // итератор указывает на позицию за последним элементом  // добавляем после последнего элемента список { 21, 22, 23 }  numbers4.insert(iter4, { 21, 22, 23 });  //numbers4 = { 1, 2, 3, 4, 5, 21, 22, 23}; |

При добавлении в контейнер deque следует учитывать, что добавление может сделать недействительными все итераторы, указатели и ссылки на элементы контейнера.

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Для удаления элементов из контейнера deque используются 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

Применение функций:

|  |
| --- |
| std::deque<**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} |

При удалении стоит учитывать, что удаление элементов из любой позиции (за исключением удаления первого и последнего элементов) делает все итераторы, указатели и ссылки на элементы deque недействительными.

Таким образом, deque, как и vector и array, поддерживает произвольный доступ к элементам контейнера, но в отличие от вектора также поддерживает добавление в начало контейнера. Кроме того, во внутренней реализации deque при изменении размера не выделяет новый массив в памяти для вмещения нового набора элементов, а манипулирует указателями.

## **СТЕК STD::STACK**

Класс **std::stack<T>** представляет **стек** - структуру данных, которая работает по принципу LIFO (last-in first-out или "последний вошел — первым вышел") — первым всегда извлекается последний добавленный элемент. Стек можно сравнить со стопкой предметов, например, стопкой тарелок - тарелки добавляются сверху, каждая последующая тарелка кладется поверх предыдущей. А если надо взять тарелку, то сначала берется та, которая в самом верху (которую положили самой последней).

Для работы со стеком надо подключать заголовочный файл <stack>. Определение пустого стека:

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::stack<std::string> stack;    // пустой стек строк  } |

### **РАЗМЕР СТЕКА**

С помощью функции **size()** можно получить количество элементов в стеке, а с помощью функции **empty()** проверить стек на наличие элементов (если возвращается true, то стек пуст):

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::stack<std::string> stack;  **if**(stack.empty())      {          std::cout << "stack is empty" << std::endl;      }      std::cout << "stack size: " << stack.size() << std::endl; // stack size: 0  } |

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ**

Для добавления в стек применяется функция **push()**, в которую передается добавляемый элемент:

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::stack<std::string> stack;      // добавляем три элемента      stack.push("Tom");      stack.push("Bob");      stack.push("Sam");        std::cout << "stack size: " << stack.size() << std::endl; // stack size: 3  } |

### **ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ**

Мы можем получить только самый верхний элемент стека - для этого применяется функция **top()**:

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::stack<std::string> stack;      stack.push("Tom");      stack.push("Bob");      stack.push("Sam");      std::cout << "Top: " << stack.top() << std::endl;   // Top: Sam  } |

В данном случае после добавления трех элементов стек будет выглядеть следующим образом:

-------

| Sam |

-------

| Bob |

-------

| Tom |

-------

На верхушке стека будет располагаться последний добавленный элемент. И с помощью функции top() можно получить этот элемент

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Для удаления элементов применяется функция **pop()**. Удаление производится в порядке, обратном добавлению:

|  |
| --- |
| stack.pop() |

Комбинируя эту функцию с функцией top() можно извлечь все элементы стека:

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::stack<std::string> stack;      stack.push("Tom");      stack.push("Bob");      stack.push("Sam");    **while** (!stack.empty())  // пока стек не пуст      {          std::cout << stack.top() << std::endl;          stack.pop();    // извлекаем верхний элемент      }  } |

В данном случае, пока стек не станет пустым, выводим на консоль верхний (последний добавленный) элемент с помощью функции top и затем извлекаем его с помощью функции pop. Консольный вывод программы:

Sam

Bob

Tom

### **ИНИЦИАЛИЗАЦИЯ ДЕКОМ**

Стек можно инициализировать другим стеком или деком (двусторонней очередью):

|  |
| --- |
| #include <iostream>  #include <stack>    **int** main()  {      std::deque<std::string> users{"Tom", "Sam", "Bob"};      std::stack<std::string> stack {users};    **while** (!stack.empty())      {          std::cout << stack.top() << std::endl;          stack.pop();      }  } |

## **ОЧЕРЕДЬ STD::QUEUE**

Класс **std::queue<T>** представляет **очередь** - контейнер, который работает по принципу FIFO (first-in first-out или "первый вошел — первым вышел") — первым всегда извлекается первый добавленный элемент. То есть это контейнер, аналогичный стандартной очереди, которая часто встречается в нашей повседневной жизни.

Для работы со очередью надо подключать заголовочный файл <queue>. Определение пустой очереди:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue;  // пустая очередь  } |

### **РАЗМЕР ОЧЕРЕДИ**

С помощью функции **size()** можно получить количество элементов в очереди, а с помощью функции **empty()** проверить очередь на наличие элементов (если возвращается true, то очередь пуста):

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue;  **if**(queue.empty())      {          std::cout << "queue is empty" << std::endl;      }      std::cout << "queue size: " << queue.size() << std::endl; // queue size: 0  } |

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ**

Для добавления в очередь применяется функция **push()**, в которую передается добавляемый элемент:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue;  // пустая очередь      // добавляем три элемента      queue.push("Tom");      queue.push("Bob");      queue.push("Sam");        std::cout << "queue size: " << queue.size() << std::endl; // queue size: 3  } |

