**Оглавление**

**Лабораторная работа №1 Перечисление. Объединение. STL контейнеры. Массив. Дек. Односвязный и двусвязный список. Вектор.**

# Лабораторная работа №1 Перечисление. Объединение. STL контейнеры. Массив. Дек. Односвязный и двусвязный список. Вектор.

Итак, начнем с самого базового понятия, от которого будем отталкиваться. Что такое STL и что такое STL-контейнеры?

**STL** – стандартная библиотека шаблонов. STL обеспечивает стандартные классы и функции, которые позволяют реализовать наиболее популярные и широко используемые алгоритмы и структуры данных. Так как фундаментом STL являются шаблоны классов, входящие в STL алгоритмы и структуры данных применяются почти ко всем типам данных. Удобно, не так ли?

**Контейнеры** – это объекты, предназначенные для хранения набора элементов. Например, вектор, линейный список, дэк. С ними мы познакомимся ниже. В каждом классе-контейнере определен набор функций для работы с ними.

**Алгоритмы** – выполняют операции над содержимым контейнера. Существует множество алгоритмов: инициализации, сортировки, поиск, замены содержимого и многие другие.

**Итераторы** – наши верные помощники для работы с контейнерами. Они играют роль указателей по отношению к контейнеру и позволяют получить доступ к содержимому контейнера. Работа с итераторами похожа во многом на работу с указателями.

**Перечисление enum** – сгруппированный набор целочисленных констант. То есть это просто набор названий из законченной сборки. Это могут быть цвета радуги, дни недели, имена людей. Используются перечисления, когда нужно объединить несколько констант в одну группу, обозначив каждую символическим, удобным для чтения названием. Объявление enum не требует выделения дополнительной памяти. Память выделяется лишь тогда, когда переменная перечисления определена.

! Одна из самых распространенных ошибок – точка с запятой в конце перечисления. Не забывайте поставить точку с запятой и разделить элементы перечисления запятыми.

|  |
| --- |
| Пример: |
| *Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить название планеты Солнечной системы, с использованием перечислений.*  В программе ниже представлено перечисление planets со списком планет, объявленное ключевым словом enum. По умолчанию нумерация перечисления будет начинаться с нуля, но мы изменили нумерацию присвоив первой константе единицу. Пользователь вводит число с клавиатуры, которое будет соответствовать названию определенной планеты. Синтаксис case planets::mercury – то же, что и case 1, напомним, что выше мы изменили нумерацию. |
| #include <iostream>  using namespace std;  enum planets {// объявим наше перечисление  mercury = 1,  venus,  earth,  mars,  jupiter,  saturn,  uran  };  int numInput(int num) { // в случае ошибки ввода наша программа не будет останавливаться  cin >> num;  if (cin.fail()) {  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  cout << "\nВы ввели не число, введите повторно!\n" << endl;  numInput(num);  }  else return num;  }  int main()  {  setlocale(LC\_ALL, "rus");  int number = 0;  cout << "Какая по расположению планета от солнца?\n"  << " 1 - меркурий,\n 2 - венера,\n 3 - земля,\n 4 - марс,\n 5 - юпитер,\n 6 - сатурн,\n 7 - уран\n 0 - выход\n";  do {  number = numInput(number);  switch (number)  {  case planets::mercury:  cout << "меркурий\n";  break;  case planets::venus:  cout << "венера\n";  break;  case planets::earth:  cout << "земля\n";  break;  case planets::mars:  cout << "марс\n";  break;  case planets::jupiter:  cout << "юпитер\n";  break;  case planets::saturn:  cout << "сатурн\n";  break;  case planets::uran:  cout << "уран\n";  break;  case 0:  break;  default:  cout << "\nНеверное значение, введите повторно!\n";  break;  }  } while (number != 0);  } |

Как мы уже выяснили, каждой константе сопоставляется некоторое числовое значение.