### **ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ**

Мы можем получить только самый первый элемент очереди - для этого применяется функция **front()** и с самый последний с помощью функции **back()**:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue;  // пустая очередь      queue.push("Tom");      queue.push("Bob");      queue.push("Sam");        std::cout << "First: " << queue.front() << std::endl;   // First: Tom      std::cout << "Last: " << queue.back() << std::endl;     // Last: Sam  } |

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Для удаления элемента из начала очереди применяется функция **pop()**:

|  |  |
| --- | --- |
| 1 | queue.pop() |

Комбинируя эту функцию с функцией front() можно извлечь все элементы стека:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue;      queue.push("Tom");      queue.push("Bob");      queue.push("Sam");    **while** (!queue.empty())  // пока очередь не пуста      {          std::cout << queue.front() << std::endl;          queue.pop();    // извлекаем первый элемент      }  } |

В данном случае, пока очередь не станет пустой, выводим на консоль первый элемент с помощью функции front и затем извлекаем его с помощью функции pop. Консольный вывод программы:

Tom

Bob

Sam

### **ИНИЦИАЛИЗАЦИЯ ДЕКОМ**

Очередь можно инициализировать другой очередь. или деком (двусторонней очередью):

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::deque<std::string> people{"Tom", "Sam", "Bob"};      std::queue<std::string> queue {people};    **while** (!queue.empty())      {          std::cout << queue.front() << std::endl;          queue.pop();      }  } |

### **ОБМЕН ЭЛЕМЕНТАМИ**

С помощью функции **swap()** можно обменять элементы двух очередей:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::queue<std::string> queue1 {std::deque<std::string>{"Alice", "Kate"}};        std::queue<std::string> queue2;      queue2.push("Tom");      queue2.push("Bob");        // обмен queue1 на queue2      queue1.swap(queue2);      // queue1 = {"Tom", "Bob"}      // queue2 = {"Alice", "Kate"}  } |

## **ОЧЕРЕДЬ ПРИОРИТЕТОВ STD::PRIORITY\_QUEUE**

Класс **std::priority\_queue<T>** представляет **очередь приоритетов** - контейнер, который, как и станлдартная очередь, работает по принципу FIFO. Данный класс определен в заголовочном файле <queue> (там же, где и класс queue), однако в плане функционала больше похож на класс stack.

Определение пустой очереди приоритетов:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<std::string> queue;  } |

### **РАЗМЕР ОЧЕРЕДИ**

С помощью функции **size()** можно получить количество элементов в стеке, а с помощью функции **empty()** проверить стек на наличие элементов (если возвращается true, то стек пуст):

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<std::string> queue;  **if**(queue.empty())      {          std::cout << "queue is empty" << std::endl;      }      std::cout << "queue size: " << queue.size() << std::endl; // queue size: 0  } |

### **ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ**

Для добавления в очередь приоритетов применяется функция **push()**, в которую передается добавляемый элемент:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<std::string> queue;        // добавляем три элемента      queue.push("Tom");      queue.push("Bob");      queue.push("Sam");        std::cout << "queue size: " << queue.size() << std::endl; // queue size: 3  } |

### **ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ**

Мы можем получить только самый первый элемент очереди - для этого применяется функция **top()**:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<std::string> queue;        // добавляем три элемента      queue.push("Sam");      queue.push("Tom");      queue.push("Bob");        std::cout << "First: " << queue.top() << std::endl; // Top: Tom  } |

Обратите внимание, что первой добавляется строка "Sam", а последней - строка "Bob", однако первой (условно более приоритетной) мы получаем строку "Tom". В данном случае мы как раз сталкиваемся с действием приоритетов. При добавлении элементов в очередь приоритетов применяется функция компаратора, которая сравнивает добавляемые элементы и располагает их в очереди в определенном порядке.

По умолчанию применяется для сравнения данных применяется функция, которая располагает первыми элементы, которые условно "больше". Например, строка "Tom" условно больше, чем "Sam" или "Bob", потому что буква "T" располагается в алфавите после "S" и "B". Соответственно очередь будет выглядеть таким образом:

Tom(1) - Sam (2) - Bob(3)

Другой пример:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<**int**> numbers;        numbers.push(4);      numbers.push(22);      numbers.push(13);        std::cout << "First: " << numbers.top() << std::endl; // Top: 22  } |

В данном случае элементы будут располагаться следующим образом:

22 - 13 - 4

### **УДАЛЕНИЕ ЭЛЕМЕНТОВ**

Для удаления элементов применяется функция **pop()**, которая извлекает элемент из начала очереди:

|  |  |
| --- | --- |
| 1 | queue.pop() |

Комбинируя эту функцию с функцией top() можно извлечь все элементы очереди:

|  |
| --- |
| #include <iostream>  #include <queue>    **int** main()  {      std::priority\_queue<std::string> queue;      queue.push("Sam");      queue.push("Tom");      queue.push("Bob");    **while** (!queue.empty())  // пока очередь не станет пустой      {          std::cout << queue.top() << std::endl;          queue.pop();    // извлекаем первый элемент      }  } |

В данном случае, пока очередь не станет пустой, выводим на консоль первый (самый приоритетный) элемент с помощью функции top и затем извлекаем его с помощью функции pop. Консольный вывод программы:

Tom

Sam

Bob