Можно назначить каждой константе индивидуальное значение или сочетать этот подход с автоустановкой. В данном случае saturn будет равно 7, а uran - 1.

enum planets {

mercury = 2,

venus,

earth,

mars,

jupiter,

saturn,

uran = 1

};

Можно даже назначить двум константам одно и то же значение.

enum planets {

mercury = 1,

merc = 1

venus,

earth,

mars,

jupiter,

saturn,

uran

};

Существует и подобный вариант:

enum planets {

mercury = 1,

merc = mercury,

venus,

earth,

mars,

jupiter,

saturn,

uran

};

В общем, существуем множество вариантов, которые можно реализовать с помощью перечислений.

**Объединения union** - позволяют определить свой тип данных и также хранят набор элементов. Похожи на структуры, но в отличие от структуры все элементы объединения имеют нулевое смещение. А это значит, что разные элементы занимают в памяти один и тот же участок, то есть в памяти они накладываются друг на друга. Объявляется объединение ключевым словом union. После объявления можно создать его переменную и присвоить ей какое-либо значение.

Объявление:

union myValues {

int x;

char y;

};

Создадим переменную объединения:

union myValues value;

Для обращения к элементам перечисления используется операция точка:

value.x = 4;

Можно определять анонимные объединения:

union {

int x;

char y;

};

Можно определить указатель на объединения. Синтаксис следующий:

(\* указатель\_на\_объединение).имя\_элемента

указатель\_на\_объединение->имя\_элемента

|  |
| --- |
| Пример: |
| Ниже представлен простой пример использования union. Как видно из примера, любое изменение, независимо от выбранной переменной из объединения буквально выталкивает уже сохранённое значение. Несложно догадаться, что на одном адресе можно хранить только что-то одно, поэтому происходит своеобразная перепись с затиранием. Поскольку для записи используется одна ячейка памяти, происходит экономия байтов. Байт — это единица информации. Для структуры с тем же описанием, что у MyUnion могло бы потребоваться (>= short + int) байтов в то время, когда для непосредственно нашего MyUnion нужно только int байтов. |
| #include <iostream>  union MyUnion  {  short x; //для x тип short  int y; //для y типом int  };  int main()  {  using namespace std;  MyUnion u; //Создаём объект u объединения MyUnion  u.x = 200; //Записываем в u.x Значение 200    cout << u.x << endl; // Выведется значение 200    u.y = 111; //Записываем в u.y значение 111  cout << u.x; //u.x было переписано последней записью, поскольку всё пишется в одну ячейку памяти  } |

**Контейнерный класс** (или ***«класс-контейнер»***) в языке C++ — это [**класс**](https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/), предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных).

**Массив** – наиболее часто используемый класс-контейнер в программировании. Из одноименного модуля <array>. Чтобы создать класс-контейнер array необходимо указать в угловых скобках тип данных в массиве и его размер:

std::array<int, 5> numbers;

*std::array* – непосредственно класс-контейнер

*int* – тип массива,

*5* – размер,

*numbers* – имя нашего контейнера.

По умолчанию все элементы имеют неопределенное значение. Мы можем инициализировать элементы частично или полностью:

std::array<int, 5> numbers {2, 3, 4}; // остальные элементы получат значение по умолчанию

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

! Если передать больше значений, возникнет ошибка.

Для **доступа к содержимому** контейнера можно использовать метод, как и при работе с массивами: указывать индекс в скобках.

**Перебор значений** можно осуществить с помощью цикла for или for-each.

**Основные функции array:**

1. size() – возвращает размер контейнера
2. at(index) – возвращает элемент контейнера по индексу в скобках
3. front() – возвращает первый элемент array
4. back() – возвращает последний элемент array
5. fill(n) – присвоить всем элементам значение в скобках

Контейнеры array можно сравнивать между собой операторами !=, ==, >, <.

|  |
| --- |
| Пример |
| Ниже представлена программа, где необходимо найти максимальный и минимальный элементы, а также их сумму. Контейнер заполняется случайными числами. |
| #include <iostream>  #include <array>  using namespace std;  int main()  {  array<int, 10> arr = { };  int max = arr[0], min = arr[0];  for (int i = 0; i < 10; i++)  {  arr[i] = rand() % 20 - 5;  if (arr[i] > max) max = arr[i];  if (arr[i] < min) min = arr[i];  cout << arr[i] << " ";  }  cout << endl << "max = " << max << endl;  cout << "min = " << min << endl;  cout << "sum = " << max + min << endl;  } |

**Дек** – представляет собой двустороннюю очередь. Для использования данного контейнера необходимо подключить заголовочный файл <deque>. Способы создания двусторонней очереди аналогичны array. Ниже представлены пару примеров:

std::deque<int> deque; - пустая очередь

std::deque<int> deque(5); - очередь из пяти элементов

std::deque<int> deque = { 1, 2, 3, 5 };

std::deque<int> deque({ 1, 2, 3, 4, 5 });

**Для получения элементов очереди можно использовать следующие операции:**

1. deque[index] – получение элемента по индексу
2. deque.at(index) – возвращает элемент дека по индексу
3. deque.front() – возвращает первый элемент дека
4. deque.back() – возвращает последний элемент дека

! Если обращаться через deque[index], результат может выйти за пределы дека, то есть значение будет неопределенным. В таком случае предпочтительнее использовать deque.at(index), так как при подобном обращении сгенерирует ошибку out\_of\_range.

**Основные функции для работы с очередью:**

1. size() – чтобы узнать размер очереди.
2. empty() – чтобы узнать, содержит очередь элементы, или она пуста
3. resize() – изменение размера очереди
   1. resize(n) – оставляет в очереди лишь n первых элементов
   2. resize(n, val) – оставляет n первых элементов, но если n больше размера очереди, то вставляет недостающее количество val. Можно использовать для добавления повторяющихся элементов очереди.

! применение функции resize может сделать некорректными все итераторы, указатели и ссылки на элементы.

1. assign() – заменить элементы очереди определенным набором.
   1. assign(il) – заменяет элементы очереди элементами il
   2. assign(n, value) – заменяет содержимое n элементами со значением val
   3. assign(begin, end) – заменяет элементами из диапазона begin – end
2. swap() – обменивает значениями две очереди
3. push\_back(val): добавляет значение val в конец очереди
4. push\_front(val): добавляет значение val в начало очереди
5. emplace\_back(val): добавляет значение val в конец очереди
6. emplace\_front(val): добавляет значение val в начало очереди
7. emplace(pos, val): вставляет элемент val на позицию pos. Возвращает итератор на добавленный элемент
8. insert(pos, val): вставляет элемент val на позицию pos, аналогично функции emplace. Возвращает итератор на добавленный элемент
9. insert(pos, n, val): вставляет n элементов val начиная с позиции pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.
10. insert(pos, begin, end): вставляет начиная с позиции pos, элементы из другого контейнера из диапазона begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.
11. insert(pos, values): вставляет список значений values начиная с позиции pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.
12. clear(p): удаляет все элементы
13. pop\_back(): удаляет последний элемент
14. pop\_front(): удаляет первый элемент
15. erase(p): удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент
16. erase(begin, end): удаляет элементы из диапазона begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

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

deque при изменении размера не выделяет новый массив в памяти для вмещения нового набора элементов, а манипулирует указателями.

|  |
| --- |
| Пример 1 |
| Ниже представлена программа, в которой находится минимальный элемент дека и добавляется в начало с выводом на экран. |
| #include <iostream>  #include <deque>  #include <algorithm>  using namespace std;  int main()  {  deque<int> num;  setlocale(LC\_ALL, "rus");  cout << "очередь: " << endl;  for (int i = 0; i < 10; i++)  {  num.push\_back(rand() % 20 - 5);  cout << num[i] << "\t";  }  cout << endl;  int max = \*max\_element(num.begin(), num.end()); // ищем минимальный элемент от начала до конца  int min = \*min\_element(num.begin(), num.end());  cout << "max = " << max << endl;  cout << "min = " << min;  num.push\_front(min);  cout << endl << "очередь с минимальным элементом в начале: " << endl;    for (int i = 0; i < num.size(); i++)  {  cout << num[i] << "\t";  }  return 0;  } |
| Пример 2 |
| В программе ниже находим произведение всех элементов дека |
| #include <iostream>  #include <deque>  #include <algorithm>  using namespace std;  int main()  {  deque<int> num;  setlocale(LC\_ALL, "rus");  cout << "очередь: " << endl;  int p = 1;  for (int i = 0; i < 10; i++)  {  num.push\_back(rand() % 20 - 5);  cout << num[i] << "\t";  p \*= num[i];  }  cout << endl << "произведение: " << p;  return 0;  } |

**Односвязный список** – такой список, где каждый элемент хранит указатель на следующий элемент. Для использования односвязного списка, необходимо подключить библиотеку forward\_list. Создание односвязного списка похоже на то, что мы видели выше в очереди и массиве, поэтому сразу перейдем к разбору полезных методов для работы с односвязным списком.

**Основные функции для работы с односвязным списком:**

1. front() – с помощью данной функции мы можем напрямую получить доступ к первому элементу списка
2. begin() – итератор на начало списка
3. end() – итератор на конец списка
4. before\_begin() – возвращает итератор на несуществующий элемент перед началом списка
5. cbefore\_begin() - возвращает константный итератор на несуществующий элемент перед началом списка. К значению по этому итератору обратиться мы не сможем, как бы сильно не хотелось
6. max\_size() – позволяет получить максимальный размер контейнера
7. insert\_after(p, val): вставляет объект val после элемента, на который указывает итератор p. Возвращает итератор на вставленный элемент.
8. insert\_after(p, n, val): вставляет n объектов val после элемента, на который указывает итератор p. Возвращает итератор на последний вставленный элемент.
9. 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.
10. insert\_after(p, il): вставляет после элемента, на который указывает итератор p, список инициализации il. Возвращает итератор на последний вставленный элемент.
11. erase\_after(p): удаляет элемент после элемента, на который указывает итератор p. Возвращает итератор на элемент после удаленного
12. 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

! Помимо вышеперечисленных вы можете использовать большинство методов, который мы перечисляли в очереди, так как большинство методов могут работать с разными типами контейнеров.

|  |
| --- |
| Пример 1 |
| В программе ниже в конец односвязного списка добавляется n элементов |
| #include <iostream>  #include <forward\_list>  using namespace std;  int numInput(int num) { // в случае ошибки ввода наша программа не будет останавливаться  cin >> num;  if (cin.fail()) {  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  cout << "\nВы ввели не число, введите повторно!\n" << endl;  numInput(num);  }  else return num;  }  int main()  {  setlocale(LC\_ALL, "rus");  forward\_list<int> newlist = { 6, 2, 8, 4, 5 }; // у нас дан певоначальный односвязный список, по желанию можете заполнить его сами  int size = 0, num = 0;  cout << "введите количество, которое хотите добавить = ";  size = numInput(size);  cout << "введите число, которое хотите добавить = ";  num = numInput(num);  cout << "первоначальный список: " << endl;  for (int n : newlist) // удобный синтаксис, который можно использовать вместо цикла for  {  cout << n << "\t";  size++;  }  cout << endl << "конечный список: " << endl;  newlist.resize(size, num);  for (int n : newlist)  {  cout << n << "\t";  }  } |
| Пример 2 |
| В программе нужно удалить все элементы односвязного списка и добавить n новых, с выводом на экран. |
| #include <iostream>  #include <forward\_list>  using namespace std;  int main()  {  setlocale(LC\_ALL, "rus");  srand((unsigned)time(NULL));  forward\_list<int> newlist;  int n;  for (int i = 0; i < 10; i++)  {  newlist.push\_front(rand() % 10);  }  cout << "первоначальный список: ";  for (int n : newlist)  {  cout << n << " ";  }  newlist.clear();    cout << endl << "введите n которое хотите добавить: " << endl;  cin >> n;  cout << "новый список: ";  for (int i = 0; i < n; i++)  {  newlist.push\_front(rand() % 10);  }  for (int n : newlist)  {  cout << n << " ";  }  return 0;  } |

**Двусвязный список** - список, где каждый элемент имеет указатели на предыдущий и последовательный элемент. Мы можем перемещаться по списку с обеих сторон. Для использования заголовочный файл – list.

Создание списка вновь похоже на то, что мы видели выше, его пропустим.

В данном контейнере не определена операция по индексу или функция at(). Зато для доступа можно использовать функции front() и back(), о которых вы уже узнали выше. Однако для доступа к элементам в центре списка, придется выполнять перебор элементов с помощью циклов, хотя и это у вас трудностей не вызовет.

Все основные функции для работы с контейнерами, опять же, применимы и для двусвязного списка.

|  |
| --- |
| Пример |
| В программе ниже в начало двусвязного списка добавляется n элементов и вывести на экран. |
| #include <iostream>  #include <list>  using namespace std;  int numInput(int num) { // в случае ошибки ввода наша программа не будет останавливаться  cin >> num;  if (cin.fail()) {  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  cout << "\nВы ввели не число, введите повторно!\n" << endl;  numInput(num);  }  else return num;  }  int main()  {  setlocale(LC\_ALL, "rus");  list<int> newlist = { 6, 2, 8, 4, 5 };  int size = 0, num = 0;  cout << "введите количество, которое хотите добавить = ";  size = numInput(size);  cout << "введите число, которое хотите добавить = ";  num = numInput(num);  for (int i = 0; i < size; i++)  {  newlist.push\_front(num);  }  cout << endl << "конечный список: " << endl;    for (int n : newlist)  {    cout << n << "\t";  }  } |

**Вектор** - контейнер, который содержит коллекцию объектов одного типа. Для использования подключаем заголовочный файл – vector.

Инициализация похожа на создание контейнеров выше. Однако есть очень важный момент, на который обратим внимание. При инициализации есть отличие круглых и фигурных скобок.

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 равно 5

std::vector<int> v3(5, 2); // вектор состоит из 5 чисел, каждое число равно 2

std::vector<int> v4{5, 2}; // вектор состоит из двух чисел 5 и 2

Обращение к элементам, как и у контейнера array.

Вектор является динамическим массивом, так что мы не ограничены размером, в отличие от array.

**Основные операции для работы с векторами:**

1. push\_back() – добавление элемента в конец контейнера
2. emplace\_back() – выполняет аналогичную задачу

Функции, перечисленные выше, вновь здесь пригодятся.

|  |
| --- |
| Пример 1 |
| В программе ниже находим максимальный и минимальный элементы вектора |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;  int main()  {  srand((unsigned)time(NULL));  vector<int> vect;  for (int i = 0; i < 10; i++)  vect.push\_back(rand() % 20 - 5); // заполняем вектор рандомными числами  vector<int>::iterator beg = vect.begin(); // начало вектора  vector<int>::iterator end = vect.end(); // конец вектора  while (beg != end)  cout << \*(beg++) << "\t";  int max = \*max\_element(vect.begin(), vect.end()); // ищем минимальный элемент от начала до конца  int min = \*min\_element(vect.begin(), vect.end()); // ищем максимальный элемент от начала до конца  cout << endl << "min = " << min << endl;  cout << "max = " << max << endl;    return 0;  } |
| Пример 2 |
| В программе ниже находим сумму элементов вектора |
| #include <iostream>  #include <vector>  #include <algorithm>  using namespace std;  int main()  {  srand((unsigned)time(NULL));  vector<int> vect;  for (int i = 0; i < 10; i++)  {  vect.push\_back(rand() % 20 - 5); // заполняем вектор рандомными числами  cout << vect.at(i) << " ";  }  int sum = 0;  for (int i = 0; i < 10; i++)  {  sum += vect.at(i);    }  cout << endl << "sum = " << sum;  return 0;  } |

! Советую посетить <https://habr.com/ru/companies/infopulse/articles/194726/>, где представлен алгоритм выбора контейнера.

Задания для самостоятельного выполнения к лабораторной работе 1:

1. Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить марку автомобиля, с использованием перечислений.
2. Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить название IT-компании, с использованием перечислений.
3. Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить название STL-контейнера, с использованием перечислений.
4. Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить имена, с использованием перечислений.
5. Написать программу, в которой после максимального элемента односвязного списка добавляется первый элемент в этом же списке.
6. Написать программу, в которой после минимального элемента односвязного списка добавляется последний элемент в этом же списке.